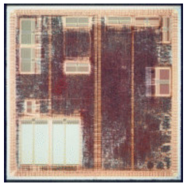
CS162
Operating Systems and Systems Programming
Lecture 1

What is an Operating System?

August 27th, 2007
Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

Who am I?



- Professor John Kubiatowicz (Prof "Kubi")
  - Background in Hardware Design
    - » Alewife project at MIT
    - » Designed CMMU, Modified SPAR C processor
    - » Helped to write operating system
  - Background in Operating Systems
    - » Worked for Project Athena (MIT)
    - » OS Developer (device drivers, network file systems)
    - » Worked on Clustered High-Availability systems (CLAM Associates)
  - Peer-to-Peer
    - » OceanStore project – Store your data for 1000 years
    - » Tapestry and Bamboo – Find you data around globe
  - Quantum Computing
    - » Well, this is just cool, but probably not apropos

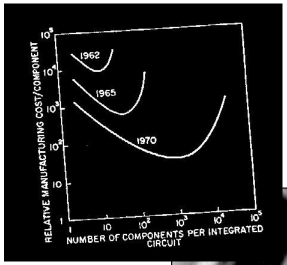
Goals for Today

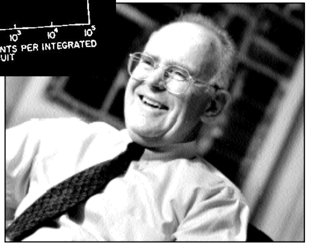
- What is an Operating System?
  - And – what is it not?
- Examples of Operating Systems design
- Why study Operating Systems?
- Oh, and "How does this class operate?"

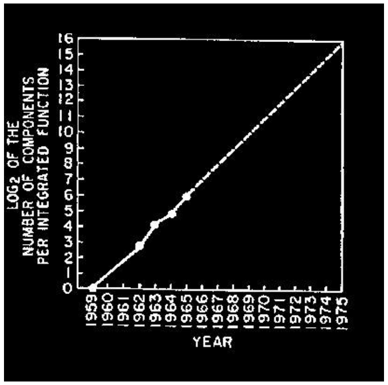
Interactive is important!
Ask Questions!

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Rapid Underlying Technology Change





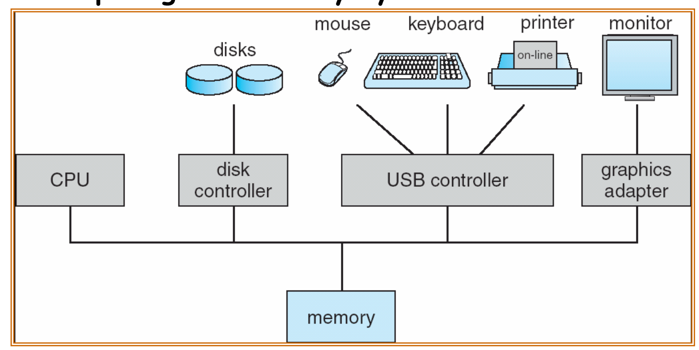


- "Cramming More Components onto Integrated Circuits"
  - Gordon Moore, Electronics, 1965

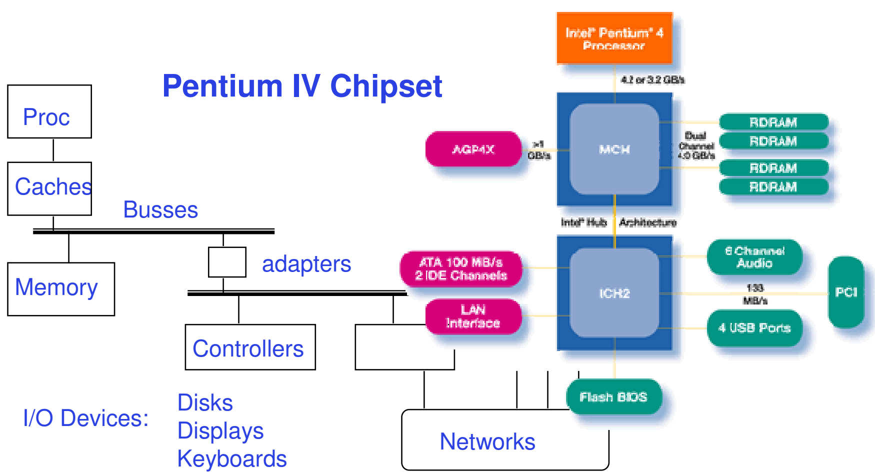
Computing Devices Everywhere

Computer System Organization

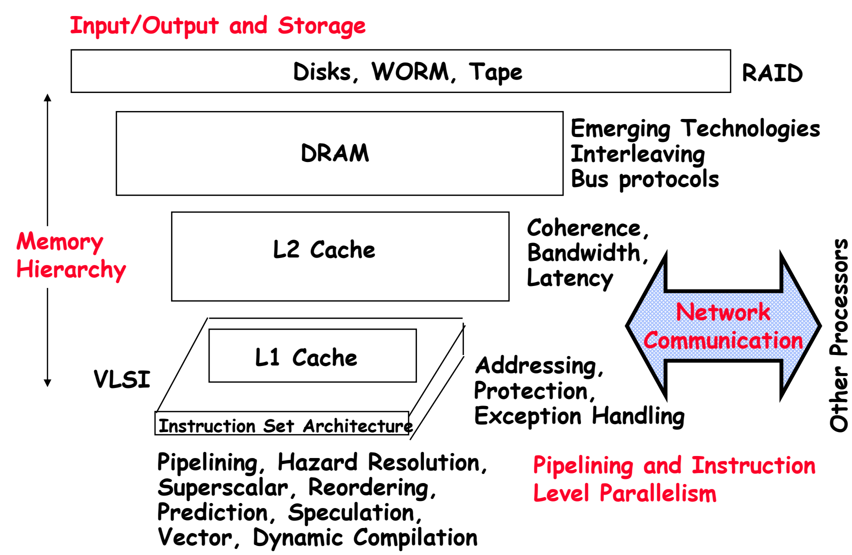
- Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



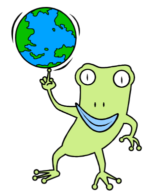
Functionality comes with great complexity!



Sample of Computer Architecture Topics



Example: Some Mars Rover Requirements



- Serious hardware limitations/complexity:
  - 20Mhz powerPC processor, 128MB of RAM
  - cameras, scientific instruments, batteries, solar panels, and locomotion equipment
  - Many independent processes work together
- Can't hit reset button very easily!
  - Must reboot itself if necessary
  - Always able to receive commands from Earth
- Individual Programs must not interfere
  - Suppose the MUT (Martian Universal Translator Module) buggy
  - Better not crash antenna positioning software!
- Further, all software may crash occasionally
  - Automatic restart with diagnostics sent to Earth
  - Periodic checkpoint of results saved?
- Certain functions time critical:
  - Need to stop before hitting something
  - Must track orbit of Earth for communication

How do we tame complexity?

- Every piece of computer hardware different
  - Different CPU
    - » Pentium, PowerPC, ColdFire, ARM, MIPS
  - Different amounts of memory, disk, …
  - Different types of devices
    - » Mice, Keyboards, Sensors, Cameras, Fingerprint readers
  - Different networking environment
    - » Cable, DSL, Wireless, Firewalls,…
- Questions:
  - Does the programmer need to write a single program that performs many independent activities?
  - Does every program have to be altered for every piece of hardware?
  - Does a faulty program crash everything?
  - Does every program have access to all hardware?

OS Tool: Virtual Machine Abstraction

Application

——— Virtual Machine Interface

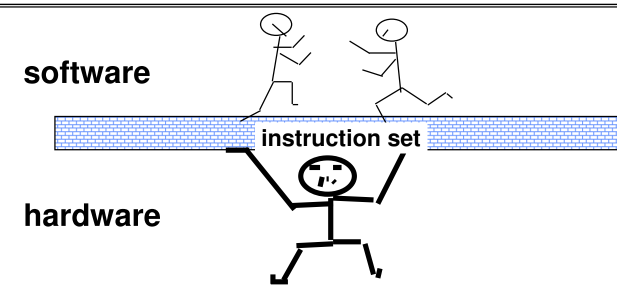
Operating System

——— Physical Machine Interface

Hardware

- Software Engineering Problem:
  - Turn hardware/software quirks ⇒ what programmers want/need
  - Optimize for convenience, utilization, security, reliability, etc…
- For Any OS area (e.g. file systems, virtual memory, networking, scheduling):
  - What's the hardware interface? (physical reality)
  - What's the application interface? (nicer abstraction)

Interfaces Provide Important Boundaries

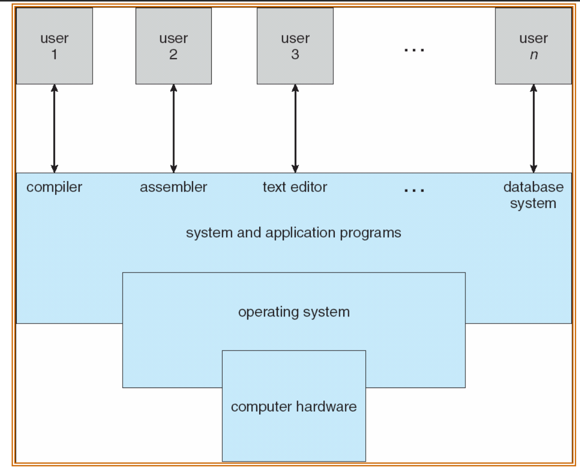


- Why do interfaces look the way that they do?
  - History, Functionality, Stupidity, Bugs, Management
  - CS152 ⇒ Machine interface
  - CS160 ⇒ Human interface
  - CS169 ⇒ Software engineering/management
- Should responsibilities be pushed across boundaries?
  - RISC architectures, Graphical Pipeline Architectures

Virtual Machines

- Software emulation of an abstract machine
  - Make it look like hardware has features you want
  - Programs from one hardware & OS on another one
- Programming simplicity
  - Each process thinks it has all memory/CPU time
  - Each process thinks it owns all devices
  - Different Devices appear to have same interface
  - Device Interfaces more powerful than raw hardware
    - » Bitmapped display ⇒ windowing system
    - » Ethernet card ⇒ reliable, ordered, networking (TCP/IP)
- Fault Isolation
  - Processes unable to directly impact other processes
  - Bugs cannot crash whole machine
- Protection and Portability
  - Java interface safe and stable across many platforms

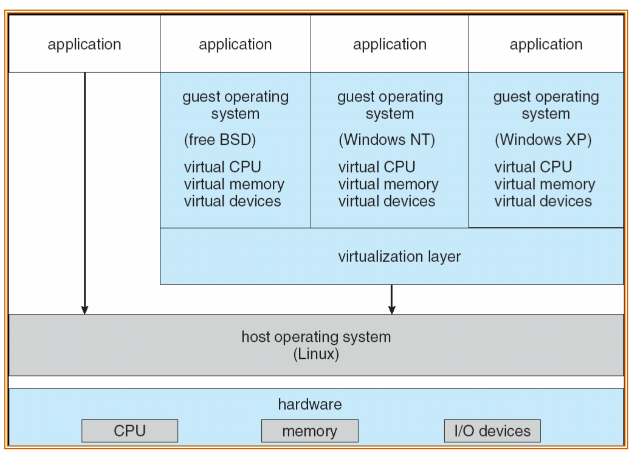
Four Components of a Computer System



Definition: *An operating system implements a virtual machine that is (hopefully) easier and safer to program and use than the raw hardware.*

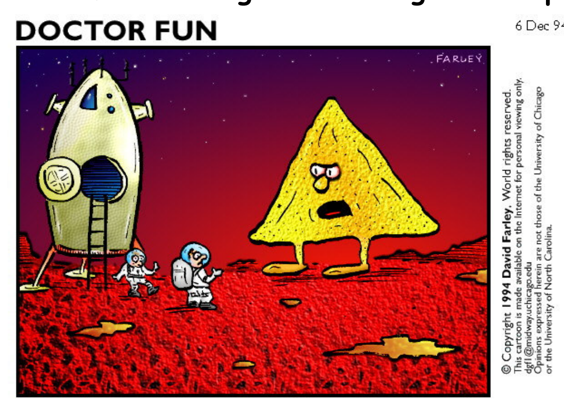
Virtual Machines (con't): Layers of OSs

- Useful for OS development
  - When OS crashes, restricted to one VM
  - Can aid testing programs on other OSs



Nachos: Virtual OS Environment

- You will be working with Nachos
  - Simulation environment
  - Hardware, interrupts, I/O
  - Execution of User Programs running on this platform



"This is the planet where nachos rule."

Course Administration

- Instructor: John Kubiatowicz (kubitron@cs.berkeley.edu)
  675 Soda Hall
  Office Hours(Tentative): M/W 3:00pm-4:00pm
- TAs: Thomas Kho (cs162-ta@cory)
  Todd Kosloff (cs162-tb@cory)
  Kelvin Lwin (cs162-tc@cory)
- Labs: Second floor of Soda Hall
- Website: http://inst.eecs.berkeley.edu/~cs162
  - Mirror: http://www.cs.berkeley.edu/~kubitron/cs162
- Webcast: http://webcast.berkeley.edu/courses/index.php
- Newsgroup: ucb.class.cs162 (use authnews.berkeley.edu)
- Course Email: cs162@cory.cs.berkeley.edu
- Reader: TBA (Stay tuned!)

Class Schedule

- Class Time: M/W 1:00-2:30AM,
  306 Soda Hall, Overflow room: 310 Soda
  - Please come to class. Lecture notes do not have everything in them. The best part of class is the interaction!
  - Also: 5% of the grade is from class participation
- Sections:
  - Important information is in the sections
  - The sections assigned to you by Telebears are temporary!
  - Every member of a project group must be in same section
  - No sections this week

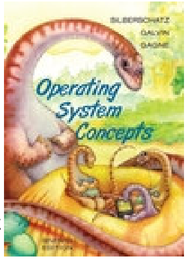
Section	Time	Location	TA
101	Th 10:00-11:00A	81 Evans	TBA
102	Th 12:00-1:00P	155 Barrows	TBA
103	Th 2:00-3:00P	75 Evans	TBA
104	Th 4:00-5:00P	B51 Hildebrand	TBA
105	F 10:00-11:00A	4 Evans	TBA

What about the overflow?

- We have a lot of students signed up:
  - 306 Soda hall can only take 108
  - 310 Soda will handle about 40-50
  - We should be able to take most of you.
    If it turns out we can't:
    - » priority to seniors and majors
    - » Concurrent Enrollment students will be lowest priority
- Alternative for sophomores and possibly juniors:
  New Course! CS194-3/CS16x
  - "Introduction to Computer Systems", Anthony Joseph
    - » Topics: Operating Systems, Networking, Security
    - » M/W 9-10:30, 306 Soda Hall
  - In the future, the existing CS162 will become an advanced operating system course, and this new course will replace it.
  - May be appropriate for a number of you
    - » Take a look at: http://www.cs.berkeley.edu/~adj/cs16x/

Textbook

- Text: Operating Systems Concepts,
  7^{th} Edition Silbershatz, Galvin, Gagne



- Online supplements
  - See "Information" link on course website
  - Includes Appendices, sample problems, etc
- Question: need 7^{th} edition?
  - No, but has new material that we may cover
  - Completely reorganized
  - Will try to give readings from both the 6^{th} and 7^{th} editions on the lecture page

Topic Coverage

Textbook: Silberschatz, Galvin, and Gagne, *Operating Systems Concepts*, 7th Ed., 2005

- 1 week: Fundamentals (Operating Systems Structures)
- 1.5 weeks: Process Control and Threads
- 2.5 weeks: Synchronization and scheduling
- 2 week: Protection, Address translation, Caching
- 1 week: Demand Paging
- 1 week: File Systems
- 2.5 weeks: Networking and Distributed Systems
- 1 week: Protection and Security
- ??: Advanced topics

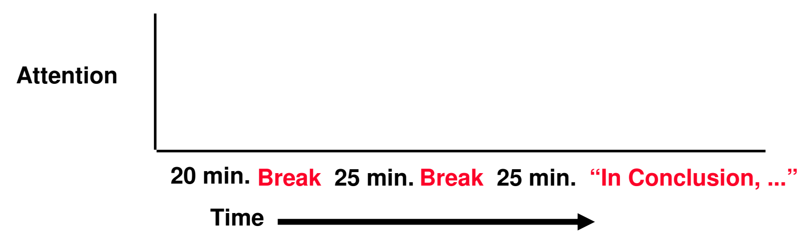
Grading

- Rough Grade Breakdown
  - Two Midterms: 15% each
    One Final: 15%
    Four Projects: 50% (i.e. 12.5% each)
    Participation: 5%
- Four Projects:
  - Phase I: Build a thread system
  - Phase II: Implement Multithreading
  - Phase III: Caching and Virtual Memory
  - Phase IV: Networking and Distributed Systems
- Late Policy:
  - Each group has 5 "slip" days.
  - For Projects, slip days deducted from *all* partners
  - 10% off per day after slip days exhausted

Group Project Simulates Industrial Environment

- Project teams have 4 or 5 members in same discussion section
  - Must work in groups in "the real world"
- Communicate with colleagues (team members)
  - Communication problems are natural
  - What have you done?
  - What answers you need from others?
  - You must document your work!!!
  - Everyone must keep an on-line notebook
- Communicate with supervisor (TAs)
  - How is the team's plan?
  - Short progress reports are required:
    - » What is the team's game plan?
    - » What is each member's responsibility?

Typical Lecture Format



- 1-Minute Review
- 20-Minute Lecture
- 5- Minute Administrative Matters
- 25-Minute Lecture
- 5-Minute Break (water, stretch)
- 25-Minute Lecture
- Instructor will come to class early & stay after to answer questions

Lecture Goal

Interactive!!!

Computing Facilities

- Every student who is enrolled should get an account form at end of lecture
  - Gives you an account of form cs162-xx@cory
  - This account is required
    - » Most of your debugging can be done on other EECS accounts, however…
    - » All of the final runs must be done on your cs162-xx account and must run on the x86 Solaris machines
- Make sure to log into your new account this week and fill out the questions
- Project Information:
  - See the "Projects and Nachos" link off the course home page
- Newsgroup (ucb.class.cs162):
  - Read this regularly!

Academic Dishonesty Policy

- Copying all or part of another person's work, or using reference material not specifically allowed, are forms of cheating and will not be tolerated. A student involved in an incident of cheating will be notified by the instructor and the following policy will apply:

  http://www.eecs.berkeley.edu/Policies/acad.dis.shtml
- The instructor may take actions such as:
  - require repetition of the subject work,
  - assign an F grade or a 'zero' grade to the subject work,
  - for serious offenses, assign an F grade for the course.
- The instructor must inform the student and the Department Chair in writing of the incident, the action taken, if any, and the student's right to appeal to the Chair of the Department Grievance Committee or to the Director of the Office of Student Conduct.
- The Office of Student Conduct may choose to conduct a formal hearing on the incident and to assess a penalty for misconduct.
- The Department will recommend that students involved in a second incident of cheating be dismissed from the University.

What does an Operating System do?

- Silerschatz and Gavin:
  "An OS is Similar to a government"
  - Begs the question: does a government do anything useful by itself?
- Coordinator and Traffic Cop:
  - Manages all resources
  - Settles conflicting requests for resources
  - Prevent errors and improper use of the computer
- Facilitator:
  - Provides facilities that everyone needs
  - Standard Libraries, Windowing systems
  - Make application programming easier, faster, less error-prone
- Some features reflect both tasks:
  - E.g. File system is needed by everyone (Facilitator)
  - But File system must be Protected (Traffic Cop)

What is an Operating System,… Really?

- Most Likely:
  - Memory Management
  - I/O Management
  - CPU Scheduling
  - Communications? (Does Email belong in OS?)
  - Multitasking/multiprogramming?
- What about?
  - File System?
  - Multimedia Support?
  - User Interface?
  - Internet Browser? ☺
- Is this only interesting to Academics??

Operating System Definition (Cont.)

- No universally accepted definition
- "Everything a vendor ships when you order an operating system" is good approximation
  - But varies wildly
- "The one program running at all times on the computer" is the **kernel**.
  - Everything else is either a system program (ships with the operating system) or an application program

What if we didn't have an Operating System?

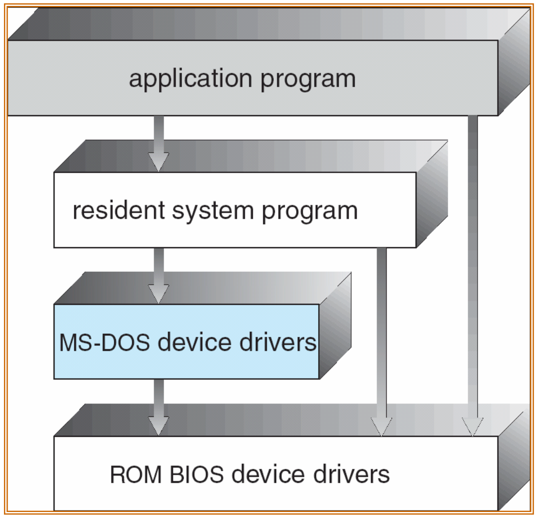
- Source Code⇒Compiler⇒Object Code⇒Hardware
- How do you get object code onto the hardware?
- How do you print out the answer?
- Once upon a time, had to Toggle in program in binary and read out answer from LED's!

Altair 8080

Simple OS: What if only one application?

- Examples:
  - Very early computers
  - Early PCs
  - Embedded controllers (elevators, cars, etc)
- OS becomes just a library of standard services
  - Standard device drivers
  - Interrupt handlers
  - Math libraries

MS-DOS Layer Structure



More thoughts on Simple OS

- What about Cell-phones, Xboxes, etc?
  - Is this organization enough?
- Can OS be encoded in ROM/Flash ROM?
- Does OS have to be software?
  - Can it be Hardware?
  - Custom Chip with predefined behavior
  - Are these even OSs?

More complex OS: Multiple Apps

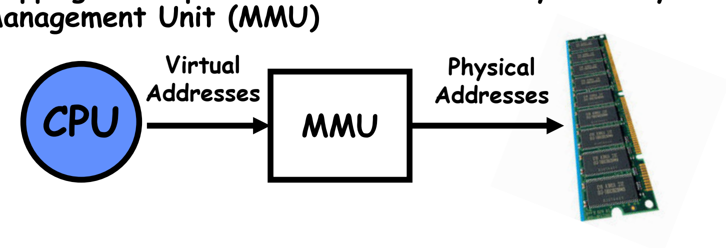
- Full Coordination and Protection
  - Manage interactions between different users
  - Multiple programs running simultaneously
  - Multiplex and protect Hardware Resources
    - » CPU, Memory, I/O devices like disks, printers, etc
- Facilitator
  - Still provides Standard libraries, facilities
- Would this complexity make sense if there were only one application that you cared about?

Example: Protecting Processes from Each Other

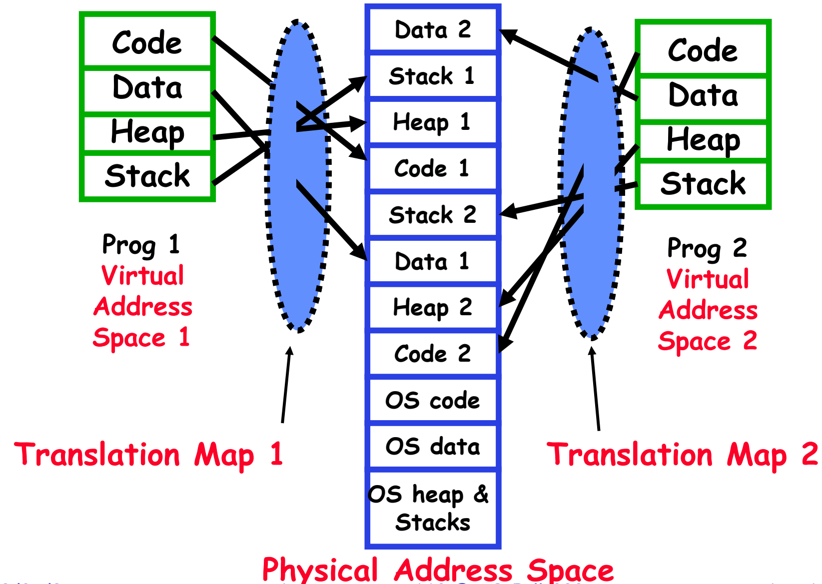
- Problem: Run multiple applications in such a way that they are protected from one another
- Goal:
  - Keep User Programs from Crashing OS
  - Keep User Programs from Crashing each other
  - [Keep Parts of OS from crashing other parts?]
- (Some of the required) Mechanisms:
  - Address Translation
  - Dual Mode Operation
- Simple Policy:
  - Programs are not allowed to read/write memory of other Programs or of Operating System

Address Translation

- Address Space
  - A group of memory addresses usable by something
  - Each program (process) and kernel has potentially different address spaces.
- Address Translation:
  - Translate from Virtual Addresses (emitted by CPU) into Physical Addresses (of memory)
  - Mapping *often* performed in Hardware by Memory Management Unit (MMU)

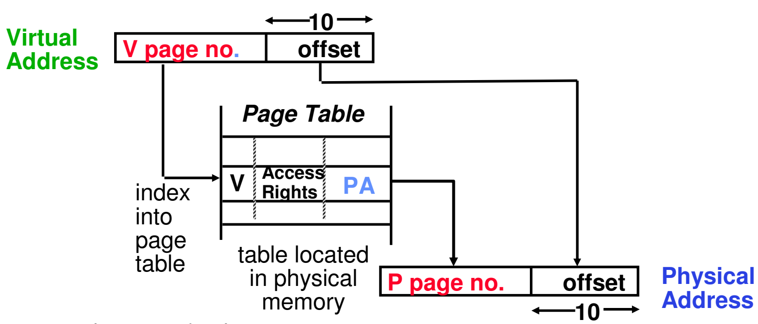


Example of Address Translation



Address Translation Details

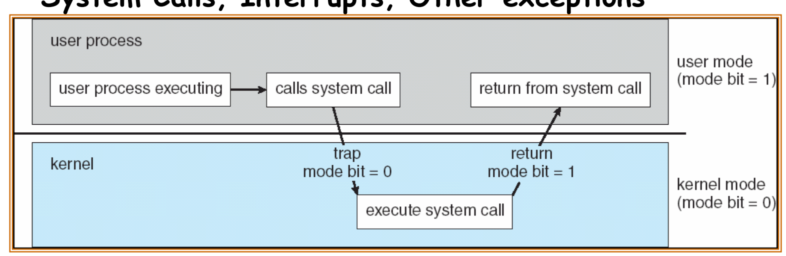
- For now, assume translation happens with table (called a Page Table):



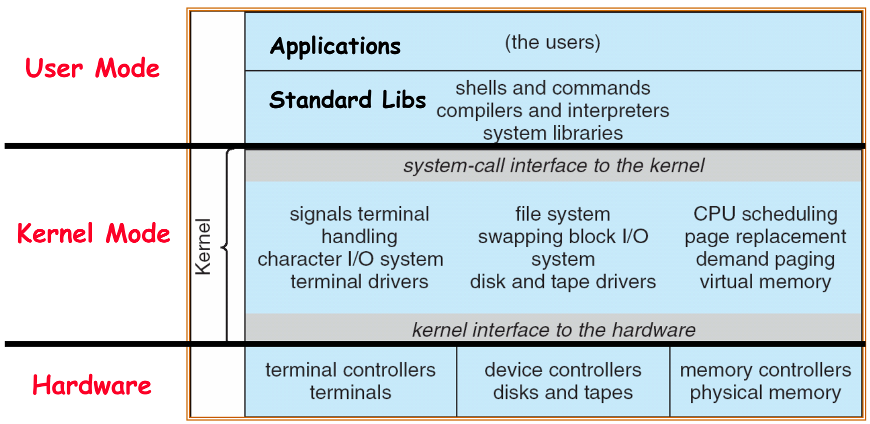
- Translation helps protection:
  - Control translations, control access
  - Should Users be able to change Page Table???

Dual Mode Operation

- Hardware provides at least two modes:
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode: Normal programs executed
- Some instructions/ops prohibited in user mode:
  - Example: cannot modify page tables in user mode
    - » Attempt to modify ⇒ Exception generated
- Transitions from user mode to kernel mode:
  - System Calls, Interrupts, Other exceptions



UNIX System Structure



OS Systems Principles

- OS as illusionist:
  - Make hardware limitations go away
  - Provide illusion of dedicated machine with infinite memory and infinite processors
- OS as government:
  - Protect users from each other
  - Allocate resources efficiently and fairly
- OS as complex system:
  - Constant tension between simplicity and functionality or performance
- OS as history teacher
  - Learn from past
  - Adapt as hardware tradeoffs change

Why Study Operating Systems?

- Learn how to build complex systems:
  - How can you manage complexity for future projects?
- Engineering issues:
  - Why is the web so slow sometimes? Can you fix it?
  - What features should be in the next mars Rover?
  - How do large distributed systems work? (Kazaa, etc)
- Buying and using a personal computer:
  - Why different PCs with same CPU behave differently
  - How to choose a processor (Opteron, Itanium, Celeron, Pentium, Hexium)? [Ok, made last one up]
  - Should you get Windows XP, 2000, Linux, Mac OS …?
  - Why does Microsoft have such a bad name?
- Business issues:
  - Should your division buy thin-clients vs PC?
- Security, viruses, and worms
  - What exposure do you have to worry about?

"In conclusion…"

- Operating systems provide a virtual machine abstraction to handle diverse hardware
- Operating systems coordinate resources and protect users from each other
- Operating systems simplify application development by providing standard services
- Operating systems can provide an array of fault containment, fault tolerance, and fault recovery

- CS162 combines things from many other areas of computer science –
  - Languages, data structures, hardware, and algorithms

CS162
Operating Systems and Systems Programming
Lecture 2

History of the World Parts 1—5
Operating Systems Structures

August 29, 2007

Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

Review: Virtual Machine Abstraction

Application

Virtual Machine Interface

Operating System

Physical Machine Interface

Hardware

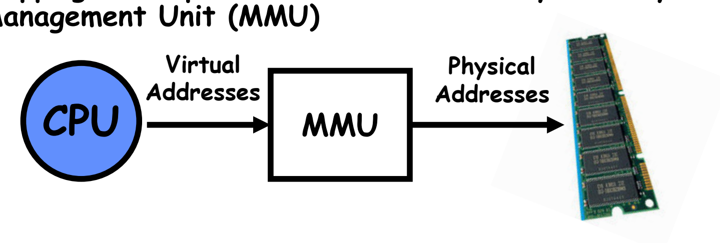
- Software Engineering Problem:
  - Turn hardware/software quirks ⇒ what programmers want/need
  - Optimize for convenience, utilization, security, reliability, etc…
- For Any OS area (e.g. file systems, virtual memory, networking, scheduling):
  - What's the hardware interface? (physical reality)
  - What's the application interface? (nicer abstraction)

Example: Protecting Processes from Each Other

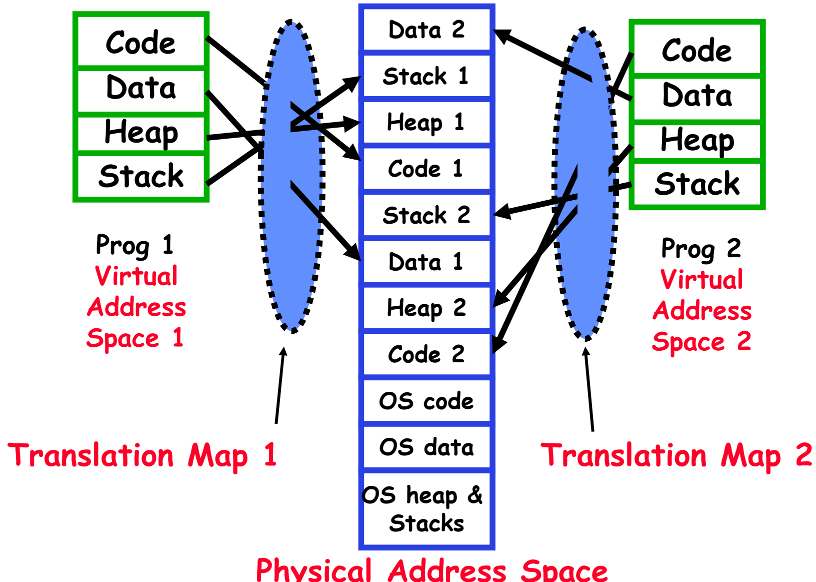
- Problem: Run multiple applications in such a way that they are protected from one another
- Goal:
  - Keep User Programs from Crashing OS
  - Keep User Programs from Crashing each other
  - [Keep Parts of OS from crashing other parts?]
- (Some of the required) Mechanisms:
  - Address Translation
  - Dual Mode Operation
- Simple Policy:
  - Programs are not allowed to read/write memory of other Programs or of Operating System

Example: Address Translation

- Address Space
  - A group of memory addresses usable by something
  - Each program (process) and kernel has potentially different address spaces.
- Address Translation:
  - Translate from Virtual Addresses (emitted by CPU) into Physical Addresses (of memory)
  - Mapping *often* performed in Hardware by Memory Management Unit (MMU)

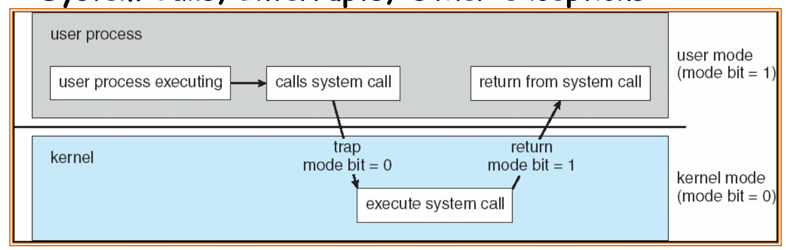


Example: Example of Address Translation



Example: Dual Mode Operation

- Hardware provides at least two modes:
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode: Normal programs executed
- Some instructions/ops prohibited in user mode:
  - Example: cannot modify page tables in user mode
    - » Attempt to modify ⇒ Exception generated
- Transitions from user mode to kernel mode:
  - System Calls, Interrupts, Other exceptions



Goals for Today

- History of Operating Systems
  - Really a history of resource-driven choices
- Operating Systems Structures
- Operating Systems Organizations

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Moore's Law Change Drives OS Change

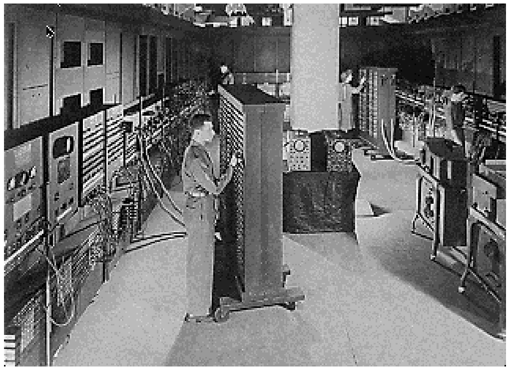
	1981	2006	Factor
CPU MHz, Cycles/inst	10 3—10	3200x4 0.25—0.5	1,280 6—40
DRAM capacity	128KB	4GB	32,768
Disk capacity	10MB	1TB	100,000
Net bandwidth	9600 b/s	1 Gb/s	110,000
# addr bits	16	32	2
#users/machine	10s	≤ 1	≤ 0.1
Price	$25,000	$4,000	0.2

Typical academic computer 1981 vs 2006

Moore's law effects

- Nothing like this in any other area of business
- Transportation in over 200 years:
  - 2 orders of magnitude from horseback @10mph to Concorde @1000mph
  - Computers do this every decade (at least until 2002)!
- What does this mean for us?
  - Techniques have to vary over time to adapt to changing tradeoffs
- I place a lot more emphasis on principles
  - The key concepts underlying computer systems
  - Less emphasis on facts that are likely to change over the next few years…
- Let's examine the way changes in $/MIP has radically changed how OS's work

Dawn of time
ENIAC: (1945—1955)

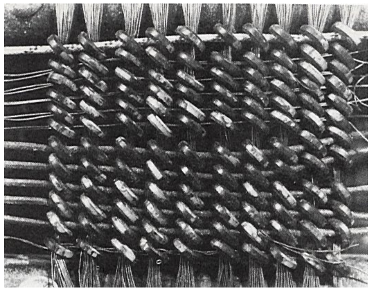


- "The machine designed by Drs. Eckert and Mauchly was a monstrosity. When it was finished, the ENIAC filled an entire room, weighed thirty tons, and consumed two hundred kilowatts of power."
- http://ei.cs.vt.edu/~history/ENIAC.Richey.HTML

History Phase 1 (1948—1970)
Hardware Expensive, Humans Cheap

- When computers cost millions of $'s, optimize for more efficient use of the hardware!
  - Lack of interaction between user and computer
- User at console: one user at a time
- Batch monitor: load program, run, print
- Optimize to better use hardware
  - When user thinking at console, computer idle⇒BAD!
  - Feed computer batches and make users wait
  - Autograder for this course is similar
- *No protection:* what if batch program has bug?

Core Memories (1950s & 60s)



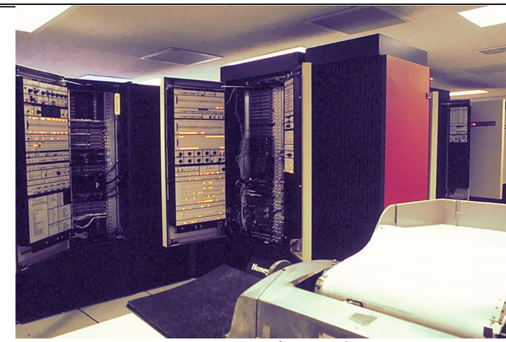
The first magnetic core memory, from the IBM 405 Alphabetical Accounting Machine.

- Core Memory stored data as magnetization in iron rings
  - Iron "cores" woven into a 2-dimensional mesh of wires
  - Origin of the term "Dump Core"
  - Rumor that IBM consulted Life Saver company
- See: http://www.columbia.edu/acis/history/core.html

History Phase 1½ (late 60s/early 70s)

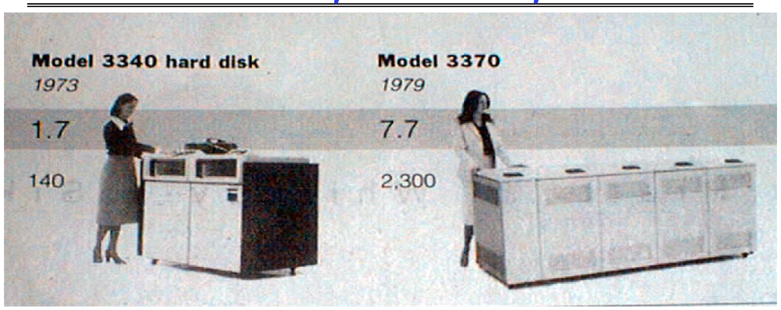
- **Data channels, Interrupts:** overlap I/O and compute
  - DMA – Direct Memory Access for I/O devices
  - I/O can be completed asynchronously
- **Multiprogramming:** several programs run simultaneously
  - Small jobs not delayed by large jobs
  - More overlap between I/O and CPU
  - Need memory protection between programs and/or OS
- **Complexity gets out of hand:**
  - Multics: announced in 1963, ran in 1969
    - » 1777 people "contributed to Multics" (30-40 core dev)
    - » Turing award lecture from Fernando Corbató (key researcher): "On building systems that will fail"
  - OS 360: released with 1000 known bugs (APARs)
    - » "Anomalous Program Activity Report"
- **OS finally becomes an important science:**
  - How to deal with complexity???
  - UNIX based on Multics, but vastly simplified

A Multics System (Circa 1976)



- The 6180 at MIT IPC, skin doors open, circa 1976:
  - "We usually ran the machine with doors open so the operators could see the AQ register display, which gave you an idea of the machine load, and for convenient access to the EXECUTE button, which the operator would push to enter BOS if the machine crashed."
- http://www.multicians.org/multics-stories.html

Early Disk History



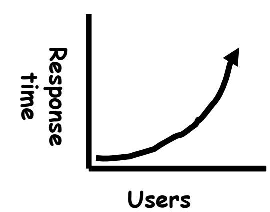
1973:
1. 7 Mbit/sq. in
140 MBytes

1979:
7. 7 Mbit/sq. in
2,300 MBytes

Contrast: Seagate 1TB, 164 GB/SQ in, 3½ in disk, 4 platters

History Phase 2 (1970 – 1985) Hardware Cheaper, Humans Expensive

- Computers available for tens of thousands of dollars instead of millions
- OS Technology maturing/stabilizing
- ***Interactive*** **timesharing:**
  - Use cheap terminals (~$1000) to let multiple users interact with the system at the same time
  - Sacrifice CPU time to get better response time
  - Users do debugging, editing, and email online
- **Problem: Thrashing**
  - Performance very non-linear response with load
  - Thrashing caused by many factors including
    - » Swapping, queueing



Administrivia: What is this CS16x???

- Why change CS162? Only minor changes since 1990's…
  - Slides!
  - Java version of Nachos
  - Content: More crypto/security, less databases and distributed filesystems
  - *Time to update again!!*
- Most CS students take CS 162 and 186
  - But, not all take EE 122, CS 169/161
  - *We'd like all students to have a basic understanding of key concepts from these classes*
- Each class introduces the same topics with class-specific biases
  - Concurrency in an Operating System versus in a Database management system
  - *Introduce concepts with a common framework*

Administrivia: CS 194-3/16x

- Mondays and Wednesdays 9-10:30 in 306 Soda
  - Taught by Anthony Josephy: high teaching ratings!
- Primary content is similar to CS 162
  - With CS 186, 161, and 169, and EE 122 topics
- 4 units with CS Upper Division credit
- 3-4 Projects (tentative)
  - Nachos Phase 1
  - Multi-core programming
  - Secure iTunes-like e-commerce site with a Peer-to-Peer content distribution network
- We need some bold students to try the course
  - Might need to be cancelled otherwise
  - Great way to get 186 & 122 material as well
  - Targeted at Sophomores/First term Juniors

Administrivia: Back to CS162

- Cs162-xx accounts:
  - Make sure you got an account form
    - » We have more forms for those of you who didn't get one
  - If you haven't logged in yet, you need to do so
- Nachos readers:
  - TBA: Will be down at Copy Central on Hearst
  - Will include lectures and printouts of all of the code
- Video archives available off lectures page
  - Just click on the title of a lecture for webcast
  - Only works for lectures that I have already given!
  - Still working on Webcast
- No slip days on first design document for each phase
  - Need to get design reviews in on time
- Don't know Java well?
  - Talk CS 9G self-paced Java course

Administriva: Almost Time for Project Signup

- Project Signup: Watch "Group/Section Assignment Link"
  - 4-5 members to a group
    - » Everyone in group must be able to *actually* attend same section
    - » The sections assigned to you by Telebears are temporary!
  - Only submit once per group!
    - » Everyone in group must have logged into their cs162-xx accounts once before you register the group
    - » Make sure that you select at least 2 potential sections
    - » Due date: Thursday 9/6 by 11:59pm
- Sections:
  - No sections tomorrow
  - Go to Telebears-assigned Section next week

Section	Time	Location	TA
101	Th 10:00-11:00A	81 Evans	Kelvin Lwin
102	Th 12:00-1:00P	155 Barrows	Kelvin Lwin
103	Th 2:00-3:00P	75 Evans	Todd Kosloff
104	Th 4:00-5:00P	B51 Hildebrand	Todd Kosloff
105	F 10:00-11:00A	4 Evans	Thomas Kho

History Phase 3 (1981—)
Hardware Very Cheap, Humans Very Expensive

- Computer costs $1K, Programmer costs $100K/year
  - If you can make someone 1% more efficient by giving them a computer, it's worth it!
  - Use computers to make people more efficient
- Personal computing:
  - Computers cheap, so give everyone a PC
- Limited Hardware Resources Initially:
  - OS becomes a subroutine library
  - One application at a time (MSDOS, CP/M, …)
- Eventually PCs become powerful:
  - OS regains all the complexity of a "big" OS
  - multiprogramming, memory protection, etc (NT,OS/2)
- Question: As hardware gets cheaper does need for OS go away?

History Phase 3 (con't)
Graphical User Interfaces

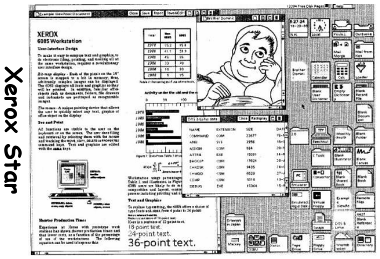
- CS160 ⇒ All about GUIs
- Xerox Star: 1981
  - Originally a research project (Alto)
  - First "mice", "windows"
- Apple Lisa/Machintosh: 1984
  - "Look and Feel" suit 1988
- Microsoft Windows:
  - Win 1.0 (1985)
  - Win 3.1 (1990)
  - Win 95 (1995)
  - Win NT (1993)
  - Win 2000 (2000)
  - Win XP (2001)

Single Level (Win 1.0, Win 3.1, Win 95)

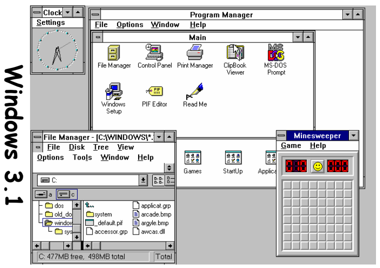
HAL/Protection (Win NT)

No HAL/ Full Prot (Win 2000, Win XP)

Xerox Star

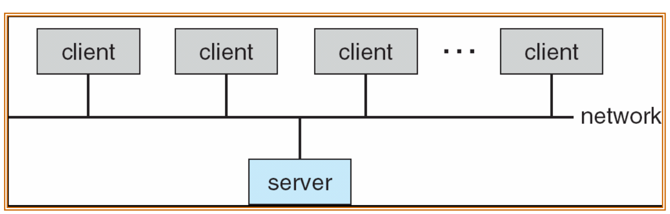


Windows 3.1



History Phase 4 (1989—): Distributed Systems

- Networking (Local Area Networking)
  - Different machines share resources
  - Printers, File Servers, Web Servers
  - Client – Server Model
- Services
  - Computing
  - File Storage



History Phase 5 (1995—): Mobile Systems

- Ubiquitous Mobile Devices
  - Laptops, PDAs, phones
  - Small, portable, and inexpensive
    - » Recently twice as many smart phones as PDAs
    - » Many computers/person!
  - Limited capabilities (memory, CPU, power, etc…)
- Wireless/Wide Area Networking
  - Leveraging the infrastructure
  - Huge distributed pool of resources extend devices
  - Traditional computers split into pieces. Wireless keyboards/mice, CPU distributed, storage remote
- Peer-to-peer systems
  - Many devices with equal responsibilities work together
  - Components of "Operating System" spread across globe

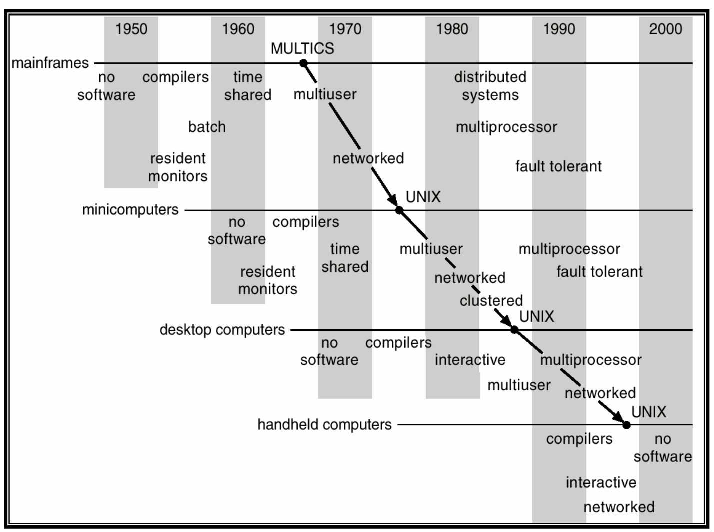
CITRIS's Model: A Societal Scale Information System

- Center for Information Technology Research in the Interest of Society
- The Network is the OS
  - Functionality spread throughout network

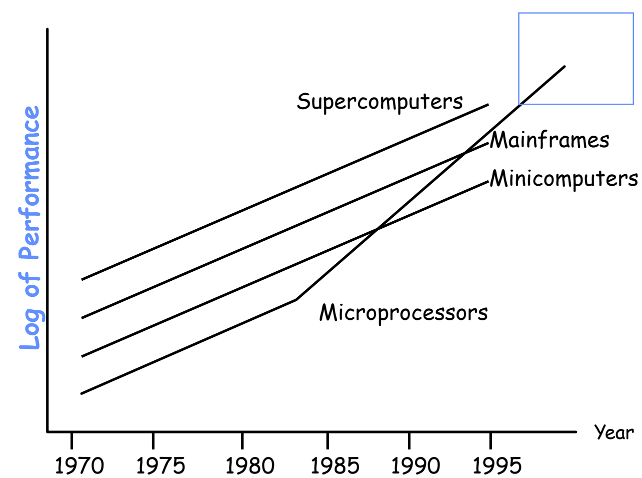
Moore's Law Reprise: Modern Laptop

	1981	2005	2006 Ultralight Laptop
CPU MHz, Cycles/inst	10 3—10	3200x4 0.25—0.5	1830 0.25—0.5
DRAM capacity	128KB	4GB	2GB
Disk capacity	10MB	1TB	100GB
Net bandwidth	9600 b/s	1 Gb/s	1 Gb/s (wired) 54 Mb/s (wireless) 2 Mb/s (wide-area)
# addr bits	16	32	32
#users/machine	10s	≤ 1	$\leq \frac{1}{4}$
Price	$25,000	$4,000	$2500

Migration of Operating-System Concepts and Features



Compare: Performance Trends (from CS152)



History of OS: Summary

- Change is continuous and OSs should adapt
  - Not: look how stupid batch processing was
  - But: Made sense at the time
- Situation today is much like the late 60s [poll]
  - Small OS: 100K lines
  - Large OS: 10M lines (5M for the browser!)
    - » 100-1000 people-years
- Complexity still reigns
  - NT under development from early 90's to late 90's
    - » Never worked very well
  - Jury still out on Windows 2000/XP
  - Windows Vista (aka "Longhorn") delayed many times
    - » Latest release date of 2005, 2006, 2007+
    - » Promised by removing some of the intended technology
- CS162: understand OSs to simplify them

Now for a quick tour of OS Structures

Operating Systems Components (What are the pieces of the OS)

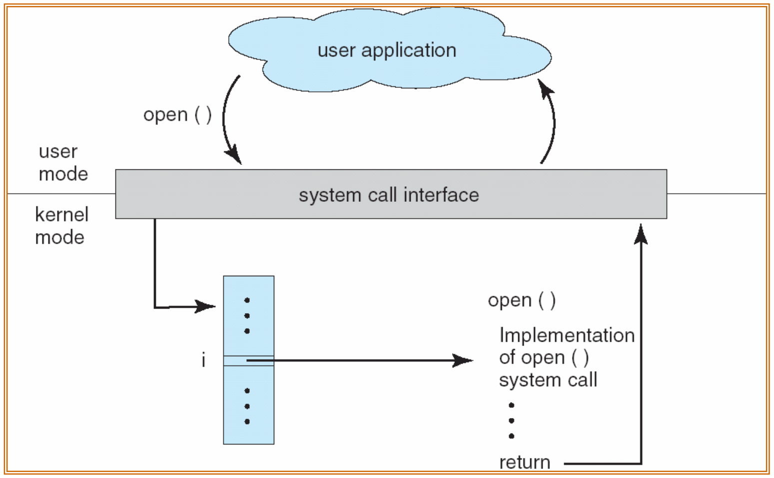
- Process Management
- Main-Memory Management
- I/O System management
- File Management
- Networking
- User Interfaces

Operating System Services (What things does the OS do?)

- Services that (more-or-less) map onto components
  - Program execution
    - » How do you execute concurrent sequences of instructions?
  - I/O operations
    - » Standardized interfaces to extremely diverse devices
  - File system manipulation
    - » How do you read/write/preserve files?
    - » Looming concern: How do you even find files???
  - Communications
    - » Networking protocols/Interface with CyberSpace?
- Cross-cutting capabilities
  - Error detection & recovery
  - Resource allocation
  - Accounting
  - Protection

System Calls (What is the API)

- See Chapter 2 of 7th edition or Chapter 3 of 6th

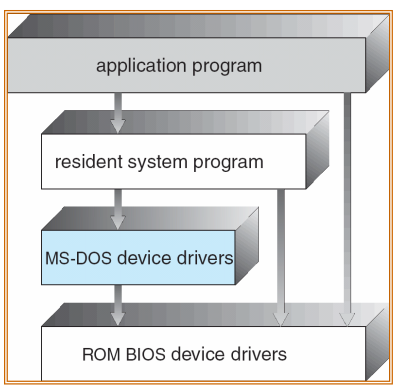


Operating Systems Structure (What is the organizational Principle?)

- Simple
  - Only one or two levels of code
- Layered
  - Lower levels independent of upper levels
- Microkernel
  - OS built from many user-level processes
- Modular
  - Core kernel with Dynamically loadable modules

Simple Structure

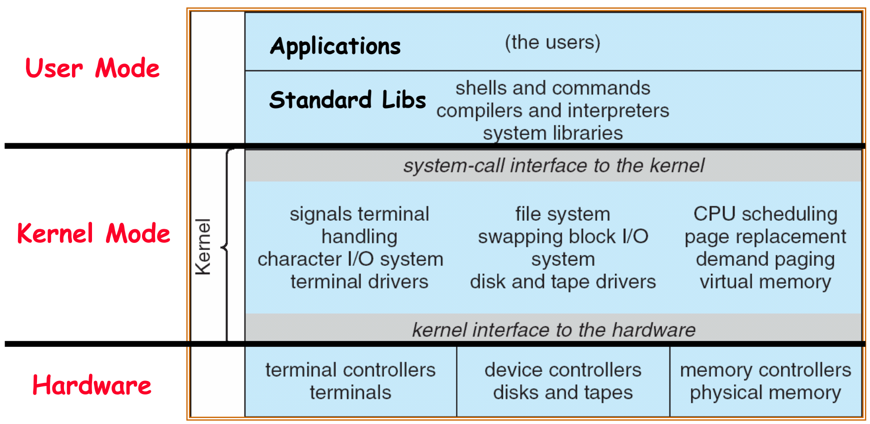
- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Interfaces and levels of functionality not well separated



UNIX: Also "Simple" Structure

- UNIX – limited by hardware functionality
- Original UNIX operating system consists of two separable parts:
  - Systems programs
  - The kernel
    - » Consists of everything below the system-call interface and above the physical hardware
    - » Provides the file system, CPU scheduling, memory management, and other operating-system functions;
    - » Many interacting functions for one level

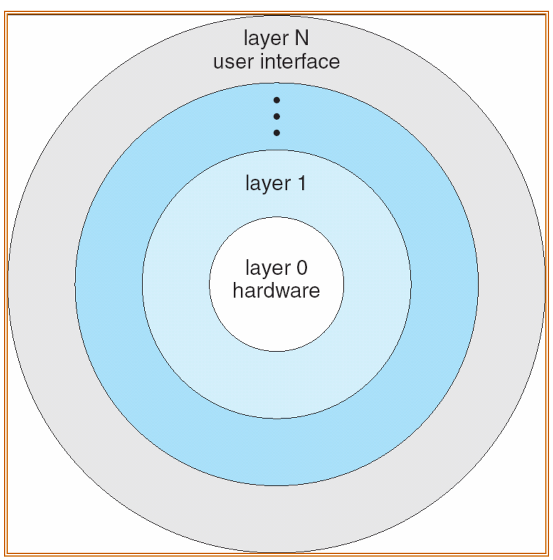
UNIX System Structure



Layered Structure

- Operating system is divided many layers (levels)
  - Each built on top of lower layers
  - Bottom layer (layer 0) is hardware
  - Highest layer (layer N) is the user interface
- Each layer uses functions (operations) and services of only lower-level layers
  - Advantage: modularity ⇒ Easier debugging/Maintenance
  - Not always possible: Does process scheduler lie above or below virtual memory layer?
    - » Need to reschedule processor while waiting for paging
    - » May need to page in information about tasks
- Important: Machine-dependent vs independent layers
  - Easier migration between platforms
  - Easier evolution of hardware platform
  - Good idea for you as well!

Layered Operating System

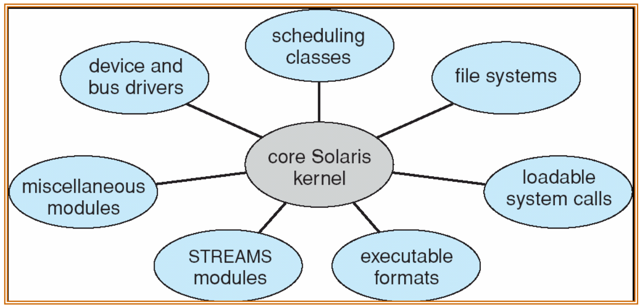


Microkernel Structure

- Moves as much from the kernel into "*user*" space
  - Small core OS 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 (less code is running in kernel mode)
  - Fault Isolation (parts of kernel protected from other parts)
  - More secure
- Detriments:
  - Performance overhead severe for naïve implementation

Modules-based Structure

- Most modern operating systems implement modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible



Implementation Issues (How is the OS implemented?)

- Policy vs. Mechanism
  - Policy: What do you want to do?
  - Mechanism: How are you going to do it?
  - Should be separated, since both change
- Algorithms used
  - Linear, Tree-based, Log Structured, etc…
- Event models used
  - threads vs event loops
- Backward compatability issues
  - Very important for Windows 2000/XP
- System generation/configuration
  - How to make generic OS fit on specific hardware

Conclusion

- Rapid Change in Hardware Leads to changing OS
  - Batch ⇒ Multiprogramming ⇒ Timeshare ⇒ Graphical UI ⇒ Ubiquitous Devices ⇒ Cyberspace/Metaverse/??
- OS features migrated from mainframes ⇒ PCs
- Standard Components and Services
  - Process Control
  - Main Memory
  - I/O
  - File System
  - UI
- Policy vs Mechanism
  - Crucial division: not always properly separated!
- Complexity is always out of control
  - However, "Resistance is NOT Useless!"

CS162
Operating Systems and Systems Programming
Lecture 3

Concurrency: Processes, Threads, and Address Spaces

September 5, 2007

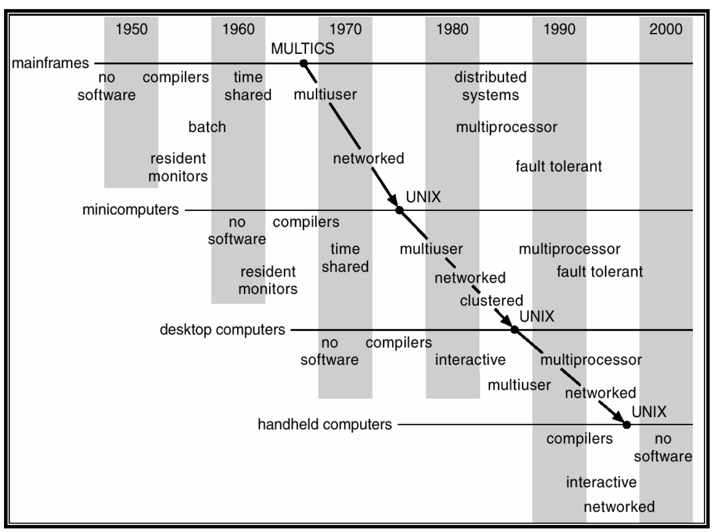
Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

Review: History of OS

- Why Study?
  - To understand how user needs and hardware constraints influenced (and will influence) operating systems
- Several Distinct Phases:
  - Hardware Expensive, Humans Cheap
    - » Eniac, … Multics
  - Hardware Cheaper, Humans Expensive
    - » PCs, Workstations, Rise of GUIs
  - Hardware Really Cheap, Humans Really Expensive
    - » Ubiquitous devices, Widespread networking
- Rapid Change in Hardware Leads to changing OS
  - Batch ⇒ Multiprogramming ⇒ Timeshare ⇒ Graphical UI ⇒ Ubiquitous Devices ⇒ Cyberspace/Metaverse/??
  - Gradual Migration of Features into Smaller Machines
- Situation today is much like the late 60s
  - Small OS: 100K lines/Large: 10M lines (5M browser!)
  - 100-1000 people-years

Review: Migration of OS Concepts and Features



Review: Implementation Issues (How is the OS implemented?)

- Policy vs. Mechanism
  - Policy: What do you want to do?
  - Mechanism: How are you going to do it?
  - Should be separated, since policies change
- Algorithms used
  - Linear, Tree-based, Log Structured, etc…
- Event models used
  - threads vs event loops
- Backward compatability issues
  - Very important for Windows 2000/XP/Vista/…
  - POSIX tries to help here
- System generation/configuration
  - How to make generic OS fit on specific hardware

Goals for Today

- How do we provide multiprogramming?
- What are Processes?
- How are they related to Threads and Address Spaces?

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

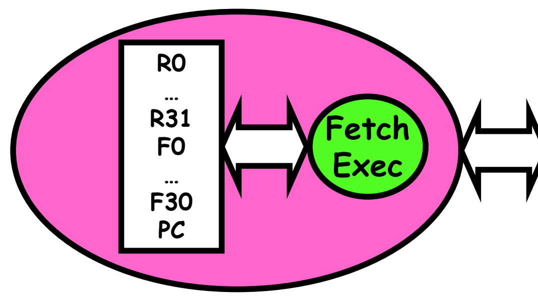
Concurrency

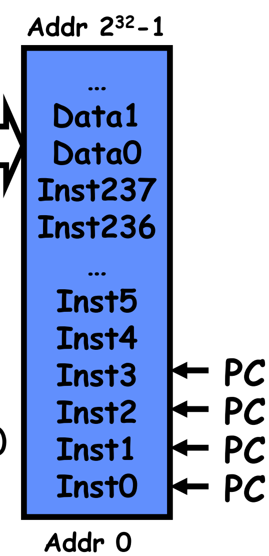
- "Thread" of execution
  - Independent Fetch/Decode/Execute loop
  - Operating in some Address space
- Uniprogramming: *one thread at a time*
  - MS/DOS, early Macintosh, Batch processing
  - Easier for operating system builder
  - Get rid concurrency by defining it away
  - Does this make sense for personal computers?
- Multiprogramming: *more than one thread at a time*
  - Multics, UNIX/Linux, OS/2, Windows NT/2000/XP, Mac OS X
  - Often called "multitasking", but multitasking has other meanings (talk about this later)

The Basic Problem of Concurrency

- The basic problem of concurrency involves resources:
  - Hardware: single CPU, single DRAM, single I/O devices
  - Multiprogramming API: users think they have exclusive access to machine
- OS Has to coordinate all activity
  - Multiple users, I/O interrupts, …
  - How can it keep all these things straight?
- Basic Idea: Use Virtual Machine abstraction
  - Decompose hard problem into simpler ones
  - Abstract the notion of an executing program
  - Then, worry about multiplexing these abstract machines
- Dijkstra did this for the "THE system"
  - Few thousand lines vs 1 million lines in OS 360 (1K bugs)

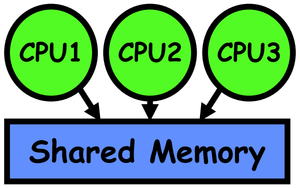
Recall (61C): What happens during execution?

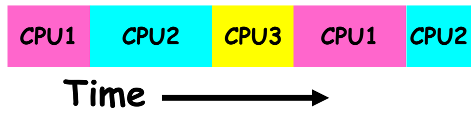




- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

How can we give the illusion of multiple processors?





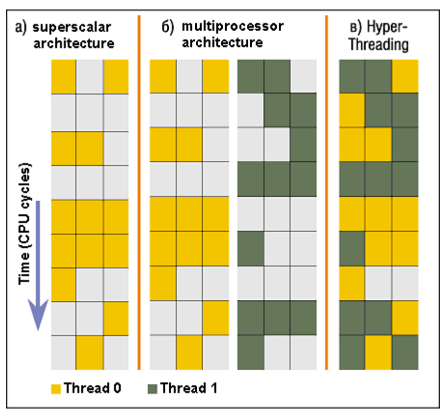
- How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Each virtual "CPU" needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How switch from one CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, and registers from new state block
- What triggers switch?
  - Timer, voluntary yield, I/O, other things

Properties of this simple multiprogramming technique

- All virtual CPUs share same non-CPU resources
  - I/O devices the same
  - Memory the same
- Consequence of sharing:
  - Each thread can access the data of every other thread (good for sharing, bad for protection)
  - Threads can share instructions (good for sharing, bad for protection)
  - Can threads overwrite OS functions?
- This (unprotected) model common in:
  - Embedded applications
  - Windows 3.1/Machintosh (switch only with yield)
  - Windows 95—ME? (switch with both yield and timer)

Modern Technique: SMT/Hyperthreading

- Hardware technique
  - Exploit natural properties of superscalar processors to provide illusion of multiple processors
  - Higher utilization of processor resources
- Can schedule each thread as if were separate CPU
  - However, not linear speedup!
  - If have multiprocessor, should schedule each processor first



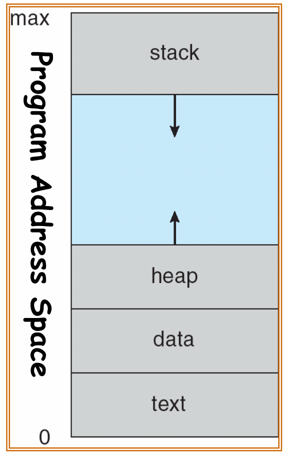
- Original technique called "Simultaneous Multithreading"
  - See http://www.cs.washington.edu/research/smt/
  - Alpha, SPARC, Pentium 4 ("Hyperthreading"), Power 5

How to protect threads from one another?

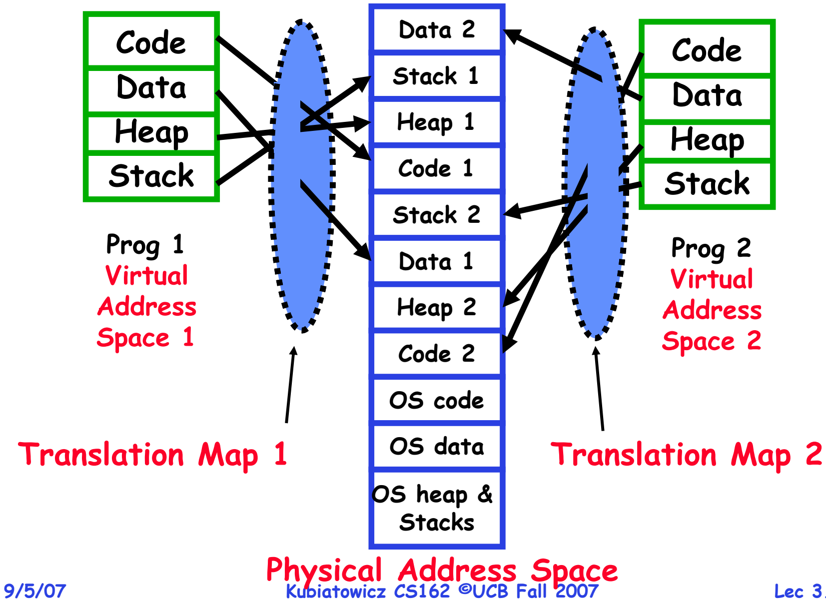
- Need three important things:
  1. Protection of memory
     - » Every task does not have access to all memory
  2. Protection of I/O devices
     - » Every task does not have access to every device
  3. Preemptive switching from task to task
     - » Use of timer
     - » Must not be possible to disable timer from usercode

Recall: Program's Address Space

- Address space ⇒ the set of accessible addresses + state associated with them:
  - For a 32-bit processor there are 2^{32} = 4 billion addresses
- What happens when you read or write to an address?
  - Perhaps Nothing
  - Perhaps acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes I/O operation
    - » (Memory-mapped I/O)
  - Perhaps causes exception (fault)



Providing Illusion of Separate Address Space: Load new Translation Map on Switch



Administriva

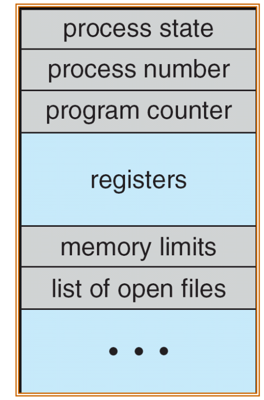
Administrivia (2)

Traditional UNIX Process

- Process: *Operating system abstraction to represent what is needed to run a single program*
  - Often called a "HeavyWeight Process"
  - Formally: a single, sequential stream of execution in its *own* address space
- Two parts:
  - Sequential Program Execution Stream
    - » Code executed as a *single, sequential* stream of execution
    - » Includes State of CPU registers
  - Protected Resources:
    - » Main Memory State (contents of Address Space)
    - » I/O state (i.e. file descriptors)
- Important: There is no concurrency in a heavyweight process

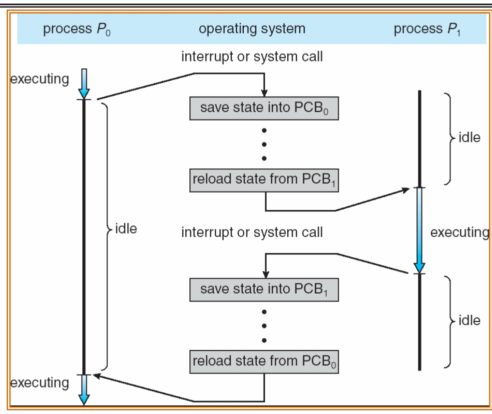
How do we multiplex processes?

- The current state of process held in a process control block (PCB):
  - This is a "snapshot" of the execution and protection environment
  - Only one PCB active at a time
- 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
  - Sample mechanisms:
    - » Memory Mapping: Give each process their own address space
    - » Kernel/User duality: Arbitrary multiplexing of I/O through system calls



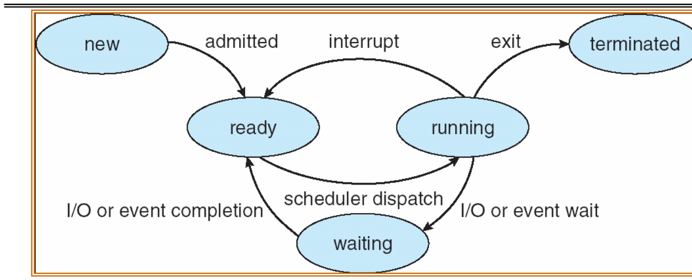
Process Control Block

CPU Switch From Process to Process



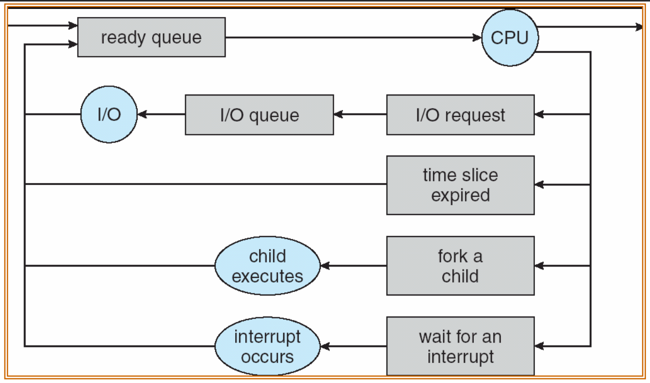
- This is also called a "context switch"
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/hyperthreading, but… contention for resources instead

Diagram of Process State



- As a process executes, it changes *state*
  - new: The process is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution

Process Scheduling

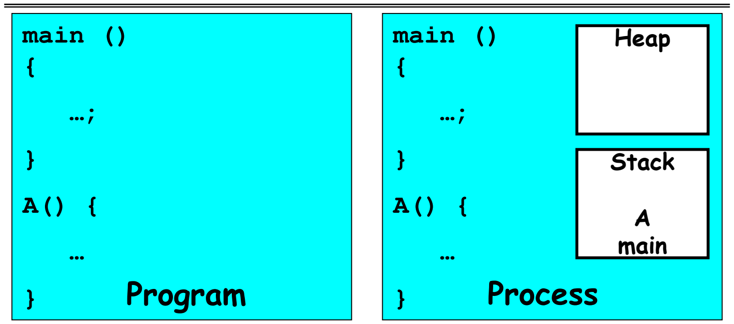


- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many algorithms possible (few weeks from now)

What does it take to create a process?

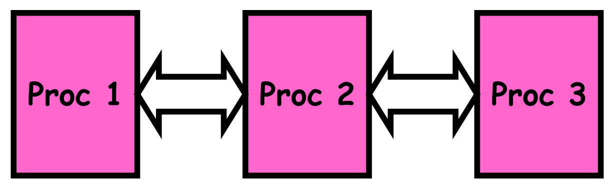
- Must construct new PCB
  - Inexpensive
- Must set up new page tables for address space
  - More expensive
- Copy data from parent process? (Unix `fork()`)
  - Semantics of Unix `fork()` are that the child process gets a complete copy of the parent memory and I/O state
  - Originally *very* expensive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

Process =? Program



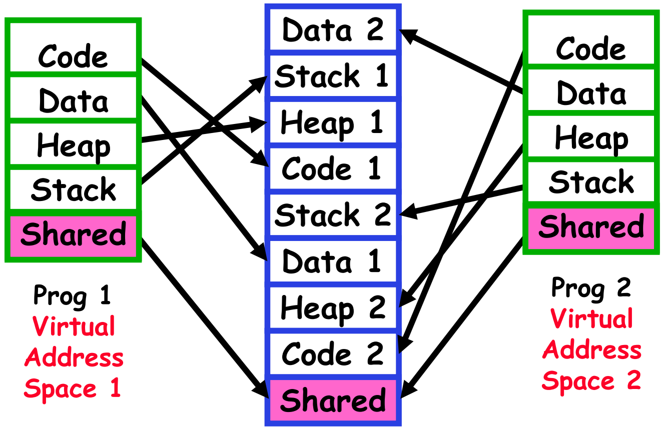
- More to a process than just a program:
  - Program is just part of the process state
  - I run emacs on lectures.txt, you run it on homework.java – Same program, different processes
- Less to a process than a program:
  - A program can invoke more than one process
  - cc starts up cpp, cc1, cc2, as, and ld

Multiple Processes Collaborate on a Task



- High Creation/memory Overhead
- (Relatively) High Context-Switch Overhead
- Need Communication mechanism:
  - Separate Address Spaces Isolates Processes
  - Shared-Memory Mapping
    - » Accomplished by mapping addresses to common DRAM
    - » Read and Write through memory
  - Message Passing
    - » `send()` and `receive()` messages
    - » Works across network

Shared Memory Communication



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

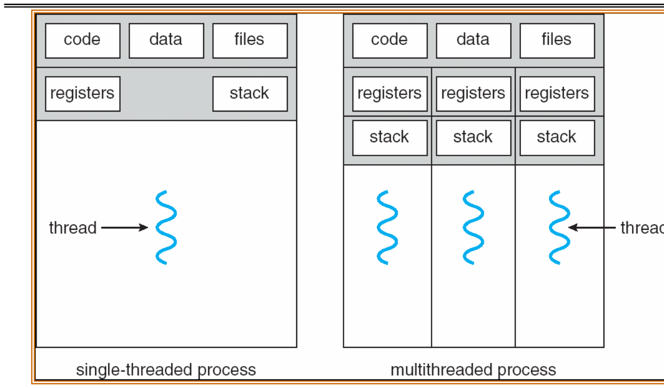
Inter-process Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - `send`(*message*) – message size fixed or variable
  - `receive`(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus, systcall/trap)
  - logical (e.g., logical properties)

Modern "Lightweight" Process with Threads

- Thread: *a sequential execution stream within process* (Sometimes called a "Lightweight process")
  - Process still contains a single Address Space
  - No protection between threads
- Multithreading: *a single program made up of a number of different concurrent activities*
  - Sometimes called multitasking, as in Ada…
- Why separate the concept of a thread from that of a process?
  - Discuss the "thread" part of a process (concurrency)
  - Separate from the "address space" (Protection)
  - Heavyweight Process ≡ Process 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
- Why have multiple threads per address space?

Examples of 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

Examples of multithreaded programs (con't)

- 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

- Some multiprocessors are actually uniprogrammed:
  - Multiple threads in one address space but one program at a time

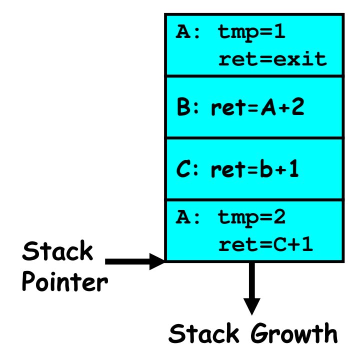
Thread State

- 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 – what is this?

- Execution Stack
  - Parameters, Temporary variables
  - return PCs are kept while called procedures are executing

Execution Stack Example

```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Classification

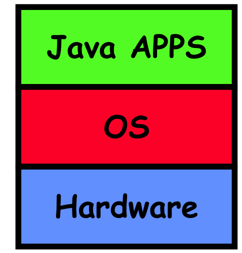
# threads Per AS: / # of addr spaces:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (Geoworks, VxWorks, JavaOS,etc) JavaOS, Pilot(PC)	Mach, OS/2, Linux Windows 9x??? Win NT to XP, Solaris, HP-UX, OS X

- Real operating systems have either
  - One or many address spaces
  - One or many threads per address space
- Did Windows 95/98/ME have real memory protection?
  - No: Users could overwrite process tables/System DLLs

Example: Implementation Java OS

- Many threads, one Address Space
- Why another OS?
  - Recommended Minimum memory sizes:
    - » UNIX + X Windows: 32MB
    - » Windows 98: 16-32MB
    - » Windows NT: 32-64MB
    - » Windows 2000/XP: 64-128MB
  - What if we want a cheap network point-of-sale computer?
    - » Say need 1000 terminals
    - » Want < 8MB
- What language to write this OS in?
  - *C/C++/ASM?* Not terribly high-level. Hard to debug.
  - *Java/Lisp?* Not quite sufficient – need direct access to HW/memory management

Java OS Structure



Summary

- Processes have two parts
  - Threads (Concurrency)
  - Address Spaces (Protection)
- Concurrency accomplished by multiplexing CPU Time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (`yield()`, I/O operations) or involuntary (timer, other interrupts)
- Protection accomplished restricting access:
  - Memory mapping isolates processes from each other
  - Dual-mode for isolating I/O, other resources
- Book talks about processes
  - When this concerns concurrency, really talking about thread portion of a process
  - When this concerns protection, talking about address space portion of a process

CS162
Operating Systems and Systems Programming
Lecture 4

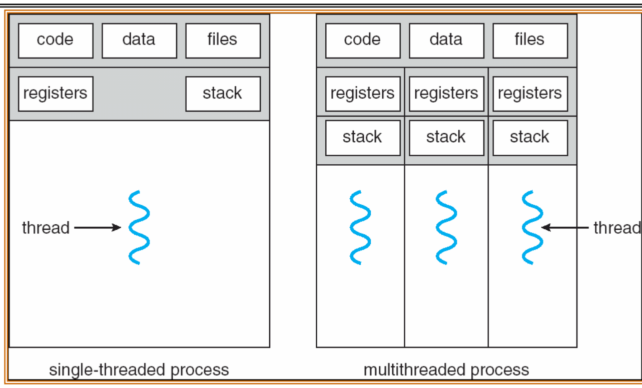
Thread Dispatching

September 10, 2007
Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

Recall: Modern Process with Multiple Threads

- Process: *Operating system abstraction to represent what is needed to run a single, multithreaded program*
- Two parts:
  - Multiple Threads
    - » Each thread is a *single, sequential stream of execution*
  - Protected Resources:
    - » Main Memory State (contents of Address Space)
    - » I/O state (i.e. file descriptors)
- Why separate the concept of a thread from that of a process?
  - Discuss the "thread" part of a process (concurrency)
  - Separate from the "address space" (Protection)
  - Heavyweight Process ≡ Process with one thread

Recall: Single and Multithreaded Processes



- Threads encapsulate concurrency
  - "Active" component of a process
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
  - "Passive" component of a process

Recall: Classification

# 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, Win 95?, Mac OS X, Win NT to XP, Solaris, HP-UX

- Real operating systems have either
  - One or many address spaces
  - One or many threads per address space
- Did Windows 95/98/ME have real memory protection?
  - No: Users could overwrite process tables/System DLLs

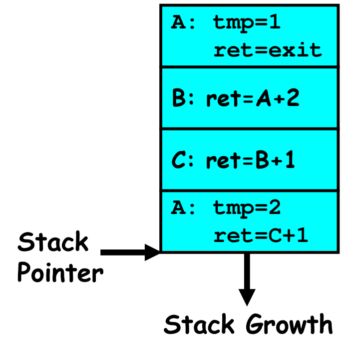
Goals for Today

- Further Understanding Threads
- Thread Dispatching
- Beginnings of Thread Scheduling

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Recall: Execution Stack Example

```
A(int tmp) {
  if (tmp<2)
    B();
  printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

MIPS: Software conventions for Registers

0	zero	constant 0	16	s0	callee saves
1	at	reserved for assembler	. . .		(callee must save)
2	v0	expression evaluation &	23	s7	
3	v1	function results	24	t8	temporary (cont'd)
4	a0	arguments	25	t9	
5	a1		26	k0	reserved for OS kernel
6	a2		27	k1	
7	a3		28	gp	Pointer to global area
8	t0	temporary: caller saves	29	sp	Stack pointer
. . .		(callee can clobber)	30	fp	frame pointer
15	t7		31	ra	Return Address (HW)

- Before calling procedure:
  - Save caller-saves regs
  - Save v0, v1
  - Save ra
- After return, assume
  - Callee-saves reg OK
  - gp,sp,fp OK (restored!)
  - Other things trashed

Single-Threaded Example

- Imagine the following C program:

```
main() {
    ComputePI("pi.txt");
    PrintClassList("clist.text");
}
```

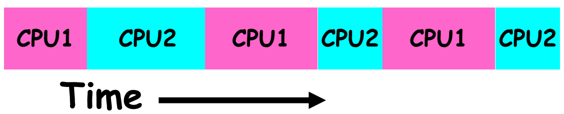
- What is the behavior here?
  - Program would never print out class list
  - Why? ComputePI would never finish

Use of Threads

- Version of program with Threads:

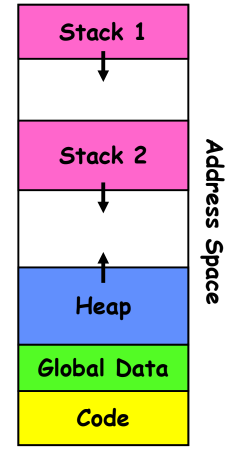
```
main() {
    CreateThread(ComputePI("pi.txt"));
    CreateThread(PrintClassList("clist.text"));
}
```

- What does "CreateThread" do?
  - Start independent thread running given procedure
- What is the behavior here?
  - Now, you would actually see the class list
  - This *should* behave as if there are two separate CPUs



Memory Footprint of Two-Thread Example

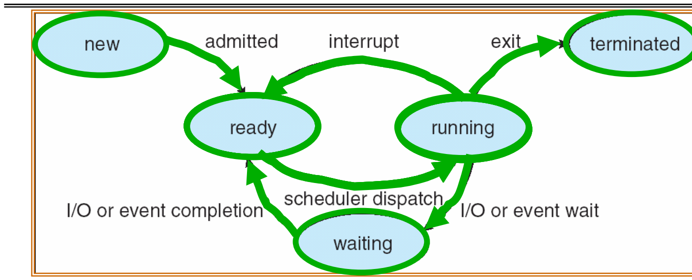
- If we stopped this program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks
- Questions:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?



Per Thread State

- Each Thread has a *Thread Control Block* (TCB)
  - Execution State: CPU registers, program counter, pointer to stack
  - Scheduling info: State (more later), priority, CPU time
  - Accounting Info
  - Various Pointers (for implementing scheduling queues)
  - Pointer to enclosing process? (PCB)?
  - Etc (add stuff as you find a need)
- In Nachos: "Thread" is a class that includes the TCB
- OS Keeps track of TCBs in protected memory
  - In Array, or Linked List, or …

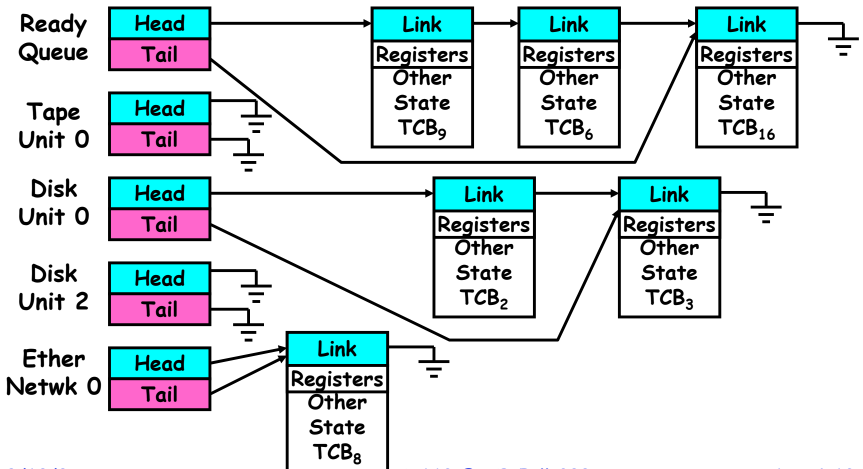
Lifecycle of a Thread (or Process)



- As a thread executes, it changes state:
  - new: The thread is being created
  - ready: The thread is waiting to run
  - running: Instructions are being executed
  - waiting: Thread waiting for some event to occur
  - terminated: The thread has finished execution
- "Active" threads are represented by their TCBs
  - TCBs organized into queues based on their state

Ready Queue And Various I/O Device Queues

- Thread not running ⇒ TCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy



Administrivia

Administrivia

Dispatch Loop

- Conceptually, the dispatching loop of the operating system looks as follows:

```
Loop {
   RunThread();
   ChooseNextThread();
   SaveStateOfCPU(curTCB);
   LoadStateOfCPU(newTCB);
}
```

- This is an *infinite* loop
  - One could argue that this is all that the OS does
- Should we ever exit this loop???
  - When would that be?

Running a thread

Consider first portion: `RunThread()`

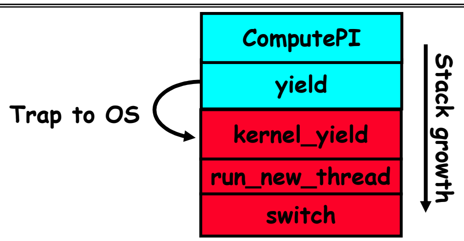
- How do I run a thread?
  - Load its state (registers, PC, stack pointer) into CPU
  - Load environment (virtual memory space, etc)
  - Jump to the PC
- How does the dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets *preempted*

Internal Events

- Blocking on I/O
  - The act of requesting I/O implicitly yields the CPU
- Waiting on a "signal" from other thread
  - Thread asks to wait and thus yields the CPU
- Thread executes a `yield()`
  - Thread volunteers to give up CPU

```
computePI() {
   while(TRUE) {
      ComputeNextDigit();
      yield();
   }
}
```


Stack for Yielding Thread



- How do we run a new thread?

```
run_new_thread() {
   newThread = PickNewThread();
   switch(curThread, newThread);
   ThreadHouseKeeping(); /* next Lecture */
}
```

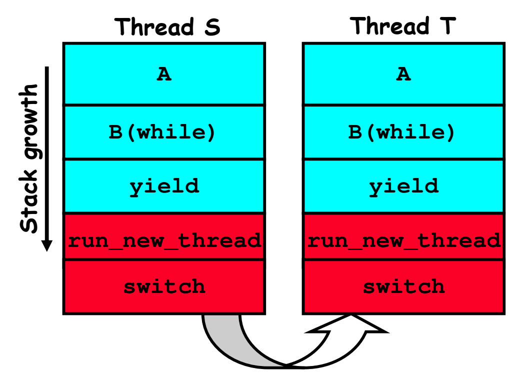
- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack
  - Maintain isolation for each thread

What do the stacks look like?

- Consider the following code blocks:

```
proc A() {
   B();

}
proc B() {
   while(TRUE) {
      yield();
   }
}
```



- Suppose we have 2 threads:
  - Threads S and T

Saving/Restoring state (often called "Context Switch)

```
Switch(tCur,tNew) {
   /* Unload old thread */
   TCB[tCur].regs.r7 = CPU.r7;
         …
   TCB[tCur].regs.r0 = CPU.r0;
   TCB[tCur].regs.sp = CPU.sp;
   TCB[tCur].regs.retpc = CPU.retpc; /*return addr*/

   /* Load and execute new thread */
   CPU.r7 = TCB[tNew].regs.r7;
         …
   CPU.r0 = TCB[tNew].regs.r0;
   CPU.sp = TCB[tNew].regs.sp;
   CPU.retpc = TCB[tNew].regs.retpc;
   return; /* Return to CPU.retpc */
}
```

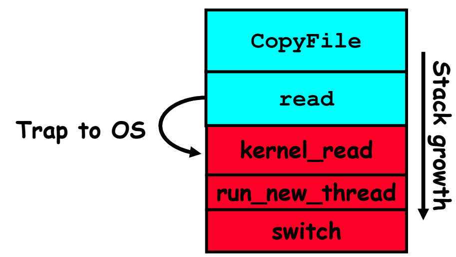

Switch Details

- How many registers need to be saved/restored?
  - MIPS 4k: 32 Int(32b), 32 Float(32b)
  - Pentium: 14 Int(32b), 8 Float(80b), 8 SSE(128b),…
  - Sparc(v7): 8 Regs(32b), 16 Int regs (32b) * 8 windows = 136 (32b)+32 Float (32b)
  - Itanium: 128 Int (64b), 128 Float (82b), 19 Other(64b)
- `retpc` is where the return should jump to.
  - In reality, this is implemented as a jump
- There is a real implementation of switch in Nachos.
  - See switch.s
    - » Normally, switch is implemented as assembly!
  - Of course, it's magical!
  - But you should be able to follow it!

Switch Details (continued)

- What if you make a mistake in implementing switch?
  - Suppose you forget to save/restore register 4
  - Get intermittent failures depending on when context switch occurred and whether new thread uses register 4
  - System will give wrong result without warning
- Can you devise an exhaustive test to test switch code?
  - No! Too many combinations and inter-leavings
- Cautionary tail:
  - For speed, Topaz kernel saved one instruction in switch()
  - Carefully documented!
    - » Only works As long as kernel size < 1MB
  - What happened?
    - » Time passed, People forgot
    - » Later, they added features to kernel (no one removes features!)
    - » Very weird behavior started happening
  - Moral of story: Design for simplicity

What happens when thread blocks on I/O?

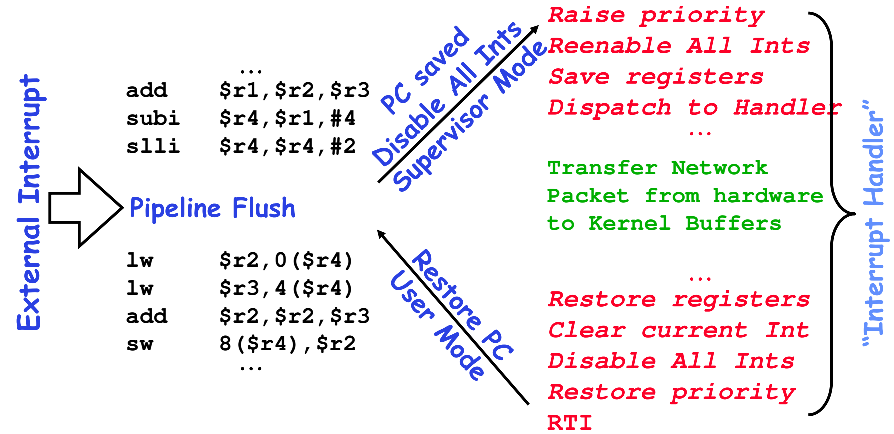


- What happens when a thread requests a block of data from the file system?
  - User code invokes a system call
  - Read operation is initiated
  - Run new thread/switch
- Thread communication similar
  - Wait for Signal/Join
  - Networking

External Events

- What happens if thread never does any I/O, never waits, and never yields control?
  - Could the ComputePI program grab all resources and never release the processor?
    - » What if it didn't print to console?
  - Must find way that dispatcher can regain control!
- Answer: Utilize External Events
  - Interrupts: signals from hardware or software that stop the running code and jump to kernel
  - Timer: like an alarm clock that goes off every some many milliseconds
- If we make sure that external events occur frequently enough, can ensure dispatcher runs

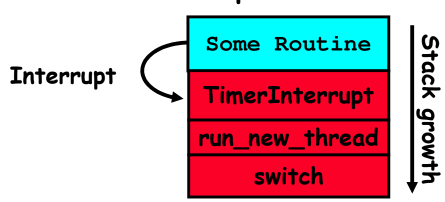
Example: Network Interrupt



- An interrupt is a hardware-invoked context switch
  - No separate step to choose what to run next
  - Always run the interrupt handler immediately

Use of Timer Interrupt to Return Control

- Solution to our dispatcher problem
  - Use the timer interrupt to force scheduling decisions



- Timer Interrupt routine:

```
TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```

- I/O interrupt: same as timer interrupt except that `DoHousekeeping()` replaced by `ServiceIO()`.

Choosing a Thread to Run

- How does Dispatcher decide what to run?
  - Zero ready threads – dispatcher loops
    - » Alternative is to create an "idle thread"
    - » Can put machine into low-power mode
  - Exactly one ready thread – easy
  - More than one ready thread: use scheduling priorities
- Possible priorities:
  - LIFO (last in, first out):
    - » put ready threads on front of list, remove from front
  - Pick one at random
  - FIFO (first in, first out):
    - » Put ready threads on back of list, pull them from front
    - » This is fair and is what Nachos does
  - Priority queue:
    - » keep ready list sorted by TCB priority field

Summary

- The state of a thread is contained in the TCB
  - Registers, PC, stack pointer
  - States: New, Ready, Running, Waiting, or Terminated
- Multithreading provides simple illusion of multiple CPUs
  - Switch registers and stack to dispatch new thread
  - Provide mechanism to ensure dispatcher regains control
- Switch routine
  - Can be very expensive if many registers
  - Must be very carefully constructed!
- Many scheduling options
  - Decision of which thread to run complex enough for complete lecture

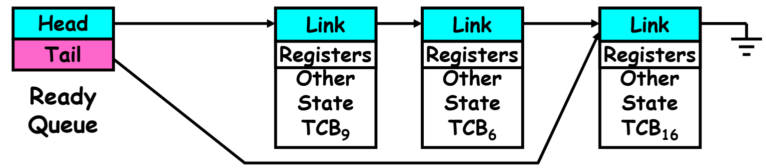
CS162
Operating Systems and Systems Programming
Lecture 5

Cooperating Threads

September 12, 2007
Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

Review: Per Thread State

- Each Thread has a *Thread Control Block* (TCB)
  - Execution State: CPU registers, program counter, pointer to stack
  - Scheduling info: State (more later), priority, CPU time
  - Accounting Info
  - Various Pointers (for implementing scheduling queues)
  - Pointer to enclosing process? (PCB)?
  - Etc (add stuff as you find a need)
- OS Keeps track of TCBs in protected memory
  - In Arrays, or Linked Lists, or …



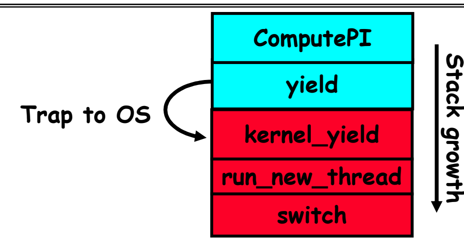
Review: Yielding through Internal Events

- Blocking on I/O
  - The act of requesting I/O implicitly yields the CPU
- Waiting on a "signal" from other thread
  - Thread asks to wait and thus yields the CPU
- Thread executes a `yield()`
  - Thread volunteers to give up CPU

```
computePI() {
    while(TRUE) {
        ComputeNextDigit();
        yield();
    }
}
```

  - Note that `yield()` must be called by programmer frequently enough!

Review: Stack for Yielding Thread



- How do we run a new thread?

```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* Later in lecture */
}
```

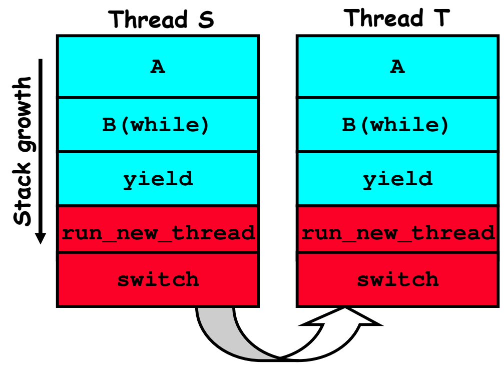
- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack
  - Maintain isolation for each thread

Review: Two Thread Yield Example

- Consider the following code blocks:

```
proc A() {
   B();

}
proc B() {
   while(TRUE) {
      yield();
   }
}
```



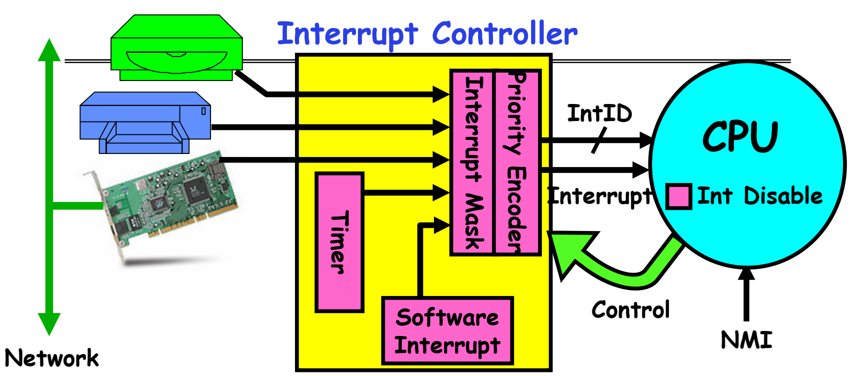
- Suppose we have 2 threads:
  - Threads S and T

Goals for Today

- More on Interrupts
- Thread Creation/Destruction
- Cooperating Threads

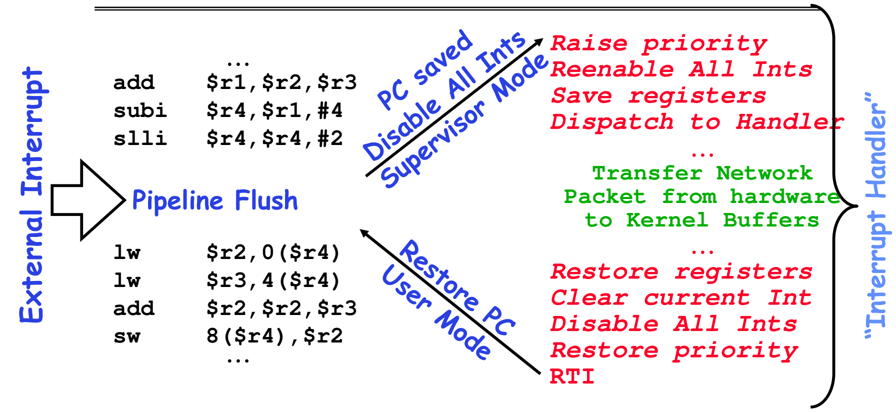
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Interrupt Controller



- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
  - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-maskable interrupt line (NMI) can't be disabled

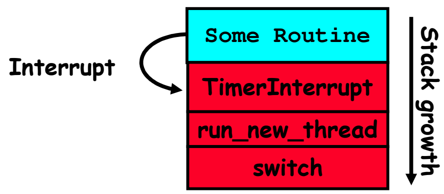
Example: Network Interrupt



- Disable/Enable All Ints ⇒ Internal CPU disable bit
  - RTI reenables interrupts, returns to user mode
- Raise/lower priority: change interrupt mask
- Software interrupts can be provided entirely in software at priority switching boundaries

Review: Preemptive Multithreading

- Use the timer interrupt to force scheduling decisions

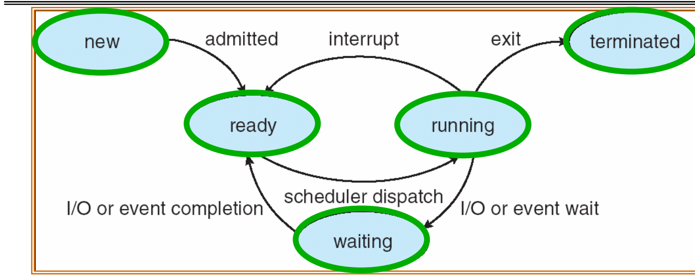


- Timer Interrupt routine:

```
TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```

- This is often called preemptive multithreading, since threads are prempted for better scheduling
  - Solves problem of user who doesn't insert yield();

Review: Lifecycle of a Thread (or Process)



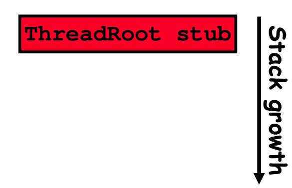
- As a thread executes, it changes state:
  - new: The thread is being created
  - ready: The thread is waiting to run
  - running: Instructions are being executed
  - waiting: Thread waiting for some event to occur
  - terminated: The thread has finished execution
- "Active" threads are represented by their TCBs
  - TCBs organized into queues based on their state

ThreadFork(): Create a New Thread

- ThreadFork() is a user-level procedure that creates a new thread and places it on ready queue
  - We called this CreateThread() earlier
- Arguments to ThreadFork()
  - Pointer to application routine (fcnPtr)
  - Pointer to array of arguments (fcnArgPtr)
  - Size of stack to allocate
- Implementation
  - Sanity Check arguments
  - Enter Kernel-mode and Sanity Check arguments again
  - Allocate new Stack and TCB
  - Initialize TCB and place on ready list (Runnable).

How do we initialize TCB and Stack?

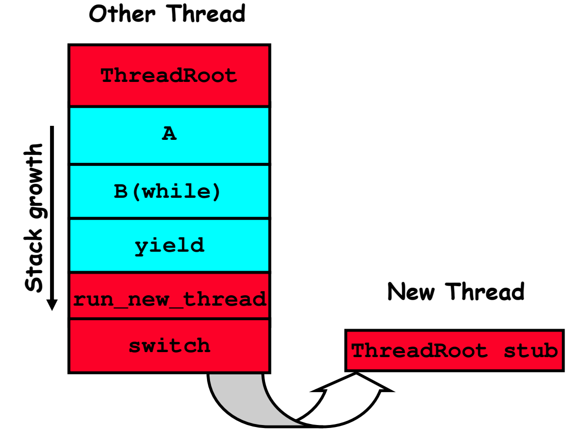
- Initialize Register fields of TCB
  - Stack pointer made to point at stack
  - PC return address ⇒ OS (asm) routine ThreadRoot()
  - Two arg registers initialized to fcnPtr and fcnArgPtr
- Initialize stack data?
  - No. Important part of stack frame is in registers (ra)
  - Think of stack frame as just before body of ThreadRoot() really gets started



Initial Stack

Administrivia

How does Thread get started?



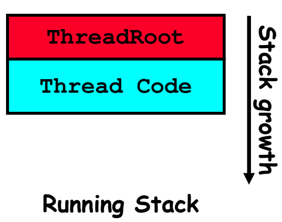
- Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`
  - This really starts the new thread

What does `ThreadRoot()` look like?

- ThreadRoot() is the root for the thread routine:

```
ThreadRoot() {
   DoStartupHousekeeping();
   UserModeSwitch(); /* enter user mode */
   Call fcnPtr(fcnArgPtr);
   ThreadFinish();
}
```

- Startup Housekeeping
  - Includes things like recording start time of thread
  - Other Statistics
- Stack will grow and shrink with execution of thread
- Final return from thread returns into ThreadRoot() which calls ThreadFinish()
  - ThreadFinish() will start at user-level



What does `ThreadFinish()` do?

- Needs to re-enter kernel mode (system call)
- "Wake up" (place on ready queue) threads waiting for this thread
  - Threads (like the parent) may be on a wait queue waiting for this thread to finish
- Can't deallocate thread yet
  - We are still running on its stack!
  - Instead, record thread as "waitingToBeDestroyed"
- Call `run_new_thread()` to run another thread:

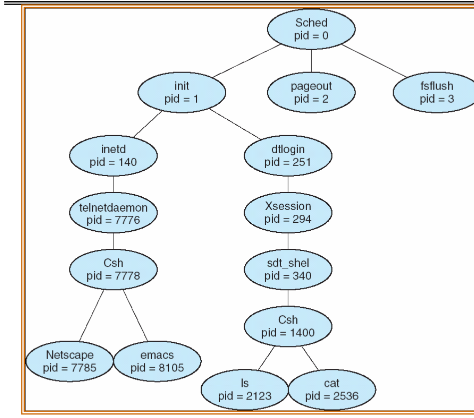
```
run_new_thread() {
   newThread = PickNewThread();
   switch(curThread, newThread);
   ThreadHouseKeeping();
}
```

  - `ThreadHouseKeeping()` notices waitingToBeDestroyed and deallocates the finished thread's TCB and stack

Additional Detail

- Thread Fork is not the same thing as UNIX fork
  - UNIX fork creates a new *process* so it has to create a new address space
  - For now, don't worry about how to create and switch between address spaces
- Thread fork is very much like an asynchronous procedure call
  - Runs procedure in separate thread
  - Calling thread doesn't wait for finish
- What if thread wants to exit early?
  - `ThreadFinish()` and `exit()` are essentially the same procedure entered at user level

Parent-Child relationship

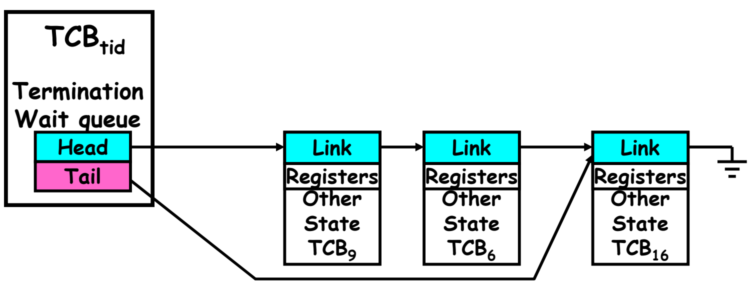


Typical process tree for Solaris system

- Every thread (and/or Process) has a parentage
  - A "parent" is a thread that creates another thread
  - A child of a parent was created by that parent

ThreadJoin() system call

- One thread can wait for another to finish with the `ThreadJoin(tid)` call
  - Calling thread will be taken off run queue and placed on waiting queue for thread `tid`
- Where is a logical place to store this wait queue?
  - On queue inside the TCB



- Similar to `wait()` system call in UNIX
  - Lets parents wait for child processes

Use of Join for Traditional Procedure Call

- A traditional procedure call is logically equivalent to doing a ThreadFork followed by ThreadJoin
- Consider the following normal procedure call of B() by A():

```
A() { B(); }
B() { Do interesting, complex stuff }
```

- The procedure A() is equivalent to A'():

```
A'() {
   tid = ThreadFork(B,null);
   ThreadJoin(tid);
}
```

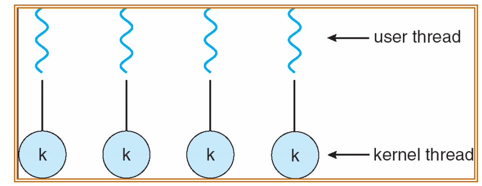
- Why not do this for every procedure?
  - Context Switch Overhead
  - Memory Overhead for Stacks

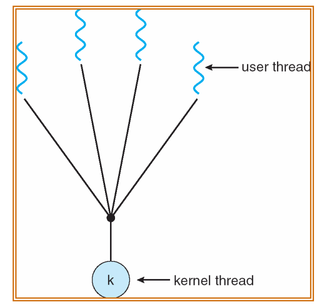
Kernel versus User-Mode threads

- We have been talking about Kernel threads
  - 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
- Even lighter weight option: User Threads
  - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-premptively relative to each other (only switch on yield())
  - Cheap
- Downside of user threads:
  - When one thread blocks on I/O, all threads block
  - Kernel cannot adjust scheduling among all threads

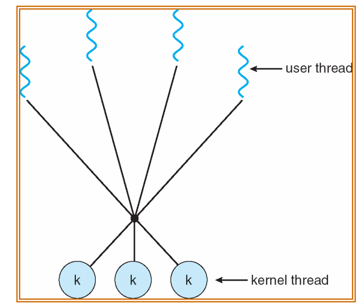
Threading models mentioned by book

Simple One-to-One Threading Model





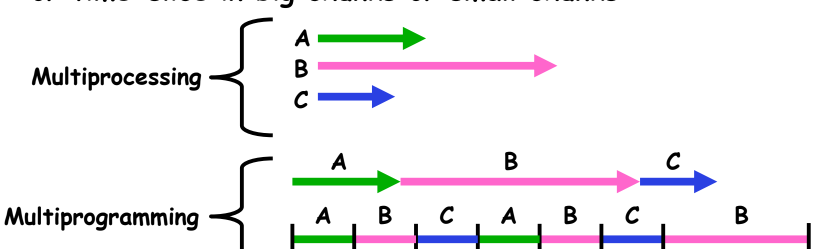
Many-to-One



Many-to-Many

Multiprocessing vs Multiprogramming

- Remember 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



Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
  - Can you test for this?
  - How can you know if your program works?
- Independent Threads:
  - No state shared with other threads
  - Deterministic ⇒ Input state determines results
  - Reproducible ⇒ Can recreate Starting Conditions, I/O
  - Scheduling order doesn't matter (if `switch()` works!!!)
- Cooperating Threads:
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible
- Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called "Heisenbugs"

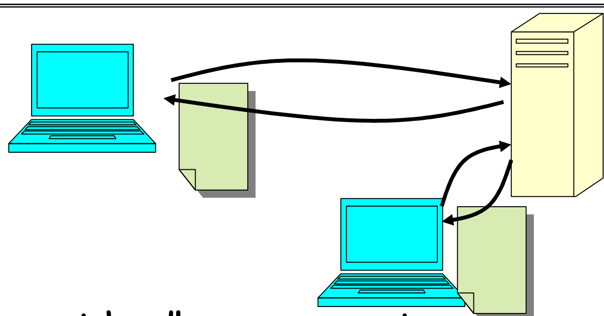
Interactions Complicate Debugging

- Is any program truly independent?
  - Every process shares the file system, OS resources, network, etc
  - Extreme example: buggy device driver causes thread A to crash "independent thread" B
- You probably don't realize how much you depend on reproducibility:
  - Example: Evil C compiler
    - » Modifies files behind your back by inserting errors into C program unless you insert debugging code
  - Example: Debugging statements can overrun stack
- Non-deterministic errors are really difficult to find
  - Example: Memory layout of kernel+user programs
    - » depends on scheduling, which depends on timer/other things
    - » Original UNIX had a bunch of non-deterministic errors
  - Example: Something which does interesting I/O
    - » User typing of letters used to help generate secure keys

Why allow cooperating threads?

- People cooperate; computers help/enhance people's lives, so computers must cooperate
  - By analogy, the non-reproducibility/non-determinism of people is a notable problem for "carefully laid plans"
- Advantage 1: Share resources
  - One computer, many users
  - One bank balance, many ATMs
    - » What if ATMs were only updated at night?
  - Embedded systems (robot control: coordinate arm & hand)
- Advantage 2: Speedup
  - Overlap I/O and computation
    - » Many different file systems do read-ahead
  - Multiprocessors – chop up program into parallel pieces
- Advantage 3: Modularity
  - More important than you might think
  - Chop large problem up into simpler pieces
    - » To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
    - » Makes system easier to extend

High-level Example: Web Server



- Server must handle many requests
- Non-cooperating version:

```
serverLoop() {
    con = AcceptCon();
    ProcessFork(ServiceWebPage(),con);
}
```

- What are some disadvantages of this technique?

Threaded Web Server

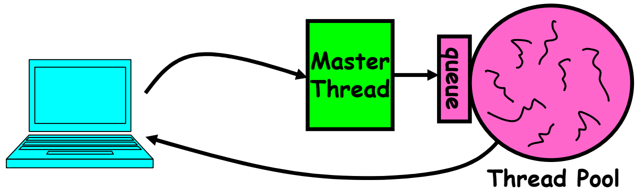
- Now, use a single process
- Multithreaded (cooperating) version:

```
serverLoop() {
    connection = AcceptCon();
    ThreadFork(ServiceWebPage(),connection);
}
```

- Looks almost the same, but has many advantages:
  - Can share file caches kept in memory, results of CGI scripts, other things
  - Threads are *much* cheaper to create than processes, so this has a lower per-request overhead
- Question: would a user-level (say one-to-many) thread package make sense here?
  - When one request blocks on disk, all block…
- What about Denial of Service attacks or digg / Slash-dot effects?

Thread Pools

- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded "pool" of worker threads, representing the maximum level of multiprogramming




```
master() {
   allocThreads(worker,queue);
   while(TRUE) {
      con=AcceptCon();
      Enqueue(queue,con);
      wakeUp(queue);
   }
}
```

```
worker(queue) {
   while(TRUE) {
      con=Dequeue(queue);
      if (con==null)
         sleepOn(queue);
      else
         ServiceWebPage(con);
   }
}
```


Summary

- Interrupts: hardware mechanism for returning control to operating system
  - Used for important/high-priority events
  - Can force dispatcher to schedule a different thread (premptive multithreading)
- New Threads Created with `ThreadFork()`
  - Create initial TCB and stack to point at `ThreadRoot()`
  - `ThreadRoot()` calls thread code, then `ThreadFinish()`
  - `ThreadFinish()` wakes up waiting threads then prepares TCB/stack for distruction
- Threads can wait for other threads using `ThreadJoin()`
- Threads may be at user-level or kernel level
- Cooperating threads have many potential advantages
  - But: introduces non-reproducibility and non-determinism
  - Need to have Atomic operations

CS162
Operating Systems and Systems Programming
Lecture 6

Synchronization

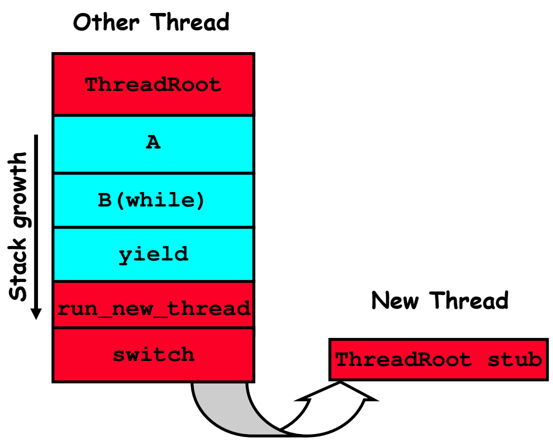
September 17, 2007

Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

Review: `ThreadFork()`: Create a New Thread

- `ThreadFork()` is a user-level procedure that creates a new thread and places it on ready queue
- Arguments to `ThreadFork()`
  - Pointer to application routine (fcnPtr)
  - Pointer to array of arguments (fcnArgPtr)
  - Size of stack to allocate
- Implementation
  - Sanity Check arguments
  - Enter Kernel-mode and Sanity Check arguments again
  - Allocate new Stack and TCB
  - Initialize TCB and place on ready list (Runnable).

Review: How does Thread get started?



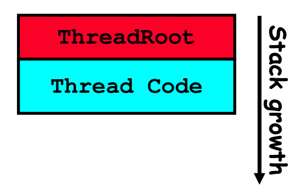
- Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`
  - This really starts the new thread

Review: What does `ThreadRoot()` look like?

- ThreadRoot() is the root for the thread routine:

```
ThreadRoot() {
  DoStartupHousekeeping();
  UserModeSwitch(); /* enter user mode */
  Call fcnPtr(fcnArgPtr);
  ThreadFinish();
}
```

- Startup Housekeeping
  - Includes things like recording start time of thread
  - Other Statistics
- Stack will grow and shrink with execution of thread



Running Stack

- Final return from thread returns into ThreadRoot() which calls ThreadFinish()
  - ThreadFinish() wake up sleeping threads

Review: Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
- Independent Threads:
  - No state shared with other threads
  - Deterministic ⇒ Input state determines results
  - Reproducible ⇒ Can recreate Starting Conditions, I/O
  - Scheduling order doesn't matter (if `switch()` works!!!)
- Cooperating Threads:
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible
- Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called "Heisenbugs"

Goals for Today

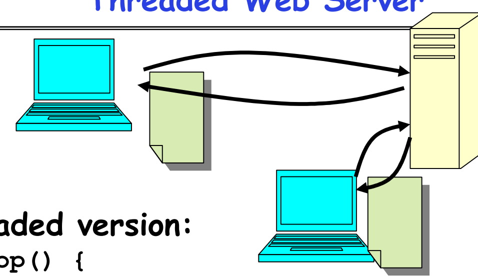
- Concurrency examples
- Need for synchronization
- Examples of valid synchronization

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Why allow cooperating threads?

- People cooperate; computers help/enhance people's lives, so computers must cooperate
  - By analogy, the non-reproducibility/non-determinism of people is a notable problem for "carefully laid plans"
- Advantage 1: Share resources
  - One computer, many users
  - One bank balance, many ATMs
    - » What if ATMs were only updated at night?
  - Embedded systems (robot control: coordinate arm & hand)
- Advantage 2: Speedup
  - Overlap I/O and computation
    - » Many different file systems do read-ahead
  - Multiprocessors – chop up program into parallel pieces
- Advantage 3: Modularity
  - More important than you might think
  - Chop large problem up into simpler pieces
    - » To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
    - » Makes system easier to extend

Threaded Web Server



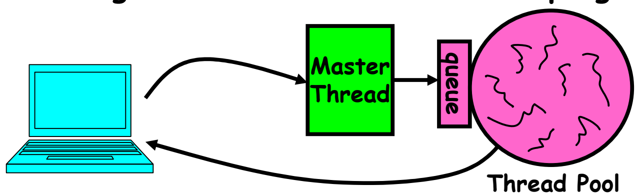
- Multithreaded version:

```
serverLoop() {
    connection = AcceptCon();
    ThreadFork(ServiceWebPage(),connection);
}
```

- Advantages of threaded version:
  - Can share file caches kept in memory, results of CGI scripts, other things
  - Threads are *much* cheaper to create than processes, so this has a lower per-request overhead
- What if too many requests come in at once?

Thread Pools

- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded "pool" of threads, representing the maximum level of multiprogramming

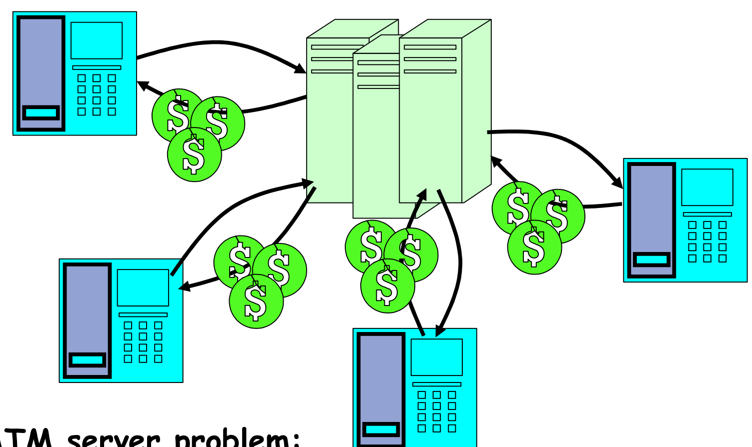



```
master() {
   allocThreads(slave,queue);
   while(TRUE) {
      con=AcceptCon();
      Enqueue(queue,con);
      wakeUp(queue);
   }
}
```

```
slave(queue) {
   while(TRUE) {
      con=Dequeue(queue);
      if (con==null)
         sleepOn(queue);
      else
         ServiceWebPage(con);
   }
}
```


Administrivia

ATM Bank Server



- ATM server problem:
  - Service a set of requests
  - Do so without corrupting database
  - Don't hand out too much money

ATM bank server example

- Suppose we wanted to implement a server process to handle requests from an ATM network:

```
BankServer() {
   while (TRUE) {
      ReceiveRequest(&op, &acctId, &amount);
      ProcessRequest(op, acctId, amount);
   }
}
ProcessRequest(op, acctId, amount) {
   if (op == deposit) Deposit(acctId, amount);
   else if …
}
Deposit(acctId, amount) {
   acct = GetAccount(acctId); /* may use disk I/O */
   acct->balance += amount;
   StoreAccount(acct); /* Involves disk I/O */
}
```

- How could we speed this up?
  - More than one request being processed at once
  - Event driven (overlap computation and I/O)
  - Multiple threads (multi-proc, or overlap comp and I/O)

Event Driven Version of ATM server

- Suppose we only had one CPU
  - Still like to overlap I/O with computation
  - Without threads, we would have to rewrite in event-driven style
- Example

```
BankServer() {
   while(TRUE) {
      event = WaitForNextEvent();
      if (event == ATMRequest)
         StartOnRequest();
      else if (event == AcctAvail)
         ContinueRequest();
      else if (event == AcctStored)
         FinishRequest();
   }
}
```

  - What if we missed a blocking I/O step?
  - What if we have to split code into hundreds of pieces which could be blocking?
  - This technique is used for graphical programming

Can Threads Make This Easier?

- Threads yield overlapped I/O and computation without "deconstructing" code into non-blocking fragments
  - One thread per request
- Requests proceeds to completion, blocking as required:

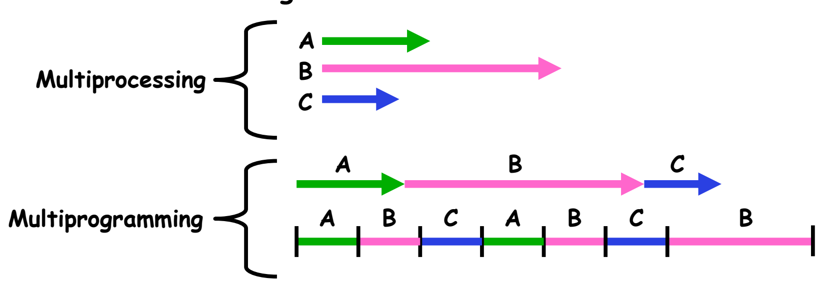
```
Deposit(acctId, amount) {
  acct = GetAccount(actId); /* May use disk I/O */
  acct->balance += amount;
  StoreAccount(acct);      /* Involves disk I/O */
}
```

- Unfortunately, shared state can get corrupted:

Thread 1	Thread 2
`load r1, acct->balance`	
	`load r1, acct->balance`
	`add r1, amount2`
	`store r1, acct->balance`
`add r1, amount1`	
`store r1, acct->balance`	

Review: Multiprocessing vs Multiprogramming

- 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



- Also recall: Hyperthreading
  - Possible to interleave threads on a per-instruction basis
  - Keep this in mind for our examples (like multiprocessing)

Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A	Thread B
x = 1;	y = 2;

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

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

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

Thread A	Thread B
x = 1;	x = 2;

  - X could be 1 or 2 (non-deterministic!)
  - Could even be 3 for serial processors:
    » Thread A writes 0001, B writes 0010.
    » Scheduling order ABABABBA yields 3!

Atomic Operations

- To understand a concurrent program, we need to know what the underlying indivisible operations are!
- **Atomic Operation**: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array

Correctness Requirements

- Threaded programs must work for all interleavings of thread instruction sequences
  - Cooperating threads inherently non-deterministic and non-reproducible
  - Really hard to debug unless carefully designed!
- Example: Therac-25
  - Machine for radiation therapy
    - » Software control of electron accelerator and electron beam/ Xray production
    - » Software control of dosage
  - Software errors caused the death of several patients
    - » A series of race conditions on shared variables and poor software design
    - » "They determined that data entry speed during editing was the key factor in producing the error condition: If the prescription data was edited at a fast pace, the overdose occurred."

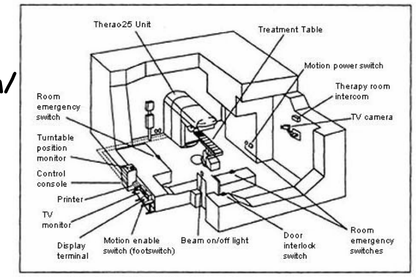
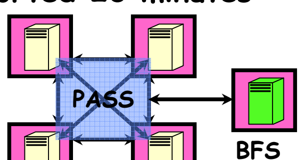


Figure 1. Typical Therac-25 facility

Space Shuttle Example

- Original Space Shuttle launch aborted 20 minutes before scheduled launch
- Shuttle has five computers:
  - Four run the "Primary Avionics Software System" (PASS)
    - » Asynchronous and real-time
    - » Runs all of the control systems
    - » Results synchronized and compared every 3 to 4 ms
  - The Fifth computer is the "Backup Flight System" (BFS)
    - » stays synchronized in case it is needed
    - » Written by completely different team than PASS
- Countdown aborted because BFS disagreed with PASS
  - A 1/67 chance that PASS was out of sync one cycle
  - Bug due to modifications in initialization code of PASS
    - » A delayed init request placed into timer queue
    - » As a result, timer queue not empty at expected time to force use of hardware clock
  - Bug not found during extensive simulation



Another Concurrent Program Example

- Two threads, A and B, compete with each other
  - One tries to increment a shared counter
  - The other tries to decrement the counter

Thread A	Thread B

i = 0;
while (i < 10)
  i = i + 1;
printf("A wins!");
	i = 0;
	while (i > -10)
	  i = i – 1;
	printf("B wins!");

- Assume that memory loads and stores are atomic, but incrementing and decrementing are *not* atomic
- Who wins? Could be either
- Is it guaranteed that someone wins? Why or why not?
- What it both threads have their own CPU running at same speed? Is it guaranteed that it goes on forever?

Hand Simulation Multiprocessor Example

Motivation: "Too much milk"

- Great thing about OS's – analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people
- Example: People need to coordinate:

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Definitions

- **Synchronization**: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We are going to show that its hard to build anything useful with only reads and writes
- **Mutual Exclusion**: ensuring that only one thread does a particular thing at a time
  - One thread *excludes* the other while doing its task
- **Critical Section**: piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code.
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing.

More Definitions

- **Lock**: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - » Important idea: all synchronization involves waiting
- For example: fix the milk problem by putting a key on the refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: roommate angry if only wants OJ

  - Of Course – We don't know how to make a lock yet

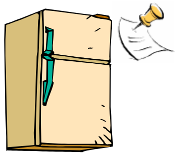
Too Much Milk: Correctness Properties

- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Always write down behavior first
  - Impulse is to start coding first, then when it doesn't work, pull hair out
  - Instead, think first, then code
- What are the correctness properties for the "Too much milk" problem???
  - Never more than one person buys
  - Someone buys if needed
- Restrict ourselves to use only atomic load and store operations as building blocks

Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of "lock")
  - Remove note after buying (kind of "unlock")
  - Don't buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {
    if (noNote) {
        leave Note;
        buy milk;
        remove note;
    }
}
```



- Result?
  - Still too much milk but only occasionally!
  - Thread can get context switched after checking milk and note but before buying milk!
- Solution makes problem worse since fails intermittently
  - Makes it really hard to debug…
  - Must work despite what the dispatcher does!

Too Much Milk: Solution #1½

- Clearly the Note is not quite blocking enough
  - Let's try to fix this by placing note first
- Another try at previous solution:

```
leave Note;
if (noMilk) {
    if (noNote) {
        leave Note;
        buy milk;
    }
}
remove note;
```

- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

Too Much Milk Solution #2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:

Thread A

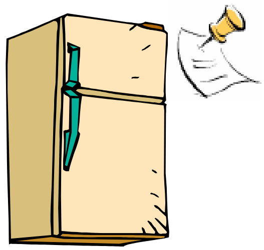
```
leave note A;
if (noNote B) {
    if (noMilk) {
        buy Milk;
    }
}
remove note A;
```

Thread B

```
leave note B;
if (noNoteA) {
    if (noMilk) {
        buy Milk;
    }
}
remove note B;
```

- Does this work?
- Possible for neither thread to buy milk
  - Context switches at exactly the wrong times can lead each to think that the other is going to buy
- Really insidious:
  - Extremely unlikely that this would happen, but will at worse possible time
  - Probably something like this in UNIX

Too Much Milk Solution #2: problem!



- *I'm* not getting milk, *You're* getting milk
- This kind of lockup is called "starvation!"

Too Much Milk Solution #3

- Here is a possible two-note solution:

Thread A

```
leave note A;
while (note B) { //X
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) { //Y
   if (noMilk) {
      buy milk;
   }
}
remove note B;
```

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At X:
  - if no note B, safe for A to buy,
  - otherwise wait to find out what will happen
- At Y:
  - if no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

Solution #3 discussion

- Our solution protects a single "Critical-Section" piece of code for each thread:

```
if (noMilk) {
   buy milk;
}
```

- Solution #3 works, but it's really unsatisfactory
  - Really complex – even for this simple an example
    - » Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of threads?
    - » Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - » This is called "busy-waiting"
- There's a better way
  - Have hardware provide better (higher-level) primitives than atomic load and store
  - Build even higher-level programming abstractions on this new hardware support

Too Much Milk: Solution #4

- Suppose we have some sort of implementation of a lock (more in a moment).
  - `Lock.Acquire()` – wait until lock is free, then grab
  - `Lock.Release()` – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- Then, our milk problem is easy:

```
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

- Once again, section of code between `Acquire()` and `Release()` called a "Critical Section"
- Of course, you can make this even simpler: suppose you are out of ice cream instead of milk
  - Skip the test since you always need more ice cream.

Where are we going with synchronization?

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

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

Summary

- Concurrent threads are a very useful abstraction
  - Allow transparent overlapping of computation and I/O
  - Allow use of parallel processing when available
- Concurrent threads introduce problems when accessing shared data
  - Programs must be insensitive to arbitrary interleavings
  - Without careful design, shared variables can become completely inconsistent
- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Showed how to protect a critical section with only atomic load and store $\Rightarrow$ pretty complex!

CS162
Operating Systems and Systems Programming
Lecture 7

Mutual Exclusion, Semaphores, Monitors, and Condition Variables

September 19, 2007

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

Review: A Concurrent Program Example

- Two threads, A and B, compete with each other
  - One tries to increment a shared counter
  - The other tries to decrement the counter

Thread A

```
i = 0;
while (i < 10)
   i = i + 1;
printf("A wins!");
```

Thread B

```
i = 0;
while (i > -10)
   i = i - 1;
printf("B wins!");
```

- Assume that memory loads and stores are atomic, but incrementing and decrementing are *not* atomic
- Who wins? Could be either
- Is it guaranteed that someone wins? Why or why not?
- What it both threads have their own CPU running at same speed? Is it guaranteed that it goes on forever?

Hand Simulation Multiprocessor Example

Review: Too Much Milk Solution #3

- Here is a possible two-note solution:

Thread A

```
leave note A;
while (note B) {\\X
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {\\Y
   if (noMilk) {
      buy milk;
   }
}
remove note B;
```

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At X:
  - if no note B, safe for A to buy,
  - otherwise wait to find out what will happen
- At Y:
  - if no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

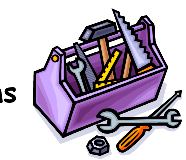
Goals for Today

- Hardware Support for Synchronization
- Higher-level Synchronization Abstractions
  - Semaphores, monitors, and condition variables
- Programming paradigms for concurrent programs

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

High-Level Picture

- The abstraction of threads is good:
  - Maintains sequential execution model
  - Allows simple parallelism to overlap I/O and computation
- Unfortunately, still too complicated to access state shared between threads
  - Consider "too much milk" example
  - Implementing a concurrent program with only loads and stores would be tricky and error-prone
- Today, we'll implement higher-level operations on top of atomic operations provided by hardware
  - Develop a "synchronization toolbox"
  - Explore some common programming paradigms



Where are we going with synchronization?

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

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level

How to implement Locks?

- **Lock**: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - » Important idea: all synchronization involves waiting
- Atomic Load/Store: get solution like Milk #3
  - Looked at this last lecture
  - Pretty complex and error prone
- Hardware Lock instruction
  - Is this a good idea?
  - Complexity?
    - » Done in the Intel 432
    - » Each feature makes hardware more complex and slow
  - What about putting a task to sleep?
    - » How do you handle the interface between the hardware and scheduler?

Naïve use of Interrupt Enable/Disable

- How can we build multi-instruction atomic operations?
  - Recall: dispatcher gets control in two ways.
    - » Internal: Thread does something to relinquish the CPU
    - » External: Interrupts cause dispatcher to take CPU
  - On a uniprocessor, can avoid context-switching by:
    - » Avoiding internal events (although virtual memory tricky)
    - » Preventing external events by disabling interrupts
- Consequently, naïve Implementation of locks:

```
LockAcquire { disable Ints; }
LockRelease { enable Ints; }
```

- Problems with this approach:
  - Can't let user do this! Consider following:

```
LockAcquire();
While(TRUE) {;}
```

  - Real-Time system—no guarantees on timing!
    - » Critical Sections might be arbitrarily long
  - What happens with I/O or other important events?
    - » "Reactor about to meltdown. Help?"

Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```
int value = FREE;
```

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```


New Lock Implementation: Discussion

- Why do we need to disable interrupts at all?
  - Avoid interruption between checking and setting lock value
  - Otherwise two threads could think that they both have lock

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {          ┐
      put thread on wait queue;  │
      Go to sleep();             │
      // Enable interrupts?      │ Critical
   } else {                      │ Section
      value = BUSY;              │
   }                             ┘
   enable interrupts;
}
```

- Note: unlike previous solution, the critical section (inside `Acquire()`) is very short
  - User of lock can take as long as they like in their own critical section: doesn't impact global machine behavior
  - Critical interrupts taken in time!

Interrupt re-enable in going to sleep

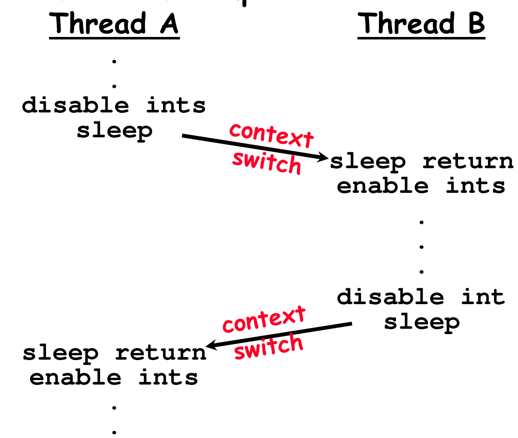
- What about re-enabling ints when going to sleep?

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
Enable Position →  put thread on wait queue;
Enable Position →  Go to sleep();
Enable Position →
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

Administrivia

How to Re-enable After Sleep()?

- In Nachos, since ints are disabled when you call sleep:
  - Responsibility of the next thread to re-enable ints
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts



Interrupt disable and enable across context switches

- An important point about structuring code:
  - In Nachos code you will see lots of comments about assumptions made concerning when interrupts disabled
  - This is an example of where modifications to and assumptions about program state can't be localized within a small body of code
  - In these cases it is possible for your program to eventually "acquire" bugs as people modify code
- Other cases where this will be a concern?
  - What about exceptions that occur after lock is acquired? Who releases the lock?

```
mylock.acquire();
a = b / 0;
mylock.release()
```


Atomic Read-Modify-Write instructions

- Problems with previous solution:
  - Can't give lock implementation to users
  - Doesn't work well on multiprocessor
    - » Disabling interrupts on all processors requires messages and would be very time consuming
- Alternative: atomic instruction sequences
  - These instructions read a value from memory and write a new value atomically
  - Hardware is responsible for implementing this correctly
    - » on both uniprocessors (not too hard)
    - » and multiprocessors (requires help from cache coherence protocol)
  - Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors

Examples of Read-Modify-Write

```
• test&set (&address) {      /* most architectures */
      result = M[address];
      M[address] = 1;
      return result;
  }
• 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;
  }
```


Implementing Locks with test&set

- Another flawed, but simple solution:

```
int value = 0; // Free
Acquire() {
   while (test&set(value)); // while busy
}
Release() {
   value = 0;
}
```

- Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
  - If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock
- Busy-Waiting: thread consumes cycles while waiting

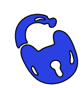
Problem: Busy-Waiting for Lock

- Positives for this solution
  - Machine can receive interrupts
  - User code can use this lock
  - Works on a multiprocessor
- Negatives
  - This is very inefficient because the busy-waiting thread will consume cycles waiting
  - Waiting thread may take cycles away from thread holding lock (no one wins!)
  - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!
- Priority Inversion problem with original Martian rover
- For semaphores and monitors, waiting thread may wait for an arbitrary length of time!
  - Thus even if busy-waiting was OK for locks, definitely not ok for other primitives
  - Homework/exam solutions should not have busy-waiting!

Better Locks using test&set

- Can we build test&set locks without busy-waiting?
  - Can't entirely, but can minimize!
  - Idea: only busy-wait to atomically check lock value

```
int guard = 0;
int value = FREE;
```



```
Acquire() {
   // Short busy-wait time
   while (test&set(guard));
   if (value == BUSY) {
      put thread on wait queue;
      go to sleep() & guard = 0;
   } else {
      value = BUSY;
      guard = 0;
   }
}
```

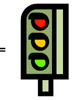
```
Release() {
   // Short busy-wait time
   while (test&set(guard));
   if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   guard = 0;
```

- Note: sleep has to be sure to reset the guard variable
  - Why can't we do it just before or just after the sleep?

Higher-level Primitives than Locks

- Goal of last couple of lectures:
  - What is the right abstraction for synchronizing threads that share memory?
  - Want as high a level primitive as possible
- Good primitives and practices important!
  - Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
  - UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs
- Synchronization is a way of coordinating multiple concurrent activities that are using shared state
  - This lecture and the next presents a couple of ways of structuring the sharing

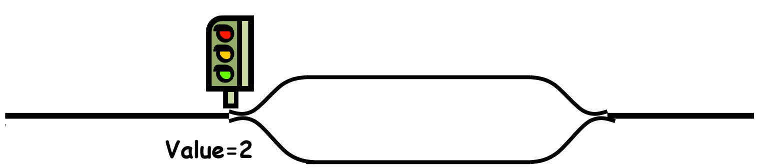
Semaphores



- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - » Think of this as the wait() operation
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - » This of this as the signal() operation
  - Note that P() stands for "*proberen*" (to test) and V() stands for "*verhogen*" (to increment) in Dutch

Semaphores Like Integers Except

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can't read or write value, except to set it initially
  - Operations must be atomic
    - » Two P's together can't decrement value below zero
    - » Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Two Uses of Semaphores

- Mutual Exclusion (initial value = 1)
  - Also called "Binary Semaphore".
  - Can be used for mutual exclusion:

```
semaphore.P();
// Critical section goes here
semaphore.V();
```

- Scheduling Constraints (initial value = 0)
  - Locks are fine for mutual exclusion, but what if you want a thread to wait for something?
  - Example: suppose you had to implement ThreadJoin which must wait for thread to terminiate:

```
Initial value of semaphore = 0
ThreadJoin {
   semaphore.P();
}
ThreadFinish {
   semaphore.V();
}
```

Producer-consumer with a bounded buffer

- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer
- Don't want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty
- Example 1: GCC compiler
  - cpp | cc1 | cc2 | as | ld
- Example 2: Coke machine
  - Producer can put limited number of cokes in machine
  - Consumer can't take cokes out if machine is empty

Correctness constraints for solution

- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine
- General rule of thumb:
  Use a separate semaphore for each constraint
  - `Semaphore fullBuffers; // consumer's constraint`
  - `Semaphore emptyBuffers;// producer's constraint`
  - `Semaphore mutex;        // mutual exclusion`

Full Solution to Bounded Buffer

```
Semaphore fullBuffer = 0; // Initially, no coke
Semaphore emptyBuffers = numBuffers;
                          // Initially, num empty slots
Semaphore mutex = 1;      // No one using machine

Producer(item) {
    emptyBuffers.P();     // Wait until space
    mutex.P();            // Wait until buffer free
    Enqueue(item);
    mutex.V();
    fullBuffers.V();      // Tell consumers there is
                          // more coke
}
Consumer() {
    fullBuffers.P();      // Check if there's a coke
    mutex.P();            // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V();     // tell producer need more
    return item;
}
```

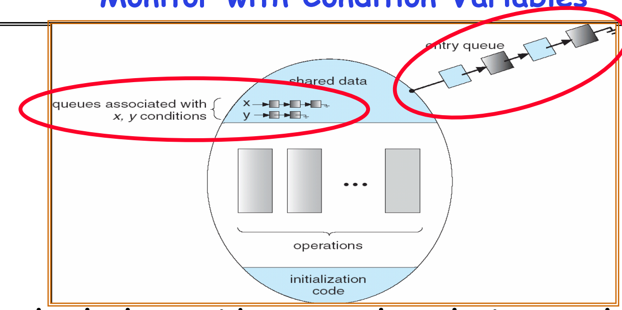

Discussion about Solution

- Why asymmetry?
  - Producer does: `emptyBuffer.P(), fullBuffer.V()`
  - Consumer does: `fullBuffer.P(), emptyBuffer.V()`
- Is order of P's important?
- Is order of V's important?
- What if we have 2 producers or 2 consumers?
  - Do we need to change anything?

Motivation for Monitors and Condition Variables

- Semaphores are a huge step up; just think of trying to do the bounded buffer with only 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?
- Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
- Definition: **Monitor**: a lock and zero or more condition variables for managing concurrent access to shared data
  - Some languages like Java provide this natively
  - Most others use actual locks and condition variables

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
  - Contrast to semaphores: Can't wait inside critical section

Simple Monitor Example

- Here is an (infinite) synchronized queue

```
Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
   lock.Acquire();           // Get Lock
   queue.enqueue(item);      // Add item
   dataready.signal();       // Signal any waiters
   lock.Release();           // Release Lock
}

RemoveFromQueue() {
   lock.Acquire();           // Get Lock
   while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
   }
   item = queue.dequeue();   // Get next item
   lock.Release();           // Release Lock
   return(item);
}
```


Summary

- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Talked about hardware atomicity primitives:
  - Disabling of Interrupts, test&set, swap, comp&swap, load-linked/store conditional
- Showed several constructions of Locks
  - Must be very careful not to waste/tie up machine resources
    - » Shouldn't disable interrupts for long
    - » Shouldn't spin wait for long
  - Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable
- Talked about Semaphores, Monitors, and Condition Variables
  - Higher level constructs that are harder to "screw up"

CS162
Operating Systems and Systems Programming
Lecture 8

Readers-Writers
Language Support for Synchronization

September 24, 2007

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

Review: Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```
int value = FREE;
```

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```


Review: How to Re-enable After Sleep()?

- In Nachos, since ints are disabled when you call sleep:
  - Responsibility of the next thread to re-enable ints
  - When the sleeping thread wakes up, returns to acquire and re-enables interrupts

```
    Thread A                  Thread B
       .
       .
  disable ints
     sleep      context switch →
                              sleep return
                              enable ints
                                   .
                                   .
                                   .
                              disable int
                                 sleep
  sleep return ← context switch
  enable ints
       .
       .
```


Review: Locks using test&set

- Can we build test&set locks without busy-waiting?
  - Can't entirely, but can minimize!
  - Idea: only busy-wait to atomically check lock value

```
int guard = 0;
int value = FREE;
```

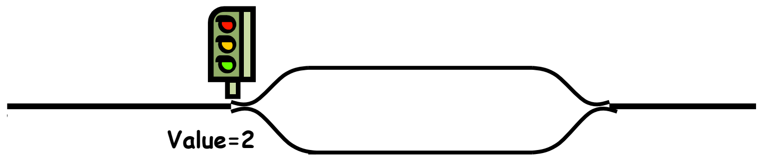
```
Acquire() {
   // Short busy-wait time
   while (test&set(guard));
   if (value == BUSY) {
      put thread on wait queue;
      go to sleep() & guard = 0;
   } else {
      value = BUSY;
      guard = 0;
   }
}
```

```
Release() {
   // Short busy-wait time
   while (test&set(guard));
   if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   guard = 0;
```

- Note: sleep has to be sure to reset the guard variable
  - Why can't we do it just before or just after the sleep?

Review: Semaphores

- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - » Think of this as the wait() operation
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - » This of this as the signal() operation
  - Only time can set integer directly is at initialization time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Goals for Today

- Continue with Synchronization Abstractions
  - Monitors and condition variables
- Readers-Writers problem and solutoin
- Language Support for Synchronization

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Review: Producer-consumer with a bounded buffer

- Problem Definition
  - Producer puts things into a shared buffer (wait if full)
  - Consumer takes them out (wait if empty)
  - Use a fixed-size buffer between them to avoid lockstep
    - » Need to synchronize access to this buffer
- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
- General rule of thumb:
  Use a separate semaphore for each constraint
  - `Semaphore fullBuffers; // consumer's constraint`
  - `Semaphore emptyBuffers;// producer's constraint`
  - `Semaphore mutex;       // mutual exclusion`

Review: Full Solution to Bounded Buffer

```
Semaphore fullBuffer = 0; // Initially, no coke
Semaphore emptyBuffers = numBuffers;
                          // Initially, num empty slots
Semaphore mutex = 1;      // No one using machine

Producer(item) {
    emptyBuffers.P();     // Wait until space
    mutex.P();            // Wait until buffer free
    Enqueue(item);
    mutex.V();
    fullBuffers.V();      // Tell consumers there is
                          // more coke
}
Consumer() {
    fullBuffers.P();      // Check if there's a coke
    mutex.P();            // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V();     // tell producer need more
    return item;
}
```

Discussion about Bounded Buffer Solution

- Why asymmetry?
  - Producer does: `emptyBuffer.P(), fullBuffer.V()`
  - Consumer does: `fullBuffer.P(), emptyBuffer.V()`
- Is order of P's important?
- Is order of V's important?
- What if we have 2 producers or 2 consumers?
  - Do we need to change anything?

Motivation for Monitors and Condition Variables

- Semaphores are a huge step up, but:
  - They are confusing because they are dual purpose:
    - » Both mutual exclusion and scheduling constraints
    - » Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious
  - Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
- Definition: **Monitor**: a lock and zero or more condition variables for managing concurrent access to shared data
  - Use of Monitors is a programming paradigm
  - Some languages like Java provide monitors in the language
- 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

Simple Monitor Example (version 1)

- Here is an (infinite) synchronized queue

```
Lock lock;
Queue queue;

AddToQueue(item) {
   lock.Acquire();          // Lock shared data
   queue.enqueue(item);     // Add item
   lock.Release();          // Release Lock
}

RemoveFromQueue() {
   lock.Acquire();          // Lock shared data
   item = queue.dequeue();// Get next item or null
   lock.Release();          // Release Lock
   return(item);            // Might return null
}
```


Condition Variables

- How do we change the RemoveFromQueue() routine to wait until something is on the queue?
  - Could do this by keeping a count of the number of things on the queue (with semaphores), but error prone
- **Condition Variable**: a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section
- Operations:
  - `Wait(&lock)`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - `Signal()`: Wake up one waiter, if any
  - `Broadcast()`: Wake up all waiters
- Rule: Must hold lock when doing condition variable ops!
  - In Birrell paper, he says can perform signal() outside of lock – IGNORE HIM (this is only an optimization)

Complete Monitor Example (with condition variable)

- Here is an (infinite) synchronized queue

```
Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
   lock.Acquire();          // Get Lock
   queue.enqueue(item);     // Add item
   dataready.signal();      // Signal any waiters
   lock.Release();          // Release Lock
}

RemoveFromQueue() {
   lock.Acquire();          // Get Lock
   while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
   }
   item = queue.dequeue();  // Get next item
   lock.Release();          // Release Lock
   return(item);
}
```


Mesa vs. Hoare monitors

- Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:

```
while (queue.isEmpty()) {
   dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue(); // Get next item
```

  – Why didn't we do this?

```
if (queue.isEmpty()) {
   dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue(); // Get next item
```

- Answer: depends on the type of scheduling
  – Hoare-style (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
  – Mesa-style (Nachos, 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

Administrivia

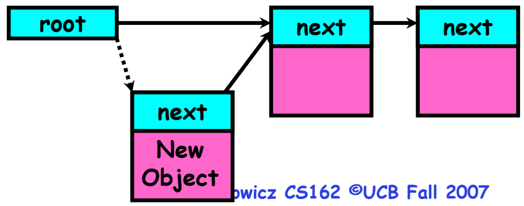
Using of Compare&Swap for queues

- compare&swap (&address, reg1, reg2) { /* 68000 */

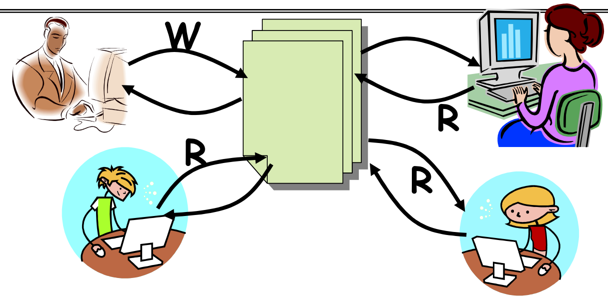
```
compare&swap (&address, reg1, reg2) { /* 68000 */
    if (reg1 == M[address]) {
       M[address] = reg2;
       return success;
    } else {
       return failure;
    }
}
```

Here is an atomic add to linked-list function:

```
addToQueue(&object) {
    do {                     // repeat until no conflict
       ld r1, M[root]        // Get ptr to current head
       st r1, M[object]      // Save link in new object
    } until (compare&swap(&root,r1,object));
}
```



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

Basic Readers/Writers Solution

- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:
  - Reader()
    ```
    Wait until no writers
    Access data base
    Check out – wake up a waiting writer
    ```
  - Writer()
    ```
    Wait until no active readers or writers
    Access database
    Check out – wake up waiting readers or writer
    ```
  - State variables (Protected by a lock called "lock"):
    - » int AR: Number of active readers; initially = 0
    - » int WR: Number of waiting readers; initially = 0
    - » int AW: Number of active writers; initially = 0
    - » int WW: Number of waiting writers; initially = 0
    - » Condition okToRead = NIL
    - » Conditioin okToWrite = NIL

Code for a Reader

```
Reader() {
  // First check self into system
  lock.Acquire();
  while ((AW + WW) > 0) { // Is it safe to read?
    WR++;                 // No. Writers exist
    okToRead.wait(&lock); // Sleep on cond var
    WR--;                 // No longer waiting
  }
  AR++;                   // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  AR--;                   // No longer active
  if (AR == 0 && WW > 0)  // No other active readers
    okToWrite.signal();   // Wake up one writer
  lock.Release();
}
```


Code for a Writer

```
Writer() {
  // First check self into system
  lock.Acquire();
  while ((AW + AR) > 0) { // Is it safe to write?
    WW++;                 // No. Active users exist
    okToWrite.wait(&lock);  // Sleep on cond var
    WW--;                 // No longer waiting
  }
  AW++;                   // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  AW--;                   // No longer active
  if (WW > 0){            // Give priority to writers
    okToWrite.signal();   // Wake up one writer
  } else if (WR > 0) {    // Otherwise, wake reader
    okToRead.broadcast(); // Wake all readers
  }
  lock.Release();
}
```

Simulation of Readers/Writers solution

- Consider the following sequence of operators:
  - R1, R2, W1, R3
- On entry, each reader checks the following:

```
while ((AW + WW) > 0) { // Is it safe to read?
  WR++;                 // No. Writers exist
  okToRead.wait(&lock); // Sleep on cond var
  WR--;                 // No longer waiting
}
AR++;                   // Now we are active!
```

- First, R1 comes along:
  AR = 1, WR = 0, AW = 0, WW = 0
- Next, R2 comes along:
  AR = 2, WR = 0, AW = 0, WW = 0
- Now, readers make take a while to access database
  - Situation: Locks released
  - Only AR is non-zero

Simulation(2)

- Next, W1 comes along:

```
while ((AW + AR) > 0) { // Is it safe to write?
  WW++;                 // No. Active users exist
  okToWrite.wait(&lock); // Sleep on cond var
  WW--;                 // No longer waiting
}
AW++;
```

- Can't start because of readers, so go to sleep:
  AR = 2, WR = 0, AW = 0, WW = 1
- Finally, R3 comes along:
  AR = 2, WR = 1, AW = 0, WW = 1
- Now, say that R2 finishes before R1:
  AR = 1, WR = 1, AW = 0, WW = 1
- Finally, last of first two readers (R1) finishes and wakes up writer:

```
if (AR == 0 && WW > 0)  // No other active readers
  okToWrite.signal();   // Wake up one writer
```


Simulation(3)

- When writer wakes up, get:
  AR = 0, WR = 1, AW = 1, WW = 0
- Then, when writer finishes:

```
if (WW > 0){             // Give priority to writers
  okToWrite.signal();    // Wake up one writer
} else if (WR > 0) {     // Otherwise, wake reader
  okToRead.broadcast();  // Wake all readers
}
```

  - Writer wakes up reader, so get:

    AR = 1, WR = 0, AW = 0, WW = 0
- When reader completes, we are finished

Questions

- Can readers starve? Consider Reader() entry code:

```
while ((AW + WW) > 0) { // Is it safe to read?
  WR++;                 // No. Writers exist
  okToRead.wait(&lock); // Sleep on cond var
  WR--;                 // No longer waiting
}
AR++;                   // Now we are active!
```

- What if we erase the condition check in Reader exit?

```
AR--;                   // No longer active
if (AR == 0 && WW > 0)  // No other active readers
  okToWrite.signal();   // Wake up one writer
```

- Further, what if we turn the signal() into broadcast()

```
AR--;                   // No longer active
okToWrite.broadcast();  // Wake up one writer
```

- Finally, what if we use only one condition variable (call it "okToContinue") instead of two separate ones?
  - Both readers and writers sleep on this variable
  - Must use broadcast() instead of signal()

Can we construct Monitors from Semaphores?

- Locking aspect is easy: Just use a mutex
- Can we implement condition variables this way?

```
Wait()   { semaphore.P(); }
Signal() { semaphore.V(); }
```

- Does this work better?

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
Signal() { semaphore.V(); }
```


Construction of Monitors from Semaphores (con't)

- Problem with previous try:
  - P and V are commutative – result is the same no matter what order they occur
  - Condition variables are NOT commutative
- Does this fix the problem?

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
Signal() {
   if semaphore queue is not empty
      semaphore.V();
}
```

  - Not legal to look at contents of semaphore queue
  - There is a race condition – signaler can slip in after lock release and before waiter executes semaphore.P()
- It is actually possible to do this correctly
  - Complex solution for Hoare scheduling in book
  - Can you come up with simpler Mesa-scheduled solution?

Monitor Conclusion

- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed
- Basic structure of monitor-based program:

```
lock
while (need to wait) {
   condvar.wait();
}
unlock

do something so no need to wait

lock

condvar.signal();

unlock
```

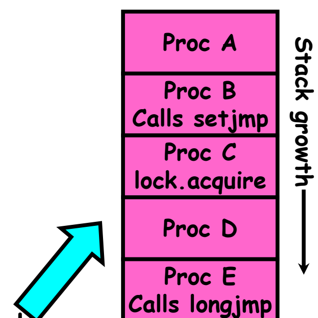
Check and/or update state variables Wait if necessary

Check and/or update state variables

C-Language Support for Synchronization

- C language: Pretty straightforward synchronization
  - Just make sure you know *all* the code paths out of a critical section

```
int Rtn() {
  lock.acquire();
  …
  if (exception) {
    lock.release();
    return errReturnCode;
  }
  …
  lock.release();
  return OK;
}
```



  - Watch out for setjmp/longjmp!
    - » Can cause a non-local jump out of procedure
    - » In example, procedure E calls longjmp, poping stack back to procedure B
    - » If Procedure C had lock.acquire, problem!

C++ Language Support for Synchronization

- Languages with exceptions like C++
  - Languages that support exceptions are problematic (easy to make a non-local exit without releasing lock)
  - Consider:

```
void Rtn() {
  lock.acquire();
  …
  DoFoo();
  …
  lock.release();
}
void DoFoo() {
  …
  if (exception) throw errException;
  …
}
```

  - Notice that an exception in DoFoo() will exit without releasing the lock

C++ Language Support for Synchronization (con't)

- Must catch all exceptions in critical sections
  - Catch exceptions, release lock, and re-throw exception:

```
void Rtn() {
  lock.acquire();
  try {
    …
    DoFoo();
    …
  } catch (…) {        // catch exception
    lock.release(); // release lock
    throw;          // re-throw the exception
  }
  lock.release();
}
void DoFoo() {
  …
  if (exception) throw errException;
  …
}
```

  - Even Better: auto_ptr<T> facility. See C++ Spec.
    - » Can deallocate/free lock regardless of exit method

Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization
- Bank Account example:

```
class Account {
  private int balance;
  // object constructor
  public Account (int initialBalance) {
    balance = initialBalance;
  }
  public synchronized int getBalance() {
    return balance;
  }
  public synchronized void deposit(int amount) {
    balance += amount;
  }
}
```

  - Every object has an associated lock which gets automatically acquired and released on entry and exit from a *synchronized* method.

Java Language Support for Synchronization (con't)

- Java also has *synchronized* statements:

```
synchronized (object) {
    …
}
```

  - Since every Java object has an associated lock, this type of statement acquires and releases the object's lock on entry and exit of the body
  - Works properly even with exceptions:

```
synchronized (object) {
  …
  DoFoo();
  …
}
void DoFoo() {
  throw errException;
}
```

Java Language Support for Synchronization (con't 2)

- In addition to a lock, every object has **a single** condition variable associated with it
  - How to wait inside a synchronization method of block:
    - » `void wait(long timeout); // Wait for timeout`
    - » `void wait(long timeout, int nanoseconds); //variant`
    - » `void wait();`
  - How to signal in a synchronized method or block:
    - » `void notify();    // wakes up oldest waiter`
    - » `void notifyAll(); // like broadcast, wakes everyone`
  - Condition variables can wait for a bounded length of time. This is useful for handling exception cases:

```
t1 = time.now();
while (!ATMRequest()) {
   wait (CHECKPERIOD);
   t2 = time.new();
   if (t2 – t1 > LONG_TIME) checkMachine();
}
```

  - Not all Java VMs equivalent!
    - » Different scheduling policies, not necessarily preemptive!

Summary

- Semaphores: Like integers with restricted interface
  - Two operations:
    - » `P()`: Wait if zero; decrement when becomes non-zero
    - » `V()`: Increment and wake a sleeping task (if exists)
    - » Can initialize value to any non-negative value
  - Use separate semaphore for each constraint
- Monitors: A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
    - » Three Operations: `Wait()`, `Signal()`, and `Broadcast()`
- Readers/Writers
  - Readers can access database when no writers
  - Writers can access database when no readers
  - Only one thread manipulates state variables at a time
- Language support for synchronization:
  - Java provides **synchronized** keyword and one condition-variable per object (with `wait()` and `notify()`)

CS162
Operating Systems and Systems Programming
Lecture 9

Tips for Working in a Project Team/ Cooperating Processes and Deadlock

September 26, 2007

Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

Review: Definition of Monitor

- Semaphores are confusing because dual purpose:
  - Both mutual exclusion and scheduling constraints
  - Cleaner idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
- **Monitor**: a lock and zero or more condition variables for managing concurrent access to shared data
  - Use of Monitors is a programming paradigm
- **Lock**: provides mutual exclusion to shared data:
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
- **Condition Variable**: a queue of threads waiting for something *inside* a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can't wait inside critical section

Review: Programming with Monitors

- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed
- Basic structure of monitor-based program:

```
lock
while (need to wait) {      } Check and/or update
   condvar.wait();            state variables
}                             Wait if necessary
unlock

do something so no need to wait

lock

condvar.signal();           } Check and/or update
                              state variables

unlock
```


Goals for Today

- Java Support for Monitors
- Tips for Programming in a Project Team
- Discussion of Deadlocks
  - Conditions for its occurrence
  - Solutions for breaking and avoiding deadlock

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization
- Bank Account example:

```
class Account {
  private int balance;
  // object constructor
  public Account (int initialBalance) {
    balance = initialBalance;
  }
  public synchronized int getBalance() {
    return balance;
  }
  public synchronized void deposit(int amount) {
    balance += amount;
  }
}
```

  - Every object has an associated lock which gets automatically acquired and released on entry and exit from a *synchronized* method.

Java Language Support for Synchronization (con't)

- Java also has *synchronized* statements:

```
synchronized (object) {
    …
}
```

  - Since every Java object has an associated lock, this type of statement acquires and releases the object's lock on entry and exit of the body
  - Works properly even with exceptions:

```
synchronized (object) {
  …
  DoFoo();
  …
}
void DoFoo() {
  throw errException;
}
```


Java Language Support for Synchronization (con't 2)

- In addition to a lock, every object has a single condition variable associated with it
  - How to wait inside a synchronization method of block:
    - » `void wait(long timeout); // Wait for timeout`
    - » `void wait(long timeout, int nanoseconds); //variant`
    - » `void wait();`
  - How to signal in a synchronized method or block:
    - » `void notify();    // wakes up oldest waiter`
    - » `void notifyAll(); // like broadcast, wakes everyone`
  - Condition variables can wait for a bounded length of time. This is useful for handling exception cases:

```
t1 = time.now();
while (!ATMRequest()) {
  wait (CHECKPERIOD);
  t2 = time.new();
  if (t2 – t1 > LONG_TIME) checkMachine();
}
```

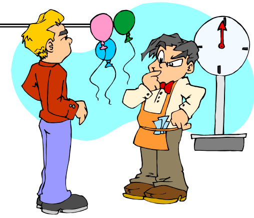
  - Not all Java VMs equivalent!
    - » Different scheduling policies, not necessarily preemptive!

Tips for Programming in a Project Team

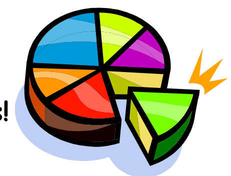
"You just have to get your synchronization right!"

- Big projects require more than one person (or long, long, long time)
  - Big OS: thousands of person-years!
- It's very hard to make software project teams work correctly
  - Doesn't seem to be as true of big construction projects
    - » Empire state building finished in one year: staging iron production thousands of miles away
    - » Or the Hoover dam: built towns to hold workers
  - Is it OK to miss deadlines?
    - » We make it free (slip days)
    - » Reality: they're very expensive as time-to-market is one of the most important things!

Big Projects



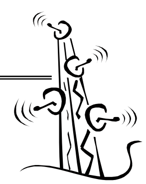
- What is a big project?
  - Time/work estimation is hard
  - Programmers are eternal optimistics (it will only take two days)!
    - » This is why we bug you about starting the project early
    - » Had a grad student who used to say he just needed "10 minutes" to fix something. Two hours later…
- Can a project be efficiently partitioned?
  - Partitionable task decreases in time as you add people
  - But, if you require communication:
    - » Time reaches a minimum bound
    - » With complex interactions, time increases!
  - Mythical person-month problem:
    - » You estimate how long a project will take
    - » Starts to fall behind, so you add more people
    - » Project takes even more time!



Techniques for Partitioning Tasks

- Functional
  - Person A implements threads, Person B implements semaphores, Person C implements locks…
  - Problem: Lots of communication across APIs
    - » If B changes the API, A may need to make changes
    - » Story: Large airline company spent $200 million on a new scheduling and booking system. Two teams "working together." After two years, went to merge software. Failed! Interfaces had changed (documented, but no one noticed). Result: would cost another $200 million to fix.
- Task
  - Person A designs, Person B writes code, Person C tests
  - May be difficult to find right balance, but can focus on each person's strengths (Theory vs systems hacker)
  - Since Debugging is hard, Microsoft has *two* testers for *each* programmer
- Most CS162 project teams are functional, but people have had success with task-based divisions

Communication



- More people mean more communication
  - Changes have to be propagated to more people
  - Think about person writing code for most fundamental component of system: everyone depends on them!
- Miscommunication is common
  - "Index starts at 0? I thought you said 1!"
- Who makes decisions?
  - Individual decisions are fast but trouble
  - Group decisions take time
  - Centralized decisions require a big picture view (someone who can be the "system architect")
- Often designating someone as the system architect can be a good thing
  - Better not be clueless
  - Better have good people skills
  - Better let other people do work

Coordination

- More people ⇒ no one can make all meetings!
  - They miss decisions and associated discussion
  - Example from earlier class: one person missed meetings and did something group had rejected
  - Why do we limit groups to 5 people?
    - » You would never be able to schedule meetings
  - Why do we require 4 people minimum?
    - » You need to experience groups to get ready for real world
- People have different work styles
  - Some people work in the morning, some at night
  - How do you decide when to meet or work together?
- What about project slippage?
  - It will happen, guaranteed!
  - Ex: phase 4, everyone busy but not talking. One person way behind. No one knew until very end – too late!
- Hard to add people to existing group
  - Members have already figured out how to work together

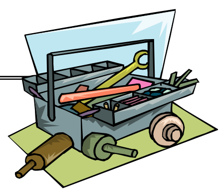
How to Make it Work?

- People are human. Get over it.
  - People will make mistakes, miss meetings, miss deadlines, etc. You need to live with it and adapt
  - It is better to anticipate problems than clean up afterwards.
- Document, document, document
  - Why Document?
    - » Expose decisions and communicate to others
    - » Easier to spot mistakes early
    - » Easier to estimate progress
  - What to document?
    - » Everything (but don't overwhelm people or no one will read)
  - Standardize!
    - » One programming format: variable naming conventions, tab indents, etc.
    - » Comments (Requires, effects, modifies)—javadoc?

Suggested Documents for You to Maintain

- Project objectives: goals, constraints, and priorities
- Specifications: the manual plus performance specs
  - This should be the first document generated and the last one finished
- Meeting notes
  - Document all decisions
  - You can often cut & paste for the design documents
- Schedule: What is your anticipated timing?
  - This document is critical!
- Organizational Chart
  - Who is responsible for what task?

Use Software Tools



- Source revision control software (CVS)
  - Easy to go back and see history
  - Figure out where and why a bug got introduced
  - Communicates changes to everyone (use CVS's features)
- Use automated testing tools
  - Write scripts for non-interactive software
  - Use "expect" for interactive software
  - Microsoft rebuilds the Longhorn/Vista kernel every night with the day's changes. Everyone is running/testing the latest software
- Use E-mail and instant messaging consistently to leave a history trail

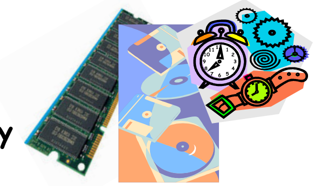
Test Continuously

- Integration tests all the time, not at 11pm on due date!
  - Write dummy stubs with simple functionality
    - » Let's people test continuously, but more work
  - Schedule periodic integration tests
    - » Get everyone in the same room, check out code, build, and test.
    - » Don't wait until it is too late!
- Testing types:
  - Unit tests: check each module in isolation (use JUnit?)
  - Daemons: subject code to exceptional cases
  - Random testing: Subject code to random timing changes
- Test early, test later, test again
  - Tendency is to test once and forget; what if something changes in some other part of the code?

Administrivia

Resources

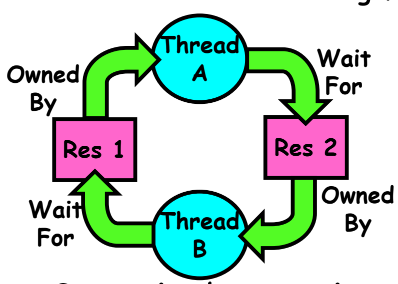
- Resources – passive entities needed by threads to do their work
  - CPU time, disk space, memory
- Two types of resources:
  - Preemptable – can take it away
    - » CPU, Embedded security chip
  - Non-preemptable – must leave it with the thread
    - » Disk space, plotter, chunk of virtual address space
    - » Mutual exclusion – the right to enter a critical section



- Resources may require exclusive access or may be sharable
  - Read-only files are typically sharable
  - Printers are not sharable during time of printing
- One of the major tasks of an operating system is to manage resources

Starvation vs Deadlock

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
    - » Example, low-priority thread waiting for resources constantly in use by high-priority threads
  - Deadlock: circular waiting for resources
    - » Thread A owns Res 1 and is waiting for Res 2
      Thread B owns Res 2 and is waiting for Res 1



  - Deadlock ⇒ Starvation but not vice versa
    - » Starvation can end (but doesn't have to)
    - » Deadlock can't end without external intervention

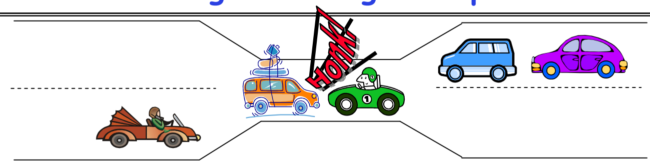
Conditions for Deadlock

- Deadlock not always deterministic – Example 2 mutexes:

Thread A	Thread B
`x.P();`	`y.P();`
`y.P();`	`x.P();`
`y.V();`	`x.V();`
`x.V();`	`y.V();`

  - Deadlock won't always happen with this code
    - » Have to have exactly the right timing ("wrong" timing?)
    - » So you release a piece of software, and you tested it, and there it is, controlling a nuclear power plant…
- Deadlocks occur with multiple resources
  - Means you can't decompose the problem
  - Can't solve deadlock for each resource independently
- Example: System with 2 disk drives and two threads
  - Each thread needs 2 disk drives to function
  - Each thread gets one disk and waits for another one

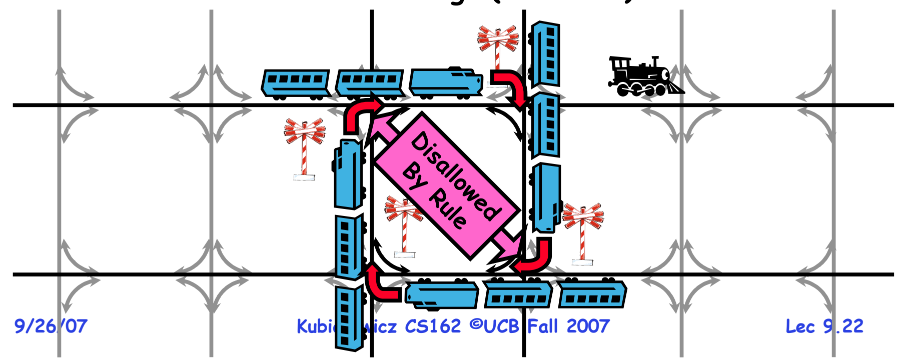
Bridge Crossing Example



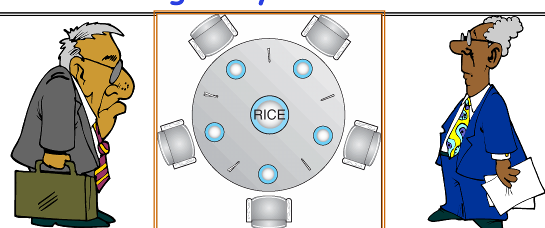
- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- For bridge: must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
  - Several cars may have to be backed up
- Starvation is possible
  - East-going traffic really fast ⇒ no one goes west

Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - » Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)



Dining Lawyers Problem

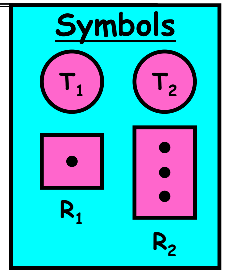


- Five chopsticks/Five lawyers (really cheap restaurant)
  - Free-for all: Lawyer will grab any one they can
  - Need two chopsticks to eat
- What if all grab at same time?
  - Deadlock!
- How to fix deadlock?
  - Make one of them give up a chopstick (Hah!)
  - Eventually everyone will get chance to eat
- How to prevent deadlock?
  - Never let lawyer take last chopstick if no hungry lawyer has two chopsticks afterwards

Four requirements for Deadlock

- Mutual exclusion
  - Only one thread at a time can use a resource.
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- Circular wait
  - There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads
    - » T_1 is waiting for a resource that is held by T_2
    - » T_2 is waiting for a resource that is held by T_3
    - » …
    - » T_n is waiting for a resource that is held by T_1

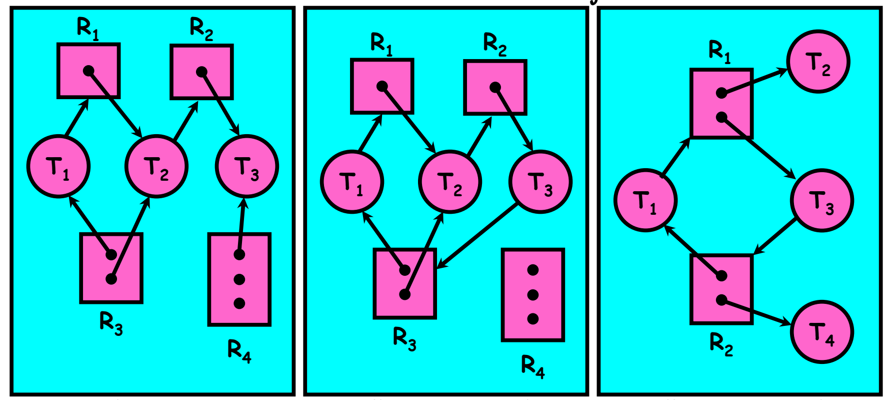
Resource-Allocation Graph



- System Model
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$
    CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances.
  - Each thread utilizes a resource as follows:
    - » `Request() / Use() / Release()`
- Resource-Allocation Graph:
  - V is partitioned into two types:
    - » $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    - » $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$

Resource Allocation Graph Examples

- Recall:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph | Allocation Graph With Deadlock | Allocation Graph With Cycle, but No Deadlock

Methods for Handling Deadlocks

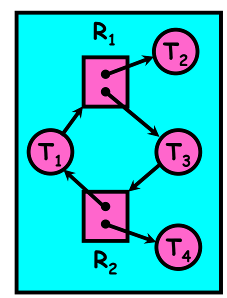
- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for forcibly preempting resources and/or terminating tasks
- Ensure that system will *never* enter a deadlock
  - Need to monitor all lock acquisitions
  - Selectively deny those that *might* lead to deadlock
- Ignore the problem and pretend that deadlocks never occur in the system
  - Used by most operating systems, including UNIX

Deadlock Detection Algorithm

- Only one of each type of resource ⇒ look for loops
- More General Deadlock Detection Algorithm
  - Let [X] represent an m-ary vector of non-negative integers (quantities of resources of each type):

    `[FreeResources]`: *Current free resources each type*
    `[Request`$_X$`]`: *Current requests from thread X*
    `[Alloc`$_X$`]`: *Current resources held by thread X*
  - See if tasks can eventually terminate on their own

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
   done = true
   Foreach node in UNFINISHED {
      if ([Request_node] <= [Avail]) {
         remove node from UNFINISHED
         [Avail] = [Avail] + [Alloc_node]
         done = false
      }
   }
} until(done)
```



  - Nodes left in UNFINISHED ⇒ deadlocked

What to do when detect deadlock?

- Terminate thread, force it to give up resources
  - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
  - Shoot a dining lawyer
  - But, not always possible – killing a thread holding a mutex leaves world inconsistent
- Preempt resources without killing off thread
  - Take away resources from thread temporarily
  - Doesn't always fit with semantics of computation
- Roll back actions of deadlocked threads
  - Hit the rewind button on TiVo, pretend last few minutes never happened
  - For bridge example, make one car roll backwards (may require others behind him)
  - Common technique in databases (transactions)
  - Of course, if you restart in exactly the same way, may reenter deadlock once again
- Many operating systems use other options

Summary

- Suggestions for dealing with Project Partners
  - Start Early, Meet Often
  - Develop Good Organizational Plan, Document Everything, Use the right tools, Develop Comprehensive Testing Plan
  - (Oh, and add 2 years to every deadline!)
- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources
- Four conditions for deadlocks
  - Mutual exclusion
    - » Only one thread at a time can use a resource
  - Hold and wait
    - » Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - » Resources are released only voluntarily by the threads
  - Circular wait
    - » $\exists$ set $\{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern

Summary (2)

- Techniques for addressing Deadlock
  - Allow system to enter deadlock and then recover
  - Ensure that system will *never* enter a deadlock
  - Ignore the problem and pretend that deadlocks never occur in the system
- Deadlock detection
  - Attempts to assess whether waiting graph can ever make progress
- Next Time: Deadlock prevention
  - Assess, for each allocation, whether it has the potential to lead to deadlock
  - Banker's algorithm gives one way to assess this

CS162
Operating Systems and Systems Programming
Lecture 10

Deadlock (cont'd)
Thread Scheduling

October 1, 2007

Prof. John Kubiatowicz

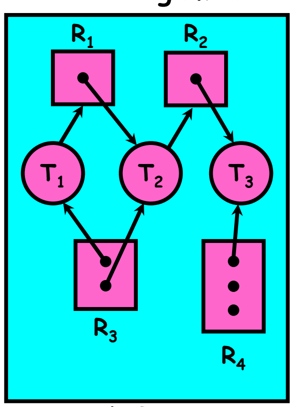
http://inst.eecs.berkeley.edu/~cs162

Review: Deadlock

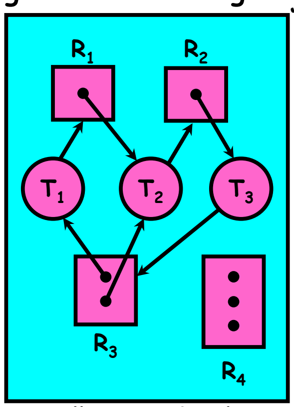
- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources
  - Deadlock⇒Starvation, but not other way around
- Four conditions for deadlocks
  - Mutual exclusion
    - » Only one thread at a time can use a resource
  - Hold and wait
    - » Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - » Resources are released only voluntarily by the threads
  - Circular wait
    - » There exists a set $\{T_1, \ldots, T_n\}$ of threads with a cyclic waiting pattern

Review: Resource Allocation Graph Examples

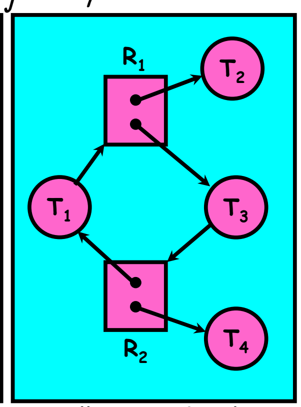
- Recall:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph



Allocation Graph With Deadlock



Allocation Graph With Cycle, but No Deadlock

Review: Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for selectively preempting resources and/or terminating tasks
- Ensure that system will *never* enter a deadlock
  - Need to monitor all lock acquisitions
  - Selectively deny those that *might* lead to deadlock
- Ignore the problem and pretend that deadlocks never occur in the system
  - used by most operating systems, including UNIX

Goals for Today

- Preventing Deadlock
- Scheduling Policy goals
- Policy Options
- Implementation Considerations

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

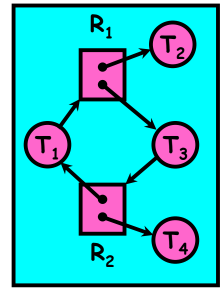
Deadlock Detection Algorithm

- Only one of each type of resource ⇒ look for loops
- More General Deadlock Detection Algorithm
  - Let [X] represent an m-ary vector of non-negative integers (quantities of resources of each type):

[FreeResources]:	Current free resources each type
$[Request_X]$:	Current requests from thread X
$[Alloc_X]$:	Current resources held by thread X

  - See if tasks can eventually terminate on their own

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
   done = true
   Foreach node in UNFINISHED {
      if ([Request_node] <= [Avail]) {
         remove node from UNFINISHED
         [Avail] = [Avail] + [Alloc_node]
         done = false
      }
   }
} until(done)
```



  - Nodes left in UNFINISHED ⇒ deadlocked

What to do when detect deadlock?

- Terminate thread, force it to give up resources
  - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
  - Shoot a dining lawyer
  - But, not always possible – killing a thread holding a mutex leaves world inconsistent
- Preempt resources without killing off thread
  - Take away resources from thread temporarily
  - Doesn't always fit with semantics of computation
- Roll back actions of deadlocked threads
  - Hit the rewind button on TiVo, pretend last few minutes never happened
  - For bridge example, make one car roll backwards (may require others behind him)
  - Common technique in databases (transactions)
  - Of course, if you restart in exactly the same way, may reenter deadlock once again
- Many operating systems use other options

Techniques for Preventing Deadlock

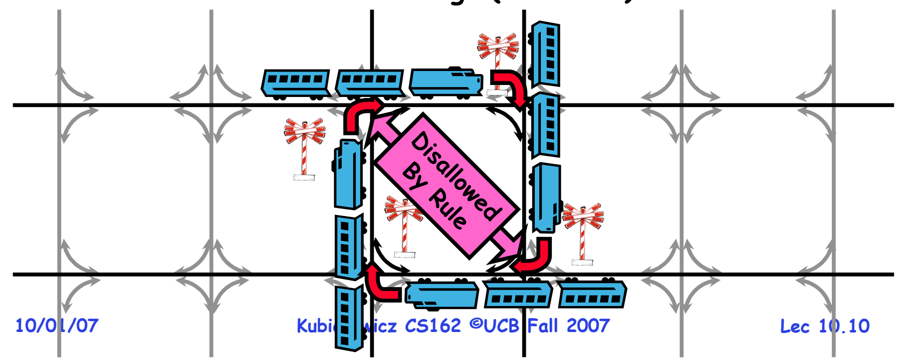
- Infinite resources
  - Include enough resources so that no one ever runs out of resources. Doesn't have to be infinite, just large
  - Give illusion of infinite resources (e.g. virtual memory)
  - Examples:
    - » Bay bridge with 12,000 lanes. Never wait!
    - » Infinite disk space (not realistic yet?)
- No Sharing of resources (totally independent threads)
  - Not very realistic
- Don't allow waiting
  - How the phone company avoids deadlock
    - » Call to your Mom in Toledo, works its way through the phone lines, but if blocked get busy signal.
  - Technique used in Ethernet/some multiprocessor nets
    - » Everyone speaks at once. On collision, back off and retry
  - Inefficient, since have to keep retrying
    - » Consider: driving to San Francisco; when hit traffic jam, suddenly you're transported back home and told to retry!

Techniques for Preventing Deadlock (con't)

- Make all threads request everything they'll need at the beginning.
  - Problem: Predicting future is hard, tend to over-estimate resources
  - Example:
    - » If need 2 chopsticks, request both at same time
    - » Don't leave home until we know no one is using any intersection between here and where you want to go; only one car on the Bay Bridge at a time
- Force all threads to request resources in a particular order preventing any cyclic use of resources
  - Thus, preventing deadlock
  - Example (x.P, y.P, z.P,…)
    - » Make tasks request disk, then memory, then…
    - » Keep from deadlock on freeways around SF by requiring everyone to go clockwise

Review: Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    - » Protocol: Always go east-west first, then north-south
  - Called "dimension ordering" (X then Y)

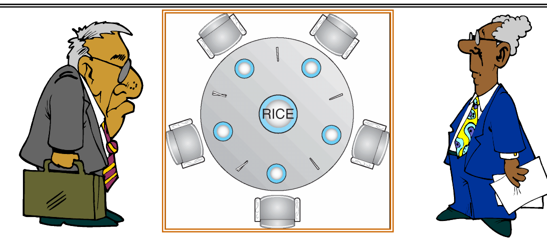


Banker's Algorithm for Preventing Deadlock

- Toward right idea:
  - State maximum resource needs in advance
  - Allow particular thread to proceed if:
    (available resources - #requested) ≥ max remaining that might be needed by any thread

- Banker's algorithm (less conservative):
  - Allocate resources dynamically
    - » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    - » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting ($[Max_{node}]-[Alloc_{node}] \le [Avail]$) for ($[Request_{node}] \le [Avail]$) Grant request if result is deadlock free (conservative!)
    - » Keeps system in a "SAFE" state, i.e. there exists a sequence $\{T_1, T_2, \ldots T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..
  - Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

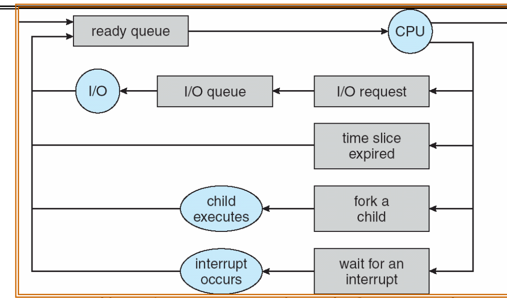
Banker's Algorithm Example



- Banker's algorithm with dining lawyers
  - "Safe" (won't cause deadlock) if when try to grab chopstick either:
    - » Not last chopstick
    - » Is last chopstick but someone will have two afterwards
  - What if k-handed lawyers? Don't allow if:
    - » It's the last one, no one would have k
    - » It's 2nd to last, and no one would have k-1
    - » It's 3rd to last, and no one would have k-2
    - » …

Administrivia

CPU Scheduling

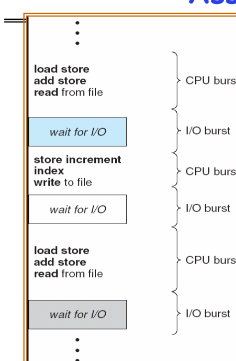


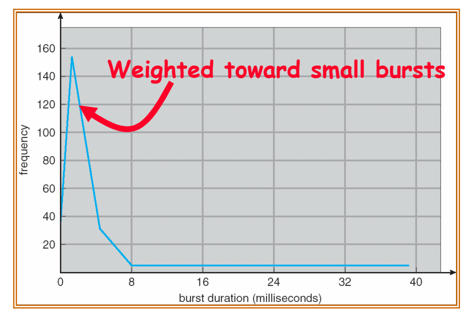
- Earlier, we talked about the life-cycle of a thread
  - Active threads work their way from Ready queue to Running to various waiting queues.
- Question: How is the OS to decide which of several tasks to take off a queue?
  - Obvious queue to worry about is ready queue
  - Others can be scheduled as well, however
- Scheduling: deciding which threads are given access to resources from moment to moment

Scheduling Assumptions

- CPU scheduling big area of research in early 70's
- Many implicit assumptions for CPU scheduling:
  - One program per user
  - One thread per program
  - Programs are independent
- Clearly, these are unrealistic but they simplify the problem so it can be solved
  - For instance: is "fair" about fairness among users or programs?
    - » If I run one compilation job and you run five, you get five times as much CPU on many operating systems
- The high-level goal: Dole out CPU time to optimize some desired parameters of system

Assumption: CPU Bursts



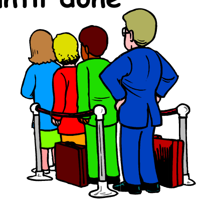


- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst

Scheduling Policy Goals/Criteria

- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    - » Time to echo a keystroke in editor
    - » Time to compile a program
    - » Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    - » Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    - » Minimize overhead (for example, context-switching)
    - » Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - » Better *average* response time by making system *less* fair

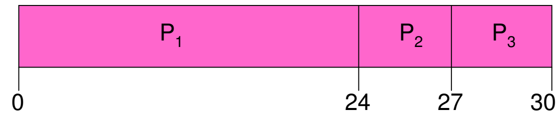
First-Come, First-Served (FCFS) Scheduling



- First-Come, First-Served (FCFS)
  - Also "First In, First Out" (FIFO) or "Run until done"
    - » In early systems, FCFS meant one program scheduled until done (including I/O)
    - » Now, means keep CPU until thread blocks
- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

  - Suppose processes arrive in the order: P_1 , P_2 , P_3 The Gantt Chart for the schedule is:



  - Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
  - Average waiting time: (0 + 24 + 27)/3 = 17
  - Average Completion time: (24 + 27 + 30)/3 = 27
- *Convoy effect:* short process behind long process

FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that processes arrive in order: P_2 , P_3 , P_1 Now, the Gantt chart for the schedule is:

  - Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
  - Average waiting time: (6 + 0 + 3)/3 = 3
  - Average Completion time: (3 + 6 + 30)/3 = 13
- In second case:
  - average waiting time is much better (before it was 17)
  - Average completion time is better (before it was 27)
- FIFO Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)
    - » Safeway: Getting milk, always stuck behind cart full of small items. Upside: get to read about space aliens!

Round Robin (RR)

- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you don't care who is behind you, on the other hand…
- Round Robin Scheme

  - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
  - After quantum expires, the process is preempted and added to the end of the ready queue.
  - *n* processes in ready queue and time quantum is $q \Rightarrow$
    - » Each process gets $1/n$ of the CPU time
    - » In chunks of at most q time units
    - » No process waits more than $(n-1)q$ time units
- Performance
  - q large $\Rightarrow$ FCFS
  - q small $\Rightarrow$ Interleaved (really small $\Rightarrow$ hyperthreading?)
  - q must be large with respect to context switch, otherwise overhead is too high (all overhead)

Example of RR with Time Quantum = 20

- **Example:**

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

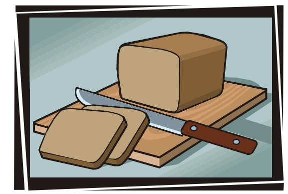
  - The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 28 48 68 88 108 112 125 145 153

  - Waiting time for $P_1=(68-20)+(112-88)=72$
    $P_2=(20-0)=20$
    $P_3=(28-0)+(88-48)+(125-108)=85$
    $P_4=(48-0)+(108-68)=88$
  - Average waiting time = $(72+20+85+88)/4=66\frac{1}{4}$
  - Average completion time = $(125+28+153+112)/4 = 104\frac{1}{2}$
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

Round-Robin Discussion



- How do you choose time slice?
  - What if too big?
    - » Response time suffers
  - What if infinite (∞)?
    - » Get back FIFO
  - What if time slice too small?
    - » Throughput suffers!
- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - » Worked ok when UNIX was used by one or two people.
    - » What if three compilations going on? 3 seconds to echo each keystroke!
  - In practice, need to balance short-job performance and long-job throughput:
    - » Typical time slice today is between 10ms – 100ms
    - » Typical context-switching overhead is 0.1ms – 1ms
    - » Roughly 1% overhead due to context-switching

Comparisons between FCFS and Round Robin

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example: 10 jobs, each take 100s of CPU time
  RR scheduler quantum of 1s
  All jobs start at the same time
- Completion Times:

Job #	FIFO	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

  - Both RR and FCFS finish at the same time
  - Average response time is much worse under RR!
    - » Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
  - Total time for RR longer even for zero-cost switch!

Earlier Example with Different Time Quantum

Best FCFS:

P_2 [8]	P_4 [24]	P_1 [53]	P_3 [68]

0 8 32 85 153

	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	$31\frac{1}{4}$
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	$61\frac{1}{4}$
	Q = 8	80	8	85	56	$57\frac{1}{4}$
	Q = 10	82	10	85	68	$61\frac{1}{4}$
	Q = 20	72	20	85	88	$66\frac{1}{4}$
	Worst FCFS	68	145	0	121	$83\frac{1}{2}$
Completion Time	Best FCFS	85	8	153	32	$69\frac{1}{2}$
	Q = 1	137	30	153	81	$100\frac{1}{2}$
	Q = 5	135	28	153	82	$99\frac{1}{2}$
	Q = 8	133	16	153	80	$95\frac{1}{2}$
	Q = 10	135	18	153	92	$99\frac{1}{2}$
	Q = 20	125	28	153	112	$104\frac{1}{2}$
	Worst FCFS	121	153	68	145	$121\frac{3}{4}$

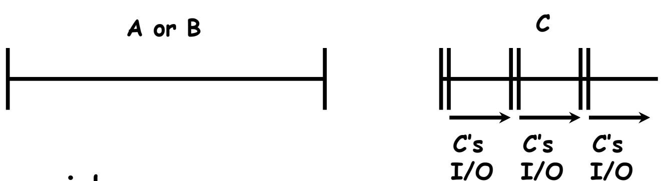
What if we Knew the Future?

- Could we always mirror best FCFS?
- Shortest Job First (SJF):
  - Run whatever job has the least amount of computation to do
  - Sometimes called "Shortest Time to Completion First" (STCF)
- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  - Sometimes called "Shortest Remaining Time to Completion First" (SRTCF)
- These can be applied either to a whole program or the current CPU burst of each program
  - Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result is better average response time

Discussion

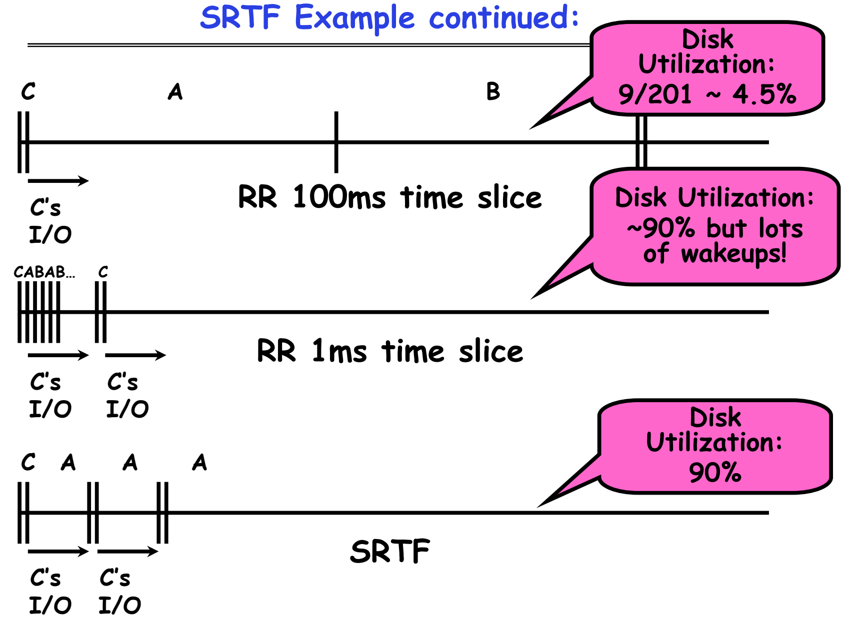
- SJF/SRTF are the best you can do at minimizing average response time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    - » SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    - » SRTF (and RR): short jobs not stuck behind long ones

Example to illustrate benefits of SRTF



- Three jobs:
  - A,B: both CPU bound, run for week
    C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FIFO:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline

SRTF Example continued:

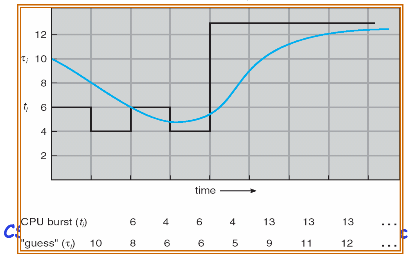


SRTF Further discussion

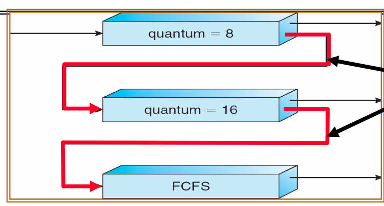
- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    - » When you submit a job, have to say how long it will take
    - » To stop cheating, system kills job if takes too long
  - But: Even non-malicious users have trouble predicting runtime of their jobs
- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal, so can't do any better
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)

Predicting the Length of the Next CPU Burst

- Adaptive: Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, etc
  - Works because programs have predictable behavior
    - » If program was I/O bound in past, likely in future
    - » If computer behavior were random, wouldn't help
- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts: Let t_{n-1}, t_{n-2}, t_{n-3}, etc. be previous CPU burst lengths. Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots)$
  - Function f could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance, exponential averaging $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$ with $(0<\alpha\leq1)$



Multi-Level Feedback Scheduling



Long-Running Compute Tasks Demoted to Low Priority

- Another method for exploiting past behavior
  - First used in CTSS
  - Multiple queues, each with different priority
    - » Higher priority queues often considered "foreground" tasks
  - Each queue has its own scheduling algorithm
    - » e.g. foreground – RR, background – FCFS
    - » Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc)
- Adjust each job's priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire, push up one level (or to top)

Scheduling Details

- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
  - Fixed priority scheduling:
    - » serve all from highest priority, then next priority, etc.
  - Time slice:
    - » each queue gets a certain amount of CPU time
    - » e.g., 70% to highest, 20% next, 10% lowest
- Countermeasure: user action that can foil intent of the OS designer
  - For multilevel feedback, put in a bunch of meaningless I/O to keep job's priority high
  - Of course, if everyone did this, wouldn't work!
- Example of Othello program:
  - Playing against competitor, so key was to do computing at higher priority the competitors.
    - » Put in printf's, ran much faster!

What about Fairness?

- What about fairness?
  - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    - » long running jobs may never get CPU
    - » In Multics, shut down machine, found 10-year-old job
  - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
  - Tradeoff: fairness gained by hurting avg response time!
- How to implement fairness?
  - Could give each queue some fraction of the CPU
    - » What if one long-running job and 100 short-running ones?
    - » Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
  - Could increase priority of jobs that don't get service
    - » What is done in UNIX
    - » This is ad hoc—what rate should you increase priorities?
    - » And, as system gets overloaded, no job gets CPU time, so everyone increases in priority ⇒ Interactive jobs suffer

Lottery Scheduling

- Yet another alternative: Lottery Scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
  - To approximate SRTF, short running jobs get more, long running jobs get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

Lottery Scheduling Example

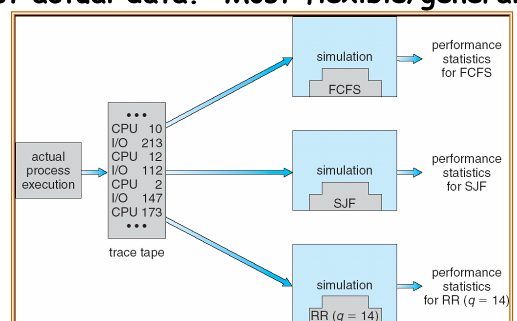
- Lottery Scheduling Example
  - Assume short jobs get 10 tickets, long jobs get 1 ticket

# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

  - What if too many short jobs to give reasonable response time?
    - » In UNIX, if load average is 100, hard to make progress
    - » One approach: log some user out

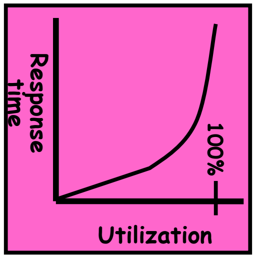
How to Evaluate a Scheduling algorithm?

- Deterministic modeling
  - takes a predetermined workload and compute the performance of each algorithm for that workload
- Queuing models
  - Mathematical approach for handling stochastic workloads
- Implementation/Simulation:
  - Build system which allows actual algorithms to be run against actual data. Most flexible/general.



A Final Word on Scheduling

- When do the details of the scheduling policy and fairness really matter?
  - When there aren't enough resources to go around
- When should you simply buy a faster computer?
  - (Or network link, or expanded highway, or …)
  - One approach: Buy it when it will pay for itself in improved response time
    - » Assuming you're paying for worse response time in reduced productivity, customer angst, etc…
    - » Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization⇒100%



- An interesting implication of this curve:
  - Most scheduling algorithms work fine in the "linear" portion of the load curve, fail otherwise
  - Argues for buying a faster X when hit "knee" of curve

Summary (Deadlock)

- Four conditions required for deadlocks
  - Mutual exclusion
    - » Only one thread at a time can use a resource
  - Hold and wait
    - » Thread holding at least one resource is waiting to acquire additional resources held by other threads
  - No preemption
    - » Resources are released only voluntarily by the threads
  - Circular wait
    - » ∃ set $\{T_1, …, T_n\}$ of threads with a cyclic waiting pattern
- Deadlock detection
  - Attempts to assess whether waiting graph can ever make progress
- Deadlock prevention
  - Assess, for each allocation, whether it has the potential to lead to deadlock
  - Banker's algorithm gives one way to assess this

Summary (Scheduling)

- Scheduling: selecting a waiting process from the ready queue and allocating the CPU to it
- FCFS Scheduling:
  - Run threads to completion in order of submission
  - Pros: Simple
  - Cons: Short jobs get stuck behind long ones
- Round-Robin Scheduling:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs
  - Cons: Poor when jobs are same length

Summary (Scheduling 2)

- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair
- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
- Lottery Scheduling:
  - Give each thread a priority-dependent number of tokens (short tasks ⇒ more tokens)
  - Reserve a minimum number of tokens for every thread to ensure forward progress/fairness

CS162
Operating Systems and Systems Programming
Lecture 11

Thread Scheduling (con't)
Protection: Address Spaces

October 3, 2007

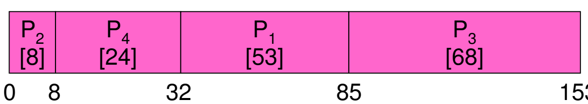
Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

Review: Last Time

- **Scheduling**: selecting a waiting process from the ready queue and allocating the CPU to it
- **FCFS Scheduling**:
  - Run threads to completion in order of submission
  - Pros: Simple (+)
  - Cons: Short jobs get stuck behind long ones (-)
- **Round-Robin Scheduling**:
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
  - Pros: Better for short jobs (+)
  - Cons: Poor when jobs are same length (-)

Review: Example with Different Time Quantum

Best FCFS:



	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	$31\frac{1}{4}$
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	$61\frac{1}{4}$
	Q = 8	80	8	85	56	$57\frac{1}{4}$
	Q = 10	82	10	85	68	$61\frac{1}{4}$
	Q = 20	72	20	85	88	$66\frac{1}{4}$
	Worst FCFS	68	145	0	121	$83\frac{1}{2}$
Completion Time	Best FCFS	85	8	153	32	$69\frac{1}{2}$
	Q = 1	137	30	153	81	$100\frac{1}{2}$
	Q = 5	135	28	153	82	$99\frac{1}{2}$
	Q = 8	133	16	153	80	$95\frac{1}{2}$
	Q = 10	135	18	153	92	$99\frac{1}{2}$
	Q = 20	125	28	153	112	$104\frac{1}{2}$
	Worst FCFS	121	153	68	145	$121\frac{3}{4}$

Review: What if we Knew the Future?

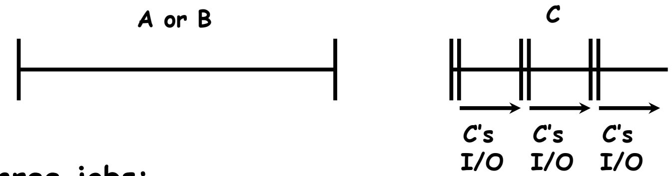
- **Shortest Job First (SJF):**
  - Run whatever job has the least computation to do
- **Shortest Remaining Time First (SRTF):**
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
- SJF/SRTF are the best you can do at minimizing average response time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - SRTF always at least as good as SJF ⇒ focus on SRTF

Goals for Today

- Finish discussion of Scheduling
- Kernel vs User Mode
- What is an Address Space?
- How is it Implemented?

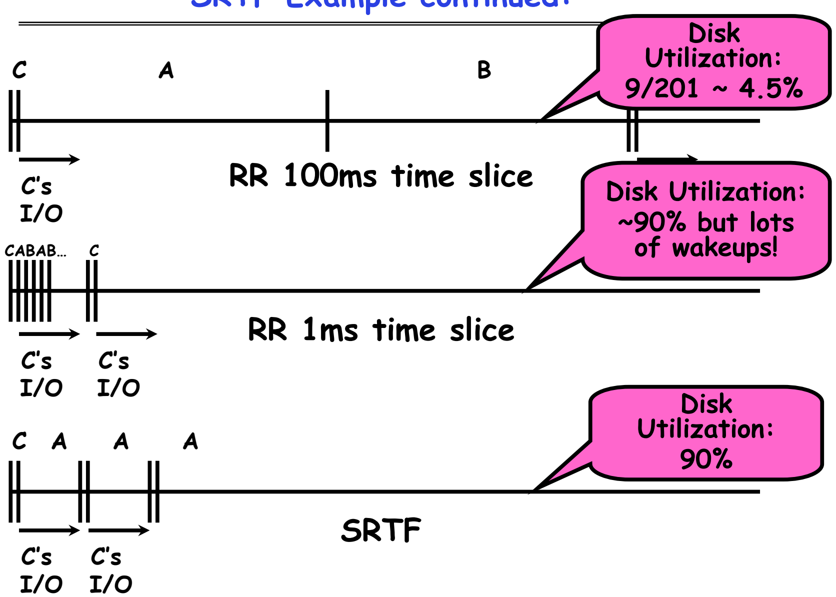
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

Example to illustrate benefits of SRTF



- Three jobs:
  - A,B: both CPU bound, run for week
    C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FIFO:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline

SRTF Example continued:

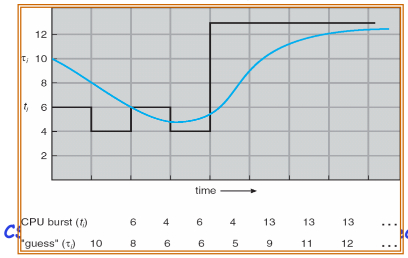


SRTF Further discussion

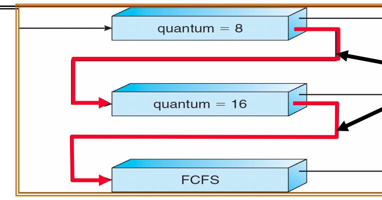
- Starvation
  - SRTF can lead to starvation if many small jobs!
  - Large jobs never get to run
- Somehow need to predict future
  - How can we do this?
  - Some systems ask the user
    - » When you submit a job, have to say how long it will take
    - » To stop cheating, system kills job if takes too long
  - But: Even non-malicious users have trouble predicting runtime of their jobs
- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal, so can't do any better
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)

Predicting the Length of the Next CPU Burst

- **Adaptive**: Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, etc
  - Works because programs have predictable behavior
    - » If program was I/O bound in past, likely in future
    - » If computer behavior were random, wouldn't help
- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts: Let t_{n-1}, t_{n-2}, t_{n-3}, etc. be previous CPU burst lengths. Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots)$
  - Function f could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance, exponential averaging $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$ with $(0<\alpha\leq1)$



Multi-Level Feedback Scheduling



Long-Running Compute Tasks Demoted to Low Priority

- Another method for exploiting past behavior
  - First used in CTSS
  - Multiple queues, each with different priority
    - » Higher priority queues often considered "foreground" tasks
  - Each queue has its own scheduling algorithm
    - » e.g. foreground – RR, background – FCFS
    - » Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc)
- Adjust each job's priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire, push up one level (or to top)

Scheduling Details

- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
  - Fixed priority scheduling:
    - » serve all from highest priority, then next priority, etc.
  - Time slice:
    - » each queue gets a certain amount of CPU time
    - » e.g., 70% to highest, 20% next, 10% lowest
- **Countermeasure**: user action that can foil intent of the OS designer
  - For multilevel feedback, put in a bunch of meaningless I/O to keep job's priority high
  - Of course, if everyone did this, wouldn't work!
- Example of Othello program:
  - Playing against competitor, so key was to do computing at higher priority the competitors.
    - » Put in printf's, ran much faster!

What about Fairness?

- What about fairness?
  - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    - » long running jobs may never get CPU
    - » In Multics, shut down machine, found 10-year-old job
  - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
  - Tradeoff: fairness gained by hurting avg response time!
- How to implement fairness?
  - Could give each queue some fraction of the CPU
    - » What if one long-running job and 100 short-running ones?
    - » Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
  - Could increase priority of jobs that don't get service
    - » What is done in UNIX
    - » This is ad hoc—what rate should you increase priorities?
    - » And, as system gets overloaded, no job gets CPU time, so everyone increases in priority⇒Interactive jobs suffer

Lottery Scheduling

- Yet another alternative: Lottery Scheduling
  - Give each job some number of lottery tickets
  - On each time slice, randomly pick a winning ticket
  - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
  - To approximate SRTF, short running jobs get more, long running jobs get fewer
  - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

Lottery Scheduling Example

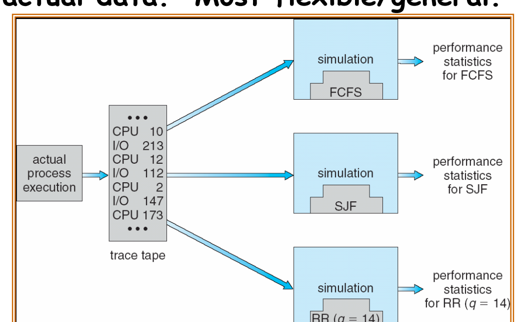
- Lottery Scheduling Example
  - Assume short jobs get 10 tickets, long jobs get 1 ticket

# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

  - What if too many short jobs to give reasonable response time?
    - » In UNIX, if load average is 100, hard to make progress
    - » One approach: log some user out

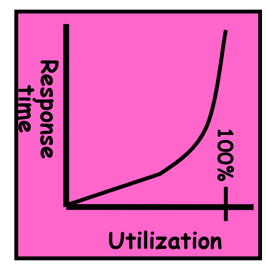
How to Evaluate a Scheduling algorithm?

- Deterministic modeling
  - takes a predetermined workload and compute the performance of each algorithm for that workload
- Queueing models
  - Mathematical approach for handling stochastic workloads
- Implementation/Simulation:
  - Build system which allows actual algorithms to be run against actual data. Most flexible/general.



A Final Word On Scheduling

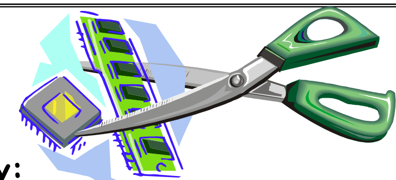
- When do the details of the scheduling policy and fairness really matter?
  - When there aren't enough resources to go around
- When should you simply buy a faster computer?
  - (Or network link, or expanded highway, or …)
  - One approach: Buy it when it will pay for itself in improved response time
    - » Assuming you're paying for worse response time in reduced productivity, customer angst, etc…
    - » Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization⇒100%



- An interesting implication of this curve:
  - Most scheduling algorithms work fine in the "linear" portion of the load curve, fail otherwise
  - Argues for buying a faster X when hit "knee" of curve

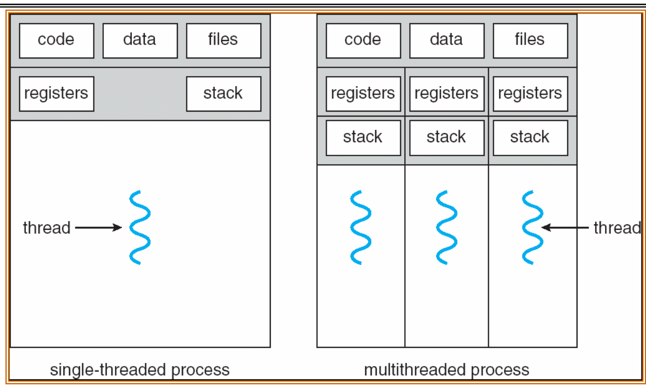
Administrivia

Virtualizing Resources



- Physical Reality:
  Different Processes/Threads share the same hardware
  - Need to multiplex CPU (Just finished: scheduling)
  - Need to multiplex use of Memory (Today)
  - Need to multiplex disk and devices (later in term)
- Why worry about memory sharing?
  - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  - Consequently, cannot just let different threads of control use the same memory
    - » Physics: two different pieces of data cannot occupy the same locations in memory
  - Probably don't want different threads to even have access to each other's memory (protection)

Recall: Single and Multithreaded Processes



- Threads encapsulate concurrency
  - "Active" component of a process
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
  - "Passive" component of a process

Important Aspects of Memory Multiplexing

- **Controlled overlap:**
  - Separate state of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
  - Conversely, would like the ability to overlap when desired (for communication)
- **Translation:**
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
  - Side effects:
    - » Can be used to avoid overlap
    - » Can be used to give uniform view of memory to programs
- **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
    - » Programs protected from themselves

Binding of Instructions and Data to Memory

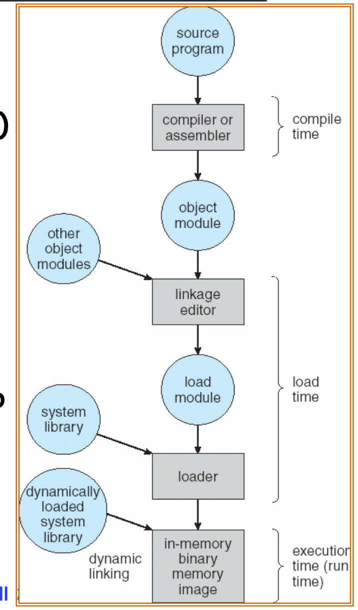
- Binding of instructions and data to addresses:
  - Choose addresses for instructions and data from the standpoint of the processor

```
data1:  dw    32                      0x300  00000020
              …                         …
start:  lw    r1,0(data1)             0x900  8C2000C0
        jal   checkit                 0x904  0C000340
loop:   addi  r1, r1, -1              0x908  2021FFFF
        bnz   r1, r0, loop            0x90C  1420FFFF
              …                         …
checkit: …                            0xD00  …
```

  - Could we place data1, start, and/or checkit at different addresses?
    - » Yes
    - » When? Compile time/Load time/Execution time
  - Related: which physical memory locations hold particular instructions or data?

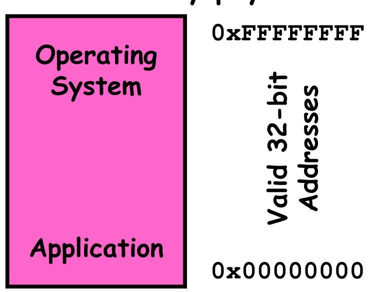
Multi-step Processing of a Program for Execution

- 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 the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes routine



Recall: Uniprogramming

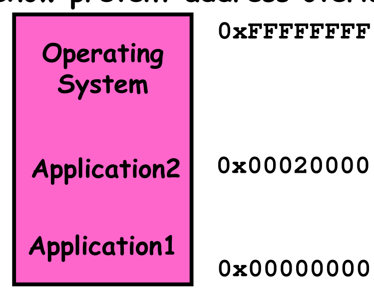
- Uniprogramming (no Translation or Protection)
  - Application always runs at same place in physical memory since only one application at a time
  - Application can access any physical address



  - Application given illusion of dedicated machine by giving it reality of a dedicated machine
- Of course, this doesn't help us with multithreading

Multiprogramming (First Version)

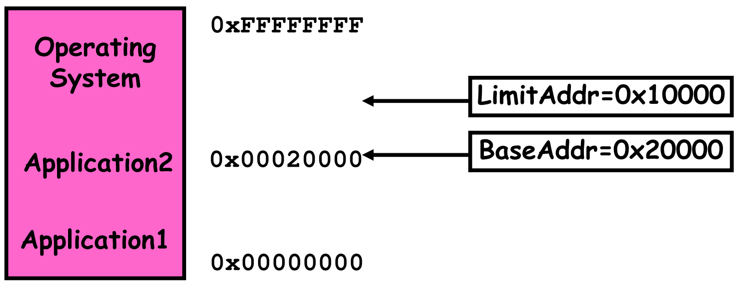
- Multiprogramming without Translation or Protection
  - Must somehow prevent address overlap between threads



  - Trick: Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
    - » Everything adjusted to memory location of program
    - » Translation done by a linker-loader
    - » Was pretty common in early days
- With this solution, no protection: bugs in any program can cause other programs to crash or even the OS

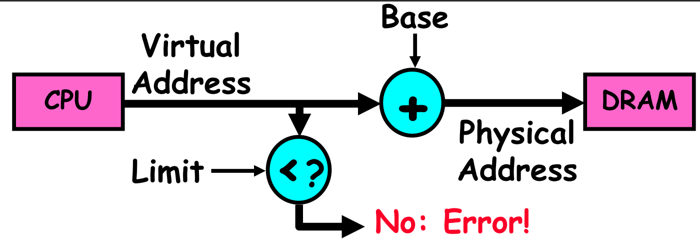
Multiprogramming (Version with Protection)

- Can we protect programs from each other without translation?



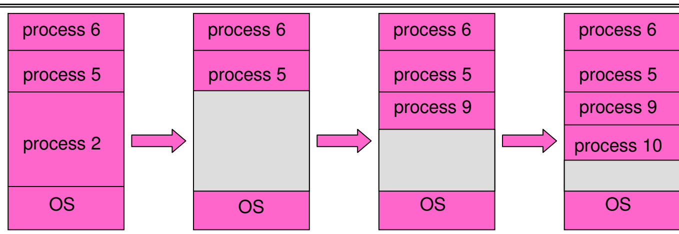
  - Yes: use two special registers *BaseAddr* and *LimitAddr* to prevent user from straying outside designated area
    - » If user tries to access an illegal address, cause an error
  - During switch, kernel loads new base/limit from TCB
    - » User not allowed to change base/limit registers

Segmentation with Base and Limit registers



- Could use base/limit for dynamic address translation (often called "segmentation"):
  - Alter address of every load/store by adding "base"
  - User allowed to read/write within segment
    - » Accesses are relative to segment so don't have to be relocated when program moved to different segment
  - User may have multiple segments available (e.g x86)
    - » Loads and stores include segment ID in opcode: x86 Example: `mov [es:bx],ax.`
    - » Operating system moves around segment base pointers as necessary

Issues with simple segmentation method

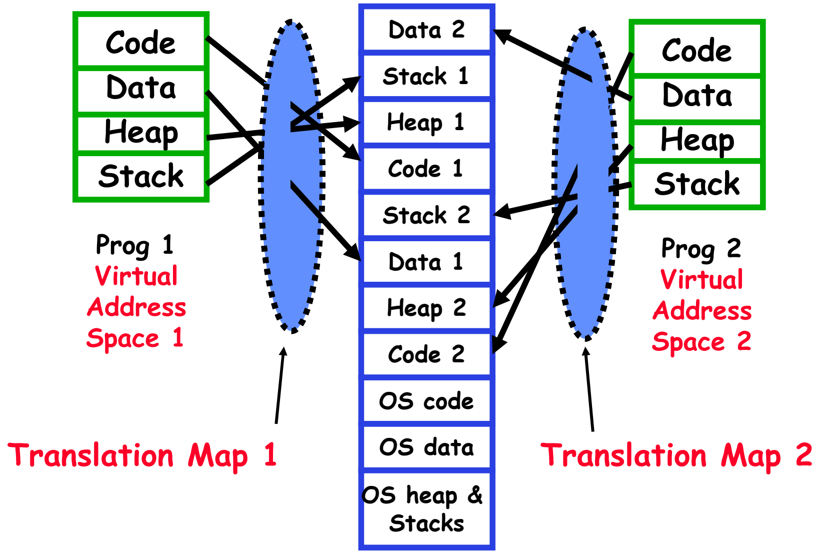


- Fragmentation problem
  - Not every process is the same size
  - Over time, memory space becomes fragmented
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by by providing multiple segments per process
- Need enough physical memory for every process

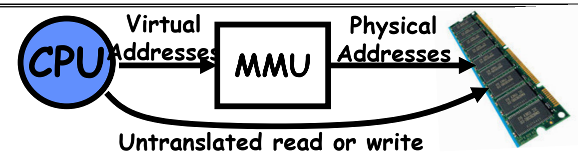
Multiprogramming (Translation and Protection version 2)

- Problem: Run multiple applications in such a way that they are protected from one another
- Goals:
  - Isolate processes and kernel from one another
  - Allow flexible translation that:
    - » Doesn't lead to fragmentation
    - » Allows easy sharing between processes
    - » Allows only part of process to be resident in physical memory
- (Some of the required) Hardware Mechanisms:
  - General Address Translation
    - » Flexible: Can fit physical chunks of memory into arbitrary places in users address space
    - » Not limited to small number of segments
    - » Think of this as providing a large number (thousands) of fixed-sized segments (called "pages")
  - Dual Mode Operation
    - » Protection base involving kernel/user distinction

Example of General Address Translation



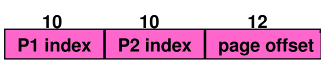
Two Views of Memory

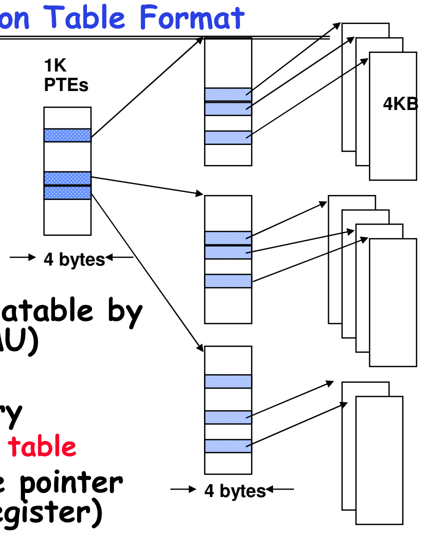


- Recall: Address Space:
  - All the addresses and state a process can touch
  - Each process and kernel has different address space
- Consequently: two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View fom memory (physical memory)
  - Translation box converts between the two views
- Translation helps to implement protection
  - If task A cannot even gain access to task B's data, no way for A to adversely affect B
- With translation, every program can be linked/loaded into same region of user address space
  - Overlap avoided through translation, not relocation

Example of Translation Table Format

Two-level Page Tables
32-bit address:





- Page: a unit of memory translatable by memory management unit (MMU)
  - Typically 1K – 8K
- Page table structure in memory
  - Each user has different page table
- Address Space switch: change pointer to base of table (hardware register)
  - Hardware traverses page table (for many architectures)
  - MIPS uses software to traverse table

Dual-Mode Operation

- Can Application Modify its own translation tables?
  - If it could, could get access to all of physical memory
  - Has to be restricted somehow
- To Assist with Protection, Hardware provides at least two modes (Dual-Mode Operation):
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode (Normal program mode)
  - Mode set with bits in special control register only accessible in kernel-mode
- Intel processor actually has four "rings" of protection:
  - PL (Priviledge Level) from 0 – 3
    - » PL0 has full access, PL3 has least
  - Privilege Level set in code segment descriptor (CS)
  - Mirrored "IOPL" bits in condition register gives permission to programs to use the I/O instructions
  - Typical OS kernels on Intel processors only use PL0 ("user") and PL3 ("kernel")

For Protection, Lock User-Programs in Asylum

- Idea: Lock user programs in padded cell with no exit or sharp objects
  - Cannot change mode to kernel mode
  - User cannot modify page table mapping
  - Limited access to memory: cannot adversely effect other processes
    - » Side-effect: Limited access to memory-mapped I/O operations (I/O that occurs by reading/writing memory locations)
  - Limited access to interrupt controller
  - What else needs to be protected?

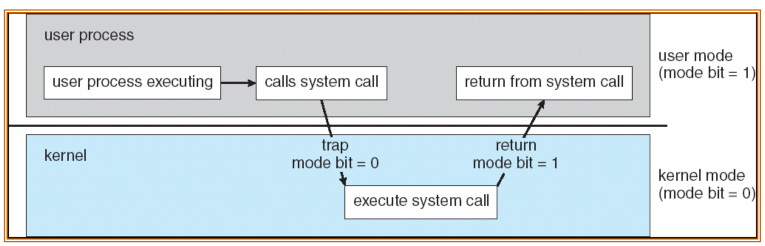
- A couple of issues
  - How to share CPU between kernel and user programs?
    - » Kinda like both the inmates and the warden in asylum are the same person. How do you manage this???
  - How do programs interact?
  - How does one switch between kernel and user modes?
    - » OS → user (kernel → user mode): getting into cell
    - » User→ OS (user → kernel mode): getting out of cell

How to get from Kernel→User

- What does the kernel do to create a new user process?
  - Allocate and initialize address-space control block
  - Read program off disk and store in memory
  - Allocate and initialize translation table
    - » Point at code in memory so program can execute
    - » Possibly point at statically initialized data
  - Run Program:
    - » Set machine registers
    - » Set hardware pointer to translation table
    - » Set processor status word for user mode
    - » Jump to start of program
- How does kernel switch between processes?
  - Same saving/restoring of registers as before
  - Save/restore PSL (hardware pointer to translation table)

User→Kernel (System Call)

- Can't let inmate (user) get out of padded cell on own
  - Would defeat purpose of protection!
  - So, how does the user program get back into kernel?



- System call: Voluntary procedure call into kernel
  - Hardware for controlled User→Kernel transition
  - Can any kernel routine be called?
    - » No! Only specific ones.
  - System call ID encoded into system call instruction
    - » Index forces well-defined interface with kernel

System Call Continued

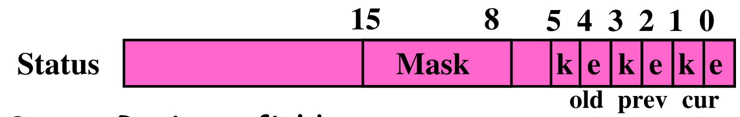
- What are some system calls?
  - I/O: open, close, read, write, lseek
  - Files: delete, mkdir, rmdir, truncate, chown, chgrp, ..
  - Process: fork, exit, wait (like join)
  - Network: socket create, set options
- Are system calls constant across operating systems?
  - Not entirely, but there are lots of commonalities
  - Also some standardization attempts (POSIX)
- What happens at beginning of system call?
    - » On entry to kernel, sets system to kernel mode
    - » Handler address fetched from table/Handler started
- System Call argument passing:
  - In registers (not very much can be passed)
  - Write into user memory, kernel copies into kernel mem
    - » User addresses must be translated!w
    - » Kernel has different view of memory than user
  - Every Argument must be explicitly checked!

User→Kernel (Exceptions: Traps and Interrupts)

- A system call instruction causes a synchronous exception (or "trap")
  - In fact, often called a software "trap" instruction
- Other sources of *Synchronous Exceptions:*
  - Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
  - Segmentation Fault (address out of range)
  - Page Fault (for illusion of infinite-sized memory)
- Interrupts are *Asynchronous Exceptions*
  - Examples: timer, disk ready, network, etc….
  - Interrupts can be disabled, traps cannot!
- On system call, exception, or interrupt:
  - Hardware enters kernel mode with interrupts disabled
  - Saves PC, then jumps to appropriate handler in kernel
  - For some processors (x86), processor also saves registers, changes stack, etc.
- Actual handler typically saves registers, other CPU state, and switches to kernel stack

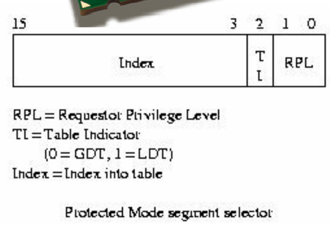
Additions to MIPS ISA to support Exceptions?

- Exception state is kept in "Coprocessor 0"
  - Use mfc0 read contents of these registers:
    - » BadVAddr (register 8): contains memory address at which memory reference error occurred
    - » Status (register 12): interrupt mask and enable bits
    - » Cause (register 13): the cause of the exception
    - » EPC (register 14): address of the affected instruction



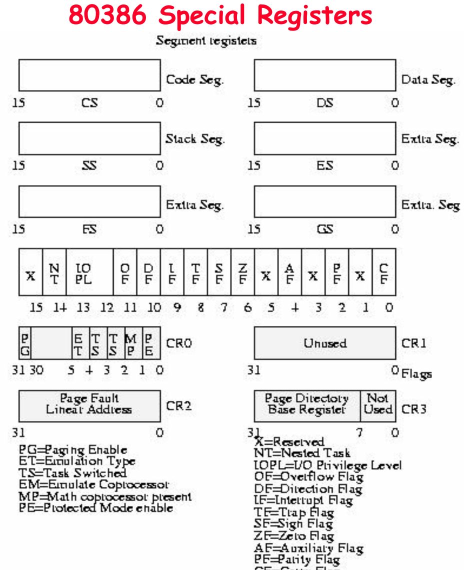
- Status Register fields:
  - Mask: Interrupt enable
    - » 1 bit for each of 5 hardware and 3 software interrupts
  - k = kernel/user: 0⇒kernel mode
  - e = interrupt enable: 0⇒interrupts disabled
  - Exception⇒6 LSB shifted left 2 bits, setting 2 LSB to 0:
    - » run in kernel mode with interrupts disabled

Intel x86 Special Registers



Typical Segment Register
Current Priority is RPL
Of Code Segment (CS)

80386 Special Registers



Communication

- Now that we have isolated processes, how can they communicate?
  - Shared memory: common mapping to physical page
    - » As long as place objects in shared memory address range, threads from each process can communicate
    - » Note that processes A and B can talk to shared memory through different addresses
    - » In some sense, this violates the whole notion of protection that we have been developing
  - If address spaces don't share memory, all inter-address space communication must go through kernel (via system calls)
    - » Byte stream producer/consumer (put/get): Example, communicate through pipes connecting stdin/stdout
    - » Message passing (send/receive): Will explain later how you can use this to build remote procedure call (RPC) abstraction so that you can have one program make procedure calls to another
    - » File System (read/write): File system is shared state!

Closing thought: Protection without Hardware

- Does protection require hardware support for translation and dual-mode behavior?
  - No: Normally use hardware, but anything you can do in hardware can also do in software (possibly expensive)
- Protection via Strong Typing
  - Restrict programming language so that you can't express program that would trash another program
  - Loader needs to make sure that program produced by valid compiler or all bets are off
  - Example languages: LISP, Ada, Modula-3 and Java
- Protection via software fault isolation:
  - Language independent approach: have compiler generate object code that provably can't step out of bounds
    - » Compiler puts in checks for every "dangerous" operation (loads, stores, etc). Again, need special loader.
    - » Alternative, compiler generates "proof" that code cannot do certain things (Proof Carrying Code)
  - Or: use virtual machine to guarantee safe behavior (loads and stores recompiled on fly to check bounds)

Summary

- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
  - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
  - Pros: Optimal (average response time)
  - Cons: Hard to predict future, Unfair
- Multi-Level Feedback Scheduling:
  - Multiple queues of different priorities
  - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
- Lottery Scheduling:
  - Give each thread a priority-dependent number of tokens (short tasks⇒more tokens)
  - Reserve a minimum number of tokens for every thread to ensure forward progress/fairness
- Evaluation of mechanisms:
  - Analytical, Queuing Theory, Simulation

Summary (2)

- Memory is a resource that must be shared
  - Controlled Overlap: only shared when appropriate
  - Translation: Change Virtual Addresses into Physical Addresses
  - Protection: Prevent unauthorized Sharing of resources
- Simple Protection through Segmentation
  - Base+limit registers restrict memory accessible to user
  - Can be used to translate as well
- Full translation of addresses through Memory Management Unit (MMU)
  - Every Access translated through page table
  - Changing of page tables only available to user
- Dual-Mode
  - Kernel/User distinction: User restricted
  - User→Kernel: System calls, Traps, or Interrupts
  - Inter-process communication: shared memory, or through kernel (system calls)

CS162
Operating Systems and Systems Programming
Lecture 12

Protection (continued)
Address Translation

October 8, 2007

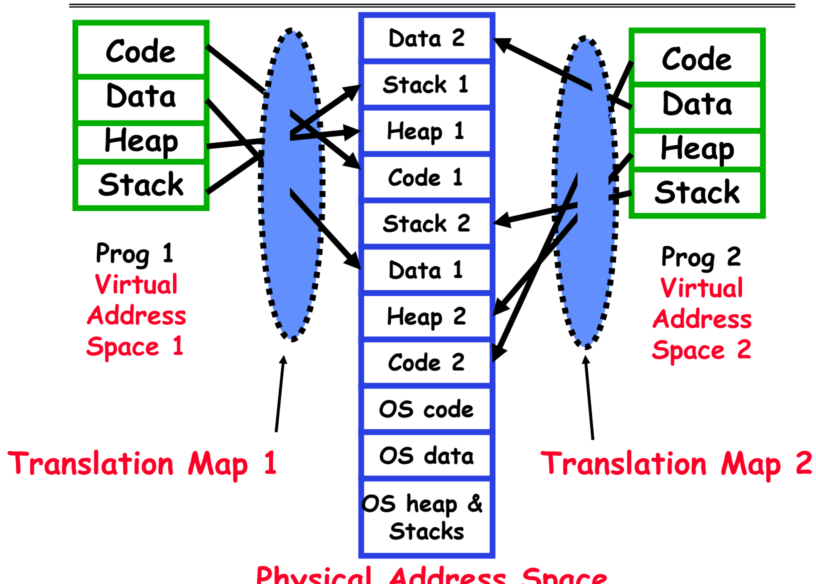
Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

Review: Important Aspects of Memory Multiplexing

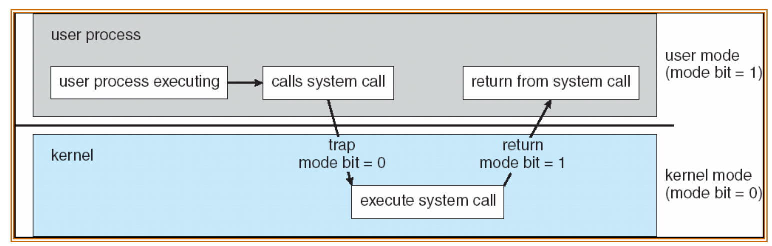
- Controlled overlap:
  - Separate state of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
  - Conversely, would like the ability to overlap when desired (for communication)
- Translation:
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
  - Side effects:
    - » Can be used to avoid overlap
    - » Can be used to give uniform view of memory to programs
- 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
    - » Programs protected from themselves

Review: General Address Translation



Review: User→Kernel (System Call)

- Can't let inmate (user) get out of padded cell on own
  - Would defeat purpose of protection!
  - So, how does the user program get back into kernel?



- System call: Voluntary procedure call into kernel
  - Hardware for controlled User→Kernel transition
  - Can any kernel routine be called?
    - » No! Only specific ones.
  - System call ID encoded into system call instruction
    - » Index forces well-defined interface with kernel

Goals for Today

- Finish discussion of protection
- Address Translation Schemes
  - Segmentation
  - Paging
  - Multi-level translation
  - Paged page tables
  - Inverted page tables
- Comparison among options

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

System Call Continued

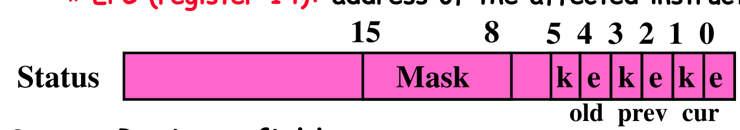
- What are some system calls?
  - I/O: open, close, read, write, lseek
  - Files: delete, mkdir, rmdir, truncate, chown, chgrp, ..
  - Process: fork, exit, wait (like join)
  - Network: socket create, set options
- Are system calls constant across operating systems?
  - Not entirely, but there are lots of commonalities
  - Also some standardization attempts (POSIX)
- What happens at beginning of system call?
  - » On entry to kernel, sets system to kernel mode
  - » Handler address fetched from table/Handler started
- System Call argument passing:
  - In registers (not very much can be passed)
  - Write into user memory, kernel copies into kernel mem
    - » User addresses must be translated!w
    - » Kernel has different view of memory than user
  - Every Argument must be explicitly checked!

User→Kernel (Exceptions: Traps and Interrupts)

- A system call instruction causes a synchronous exception (or "trap")
  - In fact, often called a software "trap" instruction
- Other sources of *Synchronous Exceptions:*
  - Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
  - Segmentation Fault (address out of range)
  - Page Fault (for illusion of infinite-sized memory)
- Interrupts are *Asynchronous Exceptions*
  - Examples: timer, disk ready, network, etc….
  - Interrupts can be disabled, traps cannot!
- On system call, exception, or interrupt:
  - Hardware enters kernel mode with interrupts disabled
  - Saves PC, then jumps to appropriate handler in kernel
  - For some processors (x86), processor also saves registers, changes stack, etc.
- Actual handler typically saves registers, other CPU state, and switches to kernel stack

Additions to MIPS ISA to support Exceptions?

- Exception state is kept in "Coprocessor 0"
  - Use mfc0 read contents of these registers:
    - » BadVAddr (register 8): contains memory address at which memory reference error occurred
    - » Status (register 12): interrupt mask and enable bits
    - » Cause (register 13): the cause of the exception
    - » EPC (register 14): address of the affected instruction



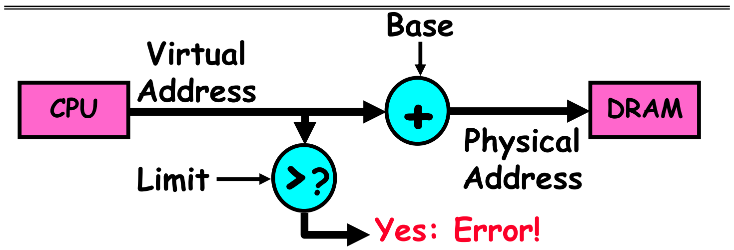
- Status Register fields:
  - Mask: Interrupt enable
    - » 1 bit for each of 5 hardware and 3 software interrupts
  - k = kernel/user: 0⇒kernel mode
  - e = interrupt enable: 0⇒interrupts disabled
  - Exception⇒6 LSB shifted left 2 bits, setting 2 LSB to 0:
    - » run in kernel mode with interrupts disabled

Communication

- Now that we have isolated processes, how can they communicate?
  - Shared memory: common mapping to physical page
    - » As long as place objects in shared memory address range, threads from each process can communicate
    - » Note that processes A and B can talk to shared memory through different addresses
    - » In some sense, this violates the whole notion of protection that we have been developing
  - If address spaces don't share memory, all inter-address space communication must go through kernel (via system calls)
    - » Byte stream producer/consumer (put/get): Example, communicate through pipes connecting `stdin/stdout`
    - » Message passing (send/receive): Will explain later how you can use this to build remote procedure call (RPC) abstraction so that you can have one program make procedure calls to another
    - » File System (read/write): File system is shared state!

Administrivia

Simple Segmentation: Base and Bounds (CRAY-1)



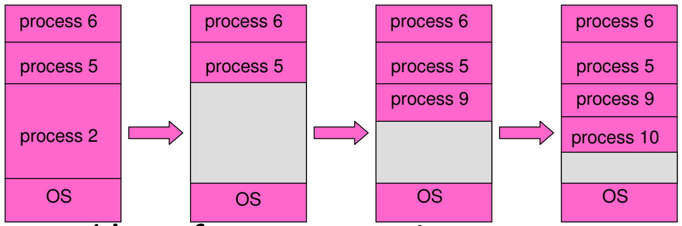
- Can use base & bounds/limit for dynamic address translation (Simple form of "segmentation"):
  - Alter every address by adding "base"
  - Generate error if address bigger than limit
- This gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
  - Program gets continuous region of memory
  - Addresses within program do not have to be relocated when program placed in different region of DRAM

Base and Limit segmentation discussion

- Provides level of indirection
  - OS can move bits around behind program's back
  - Can be used to correct if program needs to grow beyond its bounds or coalesce fragments of memory
- Only OS gets to change the base and limit!
  - Would defeat protection
- What gets saved/restored on a context switch?
  - Everything from before + base/limit values
  - Or: How about complete contents of memory (out to disk)?
    - » Called "Swapping"
- Hardware cost
  - 2 registers/Adder/Comparator
  - Slows down hardware because need to take time to do add/compare on every access
- Base and Limit Pros: Simple, relatively fast

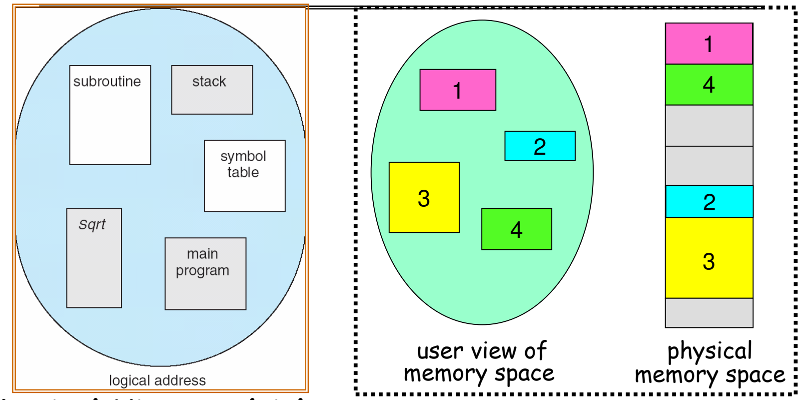
Cons for Simple Segmentation Method

- Fragmentation problem (complex memory allocation)
  - Not every process is the same size
  - Over time, memory space becomes fragmented
  - Really bad if want space to grow dynamically (e.g. heap)



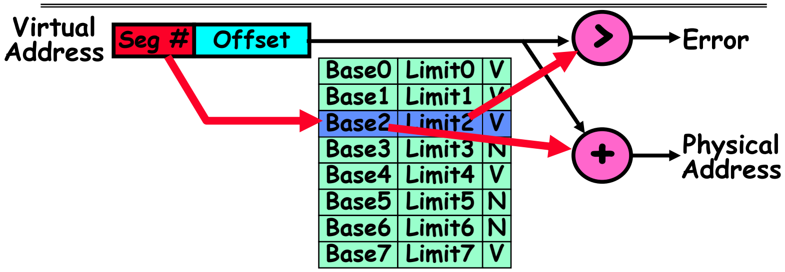
- Other problems for process maintenance
  - Doesn't allow heap and stack to grow independently
  - Want to put these as far apart in virtual memory space as possible so that they can grow as needed
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes

More Flexible Segmentation



- Logical View: multiple separate segments
  - Typical: Code, Data, Stack
  - Others: memory sharing, etc
- Each segment is given region of contiguous memory
  - Has a base and limit
  - Can reside anywhere in physical memory

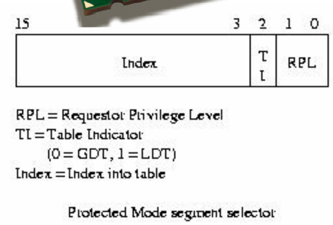
Implementation of Multi-Segment Model

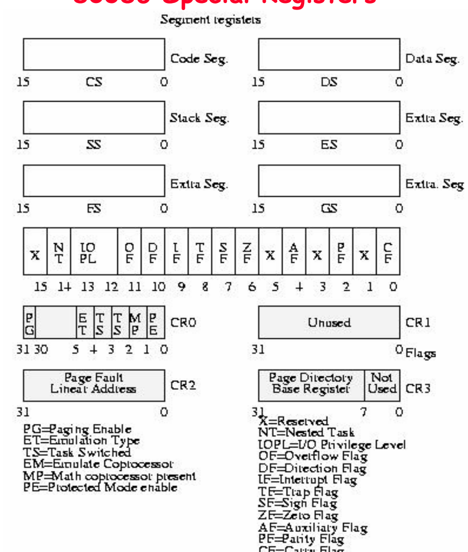


- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - » x86 Example: `mov [es:bx],ax.`
- What is "V/N"?
  - Can mark segments as invalid; requires check as well

Intel x86 Special Registers

80386 Special Registers

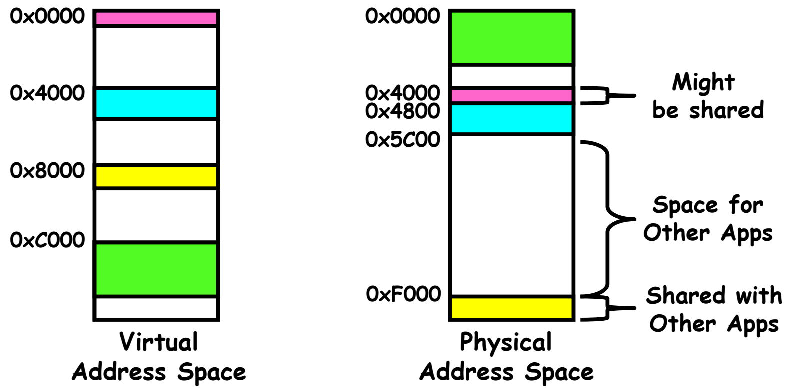




Typical Segment Register
Current Priority is RPL
Of Code Segment (CS)

Example: Four Segments (16 bit addresses)

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000



Example of segment translation

```
0x240   main:   la $a0, varx
0x244           jal strlen
 …               …
0x360   strlen: li   $v0, 0  ;count
0x364   loop:   lb   $t0, ($a0)
0x368           beq  $r0,$t1, done
 …               …
0x4050  varx    dw   0x314159
```

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

Let's simulate a bit of this code to see what happens (PC=0x240):

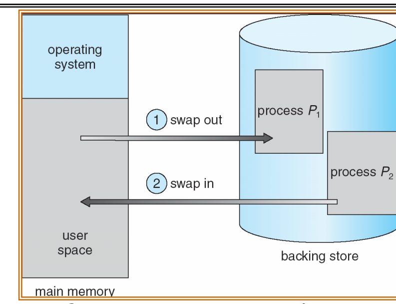
- Fetch 0x240. Virtual segment #? 0; Offset? 0x240
  Physical address? Base=0x4000, so physical addr=0x4240
  Fetch instruction at 0x4240. Get "la $a0, varx"
  Move 0x4050 → $a0, Move PC+4→PC

2. Fetch 0x244. Translated to Physical=0x4244. Get "jal strlen"
   Move 0x0248 → $ra (return address!), Move 0x0360 → PC
3. Fetch 0x360. Translated to Physical=0x4360. Get "li $v0,0"
   Move 0x0000 → $v0, Move PC+4→PC
4. Fetch 0x364. Translated to Physical=0x4364. Get "lb $t0,($a0)"
   Since $a0 is 0x4050, try to load byte from 0x4050
   Translate 0x4050. Virtual segment #? 1; Offset? 0x50
   Physical address? Base=0x4800, Physical addr = 0x4850,
   Load Byte from 0x4850→$t0, Move PC+4→PC

Observations about Segmentation

- Virtual address space has holes
  - Segmentation efficient for sparse address spaces
  - A correct program should never address gaps (except as mentioned in moment)
    - » If it does, trap to kernel and dump core
- When it is OK to address outside valid range:
  - This is how the stack and heap are allowed to grow
  - For instance, stack takes fault, system automatically increases size of stack
- Need protection mode in segment table
  - For example, code segment would be read-only
  - Data and stack would be read-write (stores allowed)
  - Shared segment could be read-only or read-write
- What must be saved/restored on context switch?
  - Segment table stored in CPU, not in memory (small)
  - Might store all of processes memory onto disk when switched (called "swapping")

Schematic View of Swapping

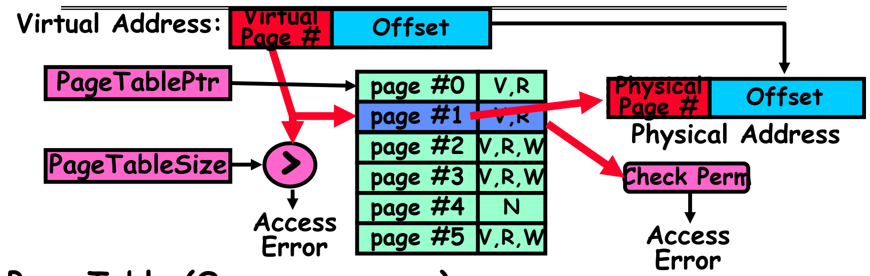


- Extreme form of Context Switch: Swapping
  - In order to make room for next process, some or all of the previous process is moved to disk
    - » Likely need to send out complete segments
  - This greatly increases the cost of context-switching
- Desirable alternative?
  - Some way to keep only active portions of a process in memory at any one time
  - Need finer granularity control over physical memory

Paging: Physical Memory in Fixed Size Chunks

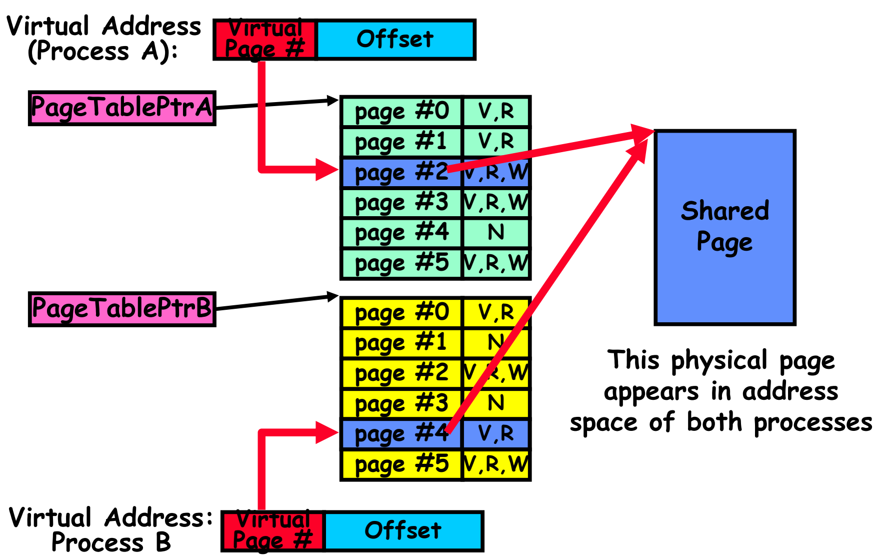
- Problems with segmentation?
  - Must fit variable-sized chunks into physical memory
  - May move processes multiple times to fit everything
  - Limited options for swapping to disk
- **Fragmentation**: wasted space
  - **External**: free gaps between allocated chunks
  - **Internal**: don't need all memory within allocated chunks
- Solution to fragmentation from segments?
  - Allocate physical memory in fixed size chunks ("pages")
  - Every chunk of physical memory is equivalent
    - » Can use simple vector of bits to handle allocation: 00110001110001101 … 110010
    - » Each bit represents page of physical memory 1⇒allocated, 0⇒free
- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    - » Typically have small pages (1K-16K)
  - Consequently: need multiple pages/segment

How to Implement Paging?

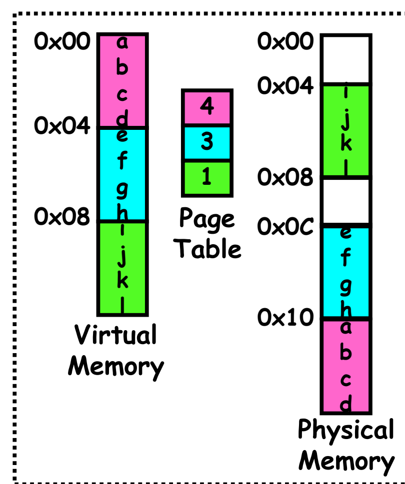


- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    - » Permissions include: Valid bits, Read, Write, etc
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - » Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    - » Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

What about Sharing?



Simple Page Table Discussion

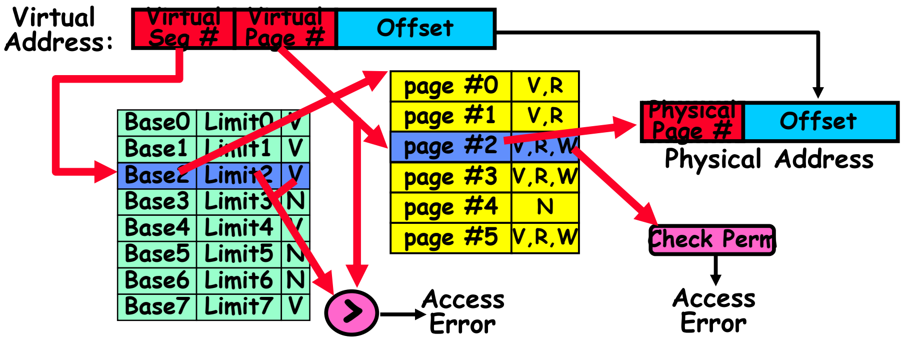


Example (4 byte pages)

- What needs to be switched on a context switch?
  - Page table pointer and limit
- Analysis
  - Pros
    - » Simple memory allocation
    - » Easy to Share
  - Con: What if address space is sparse?
    - » E.g. on UNIX, code starts at 0, stack starts at (2^{31}-1).
    - » With 1K pages, need 4 million page table entries!
  - Con: What if table really big?
    - » Not all pages used all the time ⇒ would be nice to have working set of page table in memory
- How about combining paging and segmentation?

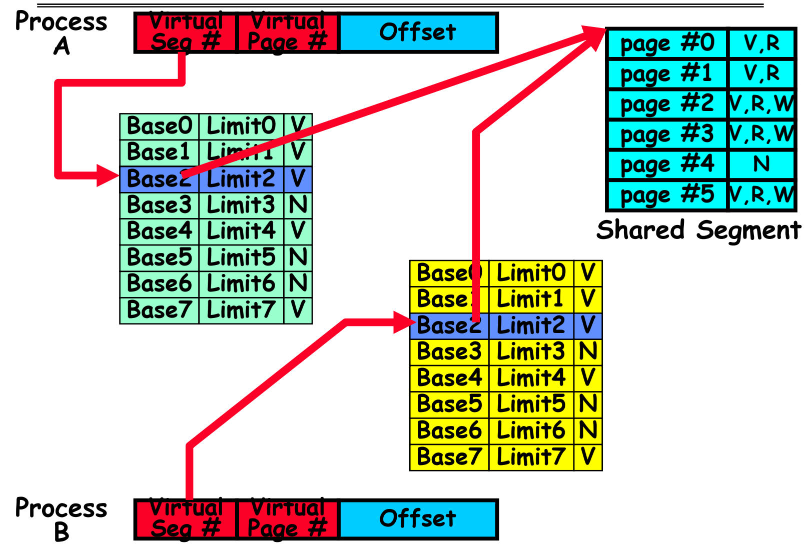
Multi-level Translation

- What about a tree of tables?
  - Lowest level page table⇒memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):

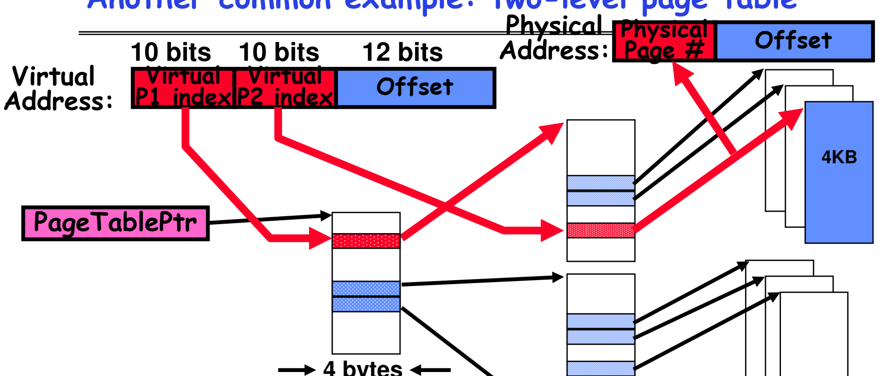


- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

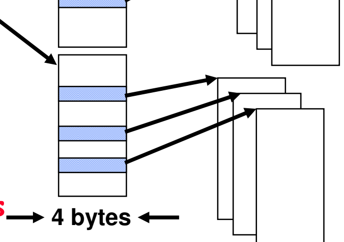
What about Sharing (Complete Segment)?



Another common example: two-level page table



- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don't need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

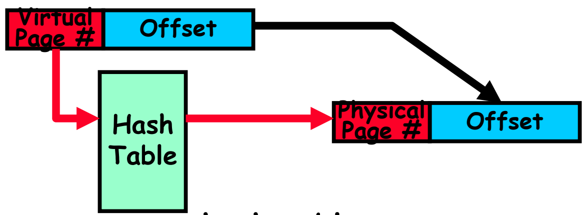


Multi-level Translation Analysis

- Pros:
  - Only need to allocate as many page table entries as we need for application
    - » In other wards, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - » Share at segment or page level (need additional reference counting)
- Cons:
  - One pointer per page (typically 4K – 16K pages today)
  - Page tables need to be contiguous
    - » However, previous example keeps tables to exactly one page in size
  - Two (or more, if >2 levels) lookups per reference
    - » Seems very expensive!

Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    - » Much of process space may be out on disk or not in use



- Answer: use a hash table
  - Called an "Inverted Page Table"
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
- Cons: Complexity of managing hash changes
  - Often in hardware!

Closing thought: Protection without Hardware

- Does protection require hardware support for translation and dual-mode behavior?
  - No: Normally use hardware, but anything you can do in hardware can also do in software (possibly expensive)
- Protection via Strong Typing
  - Restrict programming language so that you can't express program that would trash another program
  - Loader needs to make sure that program produced by valid compiler or all bets are off
  - Example languages: LISP, Ada, Modula-3 and Java
- Protection via software fault isolation:
  - Language independent approach: have compiler generate object code that provably can't step out of bounds
    - » Compiler puts in checks for every "dangerous" operation (loads, stores, etc). Again, need special loader.
    - » Alternative, compiler generates "proof" that code cannot do certain things (Proof Carrying Code)
  - Or: use virtual machine to guarantee safe behavior (loads and stores recompiled on fly to check bounds)

Summary (1/2)

- Memory is a resource that must be shared
  - Controlled Overlap: only shared when appropriate
  - Translation: Change Virtual Addresses into Physical Addresses
  - Protection: Prevent unauthorized Sharing of resources
- Dual-Mode
  - Kernel/User distinction: User restricted
  - User→Kernel: System calls, Traps, or Interrupts
  - Inter-process communication: shared memory, or through kernel (system calls)
- Exceptions
  - Synchronous Exceptions: Traps (including system calls)
  - Asynchronous Exceptions: Interrupts

Summary (2/2)

- Segment Mapping
  - Segment registers within processor
  - Segment ID associated with each access
    - » Often comes from portion of virtual address
    - » Can come from bits in instruction instead (x86)
  - Each segment contains base and limit information
    - » Offset (rest of address) adjusted by adding base
- Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Virtual page number from virtual address mapped through page table to physical page number
  - Offset of virtual address same as physical address
  - Large page tables can be placed into virtual memory
- Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space
- Inverted page table
  - Size of page table related to physical memory size

CS162
Operating Systems and Systems Programming
Lecture 13

Address Translation (con't)
Caches and TLBs

October 15, 2007

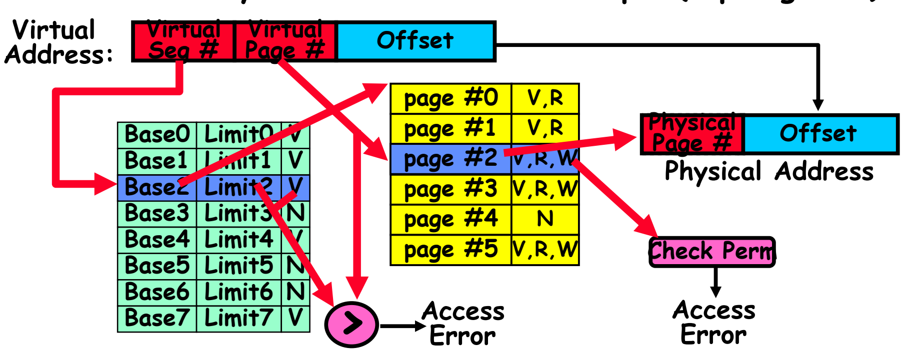
Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

Review: Exceptions: Traps and Interrupts

- A system call instruction causes a synchronous exception (or "trap")
  - In fact, often called a software "trap" instruction
- Other sources of synchronous exceptions:
  - Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
  - Segmentation Fault (address out of range)
  - Page Fault (for illusion of infinite-sized memory)
- Interrupts are Asynchronous Exceptions
  - Examples: timer, disk ready, network, etc….
  - Interrupts can be disabled, traps cannot!
- On system call, exception, or interrupt:
  - Hardware enters kernel mode with interrupts disabled
  - Saves PC, then jumps to appropriate handler in kernel
  - For some processors (x86), processor also saves registers, changes stack, etc.
- Actual handler typically saves registers, other CPU state, and switches to kernel stack

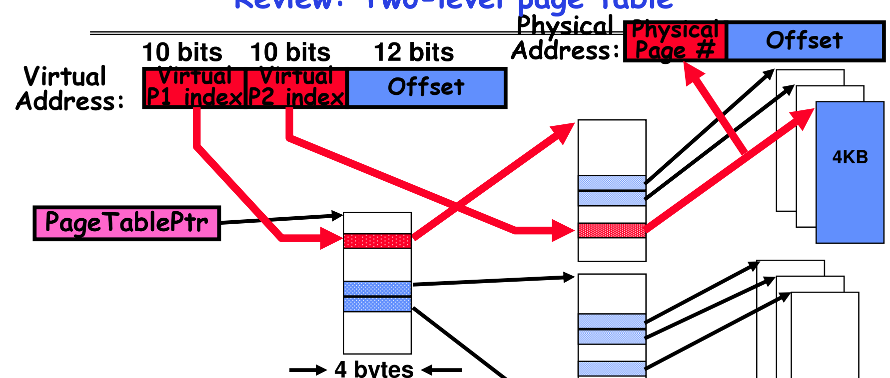
Review: Multi-level Translation

- What about a tree of tables?
  - Lowest level page table⇒memory still allocated with bitmap
  - Higher levels often segmented
- Could have any number of levels. Example (top segment):

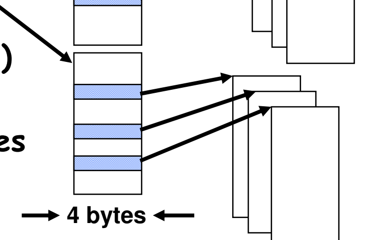


- What must be saved/restored on context switch?
  - Contents of top-level segment registers (for this example)
  - Pointer to top-level table (page table)

Review: Two-level page table



- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Sometimes, top-level page tables called "directories" (Intel)
- Each entry called a (surprise!) Page Table Entry (PTE)



Goals for Today

- Finish discussion of Address Translation
- Caching and TLBs

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

Multi-level Translation Analysis

- Pros:
  - Only need to allocate as many page table entries as we need for application
    - » In other wards, sparse address spaces are easy
  - Easy memory allocation
  - Easy Sharing
    - » Share at segment or page level (need additional reference counting)
- Cons:
  - One pointer per page (typically 4K – 16K pages today)
  - Page tables need to be contiguous
    - » However, previous example keeps tables to exactly one page in size
  - Two (or more, if >2 levels) lookups per reference
    - » Seems very expensive!
- Really starts to be a problem for 64-bit address space:
  - How big is virtual memory space vs physical memory?

What is in a PTE?

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"

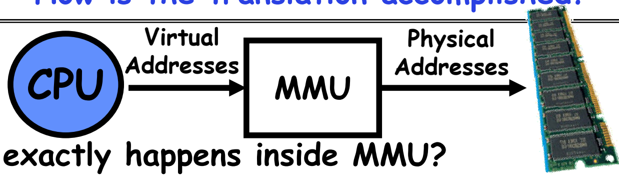
Page Frame Number (Physical Page Number)	Free (OS)	0	L	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

P: Present (same as "valid" bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: L=1⇒4MB page (directory only). Bottom 22 bits of virtual address serve as offset

Examples of how to use a PTE

- How do we use the PTE?
  - Invalid PTE can imply different things:
    - » Region of address space is actually invalid or
    - » Page/directory is just somewhere else than memory
  - Validity checked first
    - » OS can use other (say) 31 bits for location info
- Usage Example: Demand Paging
  - Keep only active pages in memory
  - Place others on disk and mark their PTEs invalid
- Usage Example: Copy on Write
  - UNIX fork gives *copy* of parent address space to child
    - » Address spaces disconnected after child created
  - How to do this cheaply?
    - » Make copy of parent's page tables (point at same memory)
    - » Mark entries in both sets of page tables as read-only
    - » Page fault on write creates two copies
- Usage Example: Zero Fill On Demand
  - New data pages must carry no information (say be zeroed)
  - Mark PTEs as invalid; page fault on use gets zeroed page
  - Often, OS creates zeroed pages in background

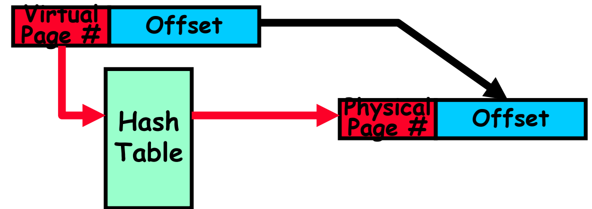
How is the translation accomplished?



- What, exactly happens inside MMU?
- One possibility: Hardware Tree Traversal
  - For each virtual address, takes page table base pointer and traverses the page table in hardware
  - Generates a "Page Fault" if it encounters invalid PTE
    - » Fault handler will decide what to do
    - » More on this next lecture
  - Pros: Relatively fast (but still many memory accesses!)
  - Cons: Inflexible, Complex hardware
- Another possibility: Software
  - Each traversal done in software
  - Pros: Very flexible
  - Cons: Every translation must invoke Fault!
- In fact, need way to cache translations for either case!

Inverted Page Table

- With all previous examples ("Forward Page Tables")
  - Size of page table is at least as large as amount of virtual memory allocated to processes
  - Physical memory may be much less
    - » Much of process space may be out on disk or not in use



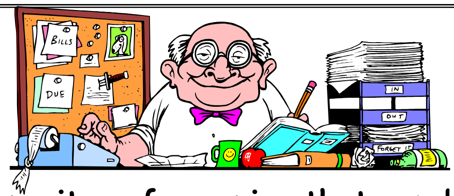
- Answer: use a hash table
  - Called an "Inverted Page Table"
  - Size is independent of virtual address space
  - Directly related to amount of physical memory
  - Very attractive option for 64-bit address spaces
- Cons: Complexity of managing hash changes
  - Often in hardware!

Closing thought: Protection without Hardware

- Does protection require hardware support for translation and dual-mode behavior?
  - No: Normally use hardware, but anything you can do in hardware can also do in software (possibly expensive)
- Protection via Strong Typing
  - Restrict programming language so that you can't express program that would trash another program
  - Loader needs to make sure that program produced by valid compiler or all bets are off
  - Example languages: LISP, Ada, Modula-3 and Java
- Protection via software fault isolation:
  - Language independent approach: have compiler generate object code that provably can't step out of bounds
    - » Compiler puts in checks for every "dangerous" operation (loads, stores, etc). Again, need special loader.
    - » Alternative, compiler generates "proof" that code cannot do certain things (Proof Carrying Code)
  - Or: use virtual machine to guarantee safe behavior (loads and stores recompiled on fly to check bounds)

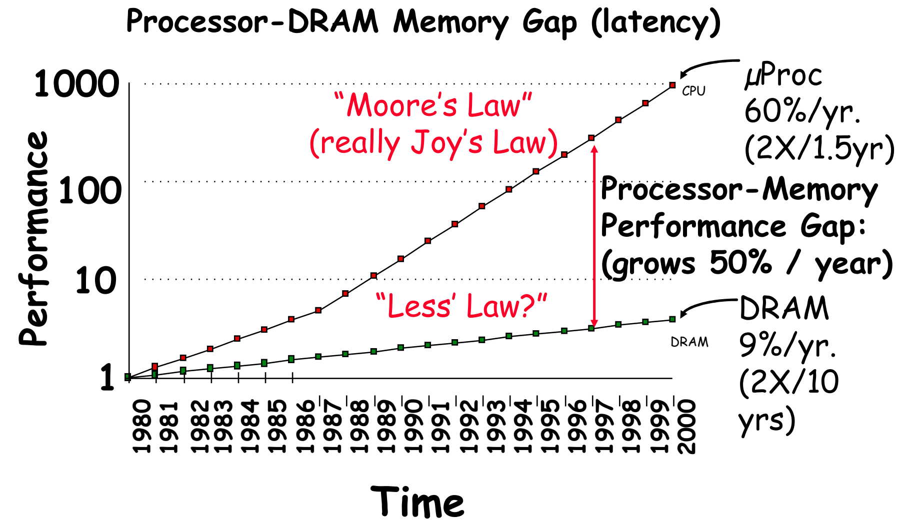
Administrivia

Caching Concept

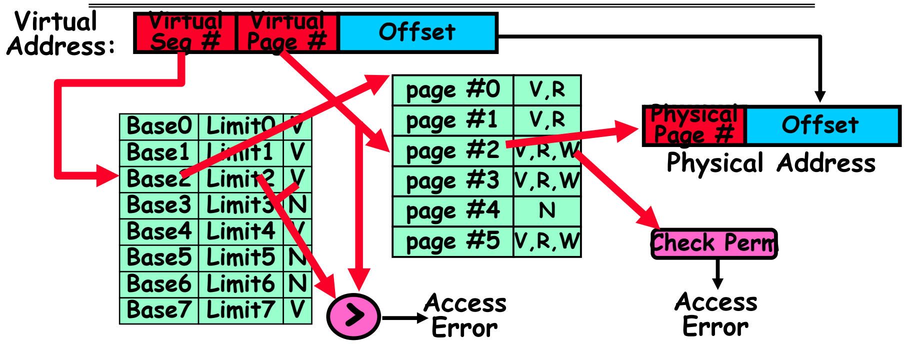


- Cache: a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant
- Caching underlies many of the techniques that are used today to make computers fast
  - Can cache: memory locations, address translations, pages, file blocks, file names, network routes, etc…
- Only good if:
  - Frequent case frequent enough and
  - Infrequent case not too expensive
- Important measure: Average Access time = (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

Why Bother with Caching?



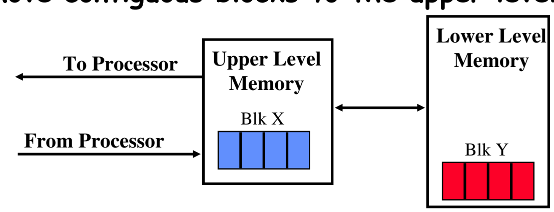
Another Major Reason to Deal with Caching



- Cannot afford to translate on every access
  - At least three DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially on disk!
- Even worse: What if we are using caching to make memory access faster than DRAM access???
- Solution? Cache translations!
  - Translation Cache: TLB ("Translation Lookaside Buffer")

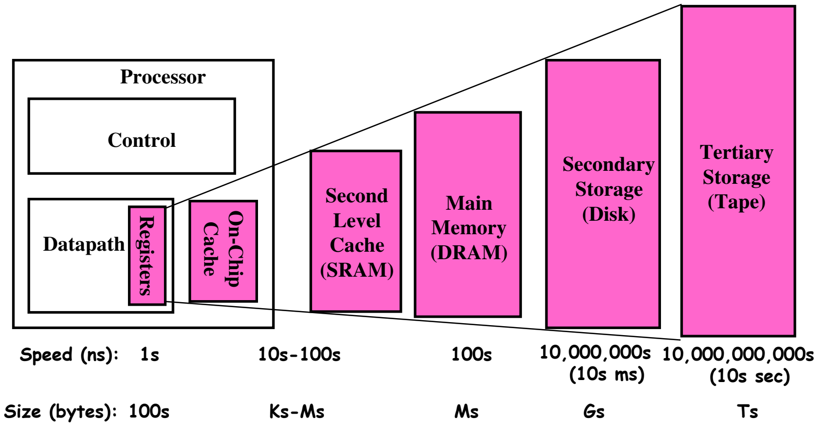
Why Does Caching Help? Locality!

- Temporal Locality (Locality in Time):
  - Keep recently accessed data items closer to processor
- Spatial Locality (Locality in Space):
  - Move contiguous blocks to the upper levels



Memory Hierarchy of a Modern Computer System

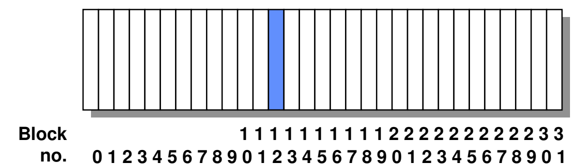
- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology



Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache

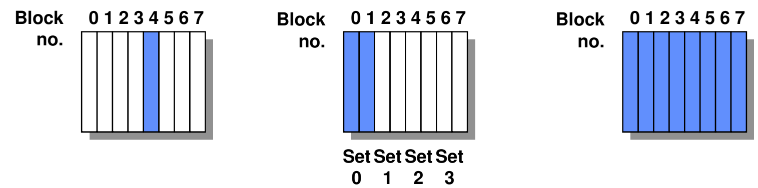
32-Block Address Space:



Direct mapped: block 12 can go only into block 4 (12 mod 8)

Set associative: block 12 can go anywhere in set 0 (12 mod 4)

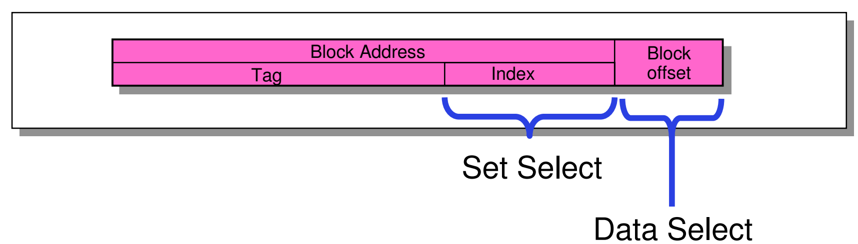
Fully associative: block 12 can go anywhere



A Summary on Sources of Cache Misses

- **Compulsory** (cold start or process migration, first reference): first access to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: If you are going to run "billions" of instruction, Compulsory Misses are insignificant
- **Capacity**:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- **Conflict** (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- **Coherence** (Invalidation): other process (e.g., I/O) updates memory

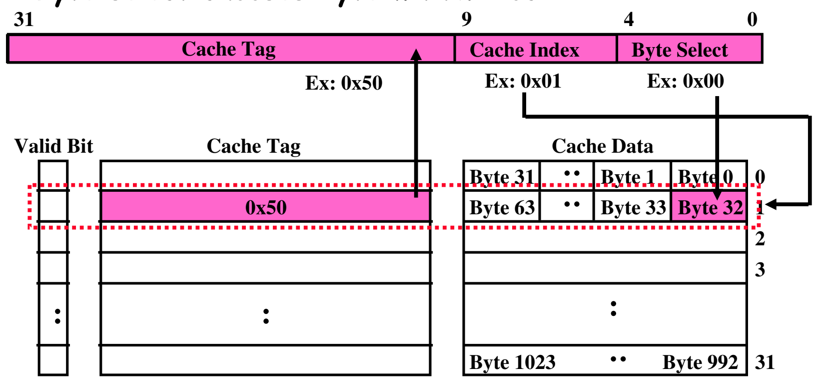
How is a Block found in a Cache?



- Index Used to Lookup Candidates in Cache
  - Index identifies the set
- Tag used to identify actual copy
  - If no candidates match, then declare cache miss
- Block is minimum quantum of caching
  - Data select field used to select data within block
  - Many caching applications don't have data select field

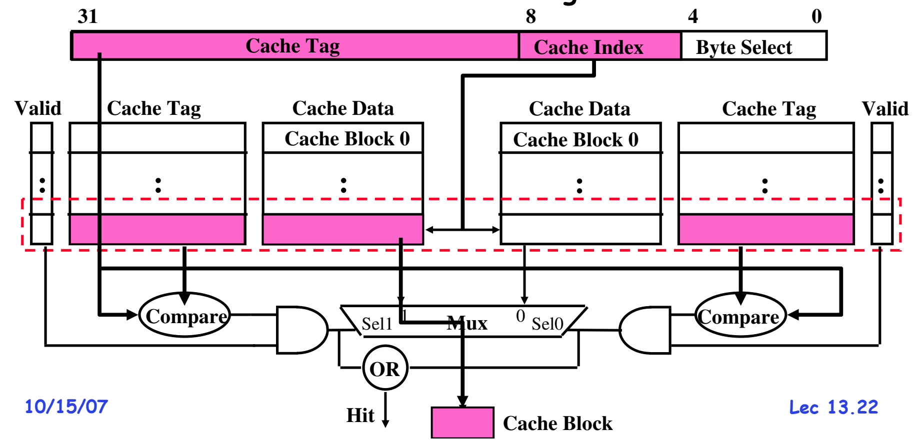
Review: Direct Mapped Cache

- Direct Mapped 2^N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2^M)
- Example: 1 KB Direct Mapped Cache with 32 B Blocks
  - Index chooses potential block
  - Tag checked to verify block
  - Byte select chooses byte within block



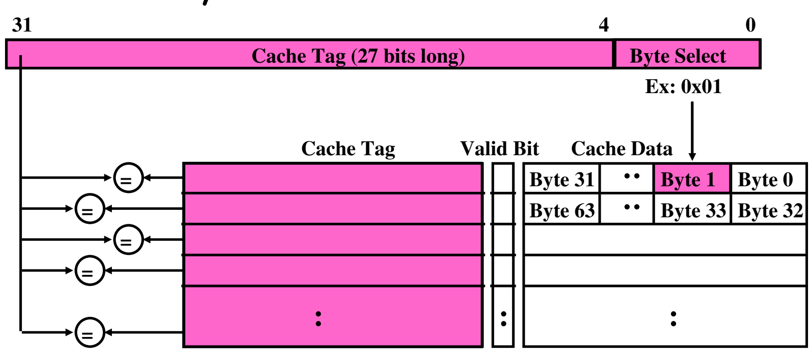
Review: Set Associative Cache

- N-way set associative: N entries per Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Cache Index selects a "set" from the cache
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result



Review: Fully Associative Cache

- Fully Associative: Every block can hold any line
  - Address does not include a cache index
  - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
  - We need N 27-bit comparators
  - Still have byte select to choose from within block



Review: Which block should be replaced on a miss?

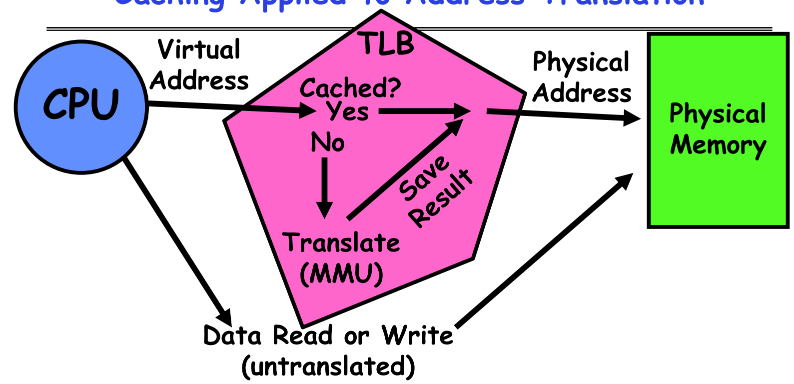
- Easy for Direct Mapped: Only one possibility
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Review: What happens on a write?

- **Write through**: The information is written to both the block in the cache and to the block in the lower-level memory
- **Write back**: The information is written only to the block in the cache.
  - Modified cache block is written to main memory only when it is replaced
  - Question is block clean or dirty?
- Pros and Cons of each?
  - WT:
    - » PRO: read misses cannot result in writes
    - » CON: Processor held up on writes unless writes buffered
  - WB:
    - » PRO: repeated writes not sent to DRAM
      processor not held up on writes
    - » CON: More complex
      Read miss may require writeback of dirty data

Caching Applied to Address Translation



- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on the same page (since accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some…
- Can we have a TLB hierarchy?
  - Sure: multiple levels at different sizes/speeds

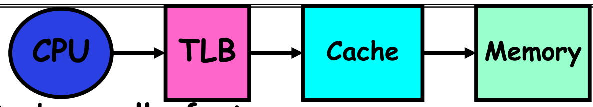
What Actually Happens on a TLB Miss?

- Hardware traversed page tables:
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    - » If PTE valid, hardware fills TLB and processor never knows
    - » If PTE marked as invalid, causes Page Fault, after which kernel decides what to do afterwards
- Software traversed Page tables (like MIPS)
  - On TLB miss, processor receives TLB fault
  - Kernel traverses page table to find PTE
    - » If PTE valid, fills TLB and returns from fault
    - » If PTE marked as invalid, internally calls Page Fault handler
- Most chip sets provide hardware traversal
  - Modern operating systems tend to have more TLB faults since they use translation for many things
  - Examples:
    - » shared segments
    - » user-level portions of an operating system

What happens on a Context Switch?

- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!
- Options?
  - Invalidate TLB: simple but might be expensive
    - » What if switching frequently between processes?
  - Include ProcessID in TLB
    - » This is an architectural solution: needs hardware
- What if translation tables change?
  - For example, to move page from memory to disk or vice versa…
  - Must invalidate TLB entry!
    - » Otherwise, might think that page is still in memory!

What TLB organization makes sense?



- Needs to be really fast
  - Critical path of memory access
    - » In simplest view: before the cache
    - » Thus, this adds to access time (reducing cache speed)
  - Seems to argue for Direct Mapped or Low Associativity
- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high!
  - This argues that cost of Conflict (Miss Time) is much higher than slightly increased cost of access (Hit Time)
- Thrashing: continuous conflicts between accesses
  - What if use low order bits of page as index into TLB?
    - » First page of code, data, stack may map to same entry
    - » Need 3-way associativity at least?
  - What if use high order bits as index?
    - » TLB mostly unused for small programs

TLB organization: include protection

- How big does TLB actually have to be?
  - Usually small: 128-512 entries
  - Not very big, can support higher associativity
- TLB usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address + other info
- What happens when fully-associative is too slow?
  - Put a small (4-16 entry) direct-mapped cache in front
  - Called a "TLB Slice"
- Example for MIPS R3000:

Virtual Address	Physical Address	Dirty	Ref	Valid	Access	ASID
0xFA00	0x0003	Y	N	Y	R/W	34
0x0040	0x0010	N	Y	Y	R	0
0x0041	0x0011	N	Y	Y	R	0

Example: R3000 pipeline includes TLB "stages"

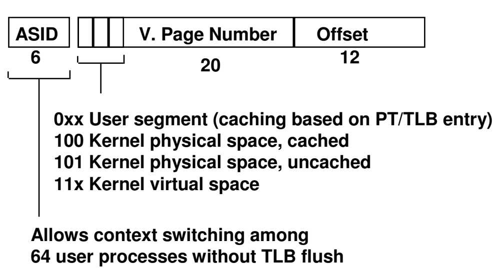
MIPS R3000 Pipeline

Inst Fetch		Dcd/ Reg	ALU / E.A		Memory	Write Reg
TLB	I-Cache	RF	Operation			WB
			E.A.	TLB	D-Cache	

TLB

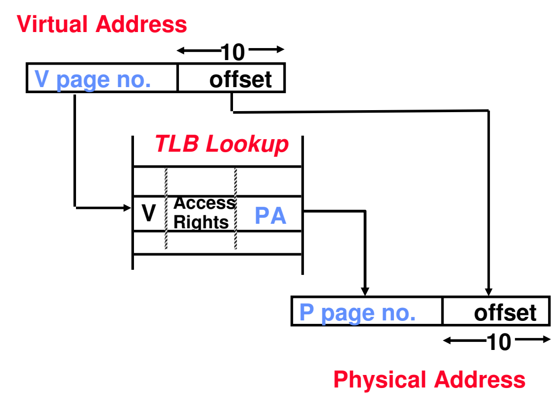
64 entry, on-chip, fully associative, software TLB fault handler

Virtual Address Space



Reducing translation time further

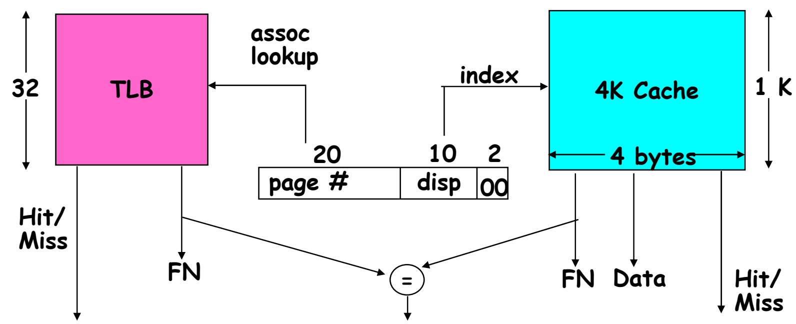
- As described, TLB lookup is in serial with cache lookup:



- Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  - Works because offset available early

Overlapping TLB & Cache Access

- Here is how this might work with a 4K cache:



- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else. See CS152/252
- Another option: Virtual Caches
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

Summary #1/2

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time.
    - » Temporal Locality: Locality in Time
    - » Spatial Locality: Locality in Space
- Three (+1) Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Conflict Misses: increase cache size and/or associativity
  - Capacity Misses: increase cache size
  - Coherence Misses: Caused by external processors or I/O devices
- Cache Organizations:
  - Direct Mapped: single block per set
  - Set associative: more than one block per set
  - Fully associative: all entries equivalent

Summary #2/2: Translation Caching (TLB)

- PTE: Page Table Entries
  - Includes physical page number
  - Control info (valid bit, writeable, dirty, user, etc)
- A cache of translations called a "Translation Lookaside Buffer" (TLB)
  - Relatively small number of entries (< 512)
  - Fully Associative (Since conflict misses expensive)
  - TLB entries contain PTE and optional process ID
- On TLB miss, page table must be traversed
  - If located PTE is invalid, cause Page Fault
- On context switch/change in page table
  - TLB entries must be invalidated somehow
- TLB is logically in front of cache
  - Thus, needs to be overlapped with cache access to be really fast

CS162
Operating Systems and Systems Programming
Lecture 14

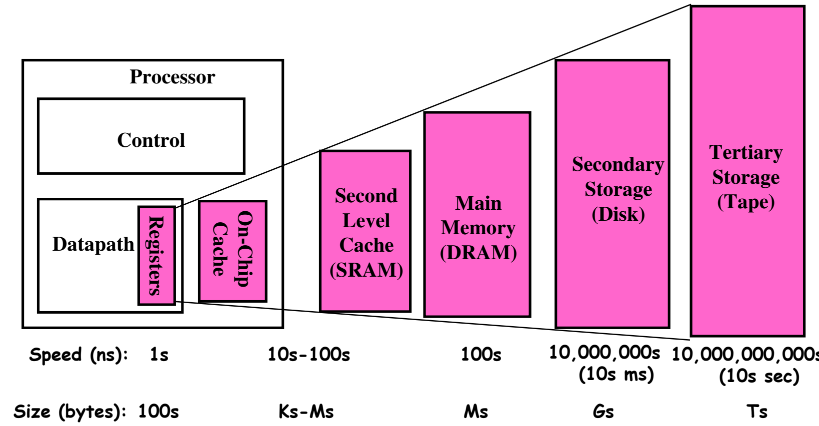
Caching and Demand Paging

October 17, 2007

Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

Review: Memory Hierarchy of a Modern Computer System

- Take advantage of the principle of locality to:
  - Present as much memory as in the cheapest technology
  - Provide access at speed offered by the fastest technology

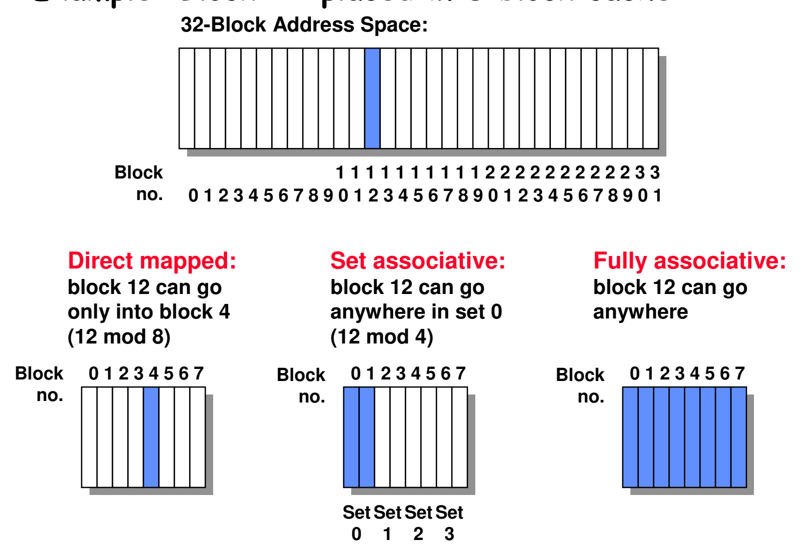


Review: A Summary on Sources of Cache Misses

- **Compulsory** (cold start): first reference to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: When running "billions" of instruction, Compulsory Misses are insignificant
- **Capacity**:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- **Conflict** (collision):
  - Multiple memory locations mapped to same cache location
  - Solutions: increase cache size, or increase associativity
- **Two others:**
  - **Coherence** (Invalidation): other process (e.g., I/O) updates memory
  - **Policy**: Due to non-optimal replacement policy

Review: Where does a Block Get Placed in a Cache?

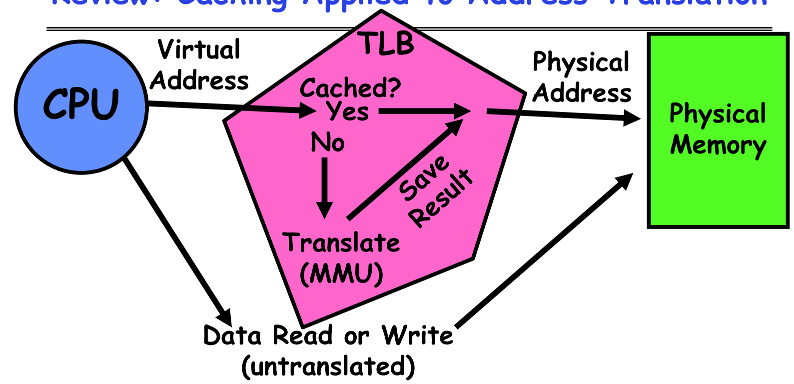
- Example: Block 12 placed in 8 block cache



Review: Other Caching Questions

- What line gets replaced on cache miss?
  - Easy for Direct Mapped: Only one possibility
  - Set Associative or Fully Associative:
    - » Random
    - » LRU (Least Recently Used)
- What happens on a write?
  - Write through: The information is written to both the cache and to the block in the lower-level memory
  - Write back: The information is written only to the block in the cache
    - » Modified cache block is written to main memory only when it is replaced
    - » Question is block clean or dirty?

Review: Caching Applied to Address Translation



- Question is one of page locality: does it exist?
  - Instruction accesses spend a lot of time on the same page (since accesses sequential)
  - Stack accesses have definite locality of reference
  - Data accesses have less page locality, but still some…
- Can we have a TLB hierarchy?
  - Sure: multiple levels at different sizes/speeds

Goals for Today

- Finish discussion of TLBs
- Concept of Paging to Disk
- Page Faults and TLB Faults
- Precise Interrupts
- Page Replacement Policies

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

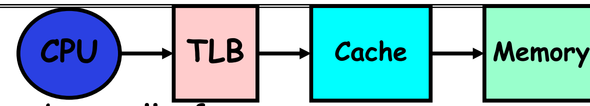
What Actually Happens on a TLB Miss?

- Hardware traversed page tables:
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    - » If PTE valid, hardware fills TLB and processor never knows
    - » If PTE marked as invalid, causes Page Fault, after which kernel decides what to do afterwards
- Software traversed Page tables (like MIPS)
  - On TLB miss, processor receives TLB fault
  - Kernel traverses page table to find PTE
    - » If PTE valid, fills TLB and returns from fault
    - » If PTE marked as invalid, internally calls Page Fault handler
- Most chip sets provide hardware traversal
  - Modern operating systems tend to have more TLB faults since they use translation for many things
  - Examples:
    - » shared segments
    - » user-level portions of an operating system

What happens on a Context Switch?

- Need to do something, since TLBs map virtual addresses to physical addresses
  - Address Space just changed, so TLB entries no longer valid!
- Options?
  - Invalidate TLB: simple but might be expensive
    - » What if switching frequently between processes?
  - Include ProcessID in TLB
    - » This is an architectural solution: needs hardware
- What if translation tables change?
  - For example, to move page from memory to disk or vice versa…
  - Must invalidate TLB entry!
    - » Otherwise, might think that page is still in memory!

What TLB organization makes sense?



- Needs to be really fast
  - Critical path of memory access
    - » In simplest view: before the cache
    - » Thus, this adds to access time (reducing cache speed)
  - Seems to argue for Direct Mapped or Low Associativity
- However, needs to have very few conflicts!
  - With TLB, the Miss Time extremely high!
  - This argues that cost of Conflict (Miss Time) is much higher than slightly increased cost of access (Hit Time)
- Thrashing: continuous conflicts between accesses
  - What if use low order bits of page as index into TLB?
    - » First page of code, data, stack may map to same entry
    - » Need 3-way associativity at least?
  - What if use high order bits as index?
    - » TLB mostly unused for small programs

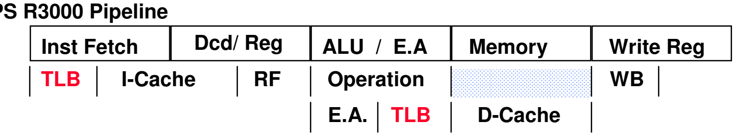
TLB organization: include protection

- How big does TLB actually have to be?
  - Usually small: 128-512 entries
  - Not very big, can support higher associativity
- TLB usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address + other info
- What happens when fully-associative is too slow?
  - Put a small (4-16 entry) direct-mapped cache in front
  - Called a "TLB Slice"
- Example for MIPS R3000:

Virtual Address	Physical Address	Dirty	Ref	Valid	Access	ASID
0xFA00	0x0003	Y	N	Y	R/W	34
0x0040	0x0010	N	Y	Y	R	0
0x0041	0x0011	N	Y	Y	R	0

Example: R3000 pipeline includes TLB "stages"

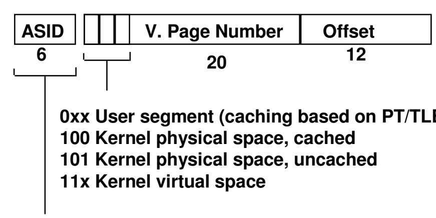
MIPS R3000 Pipeline



TLB

64 entry, on-chip, fully associative, software TLB fault handler

Virtual Address Space

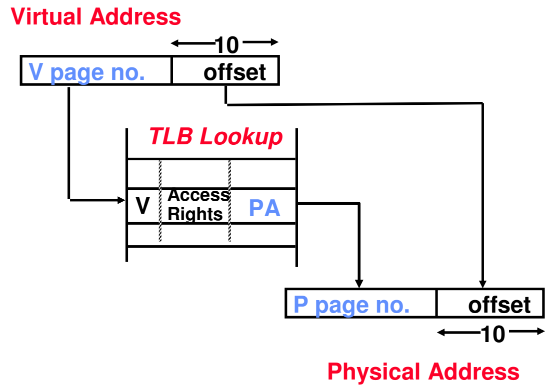


0xx User segment (caching based on PT/TLB entry)
100 Kernel physical space, cached
101 Kernel physical space, uncached
11x Kernel virtual space

Allows context switching among
64 user processes without TLB flush

Reducing translation time further

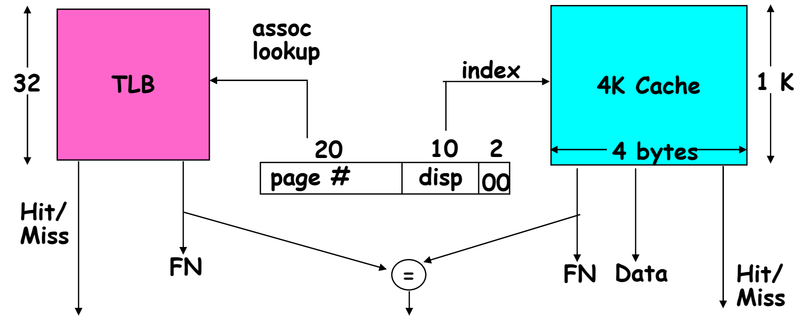
- As described, TLB lookup is in serial with cache lookup:



- Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  - Works because offset available early

Overlapping TLB & Cache Access

- Here is how this might work with a 4K cache:

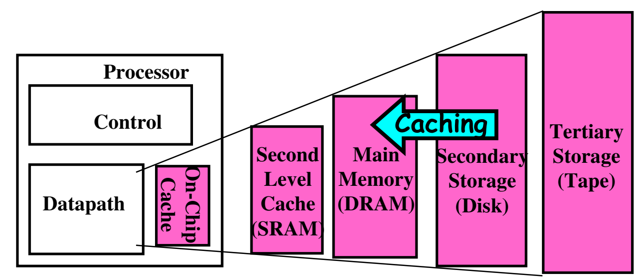


- What if cache size is increased to 8KB?
  - Overlap not complete
  - Need to do something else. See CS152/252
- Another option: Virtual Caches
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

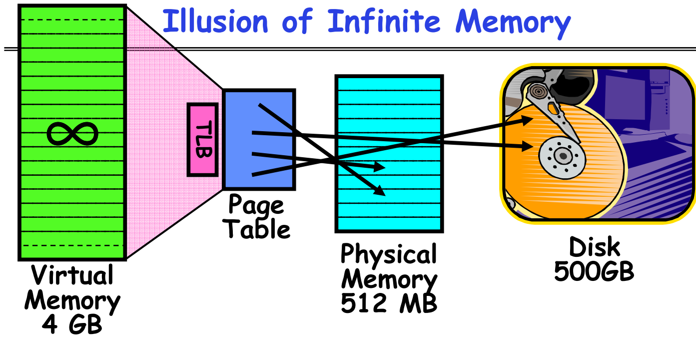
Administrative

Demand Paging

- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30%/year
- But they don't use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user's code to be in memory
- Solution: use main memory as cache for disk



Illusion of Infinite Memory



- Disk is larger than physical memory ⇒
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    - » More programs fit into memory, allowing more concurrency
- Principle: Transparent Level of Indirection (page table)
  - Supports flexible placement of physical data
    - » Data could be on disk or somewhere across network
  - Variable location of data transparent to user program
    - » Performance issue, not correctness issue

Demand Paging is Caching

- Since Demand Paging is Caching, must ask:
  - What is block size?
    - » 1 page
  - What is organization of this cache (i.e. direct-mapped, set-associative, fully-associative)?
    - » Fully associative: arbitrary virtual→physical mapping
  - How do we find a page in the cache when look for it?
    - » First check TLB, then page-table traversal
  - What is page replacement policy? (i.e. LRU, Random…)
    - » This requires more explanation… (kinda LRU)
  - What happens on a miss?
    - » Go to lower level to fill miss (i.e. disk)
  - What happens on a write? (write-through, write back)
    - » Definitely write-back. Need dirty bit!

Review: What is in a PTE?

- What is in a Page Table Entry (or PTE)?
  - Pointer to next-level page table or to actual page
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"

Page Frame Number (Physical Page Number)	Free (OS)	0	L	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

P: Present (same as "valid" bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset

Demand Paging Mechanisms

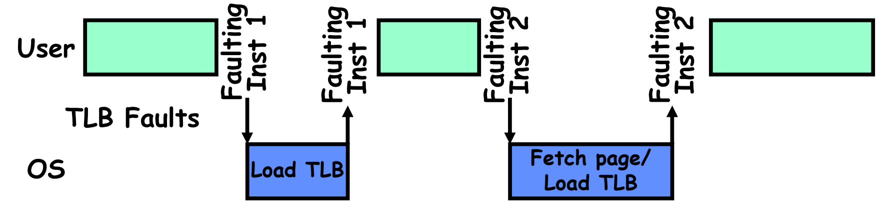
- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - » Resulting trap is a "Page Fault"
  - What does OS do on a Page Fault?:

    - » Choose an old page to replace
    - » If old page modified ("D=1"), write contents back to disk
    - » Change its PTE and any cached TLB to be invalid
    - » Load new page into memory from disk
    - » Update page table entry, invalidate TLB for new entry
    - » Continue thread from original faulting location
  - TLB for new page will be loaded when thread continued!
  - While pulling pages off disk for one process, OS runs another process from ready queue
    - » Suspended process sits on wait queue

Software-Loaded TLB

- MIPS/Snake/Nachos TLB is loaded by software
  - High TLB hit rate⇒ok to trap to software to fill the TLB, even if slower
  - Simpler hardware and added flexibility: software can maintain translation tables in whatever convenient format
- How can a process run without access to page table?
  - Fast path (TLB hit with valid=1):
    - » Translation to physical page done by hardware
  - Slow path (TLB hit with valid=0 or TLB miss)
    - » Hardware receives a "TLB Fault"
  - What does OS do on a TLB Fault?
    - » Traverse page table to find appropriate PTE
    - » If valid=1, load page table entry into TLB, continue thread
    - » If valid=0, perform "Page Fault" detailed previously
    - » Continue thread
- Everything is transparent to the user process:
  - It doesn't know about paging to/from disk
  - It doesn't even know about software TLB handling

Transparent Exceptions



- How to transparently restart faulting instructions?
  - Could we just skip it?
    - » No: need to perform load or store after reconnecting physical page
- Hardware must help out by saving:
  - Faulting instruction and partial state
    - » Need to know which instruction caused fault
    - » Is single PC sufficient to identify faulting position????
  - Processor State: sufficient to restart user thread
    - » Save/restore registers, stack, etc
- What if an instruction has side-effects?

Consider weird things that can happen

- What if an instruction has side effects?
  - Options:
    - » Unwind side-effects (easy to restart)
    - » Finish off side-effects (messy!)
  - Example 1: `mov (sp)+,10`
    - » What if page fault occurs when write to stack pointer?
    - » Did `sp` get incremented before or after the page fault?
  - Example 2: `strcpy (r1), (r2)`
    - » Source and destination overlap: can't unwind in principle!
    - » IBM S/370 and VAX solution: execute twice – once read-only
- What about "RISC" processors?
  - For instance delayed branches?
    - » Example:
      ```
      bne somewhere
      ld r1,(sp)
      ```
    - » Precise exception state consists of two PCs: PC and nPC
  - Delayed exceptions:
    - » Example:
      ```
      div r1, r2, r3
      ld r1, (sp)
      ```
    - » What if takes many cycles to discover divide by zero, but load has already caused page fault?

Precise Exceptions

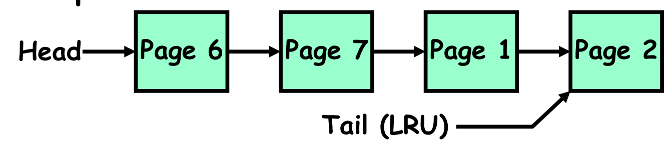
- Precise ⇒ state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations
  - Difficult in the presence of pipelining, out-of-order execution, ...
  - MIPS takes this position
- Imprecise ⇒ system software has to figure out what is where and put it all back together
- Performance goals often lead designers to forsake precise interrupts
  - system software developers, user, markets etc. usually wish they had not done this
- Modern techniques for out-of-order execution and branch prediction help implement precise interrupts

Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    - » The cost of being wrong is high: must go to disk
    - » Must keep important pages in memory, not toss them out
- What about MIN?
  - Replace page that won't be used for the longest time
  - Great, but can't really know future…
  - Makes good comparison case, however
- What about RANDOM?
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Pretty unpredictable – makes it hard to make real-time guarantees
- What about FIFO?
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages

Replacement Policies (Con't)

- What about LRU?
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- How to implement LRU? Use a list!



  - On each use, remove page from list and place at head
  - LRU page is at tail
- Problems with this scheme for paging?
  - Need to know immediately when each page used so that can change position in list…
  - Many instructions for each hardware access
- In practice, people approximate LRU (more later)

Summary

- TLB is cache on translations
  - Fully associative to reduce conflicts
  - Can be overlapped with cache access
- Demand Paging:
  - Treat memory as cache on disk
  - Cache miss $\Rightarrow$ get page from disk
- Transparent Level of Indirection
  - User program is unaware of activities of OS behind scenes
  - Data can be moved without affecting application correctness
- Software-loaded TLB
  - Fast Path: handled in hardware (TLB hit with valid=1)
  - Slow Path: Trap to software to scan page table
- Precise Exception specifies a single instruction for which:
  - All previous instructions have completed (committed state)
  - No following instructions nor actual instruction have started
- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: replace page that will be used farthest in future
  - LRU: Replace page that hasn't be used for the longest time

CS162
Operating Systems and Systems Programming
Lecture 15

Page Allocation and Replacement

October 22, 2007

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

Review: Demand Paging Mechanisms

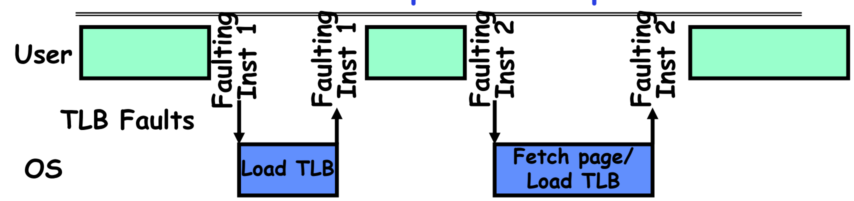
- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - » Resulting trap is a "Page Fault"
  - What does OS do on a Page Fault?:
    - » Choose an old page to replace
    - » If old page modified ("D=1"), write contents back to disk
    - » Change its PTE and any cached TLB to be invalid
    - » Load new page into memory from disk
    - » Update page table entry, invalidate TLB for new entry
    - » Continue thread from original faulting location
  - TLB for new page will be loaded when thread continued!
  - While pulling pages off disk for one process, OS runs another process from ready queue
    - » Suspended process sits on wait queue

Cache

Review: Software-Loaded TLB

- MIPS/Snake/Nachos TLB is loaded by software
  - High TLB hit rate⇒ok to trap to software to fill the TLB, even if slower
  - Simpler hardware and added flexibility: software can maintain translation tables in whatever convenient format
- How can a process run without hardware TLB fill?
  - Fast path (TLB hit with valid=1):
    - » Translation to physical page done by hardware
  - Slow path (TLB hit with valid=0 or TLB miss)
    - » Hardware receives a "TLB Fault"
  - What does OS do on a TLB Fault?
    - » Traverse page table to find appropriate PTE
    - » If valid=1, load page table entry into TLB, continue thread
    - » If valid=0, perform "Page Fault" detailed previously
    - » Continue thread
- Everything is transparent to the user process:
  - It doesn't know about paging to/from disk
  - It doesn't even know about software TLB handling

Review: Transparent Exceptions



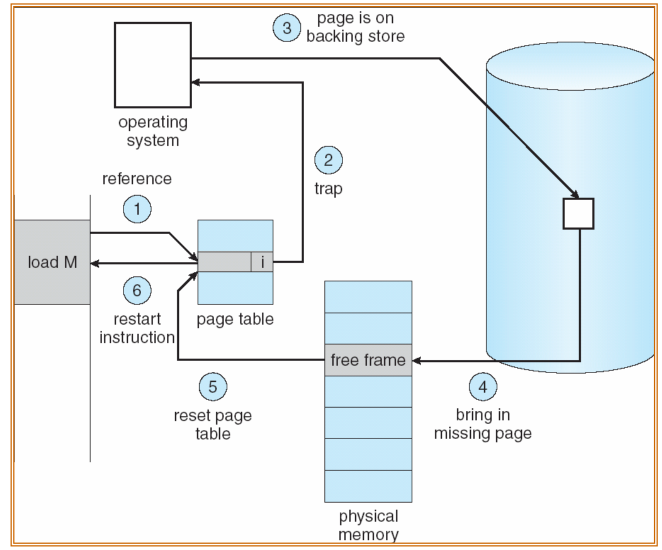
- Hardware must help out by saving:
  - Faulting instruction and partial state
  - Processor State: sufficient to restart user thread
    - » Save/restore registers, stack, etc
- Precise Exception ⇒ state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Difficult with pipelining, out-of-order execution, ...
  - MIPS takes this position
- Modern techniques for out-of-order execution and branch prediction help implement precise interrupts

Goals for Today

- Page Replacement Policies
  - Clock Algorithm
  - N^{th} chance algorithm
  - Second-Chance-List Algorithm
- Page Allocation Policies
- Working Set/Thrashing

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Steps in Handling a Page Fault



Demand Paging Example

- Since Demand Paging like caching, can compute average access time! ("Effective Access Time")
  - EAT = Hit Rate x Hit Time + Miss Rate x Miss Time
- Example:
  - Memory access time = 200 nanoseconds
  - Average page-fault service time = 8 milliseconds
  - Suppose p = Probability of miss, 1-p = Probably of hit
  - Then, we can compute EAT as follows:

$$\begin{aligned} EAT &= (1 - p) \times 200ns + p \times 8 \text{ ms} \\ &= (1 - p) \times 200ns + p \times 8{,}000{,}000ns \\ &= 200ns + p \times 7{,}999{,}800ns \end{aligned}$$

- If one access out of 1,000 causes a page fault, then EAT = 8.2 μs:
  - This is a slowdown by a factor of 40!
- What if want slowdown by less than 10%?
  - $200ns \times 1.1 < EAT \Rightarrow p < 2.5 \times 10^{-6}$
  - This is about 1 page fault in 400000!

What Factors Lead to Misses?

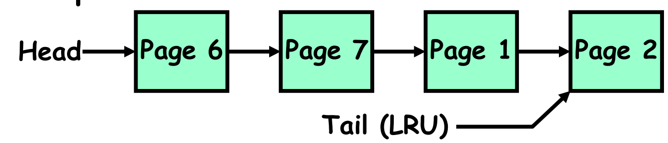
- **Compulsory Misses:**
  - Pages that have never been paged into memory before
  - How might we remove these misses?
    - » Prefetching: loading them into memory before needed
    - » Need to predict future somehow! More later.
- **Capacity Misses:**
  - Not enough memory. Must somehow increase size.
  - Can we do this?
    - » One option: Increase amount of DRAM (not quick fix!)
    - » Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!
- **Conflict Misses:**
  - Technically, conflict misses don't exist in virtual memory, since it is a "fully-associative" cache
- **Policy Misses:**
  - Caused when pages were in memory, but kicked out prematurely because of the replacement policy
  - How to fix? Better replacement policy

Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    - » The cost of being wrong is high: must go to disk
    - » Must keep important pages in memory, not toss them out
- FIFO (First In, First Out)
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages
- MIN (Minimum):
  - Replace page that won't be used for the longest time
  - Great, but can't really know future…
  - Makes good comparison case, however
- RANDOM:
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Pretty unpredictable – makes it hard to make real-time guarantees

Replacement Policies (Con't)

- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- How to implement LRU? Use a list!



  - On each use, remove page from list and place at head
  - LRU page is at tail
- Problems with this scheme for paging?
  - Need to know immediately when each page used so that can change position in list…
  - Many instructions for each hardware access
- In practice, people approximate LRU (more later)

Example: FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference stream:
  - A B C A B D A D B C B
- Consider FIFO Page replacement:

Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

  - FIFO: 7 faults.
  - When referencing D, replacing A is bad choice, since need A again right away

Example: MIN

- Suppose we have the same reference stream:
  - A B C A B D A D B C B
- Consider MIN Page replacement:

Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

  - MIN: 5 faults
  - Where will D be brought in? Look for page not referenced farthest in future.
- What will LRU do?
  - Same decisions as MIN here, but won't always be true!

Administrivia

When will LRU perform badly?

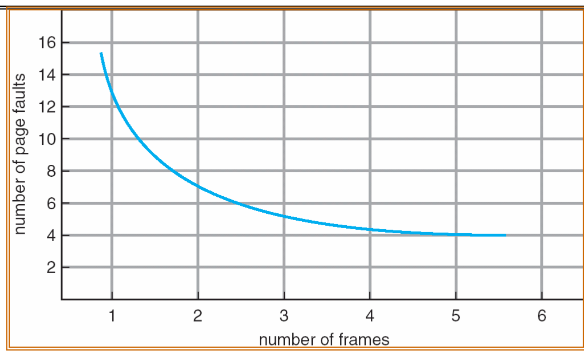
- Consider the following: A B C D A B C D A B C D
- LRU Performs as follows (same as FIFO here):

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

  - Every reference is a page fault!
- MIN Does much better:

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A									B		
2		B					C					
3			C	D								

Graph of Page Faults Versus The Number of Frames



- One desirable property: When you add memory the miss rate goes down
  - Does this always happen?
  - Seems like it should, right?
- No: BeLady's anomaly
  - Certain replacement algorithms (FIFO) don't have this obvious property!

Adding Memory Doesn't Always Help Fault Rate

- Does adding memory reduce number of page faults?
  - Yes for LRU and MIN
  - Not necessarily for FIFO! (Called Belady's anomaly)

Ref: / Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	

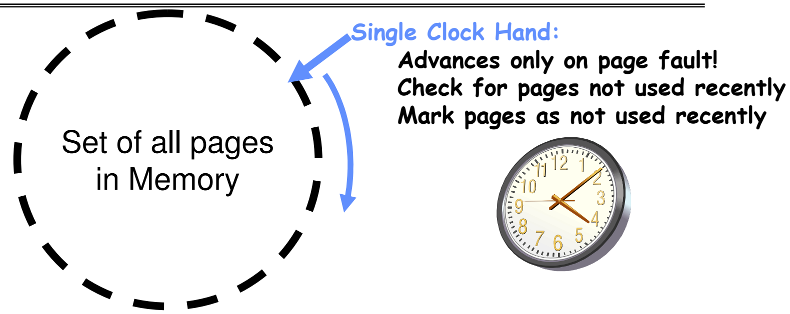
Ref: / Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

- After adding memory:
  - With FIFO, contents can be completely different
  - In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page

Implementing LRU

- Perfect:
  - Timestamp page on each reference
  - Keep list of pages ordered by time of reference
  - Too expensive to implement in reality for many reasons
- Clock Algorithm: Arrange physical pages in circle with single clock hand
  - Approximate LRU (approx to approx to MIN)
  - Replace an old page, not the oldest page
- Details:
  - Hardware "use" bit per physical page:
    - » Hardware sets use bit on each reference
    - » If use bit isn't set, means not referenced in a long time
    - » Nachos hardware sets use bit in the TLB; you have to copy this back to page table when TLB entry gets replaced
  - On page fault:
    - » Advance clock hand (not real time)
    - » Check use bit: 1→used recently; clear and leave alone
      0→selected candidate for replacement
  - Will always find a page or loop forever?
    - » Even if all use bits set, will eventually loop around⇒FIFO

Clock Algorithm: Not Recently Used



- What if hand moving slowly?
  - Good sign or bad sign?
    - » Not many page faults and/or find page quickly
- What if hand is moving quickly?
  - Lots of page faults and/or lots of reference bits set
- One way to view clock algorithm:
  - Crude partitioning of pages into two groups: young and old
  - Why not partition into more than 2 groups?

N^{th} Chance version of Clock Algorithm

- N^{th} chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - » 1⇒clear use and also clear counter (used in last sweep)
    - » 0⇒increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N? Better approx to LRU
    - » If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - » Otherwise might have to look a long way to find free page
- What about dirty pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    - » Clean pages, use N=1
    - » Dirty pages, use N=2 (and write back to disk when N=1)

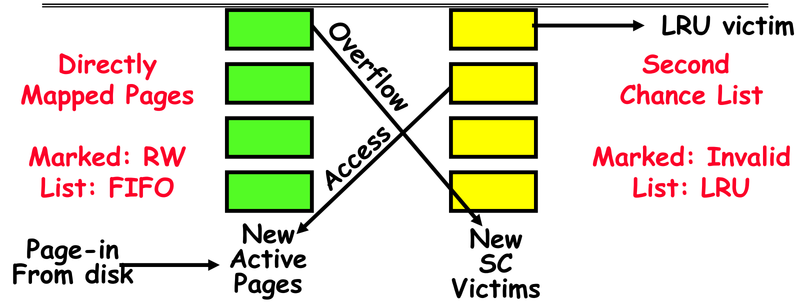
Clock Algorithms: Details

- Which bits of a PTE entry are useful to us?
  - Use: Set when page is referenced; cleared by clock algorithm
  - Modified: set when page is modified, cleared when page written to disk
  - Valid: ok for program to reference this page
  - Read-only: ok for program to read page, but not modify
    - » For example for catching modifications to code pages!
- Do we really need hardware-supported "modified" bit?
  - No. Can emulate it (BSD Unix) using read-only bit
    - » Initially, mark all pages as read-only, even data pages
    - » On write, trap to OS. OS sets software "modified" bit, and marks page as read-write.
    - » Whenever page comes back in from disk, mark read-only

Clock Algorithms Details (continued)

- Do we really need a hardware-supported "use" bit?
  - No. Can emulate it similar to above:
    - » Mark all pages as invalid, even if in memory
    - » On read to invalid page, trap to OS
    - » OS sets use bit, and marks page read-only
  - Get modified bit in same way as previous:
    - » On write, trap to OS (either invalid or read-only)
    - » Set use and modified bits, mark page read-write
  - When clock hand passes by, reset use and modified bits and mark page as invalid again
- Remember, however, that clock is just an approximation of LRU
  - Can we do a better approximation, given that we have to take page faults on some reads and writes to collect use information?
  - Need to identify an old page, not oldest page!
  - Answer: second chance list

Second-Chance List Algorithm (VAX/VMS)

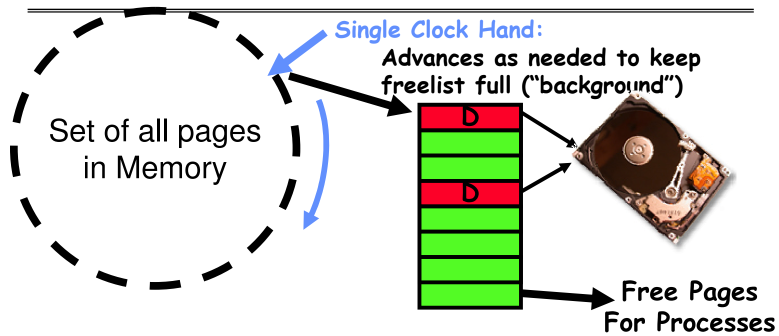


- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
  - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
  - Desired Page On SC List: move to front of Active list, mark RW
  - Not on SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list

Second-Chance List Algorithm (con't)

- How many pages for second chance list?
  - If 0 ⇒ FIFO
  - If all ⇒ LRU, but page fault on every page reference
- Pick intermediate value. Result is:
  - Pro: Few disk accesses (page only goes to disk if unused for a long time)
  - Con: Increased overhead trapping to OS (software / hardware tradeoff)
- With page translation, we can adapt to any kind of access the program makes
  - Later, we will show how to use page translation / protection to share memory between threads on widely separated machines
- Question: why didn't VAX include "use" bit?
  - Strecker (architect) asked OS people, they said they didn't need it, so didn't implement it
  - He later got blamed, but VAX did OK anyway

Free List



- Keep set of free pages ready for use in demand paging
  - Freelist filled in background by Clock algorithm or other technique ("Pageout demon")
  - Dirty pages start copying back to disk when enter list
- Like VAX second-chance list
  - If page needed before reused, just return to active set
- Advantage: Faster for page fault
  - Can always use page (or pages) immediately on fault

Demand Paging (more details)

- Does software-loaded TLB need use bit?
  Two Options:
  - Hardware sets use bit in TLB; when TLB entry is replaced, software copies use bit back to page table
  - Software manages TLB entries as FIFO list; everything not in TLB is Second-Chance list, managed as strict LRU
- Core Map
  - Page tables map virtual page → physical page
  - Do we need a reverse mapping (i.e. physical page → virtual page)?
    - » Yes. Clock algorithm runs through page frames. If sharing, then multiple virtual-pages per physical page
    - » Can't push page out to disk without invalidating all PTEs

Summary

- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past
- Clock Algorithm: Approximation to LRU
  - Arrange all pages in circular list
  - Sweep through them, marking as not "in use"
  - If page not "in use" for one pass, than can replace
- N^{th}-chance clock algorithm: Another approx LRU
  - Give pages multiple passes of clock hand before replacing
- Second-Chance List algorithm: Yet another approx LRU
  - Divide pages into two groups, one of which is truly LRU and managed on page faults.

CS162
Operating Systems and Systems Programming
Lecture 16

Page Allocation and Replacement (con't)
I/O Systems

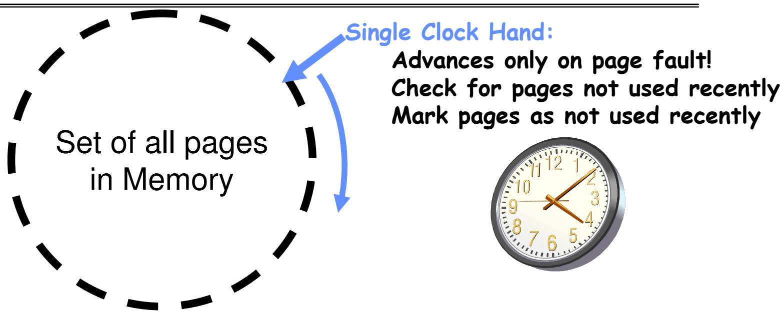
October 24, 2007

Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

Review: Page Replacement Policies

- FIFO (First In, First Out)
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages
- MIN (Minimum):
  - Replace page that won't be used for the longest time
  - Great, but can't really know future…
  - Makes good comparison case, however
- RANDOM:
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Pretty unpredictable – makes it hard to make real-time guarantees
- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.

Review: Clock Algorithm: Not Recently Used

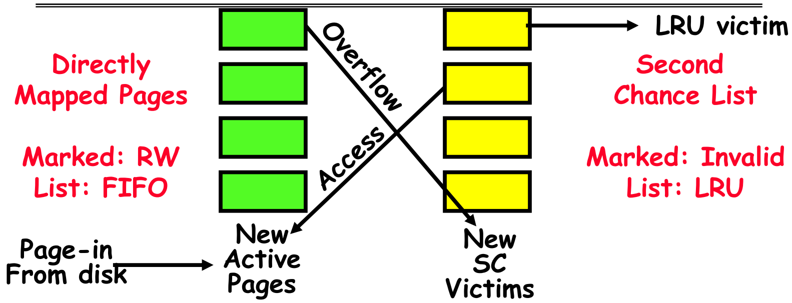


- Clock Algorithm: pages arranged in a ring
  - Hardware "use" bit per physical page:
    - » Hardware sets use bit on each reference
    - » If use bit isn't set, means not referenced in a long time
    - » Nachos hardware sets use bit in the TLB; you have to copy this back to page table when TLB entry gets replaced
  - On page fault:
    - » Advance clock hand (not real time)
    - » Check use bit: 1→used recently; clear and leave alone 0→selected candidate for replacement

Review: N^{th} Chance version of Clock Algorithm

- N^{th} chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - » 1⇒clear use and also clear counter (used in last sweep)
    - » 0⇒increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N? Better approx to LRU
    - » If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - » Otherwise might have to look a long way to find free page
- What about dirty pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
  - Common approach:
    - » Clean pages, use N=1
    - » Dirty pages, use N=2 (and write back to disk when N=1)

Review: Second-Chance List Algorithm (VAX/VMS)



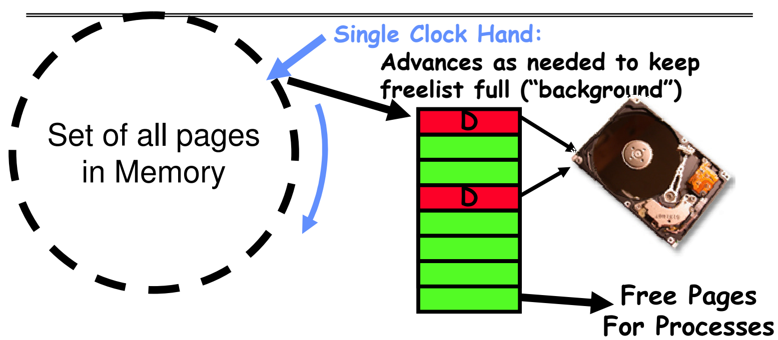
- Split memory in two: Active list (RW), SC list (Invalid)
- Access pages in Active list at full speed
- Otherwise, Page Fault
  - Always move overflow page from end of Active list to front of Second-chance list (SC) and mark invalid
  - Desired Page On SC List: move to front of Active list, mark RW
  - Not on SC list: page in to front of Active list, mark RW; page out LRU victim at end of SC list

Goals for Today

- Finish Page Allocation Policies
- Working Set/Thrashing
- I/O Systems
  - Hardware Access
  - Device Drivers

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Free List



- Keep set of free pages ready for use in demand paging
  - Freelist filled in background by Clock algorithm or other technique ("Pageout demon")
  - Dirty pages start copying back to disk when enter list
- Like VAX second-chance list
  - If page needed before reused, just return to active set
- Advantage: Faster for page fault
  - Can always use page (or pages) immediately on fault

Demand Paging (more details)

- Does software-loaded TLB need use bit? Two Options:
  - Hardware sets use bit in TLB; when TLB entry is replaced, software copies use bit back to page table
  - Software manages TLB entries as FIFO list; everything not in TLB is Second-Chance list, managed as strict LRU
- Core Map
  - Page tables map virtual page → physical page
  - Do we need a reverse mapping (i.e. physical page → virtual page)?
    - » Yes. Clock algorithm runs through page frames. If sharing, then multiple virtual-pages per physical page
    - » Can't push page out to disk without invalidating all PTEs

Allocation of Page Frames (Memory Pages)

- How do we allocate memory among different processes?
  - Does every process get the same fraction of memory? Different fractions?
  - Should we completely swap some processes out of memory?
- Each process needs *minimum* number of pages
  - Want to make sure that all processes that are loaded into memory can make forward progress
  - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
    - » instruction is 6 bytes, might span 2 pages
    - » 2 pages to handle *from*
    - » 2 pages to handle *to*
- Possible Replacement Scopes:
  - Global replacement – process selects replacement frame from set of all frames; one process can take a frame from another
  - Local replacement – each process selects from only its own set of allocated frames

Fixed/Priority Allocation

- Equal allocation (Fixed Scheme):
  - Every process gets same amount of memory
  - Example: 100 frames, 5 processes⇒process gets 20 frames
- Proportional allocation (Fixed Scheme)
  - Allocate according to the size of process
  - Computation proceeds as follows:

    s_i = size of process p_i and $S = \Sigma s_i$

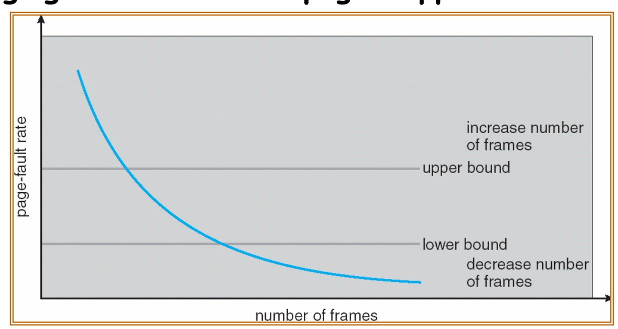
    m = total number of frames

    a_i = allocation for $p_i = \frac{s_i}{S} \times m$
- Priority Allocation:
  - Proportional scheme using priorities rather than size
    - » Same type of computation as previous scheme
  - Possible behavior: If process p_i generates a page fault, select for replacement a frame from a process with lower priority number
- Perhaps we should use an adaptive scheme instead???
  - What if some application just needs more memory?

Administrivia

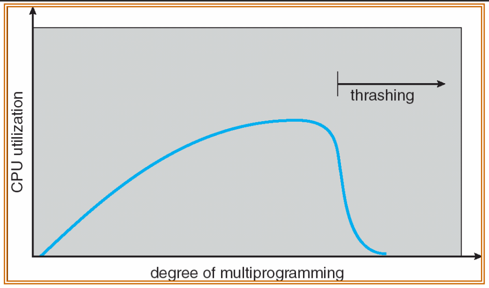
Page-Fault Frequency Allocation

- Can we reduce Capacity misses by dynamically changing the number of pages/application?



- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
- Question: What if we just don't have enough memory?

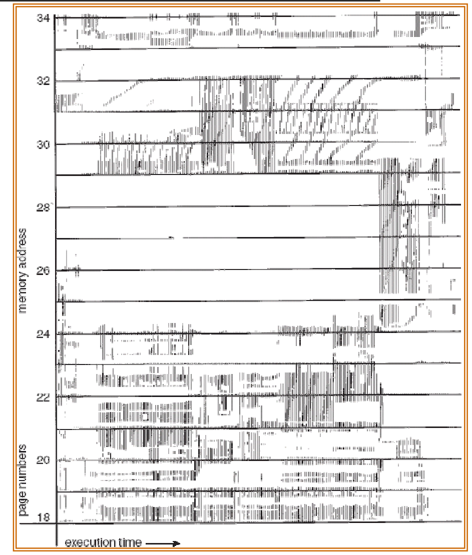
Thrashing



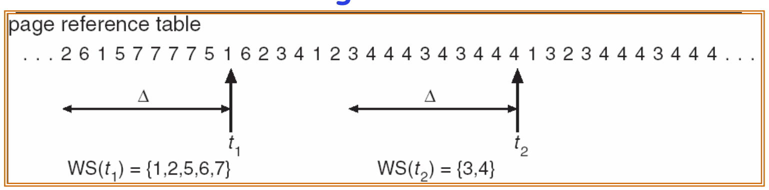
- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system spends most of its time swapping to disk
- Thrashing ≡ a process is busy swapping pages in and out
- Questions:
  - How do we detect Thrashing?
  - What is best response to Thrashing?

Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
  - Group of Pages accessed along a given time slice called the "Working Set"
  - Working Set defines minimum number of pages needed for process to behave well
- Not enough memory for Working Set⇒Thrashing
  - Better to swap out process?



Working-Set Model



- Δ ≡ working-set window ≡ fixed number of page references
  - Example: 10,000 instructions
- WS_i (working set of Process P_i) = total set of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if Δ = ∞ ⇒ will encompass entire program
- $D = \Sigma|WS_i|$ ≡ total demand frames
- if $D > m$ ⇒ Thrashing
  - Policy: if $D > m$, then suspend one of the processes
  - This can improve overall system behavior by a lot!

What about Compulsory Misses?

- Recall that compulsory misses are misses that occur the first time that a page is seen
  - Pages that are touched for the first time
  - Pages that are touched after process is swapped out/swapped back in
- Clustering:
  - On a page-fault, bring in multiple pages "around" the faulting page
  - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
- Working Set Tracking:
  - Use algorithm to try to track working set of application
  - When swapping process back in, swap in working set

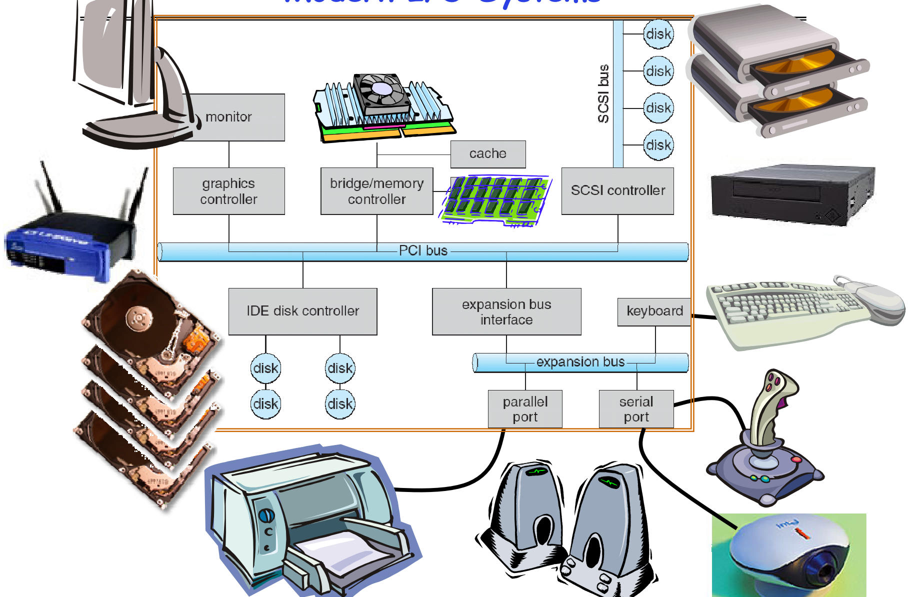
Demand Paging Summary

- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past
- Clock Algorithm: Approximation to LRU
  - Arrange all pages in circular list
  - Sweep through them, marking as not "in use"
  - If page not "in use" for one pass, than can replace
- N^{th}-chance clock algorithm: Another approx LRU
  - Give pages multiple passes of clock hand before replacing
- Second-Chance List algorithm: Yet another approx LRU
  - Divide pages into two groups, one of which is truly LRU and managed on page faults.
- Working Set:
  - Set of pages touched by a process recently
- Thrashing: a process is busy swapping pages in and out
  - Process will thrash if working set doesn't fit in memory
  - Need to swap out a process

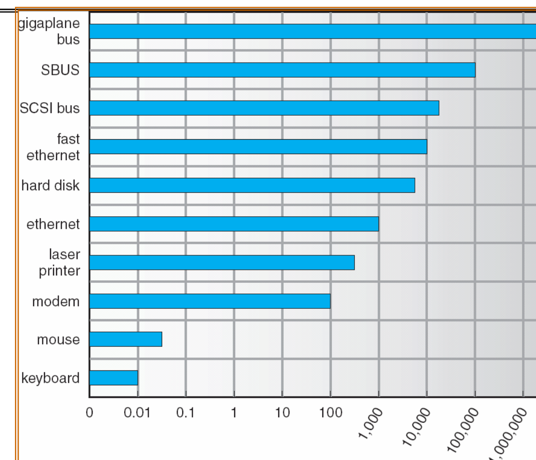
The Requirements of I/O

- So far in this course:
  - We have learned how to manage CPU, memory
- What about I/O?
  - Without I/O, computers are useless (disembodied brains?)
  - But… thousands of devices, each slightly different
    - » How can we standardize the interfaces to these devices?
  - Devices unreliable: media failures and transmission errors
    - » How can we make them reliable???
  - Devices unpredictable and/or slow
    - » How can we manage them if we don't know what they will do or how they will perform?
- Some operational parameters:
  - Byte/Block
    - » Some devices provide single byte at a time (*e.g.* keyboard)
    - » Others provide whole blocks (*e.g.* disks, networks, etc)
  - Sequential/Random
    - » Some devices must be accessed sequentially (*e.g.* tape)
    - » Others can be accessed randomly (*e.g.* disk, cd, etc.)
  - Polling/Interrupts
    - » Some devices require continual monitoring
    - » Others generate interrupts when they need service

Modern I/O Systems



Example Device-Transfer Rates (Sun Enterprise 6000)



- Device Rates vary over many orders of magnitude
  - System better be able to handle this wide range
  - Better not have high overhead/byte for fast devices!
  - Better not waste time waiting for slow devices

The Goal of the I/O Subsystem

- Provide Uniform Interfaces, Despite Wide Range of Different Devices
  - This code works on many different devices:

```
FILE fd = fopen("/dev/something","rw");
for (int i = 0; i < 10; i++) {
  fprintf(fd,"Count %d\n",i);
}
close(fd);
```

  - Why? Because code that controls devices ("device driver") implements standard interface.
- We will try to get a flavor for what is involved in actually controlling devices in rest of lecture
  - Can only scratch surface!

Want Standard Interfaces to Devices

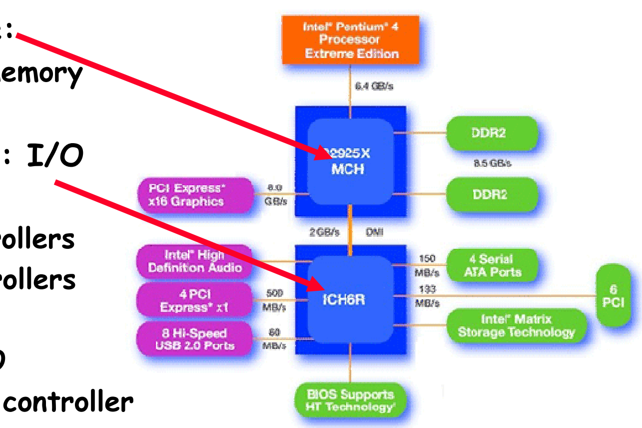
- **Block Devices:** *e.g.* disk drives, tape drives, DVD-ROM
  - Access blocks of data
  - Commands include `open()`, `read()`, `write()`, `seek()`
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- **Character Devices:** *e.g.* keyboards, mice, serial ports, some USB devices
  - Single characters at a time
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing
- **Network Devices:** *e.g.* Ethernet, Wireless, Bluetooth
  - Different enough from block/character to have own interface
  - Unix and Windows include socket interface
    - » Separates network protocol from network operation
    - » Includes `select()` functionality
  - Usage: pipes, FIFOs, streams, queues, mailboxes

How Does User Deal with Timing?

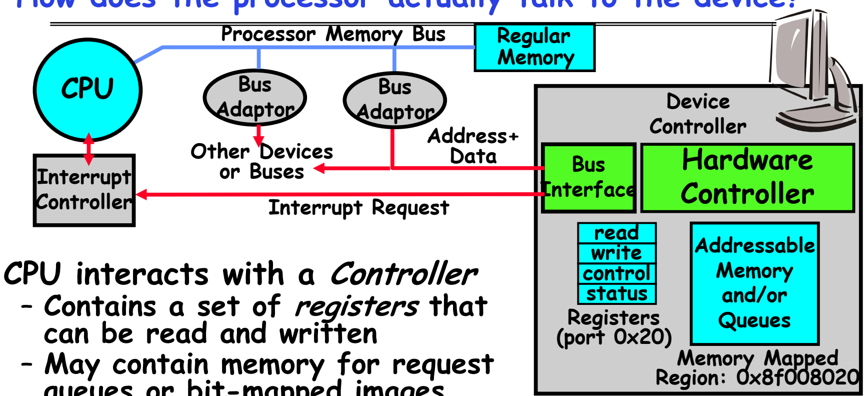
- **Blocking Interface:** "Wait"
  - When request data (e.g. `read()` system call), put process to sleep until data is ready
  - When write data (e.g. `write()` system call), put process to sleep until device is ready for data
- **Non-blocking Interface:** "Don't Wait"
  - Returns quickly from read or write request with count of bytes successfully transferred
  - Read may return nothing, write may write nothing
- **Asynchronous Interface:** "Tell Me Later"
  - When request data, take pointer to user's buffer, return immediately; later kernel fills buffer and notifies user
  - When send data, take pointer to user's buffer, return immediately; later kernel takes data and notifies user

Main components of Intel Chipset: Pentium 4

- Northbridge:
  - Handles memory
  - Graphics
- Southbridge: I/O
  - PCI bus
  - Disk controllers
  - USB controllers
  - Audio
  - Serial I/O
  - Interrupt controller
  - Timers



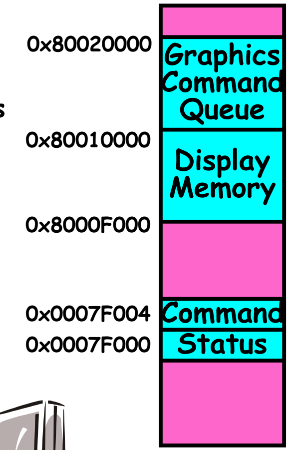
How does the processor actually talk to the device?



- CPU interacts with a *Controller*
  - Contains a set of *registers* that can be read and written
  - May contain memory for request queues or bit-mapped images
- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
  - I/O instructions: in/out instructions
    - » Example from the Intel architecture: `out 0x21,AL`
  - Memory mapped I/O: load/store instructions
    - » Registers/memory appear in physical address space
    - » I/O accomplished with load and store instructions

Example: Memory-Mapped Display Controller

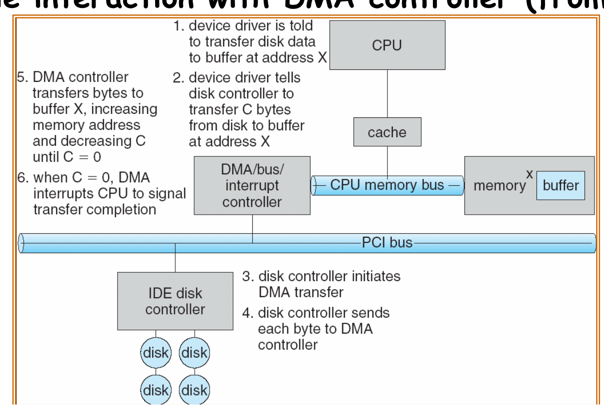
- Memory-Mapped:
  - Hardware maps control registers and display memory into physical address space
    - » Addresses set by hardware jumpers or programming at boot time
  - Simply writing to display memory (also called the "frame buffer") changes image on screen
    - » Addr: 0x8000F000—0x8000FFFF
  - Writing graphics description to command-queue area
    - » Say enter a set of triangles that describe some scene
    - » Addr: 0x80010000—0x8001FFFF
  - Writing to the command register may cause on-board graphics hardware to do something
    - » Say render the above scene
    - » Addr: 0x0007F004
- Can protect with page tables



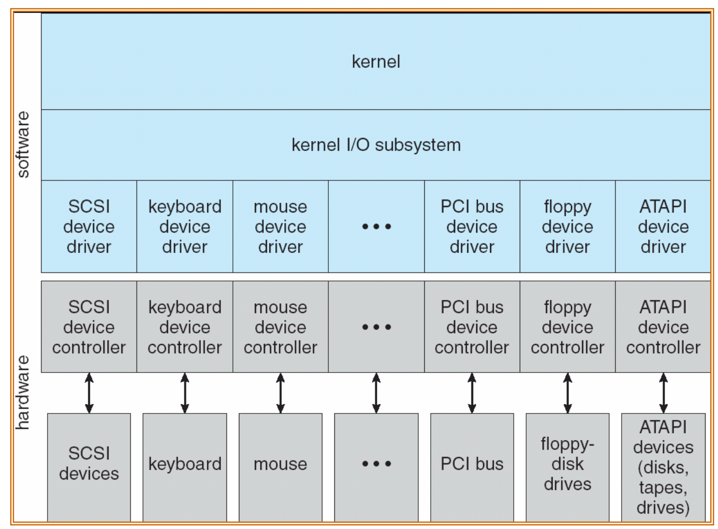
Physical Address Space

Transfering Data To/From Controller

- Programmed I/O:
  - Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size
- Direct Memory Access:
  - Give controller access to memory bus
  - Ask it to transfer data to/from memory directly
- Sample interaction with DMA controller (from book):



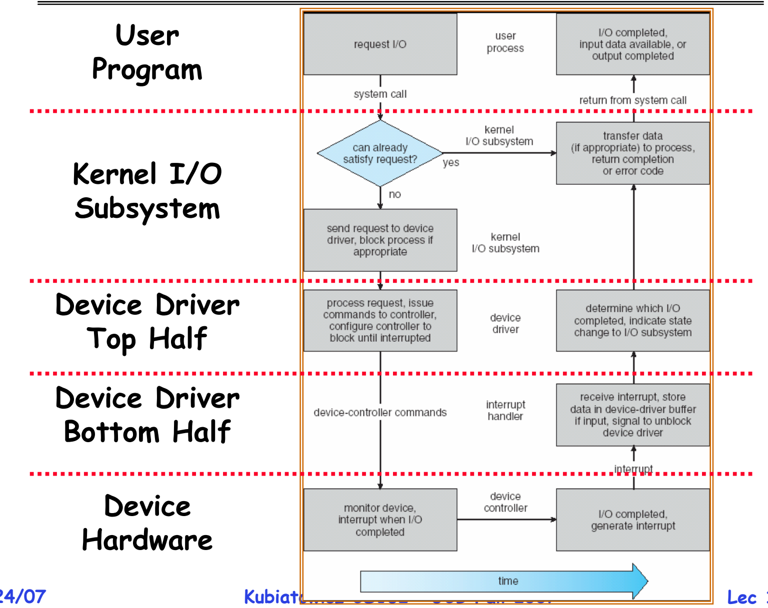
A Kernel I/O Structure



Device Drivers

- Device Driver: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call
- Device Drivers typically divided into two pieces:
  - Top half: accessed in call path from system calls
    - » Implements a set of standard, cross-device calls like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
    - » This is the kernel's interface to the device driver
    - » Top half will *start* I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    - » Gets input or transfers next block of output
    - » May wake sleeping threads if I/O now complete

Life Cycle of An I/O Request



I/O Device Notifying the OS

- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- I/O Interrupt:
  - Device generates an interrupt whenever it needs service
  - Handled in bottom half of device driver
    - » Often run on special kernel-level stack
  - Pro: handles unpredictable events well
  - Con: interrupts relatively high overhead
- Polling:
  - OS periodically checks a device-specific status register
    - » I/O device puts completion information in status register
    - » Could use timer to invoke lower half of drivers occasionally
  - Pro: low overhead
  - Con: may waste many cycles on polling if infrequent or unpredictable I/O operations
- Actual devices combine both polling and interrupts
  - For instance: High-bandwidth network device:
    - » Interrupt for first incoming packet
    - » Poll for following packets until hardware empty

Summary

- Working Set:
  - Set of pages touched by a process recently
- Thrashing: a process is busy swapping pages in and out
  - Process will thrash if working set doesn't fit in memory
  - Need to swap out a process
- I/O Devices Types:
  - Many different speeds (0.1 bytes/sec to GBytes/sec)
  - Different Access Patterns:
    - » Block Devices, Character Devices, Network Devices
  - Different Access Timing:
    - » Blocking, Non-blocking, Asynchronous
- I/O Controllers: Hardware that controls actual device
  - Processor Accesses through I/O instructions, load/store to special physical memory
  - Report their results through either interrupts or a status register that processor looks at occasionally (polling)
- Device Driver: Device-specific code in kernel

CS162
Operating Systems and Systems Programming
Lecture 17

Disk Management and File Systems

October 29, 2007

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

Review: Want Standard Interfaces to Devices

- **Block Devices:** *e.g.* disk drives, tape drives, Cdrom
  - Access blocks of data
  - Commands include `open()`, `read()`, `write()`, `seek()`
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- **Character Devices:** *e.g.* keyboards, mice, serial ports, some USB devices
  - Single characters at a time
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing
- **Network Devices:** *e.g.* Ethernet, Wireless, Bluetooth
  - Different enough from block/character to have own interface
  - Unix and Windows include socket interface
    - » Separates network protocol from network operation
    - » Includes `select()` functionality
  - Usage: pipes, FIFOs, streams, queues, mailboxes

Review: How Does User Deal with Timing?

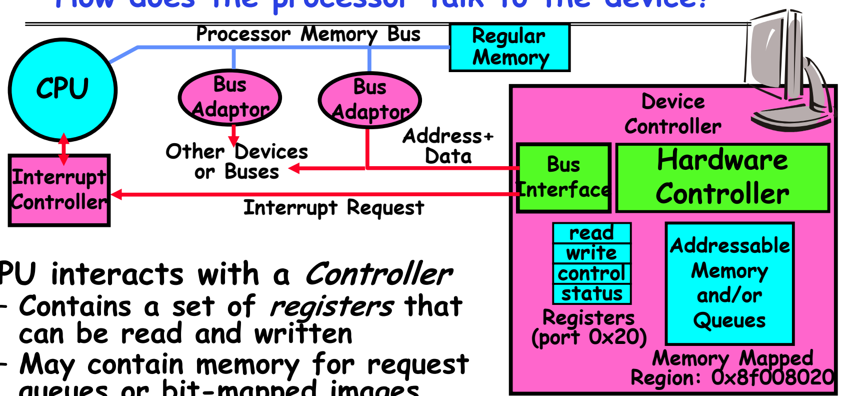
- **Blocking Interface:** "Wait"
  - When request data (e.g. `read()` system call), put process to sleep until data is ready
  - When write data (e.g. `write()` system call), put process to sleep until device is ready for data
- **Non-blocking Interface:** "Don't Wait"
  - Returns quickly from read or write request with count of bytes successfully transferred
  - Read may return nothing, write may write nothing
- **Asynchronous Interface:** "Tell Me Later"
  - When request data, take pointer to user's buffer, return immediately; later kernel fills buffer and notifies user
  - When send data, take pointer to user's buffer, return immediately; later kernel takes data and notifies user

Goals for Today

- Finish Discussing I/O Systems
  - Hardware Access
  - Device Drivers
- Disk Performance
  - Hardware performance parameters
  - Queuing Theory
- File Systems
  - Structure, Naming, Directories, and Caching

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

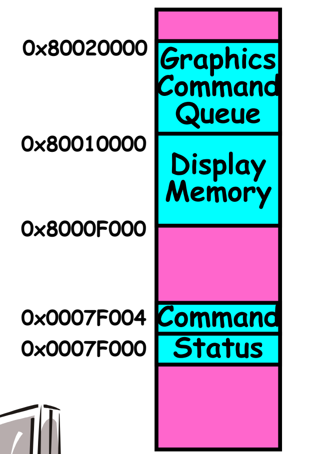
How does the processor talk to the device?



- CPU interacts with a *Controller*
  - Contains a set of *registers* that can be read and written
  - May contain memory for request queues or bit-mapped images
- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
  - I/O instructions: in/out instructions
    - » Example from the Intel architecture: `out 0x21,AL`
  - Memory mapped I/O: load/store instructions
    - » Registers/memory appear in physical address space
    - » I/O accomplished with load and store instructions

Memory-Mapped Display Controller Example

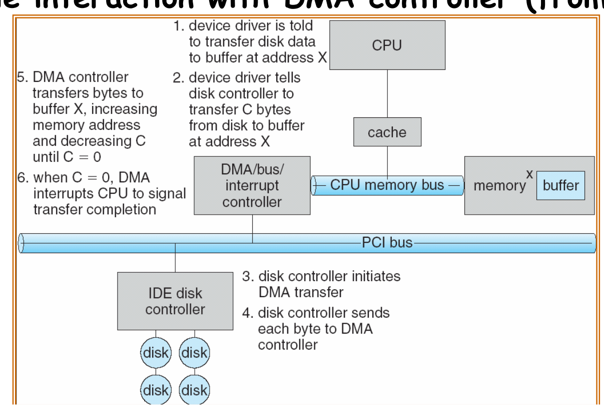
- Memory-Mapped:
  - Hardware maps control registers and display memory to physical address space
    - » Addresses set by hardware jumpers or programming at boot time
  - Simply writing to display memory (also called the "frame buffer") changes image on screen
    - » Addr: 0x8000F000—0x8000FFFF
  - Writing graphics description to command-queue area
    - » Say enter a set of triangles that describe some scene
    - » Addr: 0x80010000—0x8001FFFF
  - Writing to the command register may cause on-board graphics hardware to do something
    - » Say render the above scene
    - » Addr: 0x0007F004
- Can protect with page tables



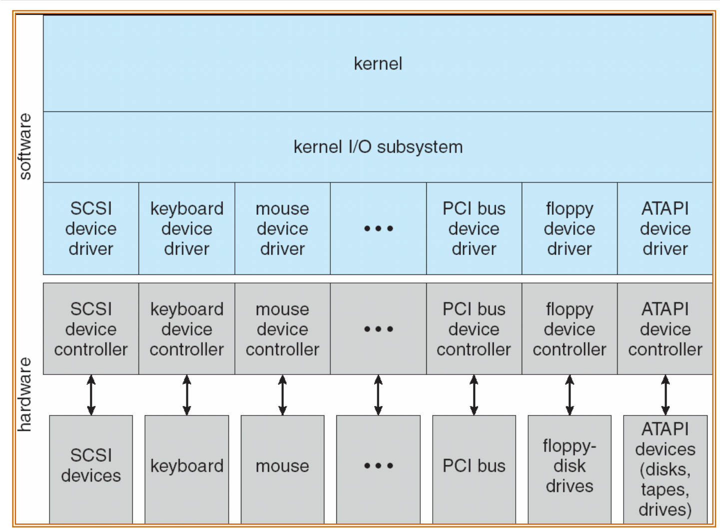
Physical Address Space

Transferring Data To/From Controller

- Programmed I/O:
  - Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size
- Direct Memory Access:
  - Give controller access to memory bus
  - Ask it to transfer data to/from memory directly
- Sample interaction with DMA controller (from book):



A Kernel I/O Structure



Device Drivers

- **Device Driver:** Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call
- Device Drivers typically divided into two pieces:
  - Top half: accessed in call path from system calls
    - » implements a set of standard, cross-device calls like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
    - » This is the kernel's interface to the device driver
    - » Top half will *start* I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    - » Gets input or transfers next block of output
    - » May wake sleeping threads if I/O now complete

Life Cycle of An I/O Request

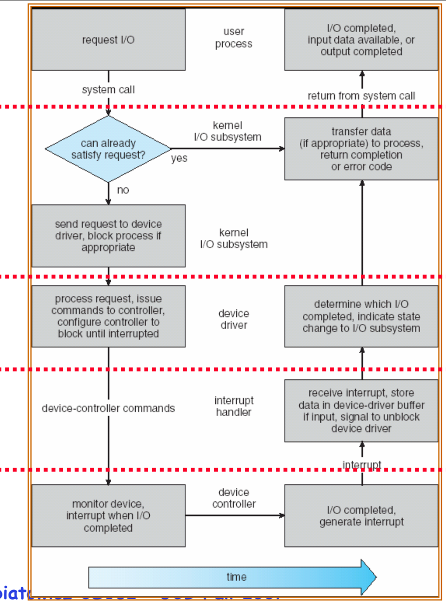
User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

Device Hardware

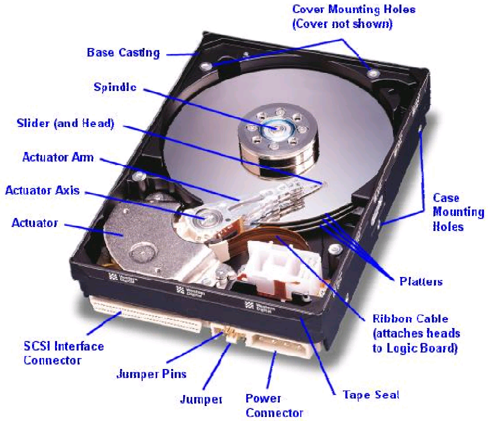


I/O Device Notifying the OS

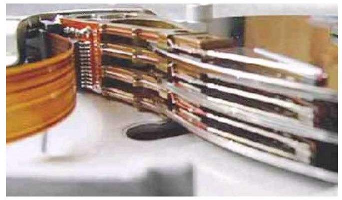
- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- **I/O Interrupt:**
  - Device generates an interrupt whenever it needs service
  - Handled in bottom half of device driver
    - » Often run on special kernel-level stack
  - Pro: handles unpredictable events well
  - Con: interrupts relatively high overhead
- **Polling:**
  - OS periodically checks a device-specific status register
    - » I/O device puts completion information in status register
    - » Could use timer to invoke lower half of drivers occasionally
  - Pro: low overhead
  - Con: may waste many cycles on polling if infrequent or unpredictable I/O operations
- Actual devices combine both polling and interrupts
  - For instance: High-bandwidth network device:
    - » Interrupt for first incoming packet
    - » Poll for following packets until hardware empty

Administrivia

Hard Disk Drives



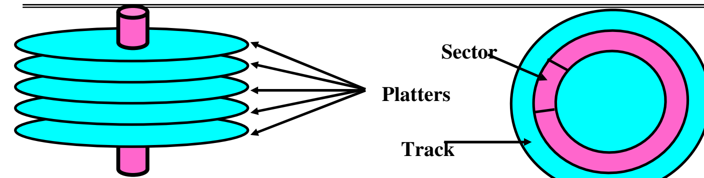
Western Digital Drive
http://www.storagereview.com/guide/



Read/Write Head
Side View

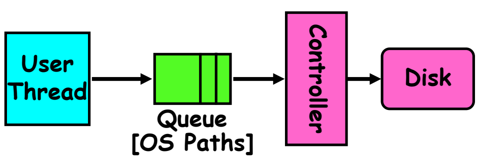
IBM/Hitachi Microdrive

Properties of a Hard Magnetic Disk

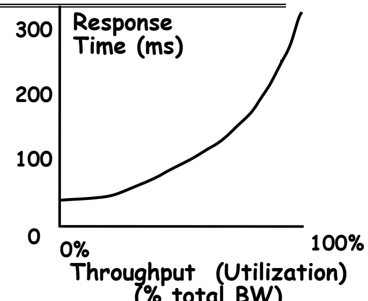


- Properties
  - Independently addressable element: sector
    - » OS always transfers groups of sectors together—"blocks"
  - A disk can access directly any given block of information it contains (random access). Can access any file either sequentially or randomly.
  - A disk can be rewritten in place: it is possible to read/modify/write a block from the disk
- Typical numbers (depending on the disk size):
  - 500 to more than 20,000 tracks per surface
  - 32 to 800 sectors per track
    - » A sector is the smallest unit that can be read or written
- Zoned bit recording
  - Constant bit density: more sectors on outer tracks
  - Speed varies with track location

Disk I/O Performance

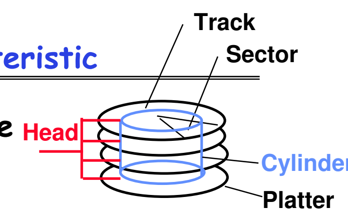


Response Time = Queue+Disk Service Time

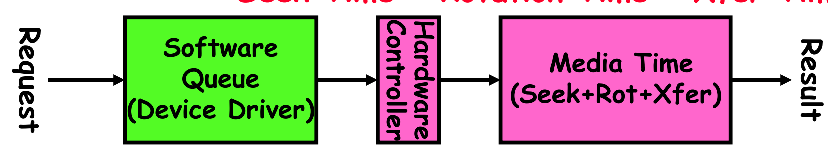


- Performance of disk drive/file system
  - Metrics: Response Time, Throughput
  - Contributing factors to latency:
    - » Software paths (can be loosely modeled by a queue)
    - » Hardware controller
    - » Physical disk media
- Queuing behavior:
  - Can lead to big increases of latency as utilization approaches 100%

Magnetic Disk Characteristic



- Cylinder: all the tracks under the head at a given point on all surface
- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track (into proper cylinder)
  - Rotational latency: wait for the desired sector to rotate under the read/write head
  - Transfer time: transfer a block of bits (sector) under the read-write head
- Disk Latency = Queueing Time + Controller time + Seek Time + Rotation Time + Xfer Time



- Highest Bandwidth:
  - Transfer large group of blocks sequentially from one track

Typical Numbers of a Magnetic Disk

- Average seek time as reported by the industry:
  - Typically in the range of 8 ms to 12 ms
  - Due to locality of disk reference may only be 25% to 33% of the advertised number
- Rotational Latency:
  - *Most* disks rotate at 3,600 to 7200 RPM (Up to 15,000RPM or more)
  - Approximately 16 ms to 8 ms per revolution, respectively
  - An average latency to the desired information is halfway around the disk: 8 ms at 3600 RPM, 4 ms at 7200 RPM
- Transfer Time is a function of:
  - Transfer size (usually a sector): 512B – 1KB per sector
  - Rotation speed: 3600 RPM to 15000 RPM
  - Recording density: bits per inch on a track
  - Diameter: ranges from 1 in to 5.25 in
  - Typical values: 2 to 50 MB per second
- Controller time depends on controller hardware
- Cost drops by factor of two per year (since 1991)

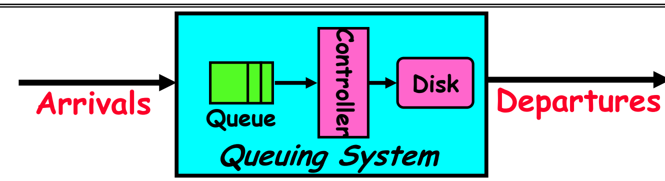
Disk Performance

- Assumptions:
  - Ignoring queuing and controller times for now
  - Avg seek time of 5ms, avg rotational delay of 4ms
  - Transfer rate of 4MByte/s, sector size of 1 KByte
- Random place on disk:
  - Seek (5ms) + Rot. Delay (4ms) + Transfer (0.25ms)
  - Roughly 10ms to fetch/put data: 100 KByte/sec
- Random place in same cylinder:
  - Rot. Delay (4ms) + Transfer (0.25ms)
  - Roughly 5ms to fetch/put data: 200 KByte/sec
- Next sector on same track:
  - Transfer (0.25ms): 4 MByte/sec
- Key to using disk effectively (esp. for filesystems) is to minimize seek and rotational delays

Disk Tradeoffs

- How do manufacturers choose disk sector sizes?
  - Need 100-1000 bits between each sector to allow system to measure how fast disk is spinning and to tolerate small (thermal) changes in track length
- What if sector was 1 byte?
  - Space efficiency – only 1% of disk has useful space
  - Time efficiency – each seek takes 10 ms, transfer rate of 50 – 100 Bytes/sec
- What if sector was 1 KByte?
  - Space efficiency – only 90% of disk has useful space
  - Time efficiency – transfer rate of 100 KByte/sec
- What if sector was 1 MByte?
  - Space efficiency – almost all of disk has useful space
  - Time efficiency – transfer rate of 4 MByte/sec

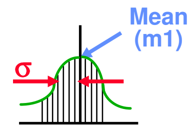
Introduction to Queuing Theory



- What about queuing time??
  - Let's apply some queuing theory
  - Queuing Theory applies to long term, steady state behavior ⇒ Arrival rate = Departure rate
- Little's Law:
  Mean # tasks in system = arrival rate x mean response time
  - Observed by many, Little was first to prove
  - Simple interpretation: you should see the same number of tasks in queue when entering as when leaving.
- Applies to any system in equilibrium, as long as nothing in black box is creating or destroying tasks
  - Typical queuing theory doesn't deal with transient behavior, only steady-state behavior

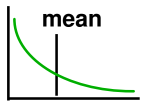
Background: Use of random distributions

- Server spends variable time with customers
  - Mean (Average) $m1 = \Sigma p(T)\times T$
  - Variance $\sigma^2 = \Sigma p(T)\times(T-m1)^2 = \Sigma p(T)\times T^2 - m1$
  - Squared coefficient of variance: $C = \sigma^2/m1^2$
    Aggregate description of the distribution.



Distribution of service times

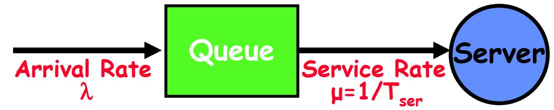
- Important values of *C*:
  - No variance or deterministic ⇒ *C*=0
  - "memoryless" or exponential ⇒ *C*=1
    - » Past tells nothing about future
    - » Many complex systems (or aggregates) well described as memoryless
  - Disk response times *C* ≈ 1.5 (majority seeks < avg)



Memoryless

A Little Queuing Theory: Some Results

- Assumptions:
  - System in equilibrium; No limit to the queue
  - Time between successive arrivals is random and memoryless



- Parameters that describe our system:
  - λ: mean number of arriving customers/second
  - T_{ser}: mean time to service a customer ("m1")
  - C: squared coefficient of variance = $\sigma^2/m1^2$
  - μ: service rate = $1/T_{ser}$
  - u: server utilization ($0\le u\le 1$): $u = \lambda/\mu = \lambda \times T_{ser}$
- Parameters we wish to compute:
  - T_q: Time spent in queue
  - L_q: Length of queue = $\lambda \times T_q$ (by Little's law)
- Results:
  - Memoryless service distribution ($C = 1$):
    - » Called M/M/1 queue: $T_q = T_{ser} \times u/(1 - u)$
  - General service distribution (no restrictions), 1 server:
    - » Called M/G/1 queue: $T_q = T_{ser} \times \frac{1}{2}(1+C) \times u/(1 - u))$

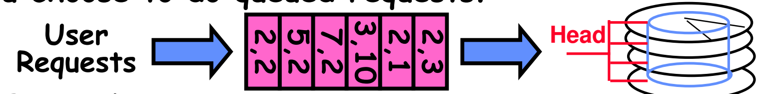
A Little Queuing Theory: An Example

- Example Usage Statistics:
  - User requests 10 x 8KB disk I/Os per second
  - Requests & service exponentially distributed (*C*=1.0)
  - Avg. service = 20 ms (From controller+seek+rot+trans)
- Questions:
  - How utilized is the disk?
    - » Ans: server utilization, $u = \lambda T_{ser}$
  - What is the average time spent in the queue?
    - » Ans: T_q
  - What is the number of requests in the queue?
    - » Ans: L_q
  - What is the avg response time for disk request?
    - » Ans: $T_{sys} = T_q + T_{ser}$
- Computation:

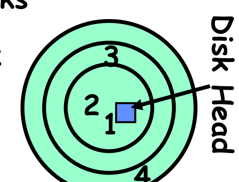
λ (avg # arriving customers/s) = 10/s
T_{ser} (avg time to service customer) = 20 ms (0.02s)
u (server utilization) = $\lambda \times T_{ser}$ = 10/s x .02s = 0.2
T_q (avg time/customer in queue) = $T_{ser} \times u/(1 - u)$
= 20 x 0.2/(1-0.2) = 20 x 0.25 = 5 ms (0 .005s)
L_q (avg length of queue) = $\lambda \times T_q$=10/s x .005s = 0.05
T_{sys} (avg time/customer in system) =$T_q + T_{ser}$= 25 ms

Disk Scheduling

- Disk can do only one request at a time; What order do you choose to do queued requests?



- FIFO Order
  - Fair among requesters, but order of arrival may be to random spots on the disk ⇒ Very long seeks
- SSTF: Shortest seek time first
  - Pick the request that's closest on the disk
  - Although called SSTF, today must include rotational delay in calculation, since rotation can be as long as seek
  - Con: SSTF good at reducing seeks, but may lead to starvation

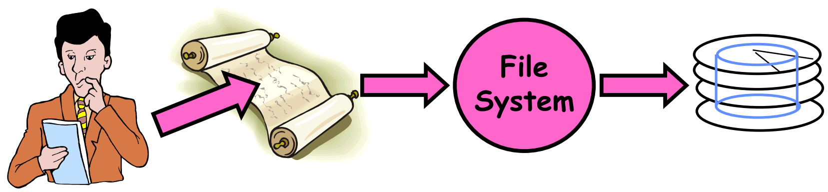


- SCAN: Implements an Elevator Algorithm: take the closest request in the direction of travel
  - No starvation, but retains flavor of SSTF
- S-SCAN: Circular-Scan: only goes in one direction
  - Skips any requests on the way back
  - Fairer than SCAN, not biased towards pages in middle

Building a File System

- **File System:** Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.
- File System Components
  - Disk Management: collecting disk blocks into files
  - Naming: Interface to find files by name, not by blocks
  - Protection: Layers to keep data secure
  - Reliability/Durability: Keeping of files durable despite crashes, media failures, attacks, etc
- User vs. System View of a File
  - User's view:
    - » Durable Data Structures
  - System's view (system call interface):
    - » Collection of Bytes (UNIX)
    - » Doesn't matter to system what kind of data structures you want to store on disk!
  - System's view (inside OS):
    - » Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
    - » Block size ≥ sector size; in UNIX, block size is 4KB

Translating from User to System View



- What happens if user says: give me bytes 2—12?
  - Fetch block corresponding to those bytes
  - Return just the correct portion of the block
- What about: write bytes 2—12?
  - Fetch block
  - Modify portion
  - Write out Block
- Everything inside File System is in whole size blocks
  - For example, `getc()`, `putc()` ⇒ buffers something like 4096 bytes, even if interface is one byte at a time
- From now on, file is a collection of blocks

Disk Management Policies

- Basic entities on a disk:
  - **File:** user-visible group of blocks arranged sequentially in logical space
  - **Directory:** user-visible index mapping names to files (next lecture)
- Access disk as linear array of sectors. Two Options:
  - Identify sectors as vectors [cylinder, surface, sector]. Sort in cylinder-major order. Not used much anymore.
  - **Logical Block Addressing (LBA).** Every sector has integer address from zero up to max number of sectors.
  - Controller translates from address ⇒ physical position
    - » First case: OS/BIOS must deal with bad sectors
    - » Second case: hardware shields OS from structure of disk
- Need way to track free disk blocks
  - Link free blocks together ⇒ too slow today
  - Use bitmap to represent free space on disk
- Need way to structure files: **File Header**
  - Track which blocks belong at which offsets within the logical file structure
  - **Optimize placement of files' disk blocks to match access and usage patterns**

Designing the File System: Access Patterns

- How do users access files?
  - Need to know type of access patterns user is likely to throw at system
- Sequential Access: bytes read in order ("give me the next X bytes, then give me next, etc")
  - Almost all file access are of this flavor
- Random Access: read/write element out of middle of array ("give me bytes i—j")
  - Less frequent, but still important. For example, virtual memory backing file: page of memory stored in file
  - Want this to be fast – don't want to have to read all bytes to get to the middle of the file
- Content-based Access: ("find me 100 bytes starting with JOSEPH")
  - Example: employee records – once you find the bytes, increase my salary by a factor of 2
  - Many systems don't provide this; instead, databases are built on top of disk access to index content (requires efficient random access)

Designing the File System: Usage Patterns

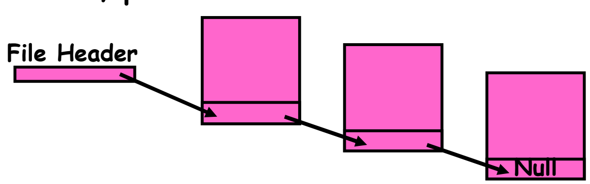
- Most files are small (for example, .login, .c files)
  - A few files are big – nachos, core files, etc.; the nachos executable is as big as all of your .class files combined
  - However, most files are small – .class's, .o's, .c's, etc.
- Large files use up most of the disk space and bandwidth to/from disk
  - May seem contradictory, but a few enormous files are equivalent to an immense # of small files
- Although we will use these observations, beware usage patterns:
  - Good idea to look at usage patterns: beat competitors by optimizing for frequent patterns
  - Except: changes in performance or cost can alter usage patterns. Maybe UNIX has lots of small files because big files are really inefficient?
- Digression, danger of predicting future:
  - In 1950's, marketing study by IBM said total worldwide need for computers was 7!
  - Company (that you haven't heard of) called "GenRad" invented oscilloscope; thought there was no market, so sold patent to Tektronix (bet you have heard of them!)

How to organize files on disk

- Goals:
  - Maximize sequential performance
  - Easy random access to file
  - Easy management of file (growth, truncation, etc)
- First Technique: Continuous Allocation
  - Use continuous range of blocks in logical block space
    - » Analogous to base+bounds in virtual memory
    - » User says in advance how big file will be (disadvantage)
  - Search bit-map for space using best fit/first fit
    - » What if not enough contiguous space for new file?
  - File Header Contains:
    - » First block/LBA in file
    - » File size (# of blocks)
  - Pros: Fast Sequential Access, Easy Random access
  - Cons: External Fragmentation/Hard to grow files
    - » Free holes get smaller and smaller
    - » Could compact space, but that would be *really* expensive
- Continuous Allocation used by IBM 360
  - Result of allocation and management cost: People would create a big file, put their file in the middle

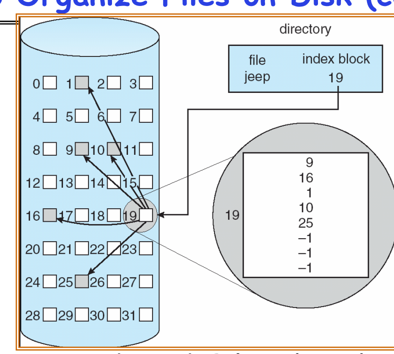
How to organize files on disk (continued)

- Second Technique: Linked List Approach
  - Each block, pointer to next on disk



  - Pros: Can grow files dynamically, Free list same as file
  - Cons: Bad Sequential Access (seek between each block), Unreliable (lose block, lose rest of file)
  - Serious Con: Bad random access!!!!
  - Technique originally from Alto (First PC, built at Xerox)
    - » No attempt to allocate contiguous blocks
- MSDOS used a similar linked approach
  - Links not in pages, but in the File Allocation Table (FAT)
    - » FAT contains an entry for each block on the disk
    - » FAT Entries corresponding to blocks of file linked together
  - Compare with Linked List Approach:
    - » Sequential access costs more unless FAT cached in memory
    - » Random access is better if FAT cached in memory

How to Organize Files on Disk (continued)



- Third Technique: Indexed Files (Nachos, VMS)
  - System Allocates file header block to hold array of pointers big enough to point to all blocks
    - » User pre-declares max file size;
  - Pros: Can easily grow up to space allocated for index Random access is fast
  - Cons: Clumsy to grow file bigger than table size Still lots of seeks: blocks may be spread over disk

Where do we still have to go?

- Still don't have good internal file structure
  - Want to minimize seeks, maximize sequential access
  - Want to be able to handle small and large files efficiently
- Don't yet know how to name/locate files
  - What is a directory?
  - How do we look up files?
- Don't yet know how to make file system fast
  - Must figure out how to use caching
- Will address these issues next time….

Summary

- I/O Controllers: Hardware that controls actual device
  - Processor Accesses through I/O instructions, load/store to special physical memory
  - Report their results through either interrupts or a status register that processor looks at occasionally (polling)
- Disk Performance:
  - Queuing time + Controller + Seek + Rotational + Transfer
  - Rotational latency: on average ½ rotation
  - Transfer time: spec of disk depends on rotation speed and bit storage density
- Queuing Latency:
  - M/M/1 and M/G/1 queues: simplest to analyze
  - As utilization approaches 100%, latency $\rightarrow \infty$

$$T_q = T_{ser} \times \tfrac{1}{2}(1+C) \times u/(1 - u))$$

- File System:
  - Transforms blocks into Files and Directories
  - Optimize for access and usage patterns
  - Maximize sequential access, allow efficient random access

CS162
Operating Systems and Systems Programming
Lecture 18

File Systems, Naming, and Directories

October 31, 2007

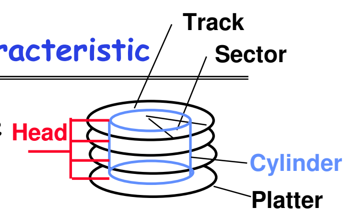
Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

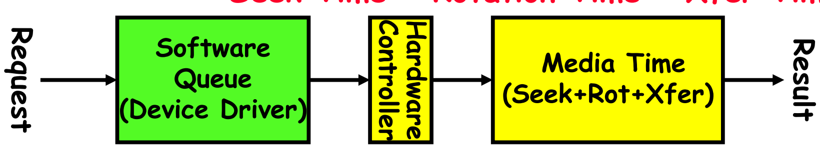
Review: Device Drivers

- **Device Driver:** Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call
- Device Drivers typically divided into two pieces:
  - Top half: accessed in call path from system calls
    - » implements a set of standard, cross-device calls like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
    - » This is the kernel's interface to the device driver
    - » Top half will *start* I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    - » Gets input or transfers next block of output
    - » May wake sleeping threads if I/O now complete

Review: Magnetic Disk Characteristic



- Cylinder: all the tracks under the head at a given point on all surface
- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track (into proper cylinder)
  - Rotational latency: wait for the desired sector to rotate under the read/write head
  - Transfer time: transfer a block of bits (sector) under the read-write head
- Disk Latency = Queueing Time + Controller time + Seek Time + Rotation Time + Xfer Time



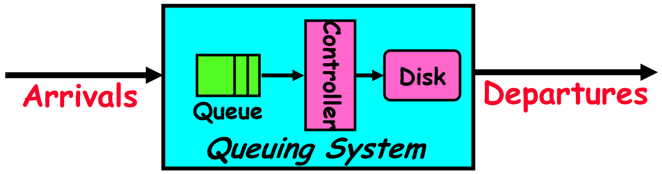
- Highest Bandwidth:
  - transfer large group of blocks sequentially from one track

Goals for Today

- Queuing Theory
- File Systems
  - Structure, Naming, Directories

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

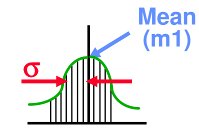
Introduction to Queuing Theory



- What about queuing time??
  - Let's apply some queuing theory
  - Queuing Theory applies to long term, steady state behavior ⇒ Arrival rate = Departure rate
- Little's Law:
  Mean # tasks in system = arrival rate x mean response time
  - Observed by many, Little was first to prove
  - Simple interpretation: you should see the same number of tasks in queue when entering as when leaving.
- Applies to any system in equilibrium, as long as nothing in black box is creating or destroying tasks
  - Typical queuing theory doesn't deal with transient behavior, only steady-state behavior

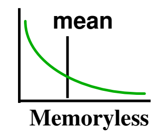
Background: Use of random distributions

- Server spends variable time with customers
  - Mean (Average) m1 = $\Sigma p(T) \times T$
  - Variance $\sigma^2 = \Sigma p(T) \times (T-m1)^2 = \Sigma p(T) \times T^2 - m1$
  - Squared coefficient of variance: $C = \sigma^2/m1^2$
  Aggregate description of the distribution.



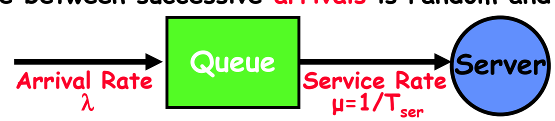
Distribution of service times

- Important values of C:
  - No variance or deterministic ⇒ C=0
  - "memoryless" or exponential ⇒ C=1
    » Past tells nothing about future
    » Many complex systems (or aggregates) well described as memoryless
  - Disk response times C ≈ 1.5 (wider variance ⇒ long tail)



A Little Queuing Theory: Some Results

- Assumptions:
  - System in equilibrium; No limit to the queue
  - Time between successive arrivals is random and memoryless



- Parameters that describe our system:
  - λ: mean number of arriving customers/second
  - T_{ser}: mean time to service a customer ("m1")
  - C: squared coefficient of variance = $\sigma^2/m1^2$
  - μ: service rate = $1/T_{ser}$
  - u: server utilization ($0 \le u \le 1$): $u = \lambda/\mu = \lambda \times T_{ser}$
- Parameters we wish to compute:
  - T_q: Time spent in queue
  - L_q: Length of queue = $\lambda \times T_q$ (by Little's law)
- Results:
  - Memoryless service distribution ($C = 1$):
    » Called M/M/1 queue: $T_q = T_{ser} \times u/(1 - u)$
  - General service distribution (no restrictions), 1 server:
    » Called M/G/1 queue: $T_q = T_{ser} \times \frac{1}{2}(1+C) \times u/(1 - u)$

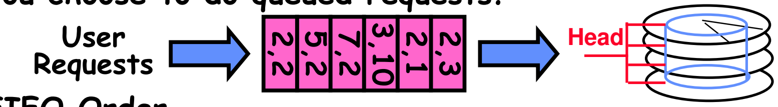
A Little Queuing Theory: An Example

- Example Usage Statistics:
  - User requests 10 x 8KB disk I/Os per second
  - Requests & service exponentially distributed (C=1.0)
  - Avg. service = 20 ms (From controller+seek+rot+trans)
- Questions:
  - How utilized is the disk?
    » Ans: server utilization, $u = \lambda T_{ser}$
  - What is the average time spent in the queue?
    » Ans: T_q
  - What is the number of requests in the queue?
    » Ans: $L_q = \lambda T_q$ (Little's law)
  - What is the avg response time for disk request?
    » Ans: $T_{sys} = T_q + T_{ser}$
- Computation:
  - λ (avg # arriving customers/s) = 10/s
  - T_{ser} (avg time to service customer) = 20 ms (0.02s)
  - u (server utilization) = $\lambda \times T_{ser}$ = 10/s x .02s = 0.2
  - T_q (avg time/customer in queue) = $T_{ser} \times u/(1 - u)$ = 20 x 0.2/(1-0.2) = 20 x 0.25 = 5 ms (0 .005s)
  - L_q (avg length of queue) = $\lambda \times T_q$=10/s x .005s = 0.05
  - T_{sys} (avg time/customer in system) =$T_q + T_{ser}$= 25 ms

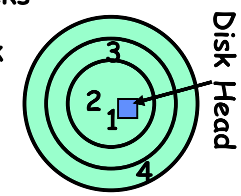
Administrivia

Disk Scheduling

- Disk can do only one request at a time; What order do you choose to do queued requests?



- FIFO Order
  - Fair among requesters, but order of arrival may be to random spots on the disk ⇒ Very long seeks
- SSTF: Shortest seek time first
  - Pick the request that's closest on the disk
  - Although called SSTF, today must include rotational delay in calculation, since rotation can be as long as seek
  - Con: SSTF good at reducing seeks, but may lead to starvation

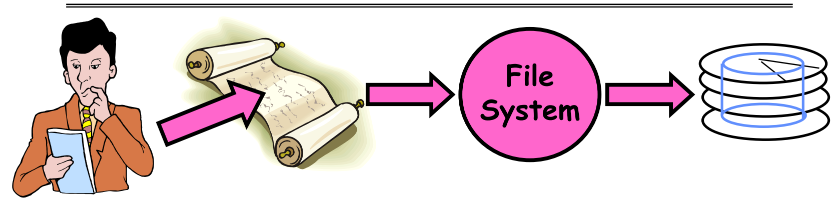


- SCAN: Implements an Elevator Algorithm: take the closest request in the direction of travel
  - No starvation, but retains flavor of SSTF
- C-SCAN: Circular-Scan: only goes in one direction
  - Skips any requests on the way back
  - Fairer than SCAN, not biased towards pages in middle

Building a File System

- **File System:** Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.
- File System Components
  - Disk Management: collecting disk blocks into files
  - Naming: Interface to find files by name, not by blocks
  - Protection: Layers to keep data secure
  - Reliability/Durability: Keeping of files durable despite crashes, media failures, attacks, etc
- User vs. System View of a File
  - User's view:
    - » Durable Data Structures
  - System's view (system call interface):
    - » Collection of Bytes (UNIX)
    - » Doesn't matter to system what kind of data structures you want to store on disk!
  - System's view (inside OS):
    - » Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
    - » Block size ≥ sector size; in UNIX, block size is 4KB

Translating from User to System View



- What happens if user says: give me bytes 2—12?
  - Fetch block corresponding to those bytes
  - Return just the correct portion of the block
- What about: write bytes 2—12?
  - Fetch block
  - Modify portion
  - Write out Block
- Everything inside File System is in whole size blocks
  - For example, `getc()`, `putc()` ⇒ buffers something like 4096 bytes, even if interface is one byte at a time
- From now on, file is a collection of blocks

Disk Management Policies

- Basic entities on a disk:
  - File: user-visible group of blocks arranged sequentially in logical space
  - Directory: user-visible index mapping names to files (next lecture)
- Access disk as linear array of sectors. Two Options:
  - Identify sectors as vectors [cylinder, surface, sector]. Sort in cylinder-major order. Not used much anymore.
  - Logical Block Addressing (LBA). Every sector has integer address from zero up to max number of sectors.
  - Controller translates from address ⇒ physical position
    - » First case: OS/BIOS must deal with bad sectors
    - » Second case: hardware shields OS from structure of disk
- Need way to track free disk blocks
  - Link free blocks together ⇒ too slow today
  - Use bitmap to represent free space on disk
- Need way to structure files: File Header
  - Track which blocks belong at which offsets within the logical file structure
  - Optimize placement of files' disk blocks to match access and usage patterns

Designing the File System: Access Patterns

- How do users access files?
  - Need to know type of access patterns user is likely to throw at system
- Sequential Access: bytes read in order ("give me the next X bytes, then give me next, etc")
  - Almost all file access are of this flavor
- Random Access: read/write element out of middle of array ("give me bytes i—j")
  - Less frequent, but still important. For example, virtual memory backing file: page of memory stored in file
  - Want this to be fast – don't want to have to read all bytes to get to the middle of the file
- Content-based Access: ("find me 100 bytes starting with KUBI")
  - Example: employee records – once you find the bytes, increase my salary by a factor of 2
  - Many systems don't provide this; instead, databases are built on top of disk access to index content (requires efficient random access)

Designing the File System: Usage Patterns

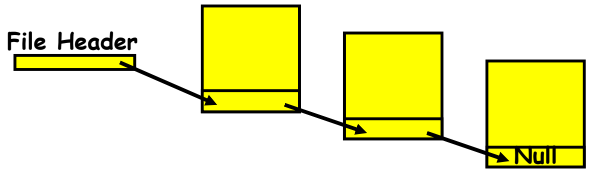
- Most files are small (for example, .login, .c files)
  - A few files are big – nachos, core files, etc.; the nachos executable is as big as all of your .class files combined
  - However, most files are small – .class's, .o's, .c's, etc.
- Large files use up most of the disk space and bandwidth to/from disk
  - May seem contradictory, but a few enormous files are equivalent to an immense # of small files
- Although we will use these observations, beware usage patterns:
  - Good idea to look at usage patterns: beat competitors by optimizing for frequent patterns
  - Except: changes in performance or cost can alter usage patterns. Maybe UNIX has lots of small files because big files are really inefficient?
- Digression, danger of predicting future:
  - In 1950's, marketing study by IBM said total worldwide need for computers was 7!
  - Company (that you haven't heard of) called "GenRad" invented oscilloscope; thought there was no market, so sold patent to Tektronix (bet you have heard of them!)

How to organize files on disk

- Goals:
  - Maximize sequential performance
  - Easy random access to file
  - Easy management of file (growth, truncation, etc)
- First Technique: Continuous Allocation
  - Use continuous range of blocks in logical block space
    - » Analogous to base+bounds in virtual memory
    - » User says in advance how big file will be (disadvantage)
  - Search bit-map for space using best fit/first fit
    - » What if not enough contiguous space for new file?
  - File Header Contains:
    - » First block/LBA in file
    - » File size (# of blocks)
  - Pros: Fast Sequential Access, Easy Random access
  - Cons: External Fragmentation/Hard to grow files
    - » Free holes get smaller and smaller
    - » Could compact space, but that would be *really* expensive
- Continuous Allocation used by IBM 360
  - Result of allocation and management cost: People would create a big file, put their file in the middle

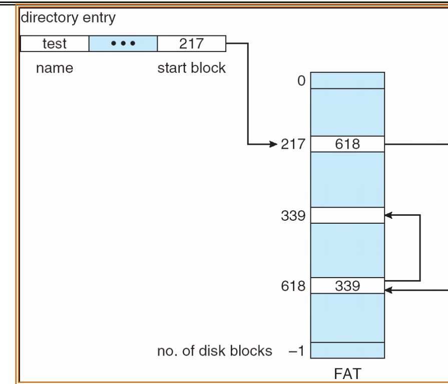
Linked List Allocation

- Second Technique: Linked List Approach
  - Each block, pointer to next on disk



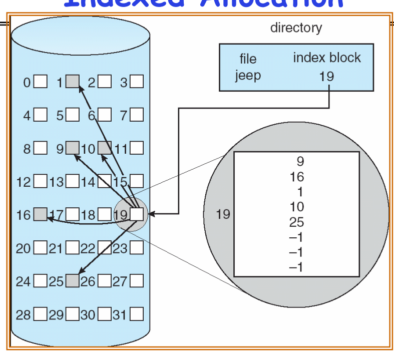
  - Pros: Can grow files dynamically, Free list same as file
  - Cons: Bad Sequential Access (seek between each block), Unreliable (lose block, lose rest of file)
  - Serious Con: Bad random access!!!!
  - Technique originally from Alto (First PC, built at Xerox)
    - » No attempt to allocate contiguous blocks

Linked Allocation: File-Allocation Table (FAT)



- MSDOS links pages together to create a file
  - Links not in pages, but in the File Allocation Table (FAT)
    - » FAT contains an entry for each block on the disk
    - » FAT Entries corresponding to blocks of file linked together
  - Access properties:
    - » Sequential access expensive unless FAT cached in memory
    - » Random access expensive always, but *really* expensive if FAT not cached in memory

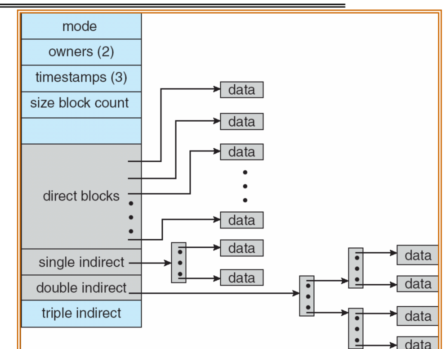
Indexed Allocation



- Third Technique: Indexed Files (Nachos, VMS)
  - System Allocates file header block to hold array of pointers big enough to point to all blocks
    - » User pre-declares max file size;
  - Pros: Can easily grow up to space allocated for index
    Random access is fast
  - Cons: Clumsy to grow file bigger than table size
    Still lots of seeks: blocks may be spread over disk

Multilevel Indexed Files (UNIX 4.1)

- Multilevel Indexed Files: Like multilevel address translation (from UNIX 4.1 BSD)
  - Key idea: efficient for small files, but still allow big files

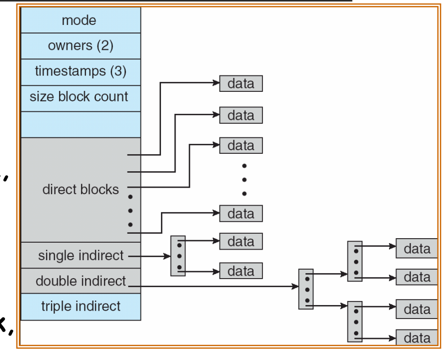


- File hdr contains 13 pointers
  - Fixed size table, pointers not all equivalent
  - This header is called an "inode" in UNIX
- File Header format:
  - First 10 pointers are to data blocks
  - Ptr 11 points to "indirect block" containing 256 block ptrs
  - Pointer 12 points to "doubly indirect block" containing 256 indirect block ptrs for total of 64K blocks
  - Pointer 13 points to a triply indirect block (16M blocks)

Multilevel Indexed Files (UNIX 4.1): Discussion

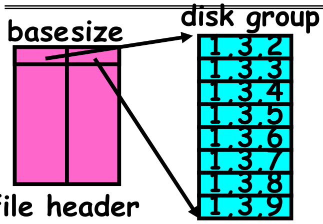
- Basic technique places an upper limit on file size that is approximately 16Gbytes
  - Designers thought this was bigger than anything anyone would need. Much bigger than a disk at the time…
  - Fallacy: today, EOS producing 2TB of data per day
- Pointers get filled in dynamically: need to allocate indirect block only when file grows > 10 blocks
  - On small files, no indirection needed

Example of Multilevel Indexed Files



- Sample file in multilevel indexed format:
  - How many accesses for block #23? (assume file header accessed on open)?
    - » Two: One for indirect block, one for data
  - How about block #5?
    - » One: One for data
  - Block #340?
    - » Three: double indirect block, indirect block, and data
- UNIX 4.1 Pros and cons
  - Pros: Simple (more or less)
    Files can easily expand (up to a point)
    Small files particularly cheap and easy
  - Cons: Lots of seeks
    Very large files must read many indirect blocks (four I/Os per block!)

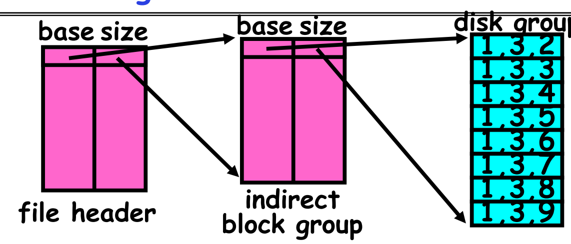
File Allocation for Cray-1 DEMOS



Basic Segmentation Structure:
Each segment contiguous on disk

- DEMOS: File system structure similar to segmentation
  - Idea: reduce disk seeks by
    - » using contiguous allocation in normal case
    - » but allow flexibility to have non-contiguous allocation
  - Cray-1 had 12ns cycle time, so CPU:disk speed ratio about the same as today (a few million instructions per seek)
- Header: table of base & size (10 "block group" pointers)
  - Each block chunk is a contiguous group of disk blocks
  - Sequential reads within a block chunk can proceed at high speed – similar to continuous allocation
- How do you find an available block group?
  - Use freelist bitmap to find block of 0's.

Large File Version of DEMOS



- What if need much bigger files?
  - If need more than 10 groups, set flag in header: BIGFILE
    - » Each table entry now points to an indirect block group
  - Suppose 1000 blocks in a block group $\Rightarrow$ 80GB max file
    - » Assuming 8KB blocks, 8byte entries$\Rightarrow$
      (10 ptrs$\times$1024 groups/ptr$\times$1000 blocks/group)*8K =80GB
- Discussion of DEMOS scheme
  - Pros: Fast sequential access, Free areas merge simply
    Easy to find free block groups (when disk not full)
  - Cons: Disk full $\Rightarrow$ No long runs of blocks (fragmentation), so high overhead allocation/access
  - Full disk $\Rightarrow$ worst of 4.1BSD (lots of seeks) with worst of continuous allocation (lots of recompaction needed)

How to keep DEMOS performing well?

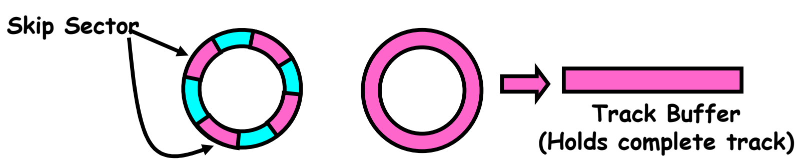
- In many systems, disks are always full
  - CS department growth: 300 GB to 1TB in a year
    - » That's 2GB/day! (Now at 3—4 TB!)
  - How to fix? Announce that disk space is getting low, so please delete files?
    - » Don't really work: people try to store their data faster
  - Sidebar: Perhaps we are getting out of this mode with new disks… However, let's assume disks full for now
- Solution:
  - Don't let disks get completely full: reserve portion
    - » Free count = # blocks free in bitmap
    - » Scheme: Don't allocate data if count < reserve
  - How much reserve do you need?
    - » In practice, 10% seems like enough
  - Tradeoff: pay for more disk, get contiguous allocation
    - » Since seeks so expensive for performance, this is a very good tradeoff

UNIX BSD 4.2

- Same as BSD 4.1 (same file header and triply indirect blocks), except incorporated ideas from DEMOS:
  - Uses bitmap allocation in place of freelist
  - Attempt to allocate files contiguously
  - 10% reserved disk space
  - Skip-sector positioning (mentioned next slide)
- Problem: When create a file, don't know how big it will become (in UNIX, most writes are by appending)
  - How much contiguous space do you allocate for a file?
  - In Demos, power of 2 growth: once it grows past 1MB, allocate 2MB, etc
  - In BSD 4.2, just find some range of free blocks
    - » Put each new file at the front of different range
    - » To expand a file, you first try successive blocks in bitmap, then choose new range of blocks
  - Also in BSD 4.2: store files from same directory near each other

Attack of the Rotational Delay

- Problem 2: Missing blocks due to rotational delay
  - Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block! Need 1 revolution/block!



  - Solution1: Skip sector positioning ("interleaving")
    - » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
  - Solution2: Read ahead: read next block right after first, even if application hasn't asked for it yet.
    - » This can be done either by OS (read ahead)
    - » By disk itself (track buffers). Many disk controllers have internal RAM that allows them to read a complete track
- Important Aside: Modern disks+controllers do many complex things "under the covers"
  - Track buffers, elevator algorithms, bad block filtering

How do we actually access files?

- All information about a file contained in its file header
  - UNIX calls this an "inode"
    - » Inodes are global resources identified by index ("inumber")
  - Once you load the header structure, all the other blocks of the file are locatable
- Question: how does the user ask for a particular file?
  - One option: user specifies an inode by a number (index).
    - » Imagine: open("14553344")
  - Better option: specify by textual name
    - » Have to map name→inumber
  - Another option: Icon
    - » This is how Apple made its money. Graphical user interfaces. Point to a file and click.
- Naming: The process by which a system translates from user-visible names to system resources
  - In the case of files, need to translate from strings (textual names) or icons to inumbers/inodes
  - For global file systems, data may be spread over globe⇒need to translate from strings or icons to some combination of physical server location and inumber

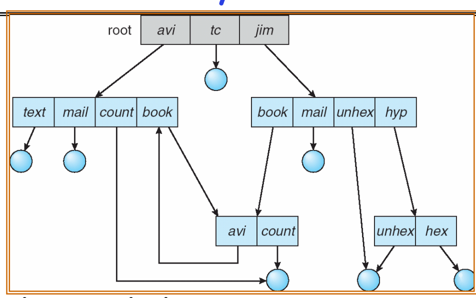
Directories

- **Directory**: a relation used for naming
  - Just a table of (file name, inumber) pairs
- How are directories constructed?
  - Directories often stored in files
    - » Reuse of existing mechanism
    - » Directory named by inode/inumber like other files
  - Needs to be quickly searchable
    - » Options: Simple list or Hashtable
    - » Can be cached into memory in easier form to search
- How are directories modified?
  - Originally, direct read/write of special file
  - System calls for manipulation: `mkdir`, `rmdir`
  - Ties to file creation/destruction
    - » On creating a file by name, new inode grabbed and associated with new file in particular directory

Directory Organization

- Directories organized into a hierarchical structure
  - Seems standard, but in early 70's it wasn't
  - Permits much easier organization of data structures
- Entries in directory can be either files or directories
- Files named by ordered set (e.g., /programs/p/list)

Directory Structure



- Not really a hierarchy!
  - Many systems allow directory structure to be organized as an acyclic graph or even a (potentially) cyclic graph
  - Hard Links: different names for the same file
    - » Multiple directory entries point at the same file
  - Soft Links: "shortcut" pointers to other files
    - » Implemented by storing the logical name of actual file
- **Name Resolution:** The process of converting a logical name into a physical resource (like a file)
  - Traverse succession of directories until reach target file
  - Global file system: May be spread across the network

Directory Structure (Con't)

- How many disk accesses to resolve "/my/book/count"?
  - Read in file header for root (fixed spot on disk)
  - Read in first data bock for root
    - » Table of file name/index pairs. Search linearly – ok since directories typically very small
  - Read in file header for "my"
  - Read in first data block for "my"; search for "book"
  - Read in file header for "book"
  - Read in first data block for "book"; search for "count"
  - Read in file header for "count"
- **Current working directory:** Per-address-space pointer to a directory (inode) used for resolving file names
  - Allows user to specify relative filename instead of absolute path (say CWD="/my/book" can resolve "count")

Where are inodes stored?

- In early UNIX and DOS/Windows' FAT file system, headers stored in special array in outermost cylinders
  - Header not stored anywhere near the data blocks. To read a small file, seek to get header, see back to data.
  - Fixed size, set when disk is formatted. At formatting time, a fixed number of inodes were created (They were each given a unique number, called an "inumber")

Where are inodes stored?

- Later versions of UNIX moved the header information to be closer to the data blocks
  - Often, inode for file stored in same "cylinder group" as parent directory of the file (makes an ls of that directory run fast).
  - Pros:
    - » Reliability: whatever happens to the disk, you can find all of the files (even if directories might be disconnected)
    - » UNIX BSD 4.2 puts a portion of the file header array on each cylinder. For small directories, can fit all data, file headers, etc in same cylinder⇒no seeks!
    - » File headers much smaller than whole block (a few hundred bytes), so multiple headers fetched from disk at same time

Summary

- Queuing Latency:
  - M/M/1 and M/G/1 queues: simplest to analyze
  - As utilization approaches 100%, latency → ∞

$$T_q = T_{ser} \times \tfrac{1}{2}(1+C) \times u/(1 - u))$$

- File System:
  - Transforms blocks into Files and Directories
  - Optimize for access and usage patterns
  - Maximize sequential access, allow efficient random access
- File (and directory) defined by header
  - Called "inode" with index called "inumber"
- Multilevel Indexed Scheme
  - Inode contains file info, direct pointers to blocks,
  - indirect blocks, doubly indirect, etc..
- DEMOS:
  - CRAY-1 scheme like segmentation
  - Emphsized contiguous allocation of blocks, but allowed to use non-contiguous allocation when necessary
- Naming: the process of turning user-visible names into resources (such as files)

CS162 Operating Systems and Systems Programming Lecture 19

File Systems continued Distributed Systems

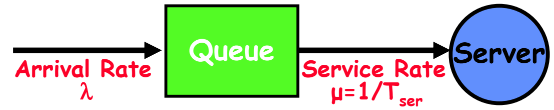
November 5, 2007

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

Review: A Little Queuing Theory: Some Results

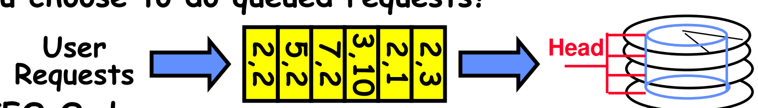
- Assumptions:
  - System in equilibrium; No limit to the queue
  - Time between successive arrivals is random and memoryless



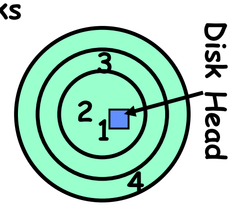
- Parameters that describe our system:
  - λ: mean number of arriving customers/second
  - T_{ser}: mean time to service a customer ("m1")
  - C: squared coefficient of variance = $\sigma^2/m1^2$
  - μ: service rate = $1/T_{ser}$
  - u: server utilization ($0 \le u \le 1$): $u = \lambda/\mu = \lambda \times T_{ser}$
- Parameters we wish to compute:
  - T_q: Time spent in queue
  - L_q: Length of queue = $\lambda \times T_q$ (by Little's law)
- Results:
  - Memoryless service distribution (C = 1):
    - » Called M/M/1 queue: $T_q = T_{ser} \times u/(1 - u)$
  - General service distribution (no restrictions), 1 server:
    - » Called M/G/1 queue: $T_q = T_{ser} \times \frac{1}{2}(1+C) \times u/(1 - u))$

Review: Disk Scheduling

- Disk can do only one request at a time; What order do you choose to do queued requests?



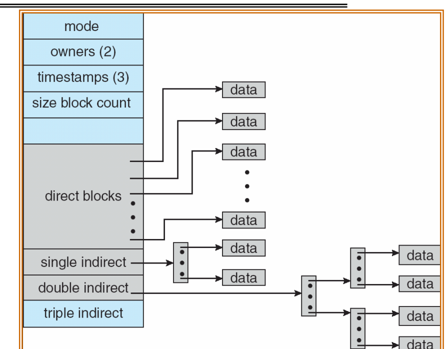
- FIFO Order
  - Fair among requesters, but order of arrival may be to random spots on the disk ⇒ Very long seeks
- SSTF: Shortest seek time first
  - Pick the request that's closest on the disk
  - Although called SSTF, today must include rotational delay in calculation, since rotation can be as long as seek
  - Con: SSTF good at reducing seeks, but may lead to starvation



- SCAN: Implements an Elevator Algorithm: take the closest request in the direction of travel
  - No starvation, but retains flavor of SSTF
- C-SCAN: Circular-Scan: only goes in one direction
  - Skips any requests on the way back
  - Fairer than SCAN, not biased towards pages in middle

Review: Multilevel Indexed Files (UNIX 4.1)

- Multilevel Indexed Files: Like multilevel address translation (from UNIX 4.1 BSD)
  - Key idea: efficient for small files, but still allow big files



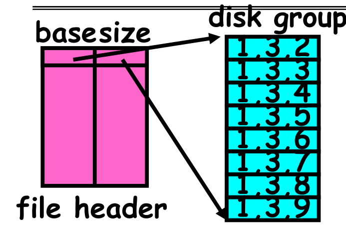
- File hdr contains 13 pointers
  - Fixed size table, pointers not all equivalent
  - This header is called an "inode" in UNIX
- File Header format:
  - First 10 pointers are to data blocks
  - Ptr 11 points to "indirect block" containing 256 block ptrs
  - Pointer 12 points to "doubly indirect block" containing 256 indirect block ptrs for total of 64K blocks
  - Pointer 13 points to a triply indirect block (16M blocks)

Goals for Today

- Finish Discussion of File Systems
  - Structure, Naming, Directories
- File Caching
- Data Durability
- Beginning of Distributed Systems Discussion

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

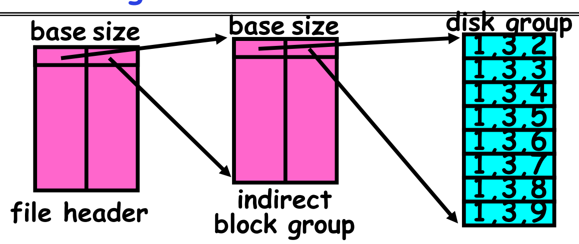
Review: File Allocation for Cray-1 DEMOS



Basic Segmentation Structure: Each segment contiguous on disk

- DEMOS: File system structure similar to segmentation
  - Idea: reduce disk seeks by
    - » using contiguous allocation in normal case
    - » but allow flexibility to have non-contiguous allocation
  - Cray-1 had 12ns cycle time, so CPU:disk speed ratio about the same as today (a few million instructions per seek)
- Header: table of base & size (10 "block group" pointers)
  - Each block chunk is a contiguous group of disk blocks
  - Sequential reads within a block chunk can proceed at high speed – similar to continuous allocation
- How do you find an available block group?
  - Use freelist bitmap to find block of 0's.

Large File Version of DEMOS



- What if need much bigger files?
  - If need more than 10 groups, set flag in header: BIGFILE
    - » Each table entry now points to an indirect block group
  - Suppose 1000 blocks in a block group ⇒ 80GB max file
    - » Assuming 8KB blocks, 8byte entries⇒ (10 ptrs×1024 groups/ptr×1000 blocks/group)*8K =80GB
- Discussion of DEMOS scheme
  - Pros: Fast sequential access, Free areas merge simply
    Easy to find free block groups (when disk not full)
  - Cons: Disk full ⇒ No long runs of blocks (fragmentation), so high overhead allocation/access
  - Full disk ⇒ worst of 4.1BSD (lots of seeks) with worst of continuous allocation (lots of recompaction needed)

How to keep DEMOS performing well?

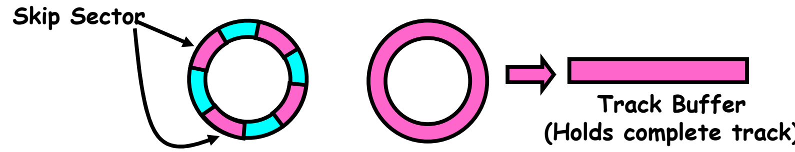
- In many systems, disks are always full
  - CS department growth: 300 GB to 1TB in a year
    - » That's 2GB/day! (Now at 3—4 TB!)
  - How to fix? Announce that disk space is getting low, so please delete files?
    - » Don't really work: people try to store their data faster
  - Sidebar: Perhaps we are getting out of this mode with new disks… However, let's assume disks full for now
- Solution:
  - Don't let disks get completely full: reserve portion
    - » Free count = # blocks free in bitmap
    - » Scheme: Don't allocate data if count < reserve
  - How much reserve do you need?
    - » In practice, 10% seems like enough
  - Tradeoff: pay for more disk, get contiguous allocation
    - » Since seeks so expensive for performance, this is a very good tradeoff

UNIX BSD 4.2

- Same as BSD 4.1 (same file header and triply indirect blocks), except incorporated ideas from DEMOS:
  - Uses bitmap allocation in place of freelist
  - Attempt to allocate files contiguously
  - 10% reserved disk space
  - Skip-sector positioning (mentioned next slide)
- Problem: When create a file, don't know how big it will become (in UNIX, most writes are by appending)
  - How much contiguous space do you allocate for a file?
  - In Demos, power of 2 growth: once it grows past 1MB, allocate 2MB, etc
  - In BSD 4.2, just find some range of free blocks
    - » Put each new file at the front of different range
    - » To expand a file, you first try successive blocks in bitmap, then choose new range of blocks
  - Also in BSD 4.2: store files from same directory near each other
- Fast File System (FFS)
  - Allocation and placement policies for BSD 4.2

Attack of the Rotational Delay

- Problem 2: Missing blocks due to rotational delay
  - Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block! Need 1 revolution/block!



  - Solution1: Skip sector positioning ("interleaving")
    - » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
  - Solution2: Read ahead: read next block right after first, even if application hasn't asked for it yet.
    - » This can be done either by OS (read ahead)
    - » By disk itself (track buffers). Many disk controllers have internal RAM that allows them to read a complete track
- Important Aside: Modern disks+controllers do many complex things "under the covers"
  - Track buffers, elevator algorithms, bad block filtering

Administrivia

How do we actually access files?

- All information about a file contained in its file header
  - UNIX calls this an "inode"
    - » Inodes are global resources identified by index ("inumber")
  - Once you load the header structure, all the other blocks of the file are locatable
- Question: how does the user ask for a particular file?
  - One option: user specifies an inode by a number (index).
    - » Imagine: open("14553344")
  - Better option: specify by textual name
    - » Have to map name→inumber
  - Another option: Icon
    - » This is how Apple made its money. Graphical user interfaces. Point to a file and click.
- Naming: The process by which a system translates from user-visible names to system resources
  - In the case of files, need to translate from strings (textual names) or icons to inumbers/inodes
  - For global file systems, data may be spread over globe⇒need to translate from strings or icons to some combination of physical server location and inumber

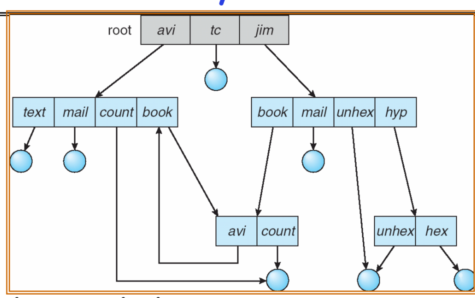
Directories

- **Directory**: a relation used for naming
  - Just a table of (file name, inumber) pairs
- How are directories constructed?
  - Directories often stored in files
    - » Reuse of existing mechanism
    - » Directory named by inode/inumber like other files
  - Needs to be quickly searchable
    - » Options: Simple list or Hashtable
    - » Can be cached into memory in easier form to search
- How are directories modified?
  - Originally, direct read/write of special file
  - System calls for manipulation: `mkdir`, `rmdir`
  - Ties to file creation/destruction
    - » On creating a file by name, new inode grabbed and associated with new file in particular directory

Directory Organization

- Directories organized into a hierarchical structure
  - Seems standard, but in early 70's it wasn't
  - Permits much easier organization of data structures
- Entries in directory can be either files or directories
- Files named by ordered set (e.g., /programs/p/list)

Directory Structure



- Not really a hierarchy!
  - Many systems allow directory structure to be organized as an acyclic graph or even a (potentially) cyclic graph
  - Hard Links: different names for the same file
    - » Multiple directory entries point at the same file
  - Soft Links: "shortcut" pointers to other files
    - » Implemented by storing the logical name of actual file
- **Name Resolution:** The process of converting a logical name into a physical resource (like a file)
  - Traverse succession of directories until reach target file
  - Global file system: May be spread across the network

Directory Structure (Con't)

- How many disk accesses to resolve "/my/book/count"?
  - Read in file header for root (fixed spot on disk)
  - Read in first data block for root
    - » Table of file name/index pairs. Search linearly – ok since directories typically very small
  - Read in file header for "my"
  - Read in first data block for "my"; search for "book"
  - Read in file header for "book"
  - Read in first data block for "book"; search for "count"
  - Read in file header for "count"
- **Current working directory:** Per-address-space pointer to a directory (inode) used for resolving file names
  - Allows user to specify relative filename instead of absolute path (say CWD="/my/book" can resolve "count")

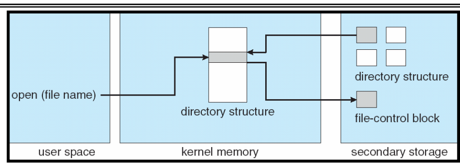
Where are inodes stored?

- In early UNIX and DOS/Windows' FAT file system, headers stored in special array in outermost cylinders
  - Header not stored near the data blocks. To read a small file, seek to get header, seek back to data.
  - Fixed size, set when disk is formatted. At formatting time, a fixed number of inodes were created (They were each given a unique number, called an "inumber")

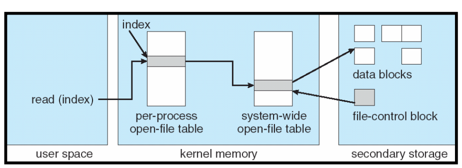
Where are inodes stored?

- Later versions of UNIX moved the header information to be closer to the data blocks
  - Often, inode for file stored in same "cylinder group" as parent directory of the file (makes an ls of that directory run fast).
  - Pros:
    - » UNIX BSD 4.2 puts a portion of the file header array on each cylinder. For small directories, can fit all data, file headers, etc in same cylinder⇒no seeks!
    - » File headers much smaller than whole block (a few hundred bytes), so multiple headers fetched from disk at same time
    - » Reliability: whatever happens to the disk, you can find many of the files (even if directories disconnected)
  - Part of the Fast File System (FFS)
    - » General optimization to avoid seeks

In-Memory File System Structures



- Open system call:
  - Resolves file name, finds file control block (inode)
  - Makes entries in per-process and system-wide tables
  - Returns index (called "file handle") in open-file table



- Read/write system calls:
  - Use file handle to locate inode
  - Perform appropriate reads or writes

File System Caching

- Key Idea: Exploit locality by caching data in memory
  - Name translations: Mapping from paths→inodes
  - Disk blocks: Mapping from block address→disk content
- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain "dirty" blocks (blocks yet on disk)
- Replacement policy? LRU
  - Can afford overhead of timestamps for each disk block
  - Advantages:
    - » Works very well for name translation
    - » Works well in general as long as memory is big enough to accommodate a host's working set of files.
  - Disadvantages:
    - » Fails when some application scans through file system, thereby flushing the cache with data used only once
    - » Example: `find . –exec grep foo {} \;`
- Other Replacement Policies?
  - Some systems allow applications to request other policies
  - Example, 'Use Once':
    - » File system can discard blocks as soon as they are used

File System Caching (con't)

- Cache Size: How much memory should the OS allocate to the buffer cache vs virtual memory?
  - Too much memory to the file system cache ⇒ won't be able to run many applications at once
  - Too little memory to file system cache ⇒ many applications may run slowly (disk caching not effective)
  - Solution: adjust boundary dynamically so that the disk access rates for paging and file access are balanced
- **Read Ahead Prefetching:** fetch sequential blocks early
  - Key Idea: exploit fact that most common file access is sequential by prefetching subsequent disk blocks ahead of current read request (if they are not already in memory)
  - Elevator algorithm can efficiently interleave groups of prefetches from concurrent applications
  - How much to prefetch?
    - » Too many imposes delays on requests by other applications
    - » Too few causes many seeks (and rotational delays) among concurrent file requests

File System Caching (con't)

- **Delayed Writes:** Writes to files not immediately sent out to disk
  - Instead, `write()` copies data from user space buffer to kernel buffer (in cache)
    - » Enabled by presence of buffer cache: can leave written file blocks in cache for a while
    - » If some other application tries to read data before written to disk, file system will read from cache
  - Flushed to disk periodically (e.g. in UNIX, every 30 sec)
  - Advantages:
    - » Disk scheduler can efficiently order lots of requests
    - » Disk allocation algorithm can be run with correct size value for a file
    - » Some files need never get written to disk! (e..g temporary scratch files written /tmp often don't exist for 30 sec)
  - Disadvantages
    - » What if system crashes before file has been written out?
    - » Worse yet, what if system crashes before a directory file has been written out? (lose pointer to inode!)

Important "ilities"

- **Availability:** the probability that the system can accept and process requests
  - Often measured in "nines" of probability. So, a 99.9% probability is considered "3-nines of availability"
  - Key idea here is independence of failures
- **Durability:** the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone
- **Reliability:** the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply availability: means that the system is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems

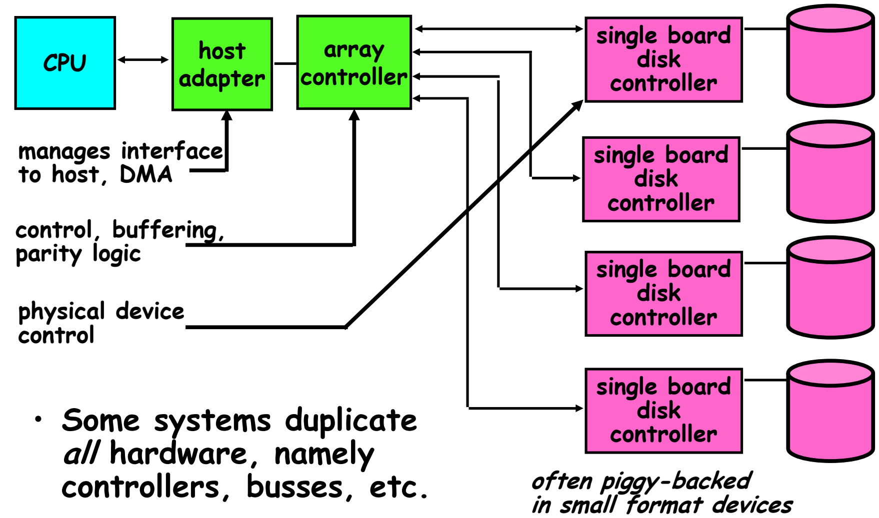
How to make file system durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
  - Can allow recovery of data from small media defects
- Make sure writes survive in short term
  - Either abandon delayed writes or
  - use special, battery-backed RAM (called non-volatile RAM or **NVRAM**) for dirty blocks in buffer cache.
- Make sure that data survives in long term
  - Need to replicate! More than one copy of data!
  - Important element: **independence of failure**
    - » Could put copies on one disk, but if disk head fails…
    - » Could put copies on different disks, but if server fails…
    - » Could put copies on different servers, but if building is struck by lightning….
    - » Could put copies on servers in different continents…
- **RAID:** Redundant Arrays of Inexpensive Disks
  - Data stored on multiple disks (redundancy)
  - Either in software or hardware
    - » In hardware case, done by disk controller; file system may not even know that there is more than one disk in use

Log Structured and Journaled File Systems

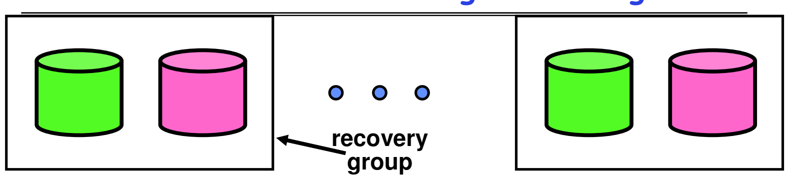
- Better reliability through use of log
  - All changes are treated as *transactions*
  - A transaction is *committed* once it is written to the log
    - » Data forced to disk for reliability
    - » Process can be accelerated with NVRAM
  - Although File system may not be updated immediately, data preserved in the log
- Difference between "Log Structured" and "Journaled"
  - In a Log Structured filesystem, data stays in log form
  - In a Journaled filesystem, Log used for recovery
- For Journaled system:
  - Log used to asynchronously update filesystem
    - » Log entries removed after used
  - After crash:
    - » Remaining transactions in the log performed ("Redo")
    - » Modifications done in way that can survive crashes
- Examples of Journaled File Systems:
  - Ext3 (Linux), XFS (Unix), etc.

Hardware RAID: Subsystem Organization



- Some systems duplicate *all* hardware, namely controllers, busses, etc.

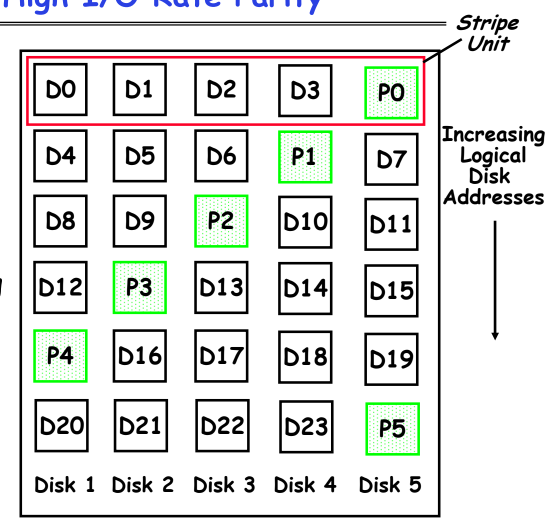
RAID 1: Disk Mirroring/Shadowing



- Each disk is fully duplicated onto its "shadow"
  - For high I/O rate, high availability environments
  - Most expensive solution: 100% capacity overhead
- Bandwidth sacrificed on write:
  - Logical write = two physical writes
  - Highest bandwidth when disk heads and rotation fully synchronized (hard to do exactly)
- Reads may be optimized
  - Can have two independent reads to same data
- Recovery:
  - Disk failure ⇒ replace disk and copy data to new disk
  - Hot Spare: idle disk already attached to system to be used for immediate replacement

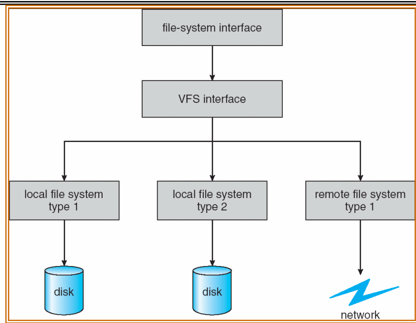
RAID 5+: High I/O Rate Parity

- Data stripped across multiple disks
  - Successive blocks stored on successive (non-parity) disks
  - Increased bandwidth over single disk
- Parity block (in green) constructed by XORing data bocks in stripe
  - $P0=D0 \oplus D1 \oplus D2 \oplus D3$
  - Can destroy any one disk and still reconstruct data
  - Suppose D3 fails, then can reconstruct: $D3=D0 \oplus D1 \oplus D2 \oplus P0$



- Later in term: talk about spreading information widely across internet for durability.

Remote File Systems: Virtual File System (VFS)

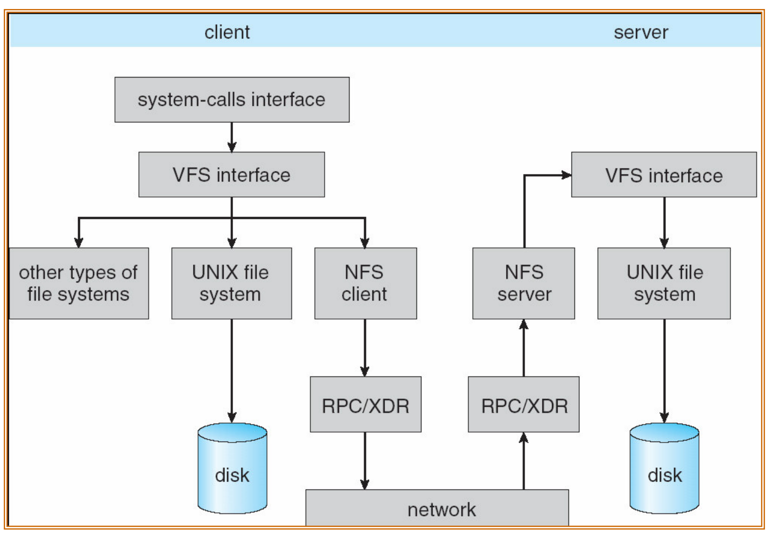


- **VFS:** Virtual abstraction similar to local file system
  - Instead of "inodes" has "vnodes"
  - Compatible with a variety of local and remote file systems
    - » provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - The API is to the VFS interface, rather than any specific type of file system

Network File System (NFS)

- Three Layers for NFS system
  - **UNIX file-system interface:** open, read, write, close calls + file descriptors
  - **VFS layer:** distinguishes local from remote files
    - » Calls the NFS protocol procedures for remote requests
  - **NFS service layer:** bottom layer of the architecture
    - » Implements the NFS protocol
- NFS Protocol: remote procedure calls (RPC) for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes/reading and writing files
- NFS servers are **stateless**; each request provides all arguments require for execution
- Modified data must be committed to the server's disk before results are returned to the client
  - lose some of the advantages of caching
  - Can lead to weird results: write file on one client, read on other, get old data

Schematic View of NFS Architecture



Conclusion

- Cray DEMOS: optimization for sequential access
  - Inode holds set of disk ranges, similar to segmentation
- 4.2 BSD Multilevel index files
  - Inode contains pointers to actual blocks, indirect blocks, double indirect blocks, etc
  - Optimizations for sequential access: start new files in open ranges of free blocks
  - Rotational Optimization
- Naming: act of translating from user-visible names to actual system resources
  - Directories used for naming for local file systems
- Important system properties
  - Availability: how often is the resource available?
  - Durability: how well is data preserved against faults?
  - Reliability: how often is resource performing correctly?
- RAID: Redundant Arrays of Inexpensive Disks
  - RAID1: mirroring, RAID5: Parity block
- VFS: Virtual File System layer
  - NFS: An example use of the VFS layer

CS162
Operating Systems and Systems Programming
Lecture 20

Reliability and Access Control / Distributed Systems

November 7, 2007

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

Review: UNIX BSD 4.2

- Inode Structure Same as BSD 4.1 (same file header and triply indirect blocks), except incorporated ideas from DEMOS:
  - Uses bitmap allocation in place of freelist
  - Attempt to allocate files contiguously
  - 10% reserved disk space
  - Skip-sector positioning
- BSD 4.2 Fast File System (FFS)
  - File Allocation and placement policies
    - » Put each new file at front of different range of blocks
    - » To expand a file, you first try successive blocks in bitmap, then choose new range of blocks
  - Inode for file stored in same "cylinder group" as parent directory of the file
  - Store files from same directory near each other
- Note: I put up the original FFS paper as reading for last lecture (and on Handouts page).

Review: File System Caching (writes)

- **Delayed Writes:** Writes to files not immediately sent out to disk
  - Instead, `write()` copies data from user space buffer to kernel buffer (in cache)
    - » Enabled by presence of buffer cache: can leave written file blocks in cache for a while
    - » If some other application tries to read data before written to disk, file system will read from cache
  - Flushed to disk periodically (e.g. in UNIX, every 30 sec)
  - Advantages:
    - » Disk scheduler can efficiently order lots of requests
    - » Disk allocation algorithm can be run with correct size value for a file
    - » Some files need never get written to disk! (e..g temporary scratch files written /tmp often don't exist for 30 sec)
  - Disadvantages
    - » What if system crashes before file has been written out?
    - » Worse yet, what if system crashes before a directory file has been written out? (lose pointer to inode!)

Review: Important "ilities"

- **Availability:** the probability that the system can accept and process requests
  - Often measured in "nines" of probability. So, a 99.9% probability is considered "3-nines of availability"
  - Key idea here is independence of failures
- **Durability:** the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone
- **Reliability:** the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply availability: means that the system is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems

Goals for Today

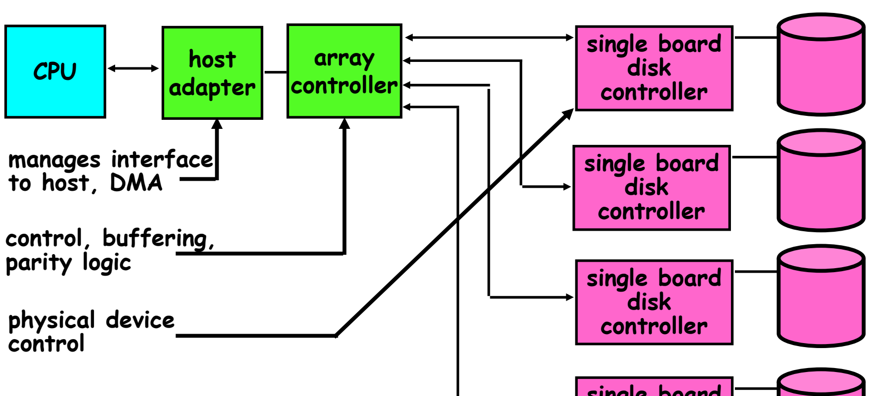
- Durability
- Authorization
- Distributed Systems

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Log Structured and Journaled File Systems

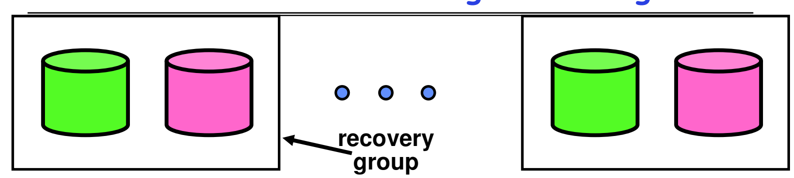
- Better reliability through use of log
  - All changes are treated as *transactions*.
    - » A transaction either happens *completely* or *not at all*
  - A transaction is *committed* once it is written to the log
    - » Data forced to disk for reliability
    - » Process can be accelerated with NVRAM
  - Although File system may not be updated immediately, data preserved in the log
- Difference between "Log Structured" and "Journaled"
  - Log Structured Filesystem (LFS): data stays in log form
  - Journaled Filesystem: Log used for recovery
- For Journaled system:
  - Log used to asynchronously update filesystem
    - » Log entries removed after used
  - After crash:
    - » Remaining transactions in the log performed ("Redo")
- Examples of Journaled File Systems:
  - Ext3 (Linux), XFS (Unix), etc.

Hardware RAID: Subsystem Organization



- Some systems duplicate *all* hardware, namely controllers, busses, etc.

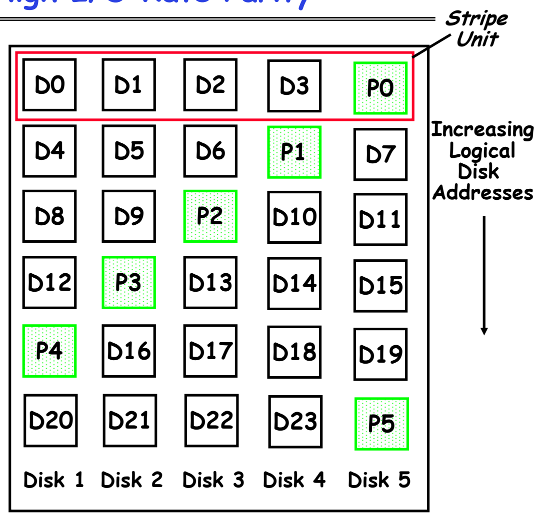
RAID 1: Disk Mirroring/Shadowing



- Each disk is fully duplicated onto its "shadow"
  - For high I/O rate, high availability environments
  - Most expensive solution: 100% capacity overhead
- Bandwidth sacrificed on write:
  - Logical write = two physical writes
  - Highest bandwidth when disk heads and rotation fully synchronized (hard to do exactly)
- Reads may be optimized
  - Can have two independent reads to same data
- Recovery:
  - Disk failure ⇒ replace disk and copy data to new disk
  - Hot Spare: idle disk already attached to system to be used for immediate replacement

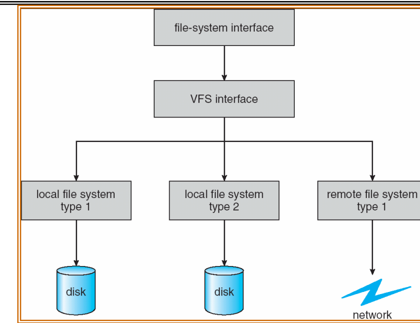
RAID 5+: High I/O Rate Parity

- Data stripped across multiple disks
  - Successive blocks stored on successive (non-parity) disks
  - Increased bandwidth over single disk
- Parity block (in green) constructed by XORing data bocks in stripe
  - $P0 = D0 \oplus D1 \oplus D2 \oplus D3$
  - Can destroy any one disk and still reconstruct data
  - Suppose D3 fails, then can reconstruct: $D3 = D0 \oplus D1 \oplus D2 \oplus P0$



- Later in term: talk about spreading information widely across internet for durability.

Remote File Systems: Virtual File System (VFS)

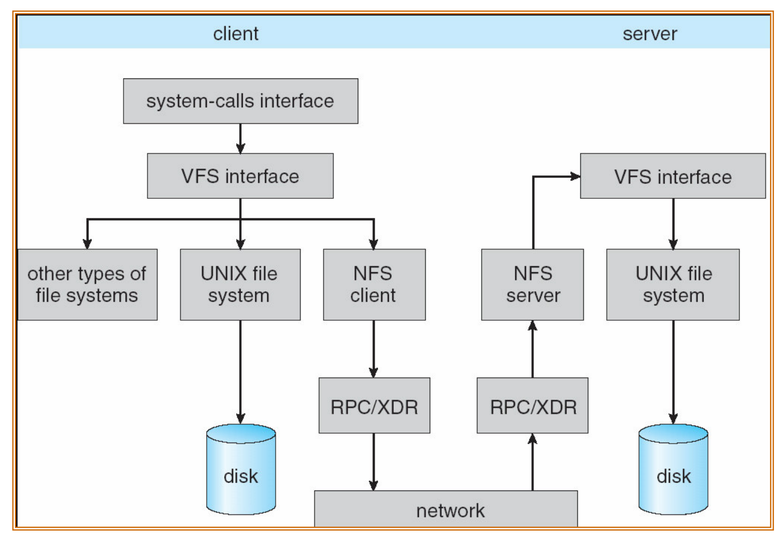


- VFS: Virtual abstraction similar to local file system
  - Instead of "inodes" has "vnodes"
  - Compatible with a variety of local and remote file systems
    - » provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - The API is to the VFS interface, rather than any specific type of file system

Network File System (NFS)

- Three Layers for NFS system
  - UNIX file-system interface: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - » Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - » Implements the NFS protocol
- NFS Protocol: remote procedure calls (RPC) for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes/reading and writing files
- NFS servers are stateless; each request provides all arguments require for execution
- Modified data must be committed to the server's disk before results are returned to the client
  - lose some of the advantages of caching
  - Can lead to weird results: write file on one client, read on other, get old data

Schematic View of NFS Architecture



Administrivia

Authorization: Who Can Do What?

- How do we decide who is authorized to do actions in the system?
- Access Control Matrix: contains all permissions in the system
  - Resources across top
    - » Files, Devices, etc…
  - Domains in columns
    - » A domain might be a user or a group of permissions
    - » E.g. above: User D3 can read F2 or execute F3
  - In practice, table would be huge and sparse!

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Authorization: Two Implementation Choices

- Access Control Lists: store permissions with object
  - Still might be lots of users!
  - UNIX limits each file to: r,w,x for owner, group, world
  - More recent systems allow definition of groups of users and permissions for each group
  - ACLs allow easy changing of an object's permissions
    - » Example: add Users C, D, and F with rw permissions
- Capability List: each process tracks which objects has permission to touch
  - Popular in the past, idea out of favor today
  - Consider page table: Each process has list of pages it has access to, not each page has list of processes …
  - Capability lists allow easy changing of a domain's permissions
    - » Example: you are promoted to system administrator and should be given access to all system files

Authorization: Combination Approach

- Users have capabilities, called "groups" or "roles"
  - Everyone with particular group access is "equivalent" when accessing group resource
  - Like passport (which gives access to country of origin)

- Objects have ACLs
  - ACLs can refer to users or groups
  - Change object permissions object by modifying ACL
  - Change broad user permissions via changes in group membership
  - Possessors of proper credentials get access

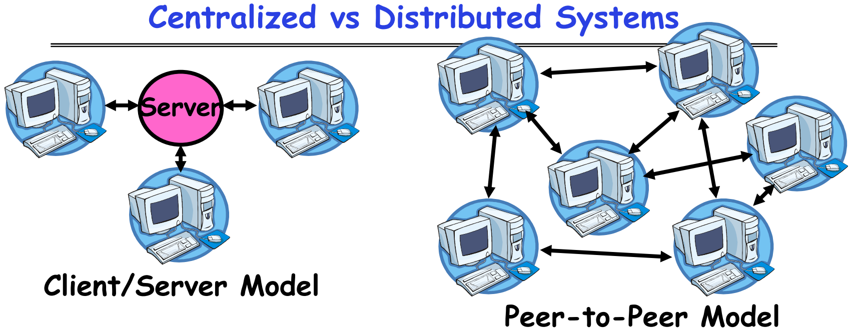
Authorization: How to Revoke?

- How does one revoke someone's access rights to a particular object?
  - Easy with ACLs: just remove entry from the list
  - Takes effect immediately since the ACL is checked on each object access
- Harder to do with capabilities since they aren't stored with the object being controlled:
  - Not so bad in a single machine: could keep all capability lists in a well-known place (e.g., the OS capability table).
  - Very hard in distributed system, where remote hosts may have crashed or may not cooperate (more in a future lecture)

Revoking Capabilities

- Various approaches to revoking capabilities:
  - Put expiration dates on capabilities and force reacquisition
  - Put epoch numbers on capabilities and revoke all capabilities by bumping the epoch number (which gets checked on each access attempt)
  - Maintain back pointers to all capabilities that have been handed out (Tough if capabilities can be copied)
  - Maintain a revocation list that gets checked on every access attempt

Centralized vs Distributed Systems



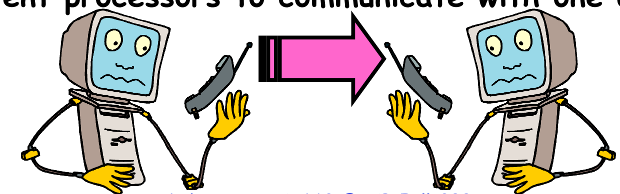
- **Centralized System:** System in which major functions are performed by a single physical computer
  - Originally, everything on single computer
  - Later: client/server model
- **Distributed System:** physically separate computers working together on some task
  - Early model: multiple servers working together
    - » Probably in the same room or building
    - » Often called a "cluster"
  - Later models: peer-to-peer/wide-spread collaboration

Distributed Systems: Motivation/Issues

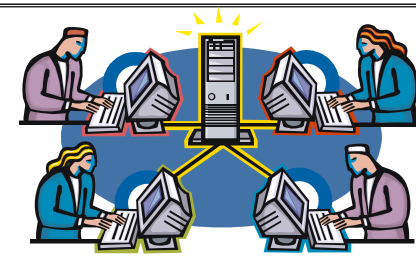
- Why do we want distributed systems?
  - Cheaper and easier to build lots of simple computers
  - Easier to add power incrementally
  - Users can have complete control over some components
  - Collaboration: Much easier for users to collaborate through network resources (such as network file systems)
- The *promise* of distributed systems:
  - Higher availability: one machine goes down, use another
  - Better durability: store data in multiple locations
  - More security: each piece easier to make secure
- Reality has been disappointing
  - Worse availability: depend on every machine being up
    - » Lamport: "a distributed system is one where I can't do work because some machine I've never heard of isn't working!"
  - Worse reliability: can lose data if any machine crashes
  - Worse security: anyone in world can break into system
- Coordination is more difficult
  - Must coordinate multiple copies of shared state information (using only a network)
  - What would be easy in a centralized system becomes a lot more difficult

Distributed Systems: Goals/Requirements

- **Transparency:** the ability of the system to mask its complexity behind a simple interface
- Possible transparencies:
  - **Location:** Can't tell where resources are located
  - **Migration:** Resources may move without the user knowing
  - **Replication:** Can't tell how many copies of resource exist
  - **Concurrency:** Can't tell how many users there are
  - **Parallelism:** System may speed up large jobs by spliting them into smaller pieces
  - **Fault Tolerance**: System may hide varoius things that go wrong in the system
- Transparency and collaboration require some way for different processors to communicate with one another

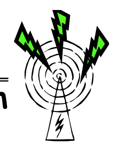


Networking Definitions

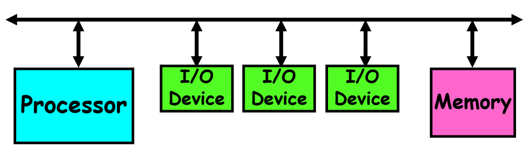


- **Network:** physical connection that allows two computers to communicate
- **Packet:** unit of transfer, sequence of bits carried over the network
  - Network carries packets from one CPU to another
  - Destination gets interrupt when packet arrives
- **Protocol:** agreement between two parties as to how information is to be transmitted

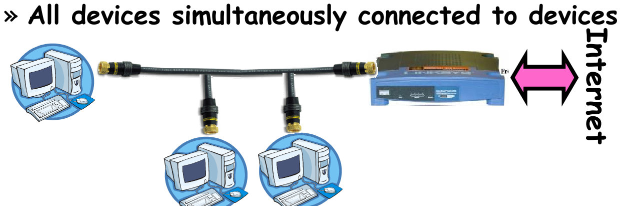
Broadcast Networks



- **Broadcast Network:** Shared Communication Medium

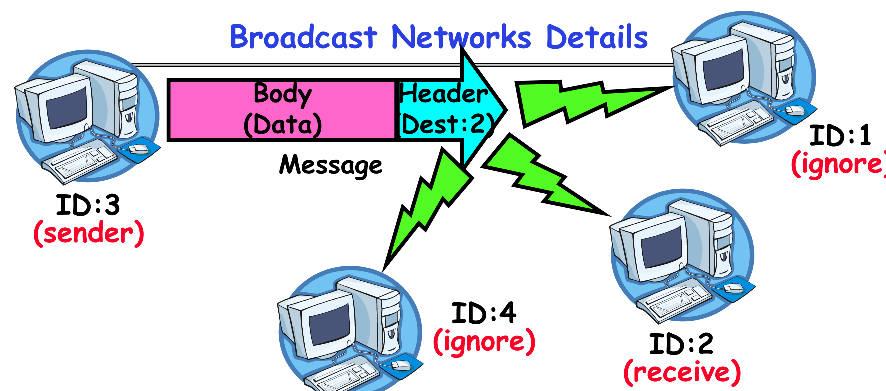


  - Shared Medium can be a set of wires
    - » Inside a computer, this is called a bus
    - » All devices simultaneously connected to devices



  - Originally, Ethernet was a broadcast network
    - » All computers on local subnet connected to one another
  - More examples (wireless: medium is air): cellular phones, GSM GPRS, EDGE, CDMA 1xRTT, and 1EvDO

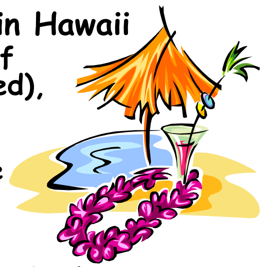
Broadcast Networks Details



- **Delivery:** When you broadcast a packet, how does a receiver know who it is for? (packet goes to everyone!)
  - Put header on front of packet: [Destination | Packet]
  - Everyone gets packet, discards if not the target
  - In Ethernet, this check is done in hardware
    - » No OS interrupt if not for particular destination
  - This is layering: we're going to build complex network protocols by layering on top of the packet

Broadcast Network Arbitration

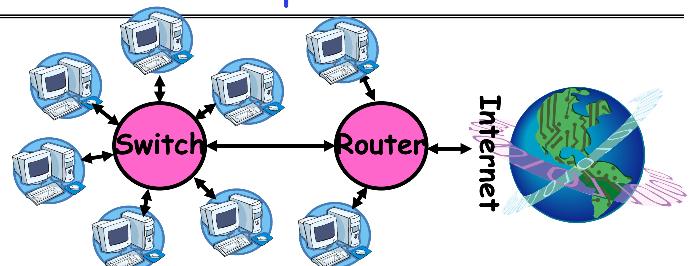
- **Arbitration:** Act of negotiating use of shared medium
  - What if two senders try to broadcast at same time?
  - Concurrent activity but can't use shared memory to coordinate!
- Aloha network (70's): packet radio within Hawaii
  - Blind broadcast, with checksum at end of packet. If received correctly (not garbled), send back an acknowledgement. If not received correctly, discard.
    - » Need checksum anyway – in case airplane flies overhead
  - Sender waits for a while, and if doesn't get an acknowledgement, re-transmits.
  - If two senders try to send at same time, both get garbled, both simply re-send later.
  - Problem: Stability: what if load increases?
    - » More collisions ⇒ less gets through ⇒more resent ⇒ more load… ⇒ More collisions…
    - » Unfortunately: some sender may have started in clear, get scrambled without finishing



Carrier Sense, Multiple Access/Collision Detection

- Ethernet (early 80's): first practical local area network
  - It is the most common LAN for UNIX, PC, and Mac
  - Use wire instead of radio, but still broadcast medium
- Key advance was in arbitration called CSMA/CD: Carrier sense, multiple access/collision detection
  - **Carrier Sense:** don't send unless idle
    - » Don't mess up communications already in process
  - **Collision Detect:** sender checks if packet trampled.
    - » If so, abort, wait, and retry.
  - **Backoff Scheme:** Choose wait time before trying again
- How long to wait after trying to send and failing?
  - What if everyone waits the same length of time? Then, they all collide again at some time!
  - Must find way to break up shared behavior with nothing more than shared communication channel
- Adaptive randomized waiting strategy:
  - **Adaptive and Random:** First time, pick random wait time with some initial mean. If collide again, pick random value from bigger mean wait time. Etc.
  - Randomness is important to decouple colliding senders
  - Scheme figures out how many people are trying to send!

Point-to-point networks

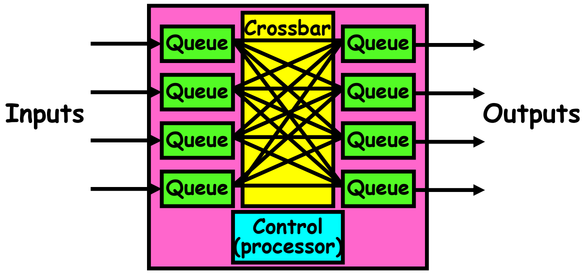


- Why have a shared bus at all? Why not simplify and only have point-to-point links + routers/switches?
  - Didn't used to be cost-effective
  - Now, easy to make high-speed switches and routers that can forward packets from a sender to a receiver.
- **Point-to-point network:** a network in which every physical wire is connected to only two computers
- **Switch:** a bridge that transforms a shared-bus configuration into a point-to-point network.
- **Router:** a device that acts as a junction between two networks to transfer data packets among them.

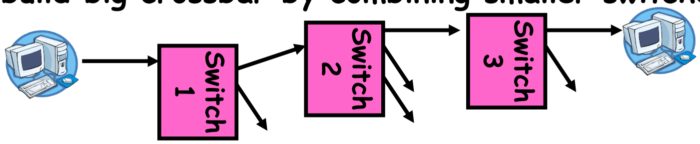
Point-to-Point Networks Discussion

- Advantages:
  - Higher link performance
    - » Can drive point-to-point link faster than broadcast link since less capacitance/less echoes (from impedance mismatches)
  - Greater aggregate bandwidth than broadcast link
    - » Can have multiple senders at once
  - Can add capacity incrementally
    - » Add more links/switches to get more capacity
  - Better fault tolerance (as in the Internet)
  - Lower Latency
    - » No arbitration to send, although need buffer in the switch
- Disadvantages:
  - More expensive than having everyone share broadcast link
  - However, technology costs now much cheaper
- Examples
  - ATM (asynchronous transfer mode)
    - » The first commercial point-to-point LAN
    - » Inspiration taken from telephone network
  - Switched Ethernet
    - » Same packet format and signaling as broadcast Ethernet, but only two machines on each ethernet.

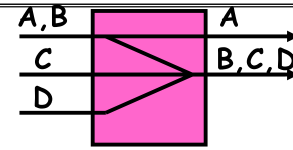
Point-to-Point Network design



- Switches look like computers: inputs, memory, outputs
  - In fact probably contains a processor
- Function of switch is to forward packet to output that gets it closer to destination
- Can build big crossbar by combining smaller switches



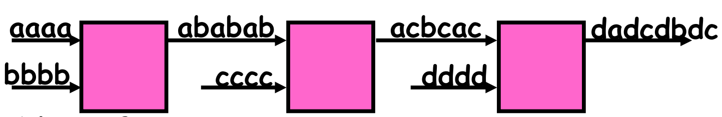
Flow control options



- What if everyone sends to the same output?
  - Congestion—packets don't flow at full rate
- In general, what if buffers fill up?
  - Need flow control policy
- Option 1: no flow control. Packets get dropped if they arrive and there's no space
  - If someone sends a lot, they are given buffers and packets from other senders are dropped
  - Internet actually works this way
- Option 2: Flow control between switches
  - When buffer fills, stop inflow of packets
  - Problem: what if path from source to destination is completely unused, but goes through some switch that has buffers filled up with unrelated traffic?

Flow Control (con't)

- Option 3: Per-flow flow control.
  - Allocate a separate set of buffers to each end-to-end stream and use separate "don't send me more" control on each end-to-end stream



- Problem: fairness
  - Throughput of each stream is entirely dependent on topology, and relationship to bottleneck
- Automobile Analogy
  - At traffic jam, one strategy is merge closest to the bottleneck
    - » Why people get off at one exit, drive 500 feet, merge back into flow
    - » Ends up slowing everybody else a huge amount
  - Also why have control lights at on-ramps
    - » Try to keep from injecting more cars than capacity of road (and thus avoid congestion)

Conclusion

- Important system properties
  - Availability: how often is the resource available?
  - Durability: how well is data preserved against faults?
  - Reliability: how often is resource performing correctly?
- Use of Log to improve Reliability
  - Journaled file systems such as ext3
- RAID: Redundant Arrays of Inexpensive Disks
  - RAID1: mirroring, RAID5: Parity block
- Authorization
  - Controlling access to resources using
    - » Access Control Lists
    - » Capabilities
- Network: physical connection that allows two computers to communicate
  - Packet: unit of transfer, sequence of bits carried over the network

CS162
Operating Systems and Systems Programming
Lecture 21

Networking

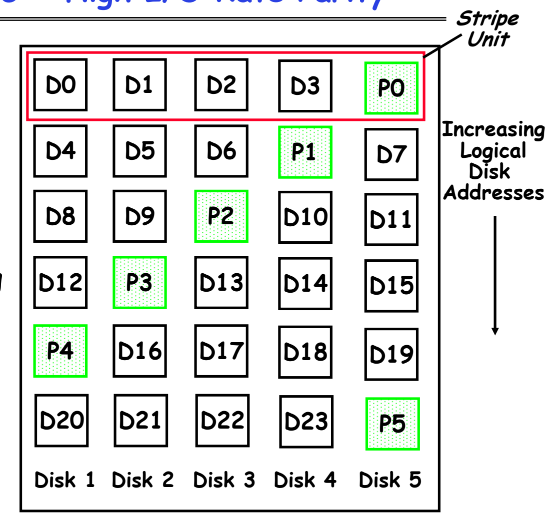
November 14, 2007

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

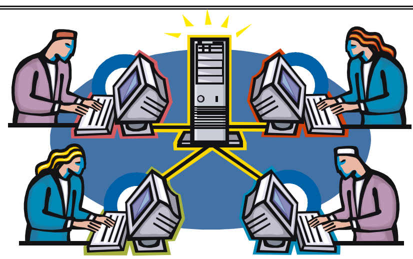
Review: RAID 5+: High I/O Rate Parity

- Data stripped across multiple disks
  - Successive blocks stored on successive (non-parity) disks
  - Increased bandwidth over single disk
- Parity block (in green) constructed by XORing data bocks in stripe
  - $P0=D0\oplus D1\oplus D2\oplus D3$
  - Can destroy any one disk and still reconstruct data
  - Suppose D3 fails, then can reconstruct: $D3=D0\oplus D1\oplus D2\oplus P0$



- Later in term: talk about spreading information widely across internet for durability.

Review: Networking Definitions



- **Network:** physical connection that allows two computers to communicate
- **Packet:** unit of transfer, sequence of bits carried over the network
  - Network carries packets from on CPU to another
  - Destination gets interrupt when packet arrives
- **Protocol:** agreement between two parties as to how information is to be transmitted

Review: Ethernet and CSMA/CD

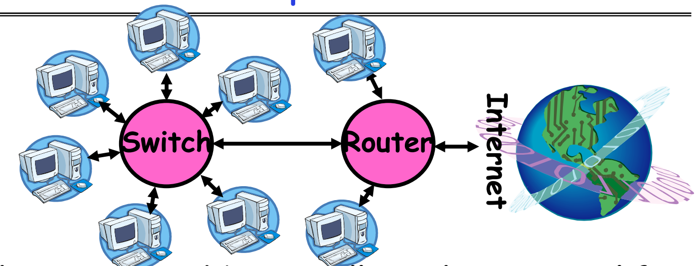
- Ethernet (early 80's): first practical local area network
  - It is the most common LAN for UNIX, PC, and Mac
  - Use wire instead of radio, but still broadcast medium
- Key advance was in arbitration called CSMA/CD: Carrier sense, multiple access/collision detection
  - **Carrier Sense:** don't send unless idle
    - » Don't mess up communications already in process
  - **Collision Detect:** sender checks if packet trampled.
    - » If so, abort, wait, and retry.
  - **Backoff Scheme:** Choose wait time before trying again
- Adaptive randomized waiting strategy:
  - **Adaptive and Random:** First time, pick random wait time with some initial mean. If collide again, pick random value from bigger mean wait time. Etc.
  - Randomness is important to decouple colliding senders
  - Scheme figures out how many people are trying to send!

Goals for Today

- Networking
  - Point-to-Point Networking
  - Routing
  - Internet Protocol (IP)

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Point-to-point networks

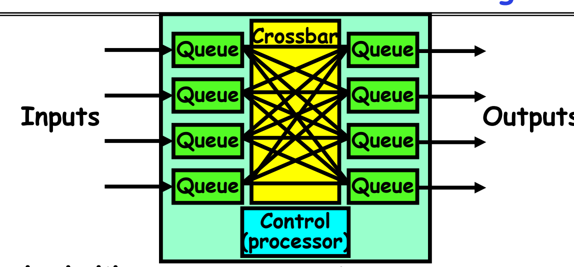


- Why have a shared bus at all? Why not simplify and only have point-to-point links + routers/switches?
  - Didn't used to be cost-effective
  - Now, easy to make high-speed switches and routers that can forward packets from a sender to a receiver.
- **Point-to-point network:** a network in which every physical wire is connected to only two computers
- **Switch:** a bridge that transforms a shared-bus (broadcast) configuration into a point-to-point network.
- **Router:** a device that acts as a junction between two networks to transfer data packets among them.

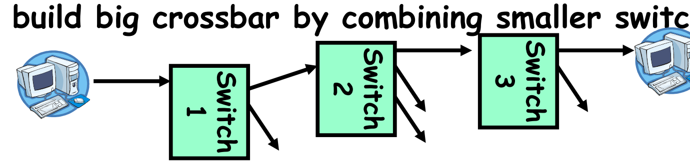
Point-to-Point Networks Discussion

- Advantages:
  - Higher link performance
    - » Can drive point-to-point link faster than broadcast link since less capacitance/less echoes (from impedance mismatches)
  - Greater aggregate bandwidth than broadcast link
    - » Can have multiple senders at once
  - Can add capacity incrementally
    - » Add more links/switches to get more capacity
  - Better fault tolerance (as in the Internet)
  - Lower Latency
    - » No arbitration to send, although need buffer in the switch
- Disadvantages:
  - More expensive than having everyone share broadcast link
  - However, technology costs now much cheaper
- Examples
  - ATM (asynchronous transfer mode)
    - » The first commercial point-to-point LAN
    - » Inspiration taken from telephone network
  - Switched Ethernet
    - » Same packet format and signaling as broadcast Ethernet, but only two machines on each ethernet.

Point-to-Point Network design

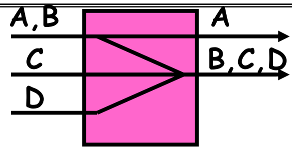


- Switches look like computers: inputs, memory, outputs
  - In fact probably contains a processor
- Function of switch is to forward packet to output that gets it closer to destination
- Can build big crossbar by combining smaller switches



- **Can perform broadcast if necessary**

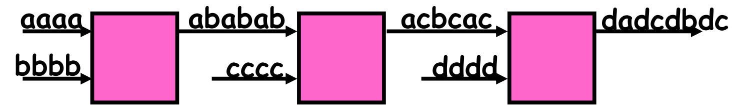
Flow control options



- What if everyone sends to the same output?
  - Congestion—packets don't flow at full rate
- In general, what if buffers fill up?
  - Need flow control policy
- Option 1: no flow control. Packets get dropped if they arrive and there's no space
  - If someone sends a lot, they are given buffers and packets from other senders are dropped
  - Internet actually works this way
- Option 2: Flow control between switches
  - When buffer fills, stop inflow of packets
  - Problem: what if path from source to destination is completely unused, but goes through some switch that has buffers filled up with unrelated traffic?

Flow Control (con't)

- Option 3: Per-flow flow control.
  - Allocate a separate set of buffers to each end-to-end stream and use separate "don't send me more" control on each end-to-end stream



- Problem: fairness
  - Throughput of each stream is entirely dependent on topology, and relationship to bottleneck
- Automobile Analogy
  - At traffic jam, one strategy is merge closest to the bottleneck
    - » Why people get off at one exit, drive 50 feet, merge back into flow
    - » Ends up slowing everybody else a huge emount
  - Also why have control lights at on-ramps
    - » Try to keep from injecting more cars than capacity of road (and thus avoid congestion)

The Internet Protocol: "IP"

- The Internet is a large network of computers spread across the globe
  - According to the Internet Systems Consortium, there were over 353 million computers as of July 2005
  - In principle, every host can speak with every other one under the right circumstances
- **IP Packet:** a network packet on the internet
- **IP Address:** a 32-bit integer used as the destination of an IP packet
  - Often written as four dot-separated integers, with each integer from 0—255 (thus representing 8x4=32 bits)
  - Example CS file server is: 169.229.60.83 ≡ 0xA9E53C53
- **Internet Host:** a computer connected to the Internet
  - Host has one or more IP addresses used for routing
    - » Some of these may be private and unavailable for routing
  - Not every computer has a unique IP address
    - » Groups of machines may share a single IP address
    - » In this case, machines have private addresses behind a "Network Address Translation" (NAT) gateway

Address Subnets

- **Subnet:** A network connecting a set of hosts with related destination addresses
- With IP, all the addresses in subnet are related by a prefix of bits
  - **Mask:** The number of matching prefix bits
    - » Expressed as a single value (e.g., 24) or a set of ones in a 32-bit value (e.g., 255.255.255.0)
- A subnet is identified by 32-bit value, with the bits which differ set to zero, followed by a slash and a mask
  - Example: 128.32.131.0/24 designates a subnet in which all the addresses look like 128.32.131.XX
  - Same subnet: 128.32.131.0/255.255.255.0
- Difference between subnet and complete network range
  - Subnet is always a subset of address range
  - Once, subnet meant single physical broadcast wire; now, less clear exactly what it means (virtualized by switches)

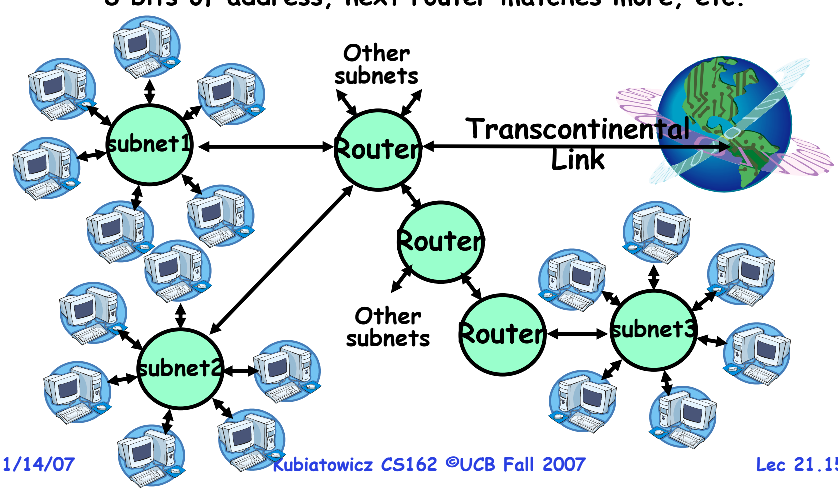
Address Ranges in IP

- IP address space divided into prefix-delimited ranges:
  - Class A: NN.0.0.0/8
    - » NN is 1–126 (126 of these networks)
    - » 16,777,214 IP addresses per network
    - » 10.xx.yy.zz is private
    - » 127.xx.yy.zz is loopback
  - Class B: NN.MM.0.0/16
    - » NN is 128–191, MM is 0-255 (16,384 of these networks)
    - » 65,534 IP addresses per network
    - » 172.[16-31].xx.yy are private
  - Class C: NN.MM.LL.0/24
    - » NN is 192–223, MM and LL 0-255 (2,097,151 of these networks)
    - » 254 IP addresses per networks
    - » 192.168.xx.yy are private
- Address ranges are often owned by organizations
  - Can be further divided into subnets

Administrivia

Hierarchical Networking: The Internet

- How can we build a network with millions of hosts?
  - Hierarchy! Not every host connected to every other one
  - Use a network of Routers to connect subnets together
    - » Routing is often by prefix: e.g. first router matches first 8 bits of address, next router matches more, etc.

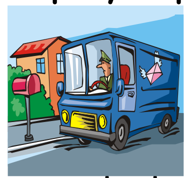


Simple Network Terminology

- Local-Area Network (LAN) – designed to cover small geographical area
  - Multi-access bus, ring, or star network
  - Speed ≈ 10 – 1000 Megabits/second
  - Broadcast is fast and cheap
  - In small organization, a LAN could consist of a single subnet. In large organizations (like UC Berkeley), a LAN contains many subnets
- Wide-Area Network (WAN) – links geographically separated sites
  - Point-to-point connections over long-haul lines (often leased from a phone company)
  - Speed ≈ 1.544 – 45 Megabits/second
  - Broadcast usually requires multiple messages

Routing

- Routing: the process of forwarding packets hop-by-hop through routers to reach their destination
  - Need more than just a destination address!
    - » Need a path
  - Post Office Analogy:
    - » Destination address on each letter is not sufficient to get it to the destination
    - » To get a letter from here to Florida, must route to local post office, sorted and sent on plane to somewhere in Florida, be routed to post office, sorted and sent with carrier who knows where street and house is…



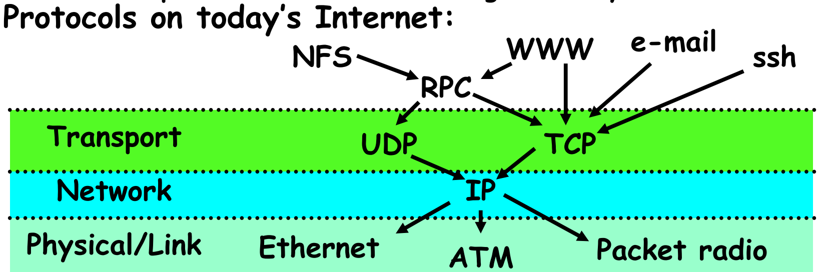
- Internet routing mechanism: routing tables
  - Each router does table lookup to decide which link to use to get packet closer to destination
  - Don't need 4 billion entries in table: routing is by subnet
  - Could packets be sent in a loop? Yes, if tables incorrect
- Routing table contains:
  - Destination address range $\rightarrow$ output link closer to destination
  - Default entry (for subnets without explicit entries)

Setting up Routing Tables

- How do you set up routing tables?
  - Internet has no centralized state!
    - » No single machine knows entire topology
    - » Topology constantly changing (faults, reconfiguration, etc)
  - Need dynamic algorithm that acquires routing tables
    - » Ideally, have one entry per subnet or portion of address
    - » Could have "default" routes that send packets for unknown subnets to a different router that has more information
- Possible algorithm for acquiring routing table
  - Routing table has "cost" for each entry
    - » Includes number of hops to destination, congestion, etc.
    - » Entries for unknown subnets have infinite cost
  - Neighbors periodically exchange routing tables
    - » If neighbor knows cheaper route to a subnet, replace your entry with neighbors entry (+1 for hop to neighbor)
- In reality:
  - Internet has networks of many different scales
  - Different algorithms run at different scales

Network Protocols

- Protocol: Agreement between two parties as to how information is to be transmitted
  - Example: system calls are the protocol between the operating system and application
  - Networking examples: many levels
    - » Physical level: mechanical and electrical network (e.g. how are 0 and 1 represented)
    - » Link level: packet formats/error control (for instance, the CSMA/CD protocol)
    - » Network level: network routing, addressing
    - » Transport Level: reliable message delivery
- Protocols on today's Internet:



Network Layering

- Layering: building complex services from simpler ones
  - Each layer provides services needed by higher layers by utilizing services provided by lower layers
- The physical/link layer is pretty limited
  - Packets are of limited size (called the "Maximum Transfer Unit or MTU: often 200-1500 bytes in size)
  - Routing is limited to within a physical link (wire) or perhaps through a switch
- Our goal in the following is to show how to construct a secure, ordered, message service routed to anywhere:

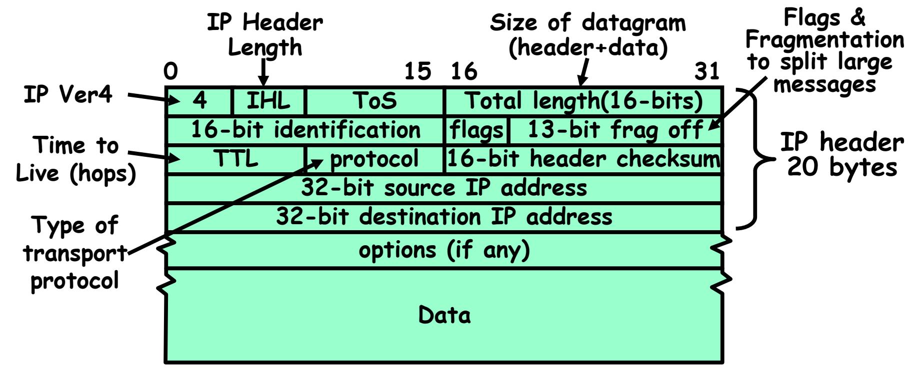
Physical Reality: Packets	Abstraction: Messages
Limited Size	Arbitrary Size
Unordered (sometimes)	Ordered
Unreliable	Reliable
Machine-to-machine	Process-to-process
Only on local area net	Routed anywhere
Asynchronous	Synchronous
Insecure	Secure

Building a messaging service

- Handling Arbitrary Sized Messages:
  - Must deal with limited physical packet size
  - Split big message into smaller ones (called fragments)
    - » Must be reassembled at destination
  - Checksum computed on each fragment or whole message
- Internet Protocol (IP): Must find way to send packets to arbitrary destination in network
  - Deliver messages unreliably ("best effort") from one machine in Internet to another
  - Since intermediate links may have limited size, must be able to fragment/reassemble packets on demand
  - Includes 256 different "sub-protocols" build on top of IP
    - » Examples: ICMP(1), TCP(6), UDP (17), IPSEC(50,51)

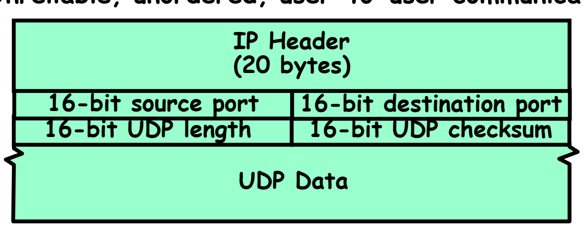
IP Packet Format

- IP Packet Format:



Building a messaging service

- Process to process communication
  - Basic routing gets packets from machine→machine
  - What we really want is routing from process→process
    - » Example: ssh, email, ftp, web browsing
  - Several IP protocols include notion of a "port", which is a 16-bit identifiers used in addition to IP addresses
    - » A communication channel (connection) defined by 5 items: [source address, source port, dest address, dest port, protocol]
- UDP: The User Datagram Protocol
  - UDP layered on top of basic IP (IP Protocol 17)
    - » Unreliable, unordered, user-to-user communication

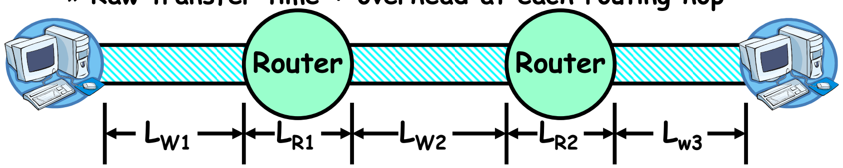


Building a messaging service (con't)

- UDP: The Unreliable Datagram Protocol
  - Datagram: an unreliable, unordered, packet sent from source user → dest user (Call it UDP/IP)
  - Important aspect: low overhead!
    - » Often used for high-bandwidth video streams
    - » Many uses of UDP considered "anti-social" – none of the "well-behaved" aspects of (say) TCP/IP
- But we need ordered messages
  - Create ordered messages on top of unordered ones
    - » IP can reorder packets! P_0,P_1 might arrive as P_1,P_0
  - How to fix this? Assign sequence numbers to packets
    - » 0,1,2,3,4…..
    - » If packets arrive out of order, reorder before delivering to user application
    - » For instance, hold onto #3 until #2 arrives, etc.
  - Sequence numbers are specific to particular connection

Performance Considerations

- Before continue, need some performance metrics
  - **Overhead:** CPU time to put packet on wire
  - **Throughput:** Maximum number of bytes per second
    - » Depends on "wire speed", but also limited by slowest router (routing delay) or by congestion at routers
  - **Latency:** time until first bit of packet arrives at receiver
    - » Raw transfer time + overhead at each routing hop



- Contributions to Latency
  - Wire latency: depends on speed of light on wire
    - » about 1–1.5 ns/foot
  - Router latency: depends on internals of router
    - » Could be < 1 ms (for a good router)
    - » Question: can router handle full wire throughput?

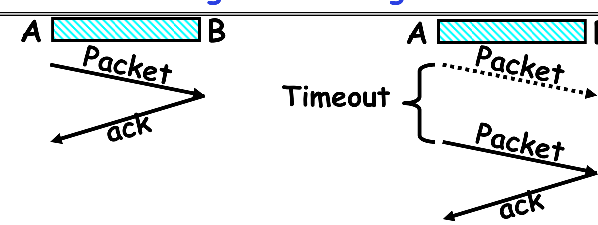
Sample Computations

- E.g.: Ethernet within Soda
  - Latency: speed of light in wire is 1.5ns/foot, which implies latency in building < 1 μs (if no routers in path)
  - Throughput: 10-1000Mb/s
  - Throughput delay: packet doesn't arrive until all bits
    - » So: 4KB/100Mb/s = 0.3 milliseconds (same order as disk!)
- E.g.: ATM within Soda
  - Latency (same as above, assuming no routing)
  - Throughput: 155Mb/s
  - Throughput delay: 4KB/155Mb/s = 200μ
- E.g.: ATM cross-country
  - Latency (assuming no routing):
    - » 3000miles * 5000ft/mile ⇒ 15 milliseconds
  - How many bits could be in transit at same time?
    - » 15ms * 155Mb/s = 290KB
  - In fact, Berkeley→MIT Latency ~ 45ms
    - » 872KB in flight if routers have wire-speed throughput
- **Requirements for good performance:**
  - **Local area: minimize overhead/improve bandwidth**
  - **Wide area: keep pipeline full!**

Reliable Message Delivery: the Problem

- All physical networks can garble and/or drop packets
  - Physical media: packet not transmitted/received
    - » If transmit close to maximum rate, get more throughput – even if some packets get lost
    - » If transmit at lowest voltage such that error correction just starts correcting errors, get best power/bit
  - Congestion: no place to put incoming packet
    - » Point-to-point network: insufficient queue at switch/router
    - » Broadcast link: two host try to use same link
    - » In any network: insufficient buffer space at destination
    - » Rate mismatch: what if sender send faster than receiver can process?
- Reliable Message Delivery
  - Reliable messages on top of unreliable packets
  - Need some way to make sure that packets actually make it to receiver
    - » Every packet received at least once
    - » Every packet received only once
  - Can combine with ordering: every packet received by process at destination exactly once and in order

Using Acknowledgements



- How to ensure transmission of packets?
  - Detect garbling at receiver via checksum, discard if bad
  - Receiver acknowledges (by sending "ack") when packet received properly at destination
  - Timeout at sender: if no ack, retransmit
- Some questions:
  - If the sender doesn't get an ack, does that mean the receiver didn't get the original message?
    - » No
  - What it ack gets dropped? Or if message gets delayed?
    - » Sender doesn't get ack, retransmits. Receiver gets message twice, acks each.

How to deal with message duplication

- Solution: put sequence number in message to identify re-transmitted packets
  - Receiver checks for duplicate #'s; Discard if detected
- Requirements:
  - Sender keeps copy of unack'ed messages
    - » Easy: only need to buffer messages
  - Receiver tracks possible duplicate messages
    - » Hard: when ok to forget about received message?
- Simple solution: Alternating-bit protocol
  - Send one message at a time; don't send next message until ack received
  - Sender keeps last message; receiver tracks sequence # of last message received
- Pros: simple, small overhead
- Con: Poor performance
  - Wire can hold multiple messages; want to fill up at (wire latency × throughput)
- Con: doesn't work if network can delay or duplicate messages arbitrarily

Conclusion

- Network: physical connection that allows two computers to communicate
  - Packet: sequence of bits carried over the network
- Broadcast Network: Shared Communication Medium
  - Transmitted packets sent to all receivers
  - Arbitration: act of negotiating use of shared medium
    - » Ethernet: Carrier Sense, Multiple Access, Collision Detect
- Point-to-point network: a network in which every physical wire is connected to only two computers
  - Switch: a bridge that transforms a shared-bus (broadcast) configuration into a point-to-point network.
- Protocol: Agreement between two parties as to how information is to be transmitted
- Internet Protocol (IP)
  - Used to route messages through routes across globe
  - 32-bit addresses, 16-bit ports
- Reliable, Ordered, Arbitrary-sized Messaging:
  - Built through protocol layering on top of unreliable, limited-sized, non-ordered packet transmission links

CS162
Operating Systems and Systems Programming
Lecture 22

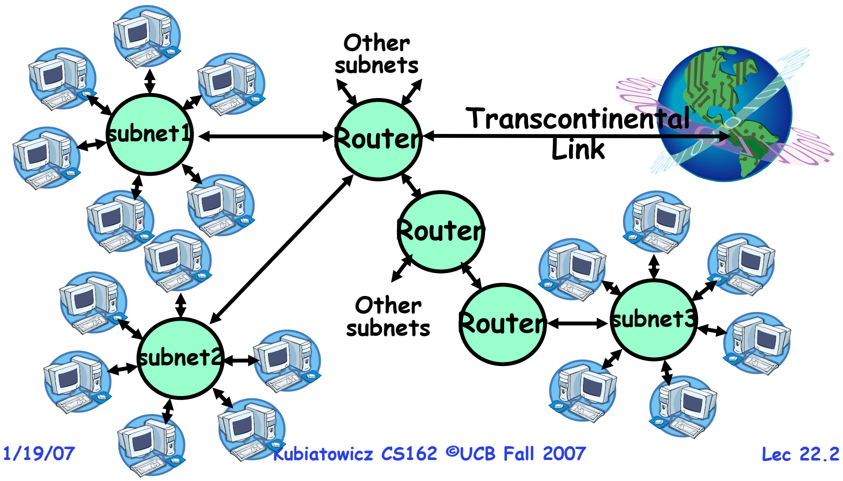
Networking II

November 19, 2007

Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

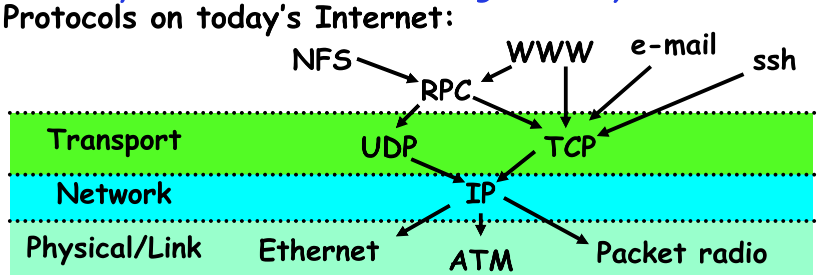
Review: Hierarchical Networking (The Internet)

- How can we build a network with millions of hosts?
  - Hierarchy! Not every host connected to every other one
  - Use a network of Routers to connect subnets together



Review: Network Protocols

- Protocol: Agreement between two parties as to how information is to be transmitted
  - Physical level: mechanical and electrical network (e.g. how are 0 and 1 represented)
  - Link level: packet formats/error control (for instance, the CSMA/CD protocol)
  - Network level: network routing, addressing
  - *Transport Level: reliable message delivery*
- Protocols on today's Internet:

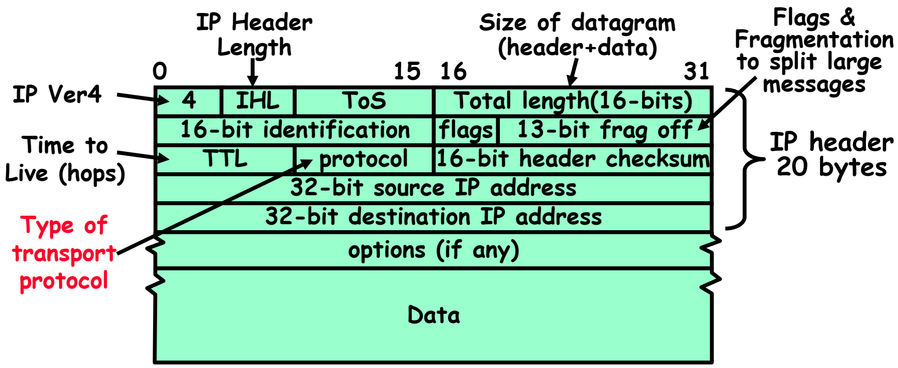


Review: Basic Networking Limitations

- The physical/link layer is pretty limited
  - Packets of limited size
    - » Maximum Transfer Unit (MTU): often 200-1500 bytes
  - Packets can get lost or garbled
  - Hardware routing limited to physical link or switch
  - Physical routers crash/links get damaged
    - » Baltimore tunnel fire (July 2001): cut major Internet links
- Handling Arbitrary Sized Messages:
  - Must deal with limited physical packet size
  - Split big message into smaller ones (called fragments)
    - » Must be reassembled at destination
    - » May happen on demand if packet routed through areas of reduced MTU (e.g. TCP)
  - Checksum computed on each fragment or whole message
- Need resilient routing algorithms to send messages on wide area
  - Multi-hop routing mechanisms
  - Redundant links/Ability to route around failed links

Review: IP Packet Format

- IP Packet Format:



- Each *protocol* represents different packet formats after first 20 bytes:
  - Examples: ICMP(1), TCP(6), UDP (17), IPSEC(50,51)

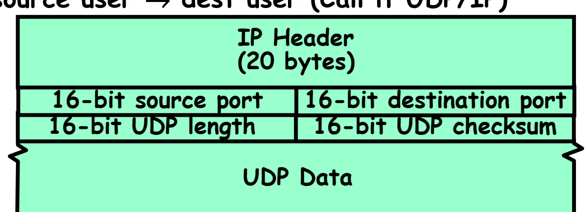
Goals for Today

- Networking
  - Reliable Messaging
    - » TCP windowing and congestion avoidance
  - Two-phase commit

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Building a messaging service

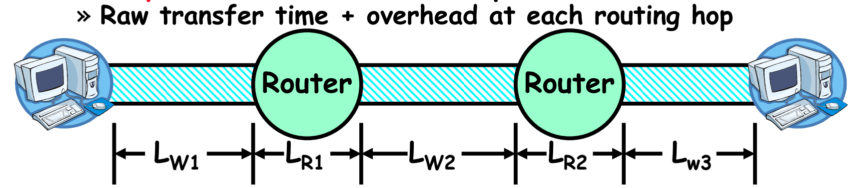
- Process to process communication
  - Basic routing gets packets from machine→machine
  - What we really want is routing from process→process
    - » Add "ports", which are 16-bit identifiers
    - » A communication channel (connection) defined by 5 items: [source addr, source port, dest addr, dest port, protocol]
- UDP: The Unreliable Datagram Protocol
  - Layered on top of basic IP (IP Protocol 17)
    - » Datagram: an unreliable, unordered, packet sent from source user → dest user (Call it UDP/IP)



  - Important aspect: low overhead!
    - » Often used for high-bandwidth video streams
    - » Many uses of UDP considered "anti-social" – none of the "well-behaved" aspects of (say) TCP/IP

Performance Considerations

- Before continue, need some performance metrics
  - Overhead: CPU time to put packet on wire
  - Throughput: Maximum number of bytes per second
    - » Depends on "wire speed", but also limited by slowest router (routing delay) or by congestion at routers
  - Latency: time until first bit of packet arrives at receiver
    - » Raw transfer time + overhead at each routing hop



- Contributions to Latency
  - Wire latency: depends on speed of light on wire
    - » about 1-1.5 ns/foot
  - Router latency: depends on internals of router
    - » Could be < 1 ms (for a good router)
    - » Question: can router handle full wire throughput?

Sample Computations

- E.g.: Ethernet within Soda
  - Latency: speed of light in wire is 1.5ns/foot, which implies latency in building < 1 μs (if no routers in path)
  - Throughput: 10-1000Mb/s
  - Throughput delay: packet doesn't arrive until all bits
    - » So: 4KB/100Mb/s = 0.3 milliseconds (same order as disk!)
- E.g.: ATM within Soda
  - Latency (same as above, assuming no routing)
  - Throughput: 155Mb/s
  - Throughput delay: 4KB/155Mb/s = 200μ
- E.g.: ATM cross-country
  - Latency (assuming no routing):
    - » 3000miles * 5000ft/mile ⇒ 15 milliseconds
  - How many bits could be in transit at same time?
    - » 15ms * 155Mb/s = 290KB
  - In fact, Berkeley→MIT Latency ~ 45ms
    - » 872KB in flight if routers have wire-speed throughput
- Requirements for good performance:
  - Local area: minimize overhead/improve bandwidth
  - Wide area: keep pipeline full!

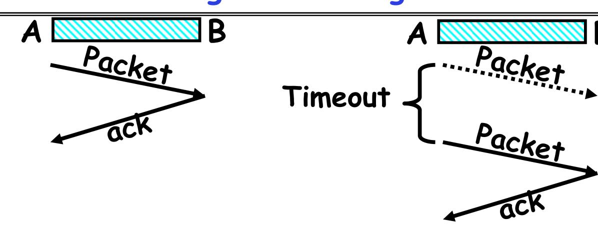
Sequence Numbers

- Ordered Messages
  - Several network services are best constructed by ordered messaging
    - » Ask remote machine to first do x, then do y, etc.
  - Unfortunately, underlying network is packet based:
    - » Packets are routed one at a time through the network
    - » Can take different paths or be delayed individually
  - IP can reorder packets! P_0,P_1 might arrive as P_1,P_0
- Solution requires queuing at destination
  - Need to hold onto packets to undo misordering
  - Total degree of reordering impacts queue size
- Ordered messages on top of unordered ones:
  - Assign sequence numbers to packets
    - » 0,1,2,3,4…..
    - » If packets arrive out of order, reorder before delivering to user application
    - » For instance, hold onto #3 until #2 arrives, etc.
  - Sequence numbers are specific to particular connection
    - » Reordering among connections normally doesn't matter
  - If restart connection, need to make sure use different range of sequence numbers than previously…

Reliable Message Delivery: the Problem

- All physical networks can garble and/or drop packets
  - Physical media: packet not transmitted/received
    - » If transmit close to maximum rate, get more throughput – even if some packets get lost
    - » If transmit at lowest voltage such that error correction just starts correcting errors, get best power/bit
  - Congestion: no place to put incoming packet
    - » Point-to-point network: insufficient queue at switch/router
    - » Broadcast link: two host try to use same link
    - » In any network: insufficient buffer space at destination
    - » Rate mismatch: what if sender send faster than receiver can process?
- Reliable Message Delivery on top of Unreliable Packets
  - Need some way to make sure that packets actually make it to receiver
    - » Every packet received at least once
    - » Every packet received at most once
  - Can combine with ordering: every packet received by process at destination exactly once and in order

Using Acknowledgements



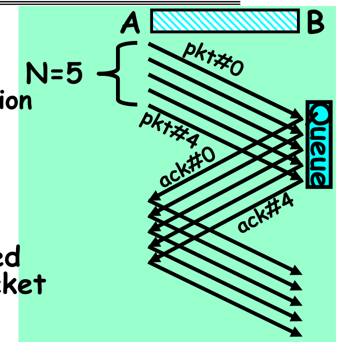
- How to ensure transmission of packets?
  - Detect garbling at receiver via checksum, discard if bad
  - Receiver acknowledges (by sending "ack") when packet received properly at destination
  - Timeout at sender: if no ack, retransmit
- Some questions:
  - If the sender doesn't get an ack, does that mean the receiver didn't get the original message?
    - » No
  - What if ack gets dropped? Or if message gets delayed?
    - » Sender doesn't get ack, retransmits. Receiver gets message twice, acks each.

How to deal with message duplication

- Solution: put sequence number in message to identify re-transmitted packets
  - Receiver checks for duplicate #'s; Discard if detected
- Requirements:
  - Sender keeps copy of unack'ed messages
    - » Easy: only need to buffer messages
  - Receiver tracks possible duplicate messages
    - » Hard: when ok to forget about received message?
- Alternating-bit protocol:
  - Send one message at a time; don't send next message until ack received
  - Sender keeps last message; receiver tracks sequence # of last message received
- Pros: simple, small overhead
- Con: Poor performance
  - Wire can hold multiple messages; want to fill up at (wire latency × throughput)
- Con: doesn't work if network can delay or duplicate messages arbitrarily

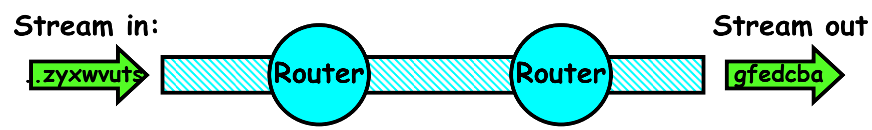
Better messaging: Window-based acknowledgements

- Window based protocol (TCP):
  - Send up to N packets without ack
    - » Allows pipelining of packets
    - » Window size (N) < queue at destination
  - Each packet has sequence number
    - » Receiver acknowledges each packet
    - » Ack says "received all packets up to sequence number X"/send more
- Acks serve dual purpose:
  - Reliability: Confirming packet received
  - Flow Control: Receiver ready for packet
    - » Remaining space in queue at receiver can be returned with ACK
- What if packet gets garbled/dropped?
  - Sender will timeout waiting for ack packet
    - » Resend missing packets⇒ Receiver gets packets out of order!
  - Should receiver discard packets that arrive out of order?
    - » Simple, but poor performance
  - Alternative: Keep copy until sender fills in missing pieces?
    - » Reduces # of retransmits, but more complex
- What if ack gets garbled/dropped?
  - Timeout and resend just the un-acknowledged packets



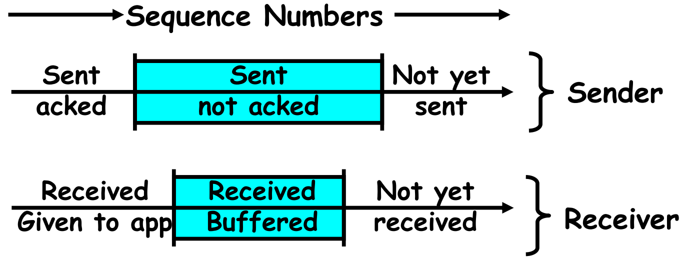
Administrivia

Transmission Control Protocol (TCP)



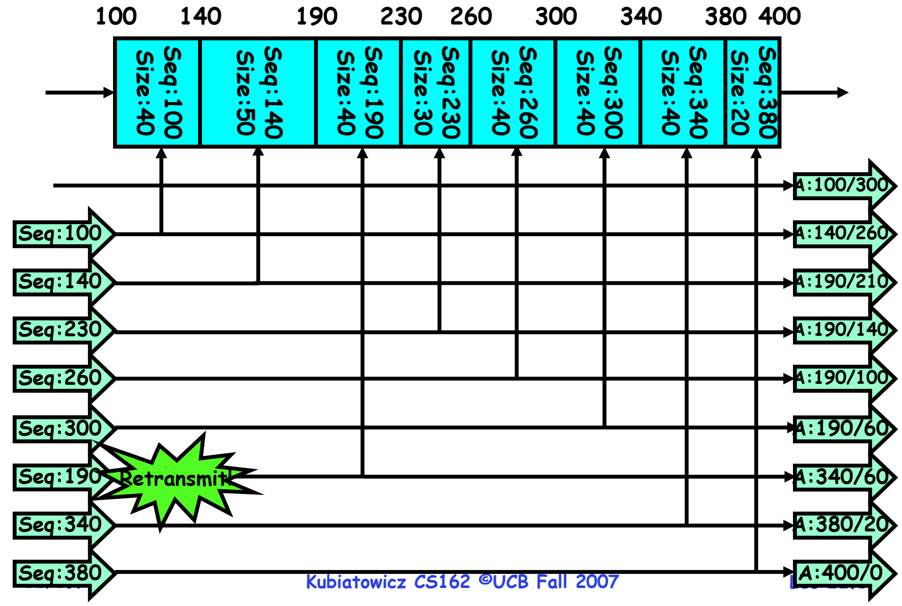
- Transmission Control Protocol (TCP)
  - TCP (IP Protocol 6) layered on top of IP
  - Reliable byte stream between two processes on different machines over Internet (read, write, flush)
- TCP Details
  - Fragments byte stream into packets, hands packets to IP
    - » IP may also fragment by itself
  - Uses window-based acknowledgement protocol (to minimize state at sender and receiver)
    - » "Window" reflects storage at receiver – sender shouldn't overrun receiver's buffer space
    - » Also, window should reflect speed/capacity of network – sender shouldn't overload network
  - Automatically retransmits lost packets
  - Adjusts rate of transmission to avoid congestion
    - » A "good citizen"

TCP Windows and Sequence Numbers

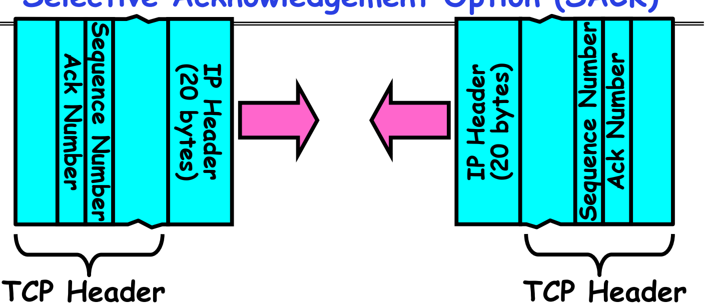


- Sender has three regions:
  - Sequence regions
    - » sent and ack'ed
    - » Sent and not ack'ed
    - » not yet sent
  - Window (colored region) adjusted by sender
- Receiver has three regions:
  - Sequence regions
    - » received and ack'ed (given to application)
    - » received and buffered
    - » not yet received (or discarded because out of order)

Window-Based Acknowledgements (TCP)



Selective Acknowledgement Option (SACK)



- Vanilla TCP Acknowledgement
  - Every message encodes Sequence number and Ack
  - Can include data for forward stream and/or ack for reverse stream
- Selective Acknowledgement
  - Acknowledgement information includes not just one number, but rather ranges of received packets
  - Must be specially negotiated at beginning of TCP setup
    - » Not widely in use (although in Windows since Windows 98)

Congestion Avoidance

- Congestion
  - How long should timeout be for re-sending messages?
    - » Too long→wastes time if message lost
    - » Too short→retransmit even though ack will arrive shortly
  - Stability problem: more congestion ⇒ ack is delayed ⇒ unnecessary timeout ⇒ more traffic ⇒ more congestion
    - » Closely related to window size at sender: too big means putting too much data into network
- How does the sender's window size get chosen?
  - Must be less than receiver's advertised buffer size
  - Try to match the rate of sending packets with the rate that the slowest link can accommodate
  - Sender uses an adaptive algorithm to decide size of N
    - » Goal: fill network between sender and receiver
    - » Basic technique: slowly increase size of window until acknowledgements start being delayed/lost
- TCP solution: "slow start" (start sending slowly)
  - If no timeout, slowly increase window size (throughput) by 1 for each ack received
  - Timeout ⇒ congestion, so cut window size in half
  - "*Additive Increase, Multiplicative Decrease*"

Sequence-Number Initialization

- How do you choose an initial sequence number?
  - When machine boots, ok to start with sequence #0?
    - » No: could send two messages with same sequence #!
    - » Receiver might end up discarding valid packets, or duplicate ack from original transmission might hide lost packet
  - Also, if it is possible to predict sequence numbers, might be possible for attacker to hijack TCP connection
- Some ways of choosing an initial sequence number:
  - Time to live: each packet has a deadline.
    - » If not delivered in X seconds, then is dropped
    - » Thus, can re-use sequence numbers if wait for all packets in flight to be delivered or to expire
  - Epoch #: uniquely identifies *which* set of sequence numbers are currently being used
    - » Epoch # stored on disk, Put in every message
    - » Epoch # incremented on crash and/or when run out of sequence #
  - Pseudo-random increment to previous sequence number
    - » Used by several protocol implementations

Use of TCP: Sockets

- Socket: an abstraction of a network I/O queue
  - Embodies one side of a communication channel
    - » Same interface regardless of location of other end
    - » Could be local machine (called "UNIX socket") or remote machine (called "network socket")
  - First introduced in 4.2 BSD UNIX: big innovation at time
    - » Now most operating systems provide some notion of socket
- Using Sockets for Client-Server (C/C++ interface):
  - On server: set up "server-socket"
    - » Create socket, Bind to protocol (TCP), local address, port
    - » Call listen(): tells server socket to accept incoming requests
    - » Perform multiple accept() calls on socket to accept incoming connection request
    - » Each successful accept() returns a new socket for a new connection; can pass this off to handler thread
  - On client:
    - » Create socket, Bind to protocol (TCP), remote address, port
    - » Perform connect() on socket to make connection
    - » If connect() successful, have socket connected to server

Socket Example (Java)

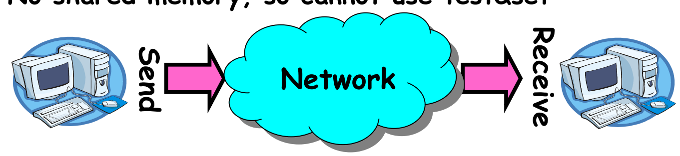
```
server:
      //Makes socket, binds addr/port, calls listen()
      ServerSocket sock = new ServerSocket(6013);
      while(true) {
         Socket client = sock.accept();
         PrintWriter pout = new
            PrintWriter(client.getOutputStream(),true);

         pout.println("Here is data sent to client!");
            …
         client.close();
      }

client:
      // Makes socket, binds addr/port, calls connect()
      Socket sock = new Socket("169.229.60.38",6013);
      BufferedReader bin =
         new BufferedReader(
            new InputStreamReader(sock.getInputStream));
      String line;
      while ((line = bin.readLine())!=null)
         System.out.println(line);
      sock.close();
```


Distributed Applications

- How do you actually program a distributed application?
  - Need to synchronize multiple threads, running on different machines
    - » No shared memory, so cannot use test&set



  - One Abstraction: send/receive messages
    - » Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
  - Mailbox (`mbox`): temporary holding area for messages
    - » Includes both destination location and queue
  - `Send(message,mbox)`
    - » Send message to remote mailbox identified by `mbox`
  - `Receive(buffer,mbox)`
    - » Wait until `mbox` has message, copy into buffer, and return
    - » If threads sleeping on this mbox, wake up one of them

Using Messages: Send/Receive behavior

- When should `send(message,mbox)` return?
  - When receiver gets message? (i.e. ack received)
  - When message is safely buffered on destination?
  - Right away, if message is buffered on source node?
- Actually two questions here:
  - When can the sender be sure that the receiver actually received the message?
  - When can sender reuse the memory containing message?
- Mailbox provides 1-way communication from T1→T2
  - T1→buffer→T2
  - Very similar to producer/consumer
    - » Send = V, Receive = P
    - » However, can't tell if sender/receiver is local or not!

Messaging for Producer-Consumer Style

- Using send/receive for producer-consumer style:

```
Producer:
   int msg1[1000];
   while(1) {
      prepare message;
      send(msg1,mbox);
   }
Consumer:
   int buffer[1000];
   while(1) {
      receive(buffer,mbox);
      process message;
   }
```

Send Message

Receive Message

- No need for producer/consumer to keep track of space in mailbox: handled by send/receive
  - One of the roles of the window in TCP: window is size of buffer on far end
  - Restricts sender to forward only what will fit in buffer

Messaging for Request/Response communication

- What about two-way communication?
  - Request/Response
    - » Read a file stored on a remote machine
    - » Request a web page from a remote web server
  - Also called: client-server
    - » Client ≡ requester, Server ≡ responder
    - » Server provides "service" (file storage) to the client
- Example: File service

```
Client: (requesting the file)
   char response[1000];

   send("read rutabaga", server_mbox);
   receive(response, client_mbox);

Consumer: (responding with the file)
   char command[1000], answer[1000];

   receive(command, server_mbox);
   decode command;
   read file into answer;
   send(answer, client_mbox);
```

Request File

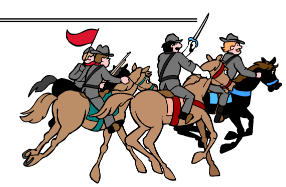
Get Response

Receive Request

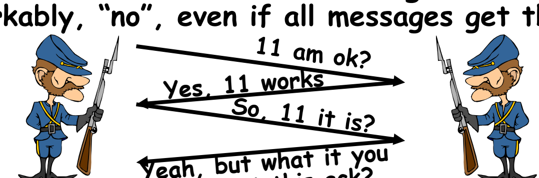
Send Response

General's Paradox

- General's paradox:
  - Constraints of problem:
    - » Two generals, on separate mountains
    - » Can only communicate via messengers
    - » Messengers can be captured
  - Problem: need to coordinate attack
    - » If they attack at different times, they all die
    - » If they attack at same time, they win
  - Named after Custer, who died at Little Big Horn because he arrived a couple of days too early



- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  - Remarkably, "no", even if all messages get through



  - No way to be sure last message gets through!

Two-Phase Commit

- Since we can't solve the General's Paradox (i.e. simultaneous action), let's solve a related problem
  - Distributed transaction: Two machines agree to do something, or not do it, atomically
- Two-Phase Commit protocol does this
  - Use a persistent, stable log on each machine to keep track of whether commit has happened
    - » If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash
  - Prepare Phase:
    - » The global coordinator requests that all participants will promise to commit or rollback the transaction
    - » Participants record promise in log, then acknowledge
    - » If anyone votes to abort, coordinator writes "abort" in its log and tells everyone to abort; each records "abort" in log
  - Commit Phase:
    - » After all participants respond that they are prepared, then the coordinator writes "commit" to its log
    - » Then asks all nodes to commit; they respond with ack
    - » After receive acks, coordinator writes "got commit" to log
  - Log can be used to complete this process such that all machines either commit or don't commit

Two phase commit example

- Simple Example: A≡ATM machine, B≡The Bank
  - Phase 1:
    - » A writes "Begin transaction" to log
      A→B: OK to transfer funds to me?
    - » Not enough funds:
      B→A: transaction aborted; A writes "Abort" to log
    - » Enough funds:
      B: Write new account balance to log
      B→A: OK, I can commit
  - Phase 2: A can decide for both whether they will commit
    - » A: write new account balance to log
    - » Write "commit" to log
    - » Send message to B that commit occurred; wait for ack
    - » Write "Got Commit" to log
- What if B crashes at beginning?
  - Wakes up, does nothing; A will timeout, abort and retry
- What if A crashes at beginning of phase 2?
  - Wakes up, sees transaction in progress; sends "abort" to B
- What if B crashes at beginning of phase 2?
  - B comes back up, look at log; when A sends it "Commit" message, it will say, oh, ok, commit

Distributed Decision Making Discussion

- Two-Phase Commit: Blocking
  - A Site can get stuck in a situation where it cannot continue until some other site (usually the coordinator) recovers.
  - Example of how this could happen:
    - » Participant site B writes a "prepared to commit" record to its log, sends a "yes" vote to the coordintor (site A) and crashes
    - » Site A crashes
    - » Site B wakes up, check its log, and realizes that it has voted "yes" on the update. It sends a message to site A asking what happened. At this point, B cannot change its mind and decide to abort, because update may have committed
    - » B is blocked until A comes back
  - Blocking is problematic because a blocked site must hold resources (locks on updated items, pagespinned in memory, etc) until it learns fate of update
- Alternative: There are alternatives such as "Three Phase Commit" which don't have this blocking problem

Conclusion

- Layering: building complex services from simpler ones
- Datagram: an independent, self-contained network message whose arrival, arrival time, and content are not guaranteed
- Performance metrics
  - Overhead: CPU time to put packet on wire
  - Throughput: Maximum number of bytes per second
  - Latency: time until first bit of packet arrives at receiver
- Arbitrary Sized messages:
  - Fragment into multiple packets; reassemble at destination
- Ordered messages:
  - Use sequence numbers and reorder at destination
- Reliable messages:
  - Use Acknowledgements
  - Want a window larger than 1 in order to increase throughput
- TCP: Reliable byte stream between two processes on different machines over Internet (read, write, flush)
- Two-phase commit: distributed decision making

CS162
Operating Systems and Systems Programming
Lecture 23

Network Communication Abstractions / Remote Procedure Call

November 21, 2007

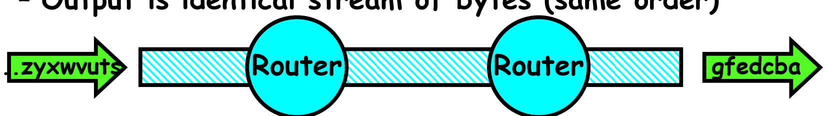
Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs162

Review: Reliable Networking

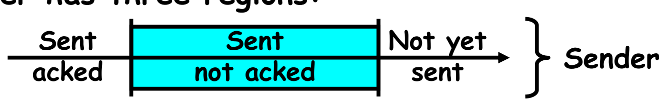
- Layering: building complex services from simpler ones
- Datagram: an independent, self-contained network message whose arrival, arrival time, and content are not guaranteed
- Performance metrics
  - Overhead: CPU time to put packet on wire
  - Throughput: Maximum number of bytes per second
  - Latency: time until first bit of packet arrives at receiver
- Arbitrary Sized messages:
  - Fragment into multiple packets; reassemble at destination
- Ordered messages:
  - Use sequence numbers and reorder at destination
- Reliable messages:
  - Use Acknowledgements
  - Want a window larger than 1 in order to increase throughput

Review: TCP Windows and Sequence Numbers

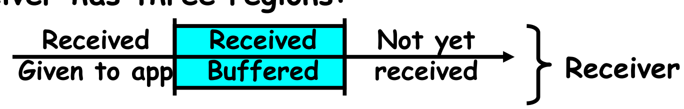
- TCP provides a stream abstraction:
  - Reliable byte stream between two processes on different machines over Internet (read, write, flush)
  - Input is an unbounded stream of bytes
  - Output is identical stream of bytes (same order)



- Sender has three regions:



  - Window (colored region) adjusted by sender
- Receiver has three regions:



  - Maximum size of window advertised to sender at setup

Review: Congestion Avoidance

- Two issues
  - Choose appropriate message timeout value
    - » Too long→wastes time if message lost
    - » Too short→retransmit even though ack will arrive shortly
  - Choose appropriate sender's window
    - » Try to match the rate of sending packets with the rate that the slowest link can accommodate
    - » Max is receiver's advertised window size
- TCP solution: "slow start" (start sending slowly)
  - Measure/estimate Round-Trip Time
  - Use adaptive algorithm to fill network (compute win size)
    - » Basic technique: slowly increase size of window until acknowledgements start being delayed/lost
  - Set window size to one packet
  - If no timeout, slowly increase window size (throughput)
    - » 1 packet per ACK, up to receiver's advertised buffer size
  - Timeout ⇒ congestion, so cut window size in half
  - "*Additive Increase, Multiplicative Decrease*"

Goals for Today

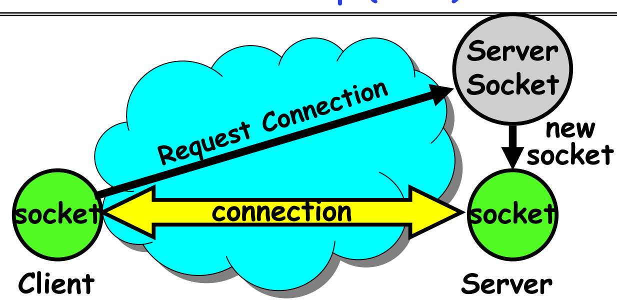
- Messages
  - Send/receive
  - One vs. two-way communication
- Distributed Decision Making
  - Two-phase commit/Byzantine Commit
- Remote Procedure Call

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Use of TCP: Sockets

- **Socket:** an abstraction of a network I/O queue
  - Embodies one side of a communication channel
    - » Same interface regardless of location of other end
    - » Could be local machine (called "UNIX socket") or remote machine (called "network socket")
  - First introduced in 4.2 BSD UNIX: big innovation at time
    - » Now most operating systems provide some notion of socket
- Using Sockets for Client-Server (C/C++ interface):
  - On server: set up "server-socket"
    - » Create socket, Bind to protocol (TCP), local address, port
    - » call listen(): tells server socket to accept incoming requests
    - » Perform multiple accept() calls on socket to accept incoming connection request
    - » Each successful accept() returns a new socket for a new connection; can pass this off to handler thread
  - On client:
    - » Create socket, Bind to protocol (TCP), remote address, port
    - » Perform connect() on socket to make connection
    - » If connect() successful, have socket connected to server

Socket Setup (Con't)



- Things to remember:
  - Connection requires 5 values:
    [Src Addr, Src Port, Dst Addr, Dst Port, Protocol]
  - Often, Src Port "randomly" assigned
    - » Done by OS during client socket setup
  - Dst Port often "well known"
    - » 80 (web), 443 (secure web), 25 (sendmail), etc
    - » Well-known ports from 0—1023

Socket Example (Java)

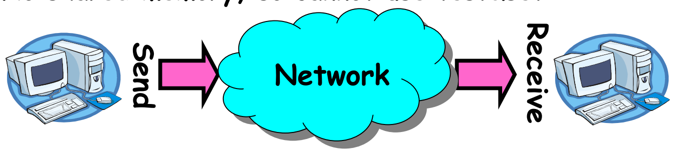
```
server:
      //Makes socket, binds addr/port, calls listen()
      ServerSocket sock = new ServerSocket(6013);
      while(true) {
        Socket client = sock.accept();
        PrintWriter pout = new
          PrintWriter(client.getOutputStream(),true);

        pout.println("Here is data sent to client!");
          …
        client.close();
      }

client:
      // Makes socket, binds addr/port, calls connect()
      Socket sock = new Socket("169.229.60.38",6018);
      BufferedReader bin =
        new BufferedReader(
          new InputStreamReader(sock.getInputStream));
      String line;
      while ((line = bin.readLine())!=null)
        System.out.println(line);
      sock.close();
```

Distributed Applications

- How do you actually program a distributed application?
  - Need to synchronize multiple threads, running on different machines
    - » No shared memory, so cannot use test&set



  - One Abstraction: send/receive messages
    - » Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
  - Mailbox (`mbox`): temporary holding area for messages
    - » Includes both destination location and queue
  - `Send(message,mbox)`
    - » Send message to remote mailbox identified by `mbox`
  - `Receive(buffer,mbox)`
    - » Wait until `mbox` has message, copy into buffer, and return
    - » If threads sleeping on this mbox, wake up one of them

Using Messages: Send/Receive behavior

- When should `send(message,mbox)` return?
  - When receiver gets message? (i.e. ack received)
  - When message is safely buffered on destination?
  - Right away, if message is buffered on source node?
- Actually two questions here:
  - When can the sender be sure that receive actually received the message?
  - When can sender reuse the memory containing message?
- Mailbox provides 1-way communication from T1→T2
  - T1→buffer→T2
  - Very similar to producer/consumer
    - » Send = V, Receive = P
    - » However, can't tell if sender/receiver is local or not!

Messaging for Producer-Consumer Style

- Using send/receive for producer-consumer style:

```
Producer:
   int msg1[1000];
   while(1) {
      prepare message;
      send(msg1,mbox);
   }
Consumer:
   int buffer[1000];
   while(1) {
      receive(buffer,mbox);
      process message;
   }
```

Send Message (send(msg1,mbox);)

Receive Message (receive(buffer,mbox);)

- No need for producer/consumer to keep track of space in mailbox: handled by send/receive
  - One of the roles of the window in TCP: window is size of buffer on far end
  - Restricts sender to forward only what will fit in buffer

Messaging for Request/Response communication

- What about two-way communication?
  - Request/Response
    - » Read a file stored on a remote machine
    - » Request a web page from a remote web server
  - Also called: client-server
    - » Client ≡ requester, Server ≡ responder
    - » Server provides "service" (file storage) to the client
- Example: File service

```
Client: (requesting the file)
   char response[1000];

   send("read rutabaga", server_mbox);
   receive(response, client_mbox);

Server: (responding with the file)
   char command[1000], answer[1000];

   receive(command, server_mbox);
   decode command;
   read file into answer;
   send(answer, client_mbox);
```

Request File (send("read rutabaga", server_mbox);)

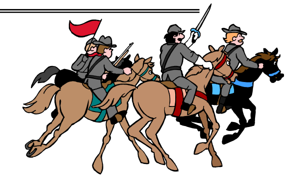
Get Response (receive(response, client_mbox);)

Receive Request (receive(command, server_mbox);)

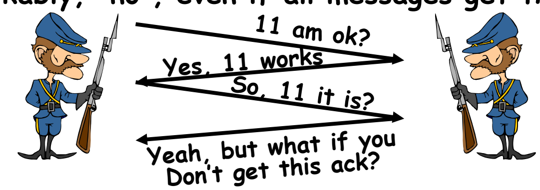
Send Response (send(answer, client_mbox);)

Administrivia

General's Paradox



- General's paradox:
  - Constraints of problem:
    - » Two generals, on separate mountains
    - » Can only communicate via messengers
    - » Messengers can be captured
  - Problem: need to coordinate attack
    - » If they attack at different times, they all die
    - » If they attack at same time, they win
  - Named after Custer, who died at Little Big Horn because he arrived a couple of days too early
- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  - Remarkably, "no", even if all messages get through



  - No way to be sure last message gets through!

Two-Phase Commit

- Since we can't solve the General's Paradox (i.e. simultaneous action), let's solve a related problem
  - Distributed transaction: Two machines agree to do something, or not do it, atomically
- Two-Phase Commit protocol does this
  - Use a persistent, stable log on each machine to keep track of whether commit has happened
    - » If a machine crashes, when it wakes up it first checks its log to recover state of world at time of crash
  - Prepare Phase:
    - » The global coordinator requests that all participants will promise to commit or rollback the transaction
    - » Participants record promise in log, then acknowledge
    - » If anyone votes to abort, coordinator writes "Abort" in its log and tells everyone to abort; each records "Abort" in log
  - Commit Phase:
    - » After all participants respond that they are prepared, then the coordinator writes "Commit" to its log
    - » Then asks all nodes to commit; they respond with ack
    - » After receive acks, coordinator writes "Got Commit" to log
  - Log can be used to complete this process such that all machines either commit or don't commit

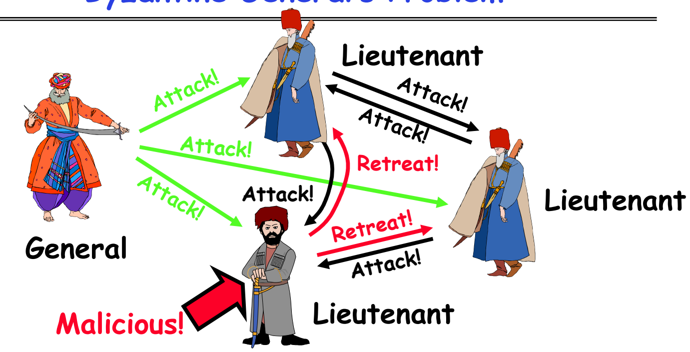
Two phase commit example

- Simple Example: A≡WellsFargo Bank, B≡Bank of America
  - Phase 1: Prepare Phase
    - » A writes "Begin transaction" to log
      A→B: OK to transfer funds to me?
    - » Not enough funds:
      B→A: transaction aborted; A writes "Abort" to log
    - » Enough funds:
      B: Write new account balance & promise to commit to log
      B→A: OK, I can commit
  - Phase 2: A can decide for both whether they will commit
    - » A: write new account balance to log
    - » Write "Commit" to log
    - » Send message to B that commit occurred; wait for ack
    - » Write "Got Commit" to log
- What if B crashes at beginning?
  - Wakes up, does nothing; A will timeout, abort and retry
- What if A crashes at beginning of phase 2?
  - Wakes up, sees that there is a transaction in progress; sends "Abort" to B
- What if B crashes at beginning of phase 2?
  - B comes back up, looks at log; when A sends it "Commit" message, it will say, "oh, ok, commit"

Distributed Decision Making Discussion

- Why is distributed decision making desirable?
  - Fault Tolerance!
  - A group of machines can come to a decision even if one or more of them fail during the process
    - » Simple failure mode called "failstop" (different modes later)
  - After decision made, result recorded in multiple places
- Undesirable feature of Two-Phase Commit: Blocking
  - One machine can be stalled until another site recovers:
    - » Site B writes `"prepared to commit"` record to its log, sends a "yes" vote to the coordinator (site A) and crashes
    - » Site A crashes
    - » Site B wakes up, check its log, and realizes that it has voted "yes" on the update. It sends a message to site A asking what happened. At this point, B cannot decide to abort, because update may have committed
    - » B is blocked until A comes back
  - A blocked site holds resources (locks on updated items, pages pinned in memory, etc) until learns fate of update
- Alternative: There are alternatives such as "Three Phase Commit" which don't have this blocking problem
- What happens if one or more of the nodes is malicious?
  - Malicious: attempting to compromise the decision making

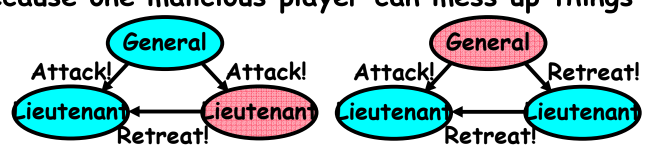
Byzantine General's Problem



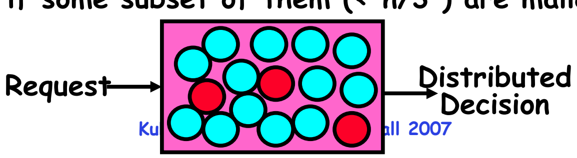
- Byazantine General's Problem (n players):
  - One General
  - n-1 Lieutenants
  - Some number of these (f) can be insane or malicious
- The commanding general must send an order to his n-1 lieutenants such that:
  - IC1: All loyal lieutenants obey the same order
  - IC2: If the commanding general is loyal, then all loyal lieutenants obey the order he sends

Byzantine General's Problem (con't)

- Impossibility Results:
  - Cannot solve Byzantine General's Problem with n=3 because one malicious player can mess up things



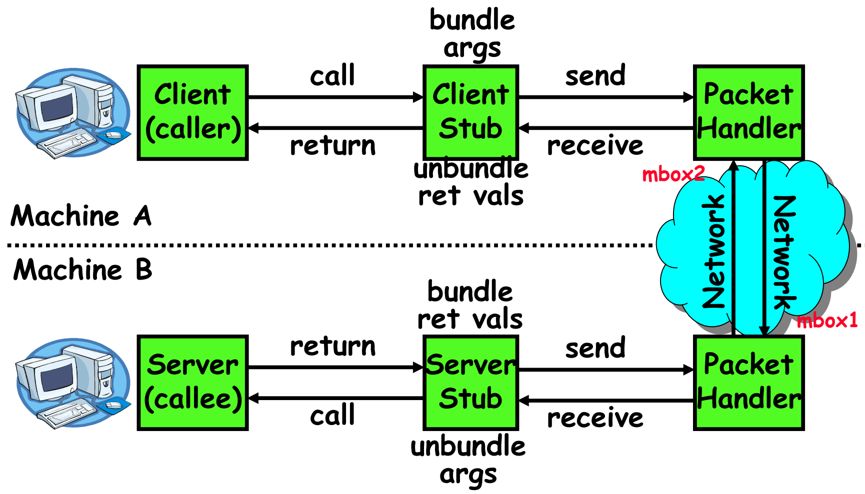
  - With f faults, need n > 3f to solve problem
- Various algorithms exist to solve problem
  - Original algorithm has #messages exponential in n
  - Newer algorithms have message complexity $O(n^2)$
    - » One from MIT, for instance (Castro and Liskov, 1999)
- Use of BFT (Byzantine Fault Tolerance) algorithm
  - Allow multiple machines to make a coordinated decision even if some subset of them (< n/3) are malicious



Remote Procedure Call

- Raw messaging is a bit too low-level for programming
  - Must wrap up information into message at source
  - Must decide what to do with message at destination
  - May need to sit and wait for multiple messages to arrive
- Better option: Remote Procedure Call (RPC)
  - Calls a procedure on a remote machine
  - Client calls:
    `remoteFileSystem→Read("rutabaga");`
  - Translated automatically into call on server:
    `fileSys→Read("rutabaga");`
- Implementation:
  - Request-response message passing (under covers!)
  - "Stub" provides glue on client/server
    - » Client stub is responsible for "marshalling" arguments and "unmarshalling" the return values
    - » Server-side stub is responsible for "unmarshalling" arguments and "marshalling" the return values.
- Marshalling involves (depending on system)
  - Converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.

RPC Information Flow



RPC Details

- Equivalence with regular procedure call
  - Parameters ⇔ Request Message
  - Result ⇔ Reply message
  - Name of Procedure: Passed in request message
  - Return Address: mbox2 (client return mail box)
- Stub generator: Compiler that generates stubs
  - Input: interface definitions in an "interface definition language (IDL)"
    - » Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language
    - » Code for client to pack message, send it off, wait for result, unpack result and return to caller
    - » Code for server to unpack message, call procedure, pack results, send them off
- Cross-platform issues:
  - What if client/server machines are different architectures or in different languages?
    - » Convert everything to/from some canonical form
    - » Tag every item with an indication of how it is encoded (avoids unnecessary conversions).

RPC Details (continued)

- How does client know which mbox to send to?
  - Need to translate name of remote service into network endpoint (Remote machine, port, possibly other info)
  - Binding: the process of converting a user-visible name into a network endpoint
    - » This is another word for "naming" at network level
    - » Static: fixed at compile time
    - » Dynamic: performed at runtime
- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    - » Name service provides dynmaic translation of service→mbox
  - Why dynamic binding?
    - » Access control: check who is permitted to access service
    - » Fail-over: If server fails, use a different one
- What if there are multiple servers?
  - Could give flexibility at binding time
    - » Choose unloaded server for each new client
  - Could provide same mbox (router level redirect)
    - » Choose unloaded server for each new request
    - » Only works if no state carried from one call to next
- What if multiple clients?
  - Pass pointer to client-specific return mbox in request

Problems with RPC

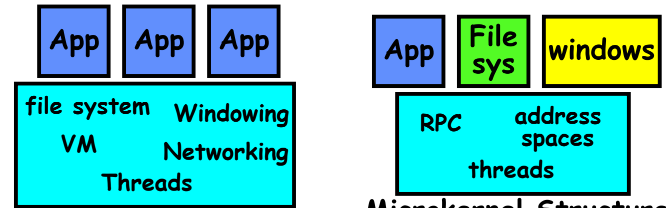
- Non-Atomic failures
  - Different failure modes in distributed system than on a single machine
  - Consider many different types of failures
    - » User-level bug causes address space to crash
    - » Machine failure, kernel bug causes all processes on same machine to fail
    - » Some machine is compromised by malicious party
  - Before RPC: whole system would crash/die
  - After RPC: One machine crashes/compromised while others keep working
  - Can easily result in inconsistent view of the world
    - » Did my cached data get written back or not?
    - » Did server do what I requested or not?
  - Answer? Distributed transactions/Byzantine Commit
- Performance
  - Cost of Procedure call « same-machine RPC « network RPC
  - Means programmers must be aware that RPC is not free
    - » Caching can help, but may make failure handling complex

Cross-Domain Communication/Location Transparency

- How do address spaces communicate with one another?
  - Shared Memory with Semaphores, monitors, etc…
  - File System
  - Pipes (1-way communication)
  - "Remote" procedure call (2-way communication)
- RPC's can be used to communicate between address spaces on different machines or the same machine
  - Services can be run wherever it's most appropriate
  - Access to local and remote services looks the same
- Examples of modern RPC systems:
  - CORBA (Common Object Request Broker Architecture)
  - DCOM (Distributed COM)
  - RMI (Java Remote Method Invocation)

Microkernel operating systems

- Example: split kernel into application-level servers.
  - File system looks remote, even though on same machine



Monolithic Structure   Microkernel Structure

- Why split the OS into separate domains?
  - Fault isolation: bugs are more isolated (build a firewall)
  - Enforces modularity: allows incremental upgrades of pieces of software (client or server)
  - Location transparent: service can be local or remote
    - » For example in the X windowing system: Each X client can be on a separate machine from X server; Neither has to run on the machine with the frame buffer.

Conclusion

- **TCP:** Reliable byte stream between two processes on different machines over Internet (read, write, flush)
  - Uses window-based acknowledgement protocol
  - Congestion-avoidance dynamically adapts sender window to account for congestion in network
- **Two-phase commit:** distributed decision making
  - First, make sure everyone guarantees that they will commit if asked (prepare)
  - Next, ask everyone to commit
- **Byzantine General's Problem:** distributed decision making with malicious failures
  - One general, n-1 lieutenants: some number of them may be malicious (often "f" of them)
  - All non-malicious lieutenants must come to same decision
  - If general not malicious, lieutenants must follow general
  - Only solvable if $n \geq 3f+1$
- **Remote Procedure Call (RPC):** Call procedure on remote machine
  - Provides same interface as procedure
  - Automatic packing and unpacking of arguments without user programming (in stub)

CS162
Operating Systems and Systems Programming
Lecture 24

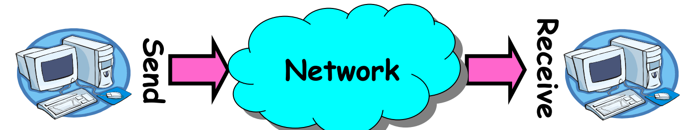
Testing Methodologies/ Distributed File Systems

November 26, 2007

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

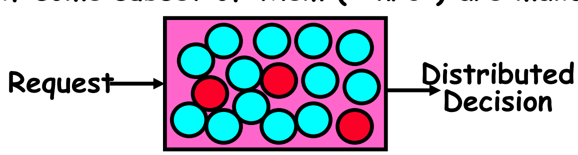
Review: Distributed Applications



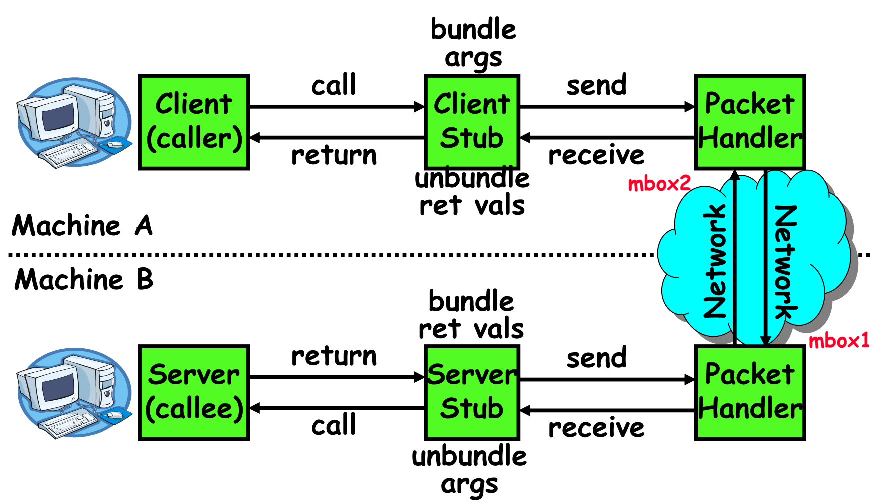
- Message Abstraction: send/receive messages
  - Already atomic: no receiver gets portion of a message and two receivers cannot get same message
- Interface:
  - Mailbox (`mbox`): temporary holding area for messages
    - » Includes both destination location and queue
  - `Send(message,mbox)`
    - » Send message to remote mailbox identified by `mbox`
  - `Receive(buffer,mbox)`
    - » Wait until `mbox` has message, copy into buffer, and return
    - » If threads sleeping on this mbox, wake up one of them
- Two-phase commit: distributed decision making
  - First, make sure everyone guarantees that they will commit if asked (prepare)
  - Next, ask everyone to commit

Review: Byzantine General's Problem

- Byazantine General's Problem (n players):
  - One General
  - n-1 Lieutenants
  - Some number of these (f<n/3) can be insane or malicious
- The commanding general must send an order to his n-1 lieutenants such that:
  - IC1: All loyal lieutenants obey the same order
  - IC2: If the commanding general is loyal, then all loyal lieutenants obey the order he sends
- Various algorithms exist to solve problem
  - Newer algorithms have message complexity $O(n^2)$
- Use of BFT (Byzantine Fault Tolerance) algorithm
  - Allow multiple machines to make a coordinated decision even if some subset of them (< n/3) are malicious



Review: RPC Information Flow



Goals for Today

- Finish RPC
- Testing Methodologies
- Examples of Distributed File Systems
- Cache Coherence Protocols

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Slides on Testing from George Necula (CS169)
Many slides generated from my lecture notes by Kubiatowicz.

RPC Details

- Equivalence with regular procedure call
  - Parameters ⇔ Request Message
  - Result ⇔ Reply message
  - Name of Procedure: Passed in request message
  - Return Address: mbox2 (client return mail box)
- Stub generator: Compiler that generates stubs
  - Input: interface definitions in an "interface definition language (IDL)"
    - » Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language
    - » Code for client to pack message, send it off, wait for result, unpack result and return to caller
    - » Code for server to unpack message, call procedure, pack results, send them off
- Cross-platform issues:
  - What if client/server machines are different architectures or in different languages?
    - » Convert everything to/from some canonical form
    - » Tag every item with an indication of how it is encoded (avoids unnecessary conversions).

RPC Details (continued)

- How does client know which mbox to send to?
  - Need to translate name of remote service into network endpoint (Remote machine, port, possibly other info)
  - **Binding:** the process of converting a user-visible name into a network endpoint
    - » This is another word for "naming" at network level
    - » Static: fixed at compile time
    - » Dynamic: performed at runtime
- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    - » Name service provides dynmaic translation of service→mbox
  - Why dynamic binding?
    - » Access control: check who is permitted to access service
    - » Fail-over: If server fails, use a different one
- What if there are multiple servers?
  - Could give flexibility at binding time
    - » Choose unloaded server for each new client
  - Could provide same mbox (router level redirect)
    - » Choose unloaded server for each new request
    - » Only works if no state carried from one call to next
- What if multiple clients?
  - Pass pointer to client-specific return mbox in request

Problems with RPC

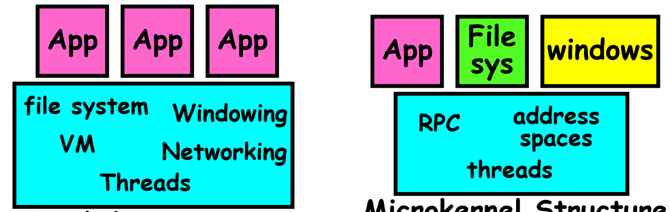
- Non-Atomic failures
  - Different failure modes in distributed system than on a single machine
  - Consider many different types of failures
    - » User-level bug causes address space to crash
    - » Machine failure, kernel bug causes all processes on same machine to fail
    - » Some machine is compromised by malicious party
  - Before RPC: whole system would crash/die
  - After RPC: One machine crashes/compromised while others keep working
  - Can easily result in inconsistent view of the world
    - » Did my cached data get written back or not?
    - » Did server do what I requested or not?
  - Answer? Distributed transactions/Byzantine Commit
- Performance
  - Cost of Procedure call « same-machine RPC « network RPC
  - Means programmers must be aware that RPC is not free
    - » Caching can help, but may make failure handling complex

Cross-Domain Communication/Location Transparency

- How do address spaces communicate with one another?
  - Shared Memory with Semaphores, monitors, etc…
  - File System
  - Pipes (1-way communication)
  - "Remote" procedure call (2-way communication)
- RPC's can be used to communicate between address spaces on different machines on the same machine
  - Services can be run wherever it's most appropriate
  - Access to local and remote services looks the same
- Examples of modern RPC systems:
  - CORBA (Common Object Request Broker Architecture)
  - DCOM (Distributed COM)
  - RMI (Java Remote Method Invocation)

Microkernel operating systems

- Example: split kernel into application-level servers.
  - File system looks remote, even though on same machine

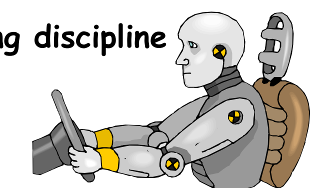


Monolithic Structure   Microkernel Structure

- Why split the OS into separate domains?
  - Fault isolation: bugs are more isolated (build a firewall)
  - Enforces modularity: allows incremental upgrades of pieces of software (client or server)
  - Location transparent: service can be local or remote
    - » For example in the X windowing system: Each X client can be on a separate machine from X server; Neither has to run on the machine with the frame buffer.

Administrivia

Role of Testing



- Testing is basic to every engineering discipline
  - Design a drug
  - Manufacture an airplane
  - Etc.
- Why?
  - Because our ability to predict how our creations will behave is imperfect
  - We need to check our work, because we will make mistakes
- Some Testing Goals:
  - Reveal faults
  - Establish confidence
    - » of reliability
    - » of (probable) correctness
    - » of detection (therefore absence) of particular faults
  - Clarify/infer the specification

Typical Software Licence

- 11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
- 12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Independent Testing

- Programmers never believe they made mistake
  - Plus a vested interest in not finding mistakes
- Design and programming are constructive tasks
  - Testers must seek to break the software
- Wrong conclusions:
  - The developer should not be testing at all
    - » Instead: "Test before you code"
  - Testers get involved once software is done
    - » Instead: Testers involved at all stages
  - Toss the software over the wall for testing
    - » Instead: Testers and developers collaborate in developing the test suite
  - Testing team is responsible for assuring quality
    - » Instead: Quality is assured by a good software process

Principles of Testability

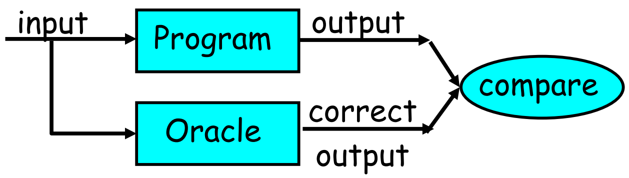
- Testers have two jobs
  - Clarify the specification
  - Find (important) bugs
- Avoid unpredictable results
  - No unnecessary non-deterministic behavior
- Design in self-checking
  - Have system check its own work (Asserts!)
  - May require adding some redundancy to the code
- Avoid system state
  - System retains nothing across units of work
    - » A transaction, a session, etc.
  - System returns to well-known state after each task
    - » Easiest system to test (or to recover from failure)
- Minimize interactions between features
  - Number of interactions can easily grow huge
  - Rich breeding ground for bugs
- Have a test interface

Testing Frameworks

- Key components of a test system are
  - Building the system to test
    - » May build many different versions to test
  - Running the tests
  - Deciding whether tests passed/failed
    - » Sometimes a non-trivial task (e.g., compilers) !
  - Reporting results
- Testing frameworks provide these functions
  - E.g., Tinderbox, JUnit

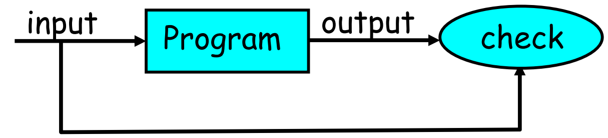
What is an Oracle?

- Oracle = alternative realization of the specification



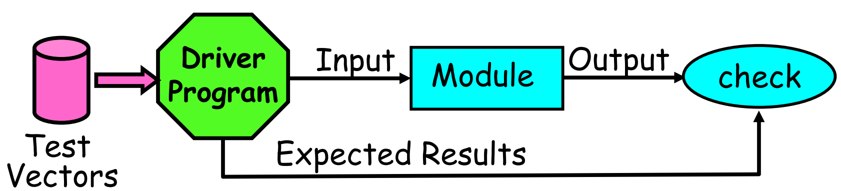
- Examples of oracles
  - The "eyeball oracle"
    - » Expensive, not dependable, lack of automation
  - A prototype, or sub-optimal implementation
    - » E.g., bubble-sort as oracle for quick sort
  - A manual list of expected results

Result Checking



- Easy to check the result of some algorithms
  - E.g., computing roots of polynomials, vs. checking that the result is correct
  - E.g., executing a query, vs. checking that the results meet the conditions
    - » Not easy to check that you got all results though !

Data Driven Tests



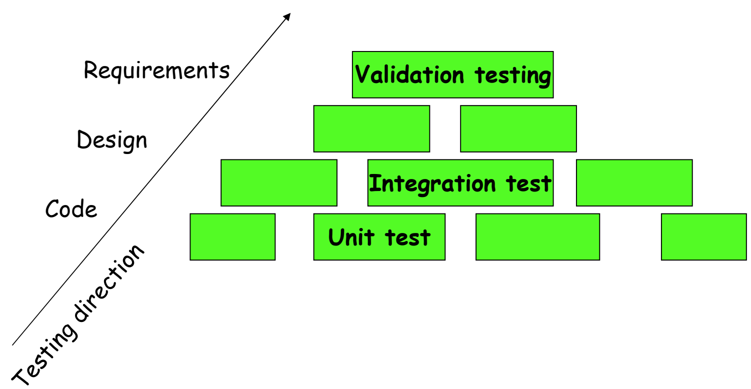
- Build a database of event tuples (or test vectors)
  - E.g: < Input1, Input2, Input3, Input4, Result>
  - So: <3, 4, "hello", 5, 42>
    <3, 5, "goodbye", 5, failure>
- A test is a series of such events chained together
  - Produce a high-level "driver" program to apply tuples to the system under test
    - » Tuples could be in a file and read in by driver program
  - Can be completely automatic

Assertions

- Use assert(…) liberally
  - Documents important invariants
  - Makes your code self-checking
  - And does it on *every* execution !
  - You still have to worry about coverage

- May need to write functions that check invariants
- Opinion: Most programmers don't use assert enough

Testing Strategies



Unit Tests

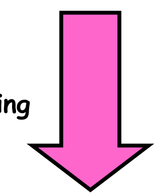
- Focus on smallest unit of design
  - A procedure, a class, a component
- Test the following
  - Local data structures
  - Basic algorithm
  - Boundary conditions
  - Error handling
- Good idea to plan unit tests ahead

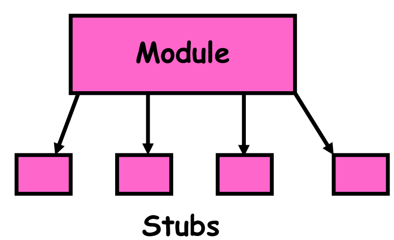
Integration Testing

- If all parts work, how come the whole doesn't?
- For software, the whole is more than the sum of the parts
  - Individual imprecision is magnified (e.g., races)
  - Unclear interface design
- Don't try the "big bang" integration !
- Do incremental integration
  - Top-down integration
  - Bottom-up integration

Top-Down Integration

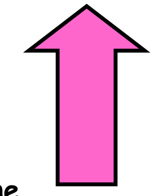
- Test the main control module first
- Slowly replace stubs with real code
  - Can go depth-first
    - » Along a favorite path, to create quickly a working system
  - Or, breadth first
- Problem: you may need complex stubs to test higher-levels

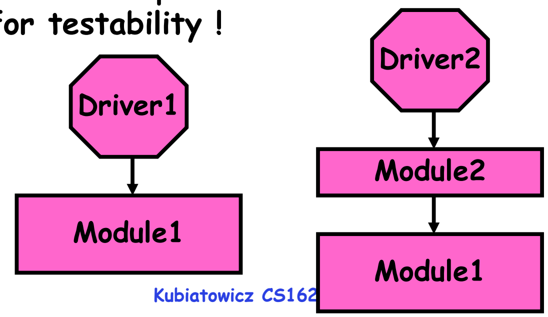




Bottom-Up Integration

- Integrate already tested modules
- No stubs, but need drivers
  - Often the drivers are easier to write
- Example:
  - Financial code that depends on subroutine for computing roots of polynomials
  - We cannot test the code without the subroutine
    - » A simple stub might not be enough
  - We can develop and test the subroutine first
- Plan for testability !





Stress Testing

- Push system into extreme situations
  - And see if it still works . . .
- Stress
  - Performance
    - » Feed data at very high rates
  - Interfaces
    - » Replace APIs with badly behaved stubs
  - Internal structures
    - » Works for any size array? Try sizes 0 and 1.
  - Resources
    - » Set memory artificially low.
    - » Same for # of file descriptors, network connections, etc.

Stress Testing (Cont.)

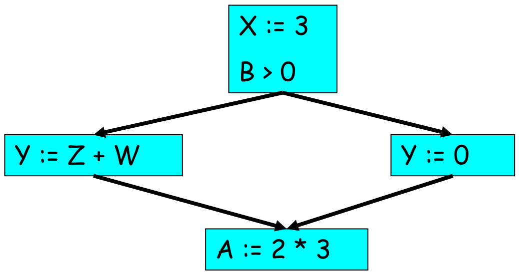
- Stress testing will find many obscure bugs
  - Explores the corner cases of the design

  "Bugs lurk in corners, and congregate at boundaries"
- Some may not be worth fixing
  - Bugs too unlikely to arise in practice
- A corner case now is tomorrow's common case
  - Data rates, data sizes always increasing
  - Your software will be stressed

Code Coverage

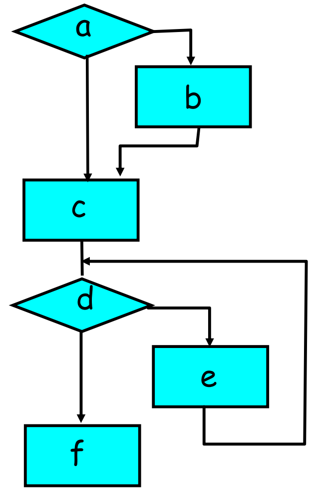
- Code Coverage
  - Make sure that code is *covered*
- Control flow coverage criteria: Make sure you have tests that exercise every…
  - Statement (node, basic block) coverage
  - Branch (edge) and condition coverage
  - Data flow (syntactic dependency) coverage
- More sophisticated coverage criteria increase the #units to be covered in a program

Control Flow Graphs: The One Slide Tutorial



- A graph
- Nodes are *basic blocks*
  - Maximal single-entry, jump-exit code segments
- Edges are transfers of control between basic blocks
  - E.g. Branches.

Basic structural criteria (ex.)

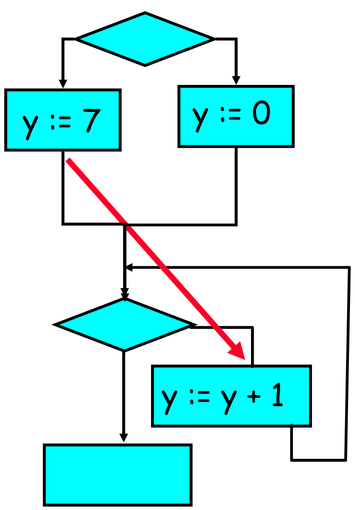


Edge ac is required by all-edges but not by all-nodes coverage

- abcdedf – all nodes

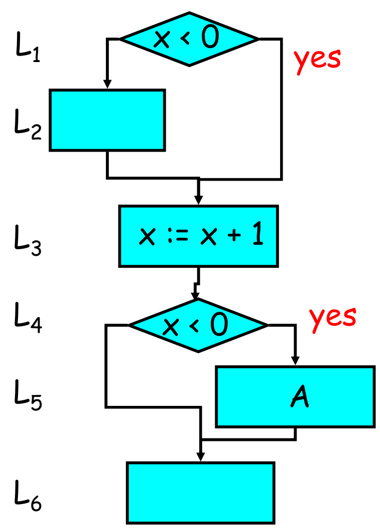
Typical loop coverage criterion would require zero iterations (cdf), one iteration (cdedf), and multiple iterations (cdededed...df)

Data flow coverage criteria (ex.)



- An untested def-use association could hide an erroneous computation
  - Even though we have all-node and all-edge coverage
- This suggests all paths coverage

All Paths Coverage



- There could be an exponential number of paths in a acyclic program
  - 2 conditionals) 4 max combinations
- Many are not reachable:
  L_1-L_2-L_3-L_4-L_6
- We choose
  - x = 0: L_1-L_2-L_3-L_4-L_5-L_6
  - x = -1: L_1-L_3-L_4-L_6
  - x = -2: L_1-L_3-L_4-L_5-L_6

Code Coverage (Cont.)

- Code coverage has proven value
  - It's a real metric, though far from perfect
- But 100% coverage does not mean no bugs
  - Many bugs lurk in corner cases
  - E.g., a bug visible after loop executes 1,025 times
- And 100% coverage is almost never achieved
  - Products ship with < 60% coverage
  - High coverage may not even be economically desirable
    - » May be better to focus on tricky parts
- Code coverage helps identify weak test suites
  - Tricky bits with low coverage are a danger sign
  - Areas with low coverage suggest something is missing in the test suite

Code Inspections

- Problem: Testing is weak
  - Can never test more than a tiny fraction of possibilities
  - Testers don't know as much about the code as the developers
    - » But developers can only do so much testing
- Here's an idea: Understand the code!
  - One person explains to a group of programmers how a piece of code works
- Key points
  - Don't try to read too much code at one sitting
    - » A few pages at most
  - Everyone comes prepared
    - » Distribute code beforehand
  - No blame
    - » Goal is to understand, clarify code, not roast programmers

Experience with Inspections

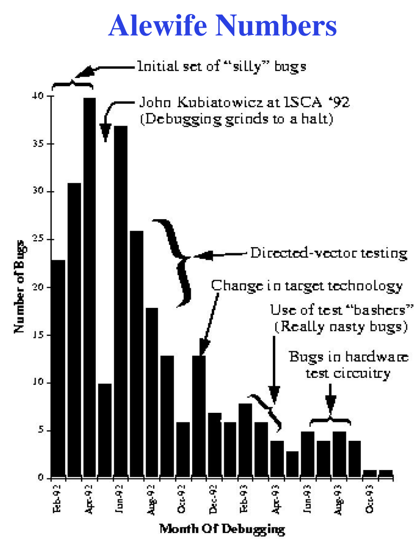
- Inspections work!
  - Finds 70%-90% of bugs in studies
  - Dramatically reduces cost of finding bugs
- Other advantages
  - Teaches everyone the code
  - Finds bugs earlier than testing
- Bottom line: More than pays for itself

Regression Testing

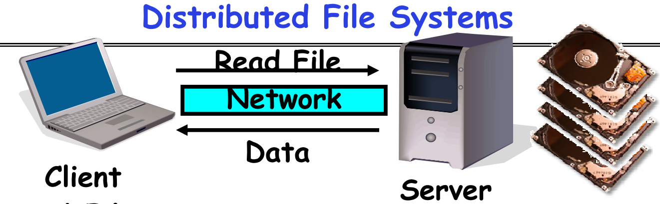
- Idea
  - When you find a bug,
  - Write a test that exhibits the bug,
  - And always run that test when the code changes,
  - So that the bug doesn't reappear
- Without regression testing, it is surprising how often old bugs reoccur
  - Regression testing ensures forward progress
  - We never go back to old bugs
- Regression testing can be manual or automatic
  - Ideally, run regressions after every change
  - To detect problems as quickly as possible
- But, regression testing is expensive
  - Limits how often it can be run in practice
  - Reducing cost is a long-standing research problem

Testing: When are you done?

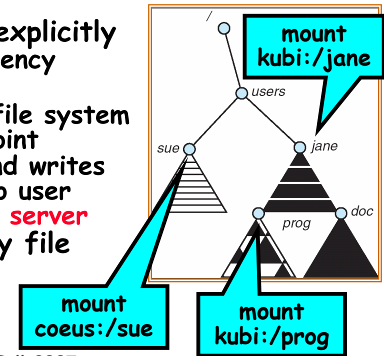
- When you run out of time?
- Consider rate of bug finding
  - Rate is high ⇒ NOT DONE
  - Rate is low ⇒ May need new testing methodology
- Coverage Metrics
  - How well did you cover the design with test cases?
- Types of testing:
  - Directed Testing – test explicit behavior
  - Random Testing – apply random values or orderings
  - Daemons – continuous error/unexpected behavior insertion



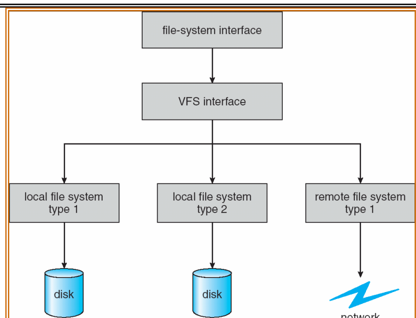
Distributed File Systems



- Distributed File System:
  - Transparent access to files stored on a remote disk
- Naming choices (always an issue):
  - *Hostname*:localname: Name files explicitly
    - » No location or migration transparency
  - *Mounting* of remote file systems
    - » System manager mounts remote file system by giving name and local mount point
    - » Transparent to user: all reads and writes look like local reads and writes to user e.g. /users/sue/foo→/sue/foo on server
  - *A single, global name space:* every file in the world has unique name
    - » Location Transparency: servers can change and files can move without involving user

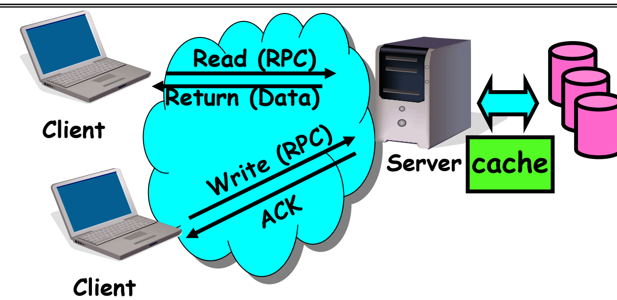


Virtual File System (VFS)



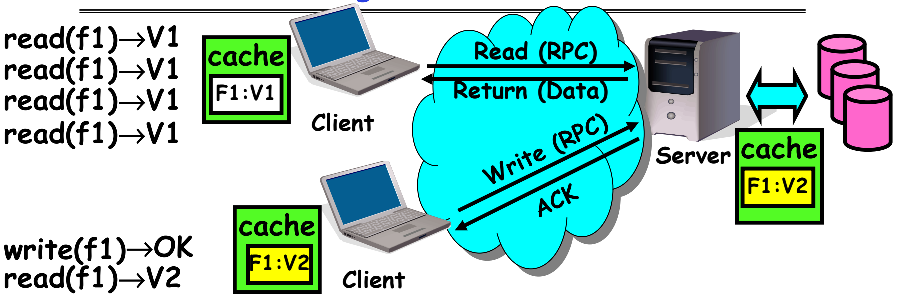
- VFS: Virtual abstraction similar to local file system
  - Instead of "inodes" has "vnodes"
  - Compatible with a variety of local and remote file systems
    - » provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - The API is to the VFS interface, rather than any specific type of file system

Simple Distributed File System



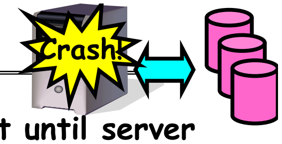
- Remote Disk: Reads and writes forwarded to server
  - Use RPC to translate file system calls
  - No local caching/can be caching at server-side
- Advantage: Server provides completely consistent view of file system to multiple clients
- Problems? Performance!
  - Going over network is slower than going to local memory
  - Lots of network traffic/not well pipelined
  - Server can be a bottleneck

Use of caching to reduce network load



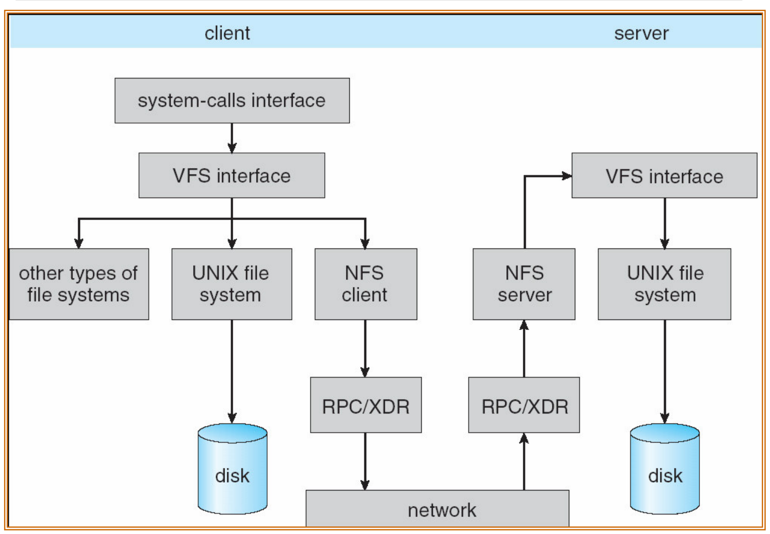
- Idea: Use caching to reduce network load
  - In practice: use buffer cache at source and destination
- Advantage: if open/read/write/close can be done locally, don't need to do any network traffic…fast!
- Problems:
  - Failure:
    - » Client caches have data not committed at server
  - Cache consistency!
    - » Client caches not consistent with server/each other

Failures



- What if server crashes? Can client wait until server comes back up and continue as before?
  - Any data in server memory but not on disk can be lost
  - Shared state across RPC: What if server crashes after seek? Then, when client does "read", it will fail
  - Message retries: suppose server crashes after it does UNIX "rm foo", but before acknowledgment?
    - » Message system will retry: send it again
    - » How does it know not to delete it again? (could solve with two-phase commit protocol, but NFS takes a more ad hoc approach)
- Stateless protocol: A protocol in which all information required to process a request is passed with request
  - Server keeps no state about client, except as hints to help improve performance (e.g. a cache)
  - Thus, if server crashes and restarted, requests can continue where left off (in many cases)
- What if client crashes?
  - Might lose modified data in client cache

Schematic View of NFS Architecture



Network File System (NFS)

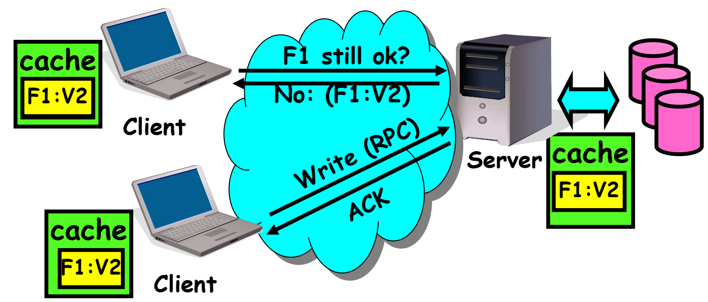
- Three Layers for NFS system
  - UNIX file-system interface: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - » Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - » Implements the NFS protocol
- NFS Protocol: RPC for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes/reading and writing files
- Write-through caching: Modified data committed to server's disk before results are returned to the client
  - lose some of the advantages of caching
  - time to perform write() can be long
  - Need some mechanism for readers to eventually notice changes! (more on this later)

NFS Continued

- NFS servers are stateless; each request provides all arguments require for execution
  - E.g. reads include information for entire operation, such as `ReadAt(inumber,position)`, not `Read(openfile)`
  - No need to perform network open() or close() on file – each operation stands on its own
- Idempotent: Performing requests multiple times has same effect as performing it exactly once
  - Example: Server crashes between disk I/O and message send, client resend read, server does operation again
  - Example: Read and write file blocks: just re-read or re-write file block – no side effects
  - Example: What about "remove"? NFS does operation twice and second time returns an advisory error
- Failure Model: Transparent to client system
  - Is this a good idea? What if you are in the middle of reading a file and server crashes?
  - Options (NFS Provides both):
    - » Hang until server comes back up (next week?)
    - » Return an error. (Of course, most applications don't know they are talking over network)

NFS Cache consistency

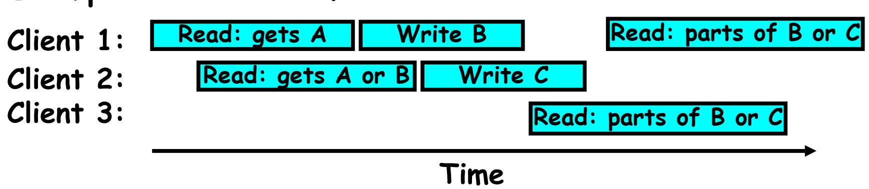
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - » Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.



  - What if multiple clients write to same file?
    - » In NFS, can get either version (or parts of both)
    - » Completely arbitrary!

Sequential Ordering Constraints

- What sort of cache coherence might we expect?
  - i.e. what if one CPU changes file, and before it's done, another CPU reads file?
- Example: Start with file contents = "A"



- What would we actually want?
  - Assume we want distributed system to behave exactly the same as if all processes are running on single system
    - » If read finishes before write starts, get old copy
    - » If read starts after write finishes, get new copy
    - » Otherwise, get either new or old copy
  - For NFS:
    - » If read starts more than 30 seconds after write, get new copy; otherwise, could get partial update

NFS Pros and Cons

- NFS Pros:
  - Simple, Highly portable
- NFS Cons:
  - Sometimes inconsistent!
  - Doesn't scale to large # clients
    - » Must keep checking to see if caches out of date
    - » Server becomes bottleneck due to polling traffic

Andrew File System

- Andrew File System (AFS, late 80's) → DCE DFS (commercial product)
- Callbacks: Server records who has copy of file
  - On changes, server immediately tells all with old copy
  - No polling bandwidth (continuous checking) needed
- Write through on close
  - Changes not propagated to server until close()
  - Session semantics: updates visible to other clients only after the file is closed
    - » As a result, do not get partial writes: all or nothing!
    - » Although, for processes on local machine, updates visible immediately to other programs who have file open
- In AFS, everyone who has file open sees old version
  - Don't get newer versions until reopen file

Andrew File System (con't)

- Data cached on local disk of client as well as memory
  - On open with a cache miss (file not on local disk):
    - » Get file from server, set up callback with server
  - On write followed by close:
    - » Send copy to server; tells all clients with copies to fetch new version from server on next open (using callbacks)
- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client: go ask everyone "who has which files cached?"
- AFS Pro: Relative to NFS, less server load:
  - Disk as cache ⇒ more files can be cached locally
  - Callbacks ⇒ server not involved if file is read-only
- For both AFS and NFS: central server is bottleneck!
  - Performance: all writes→server, cache misses→server
  - Availability: Server is single point of failure
  - Cost: server machine's high cost relative to workstation

Conclusion

- Remote Procedure Call (RPC): Call procedure on remote machine
  - Provides same interface as procedure
  - Automatic packing and unpacking of arguments without user programming (in stub)
- Testing Goals
  - Reveal faults
  - Clarify Specification
- Testing Frameworks:
  - Provide mechanism for applying tests (driver), checking results, reporting problems
  - Oracle: simpler version of code for testing outputs
  - Assertions: Documents (and checks) important invariants
- Levels of Tests:
  - Unit testing: per module
  - Integration Testing: tying modules together
  - Regression Testing: making sure bugs don't reappear

Conclusion (2)

- VFS: Virtual File System layer
  - Provides mechanism which gives same system call interface for different types of file systems
- Distributed File System:
  - Transparent access to files stored on a remote disk
    - » NFS: Network File System
    - » AFS: Andrew File System
  - Caching for performance
- Cache Consistency: Keeping contents of client caches consistent with one another
  - If multiple clients, some reading and some writing, how do stale cached copies get updated?
  - NFS: check periodically for changes
  - AFS: clients register callbacks so can be notified by server of changes

CS162
Operating Systems and Systems Programming
Lecture 25

Protection and Security in Distributed Systems

November 28, 2007

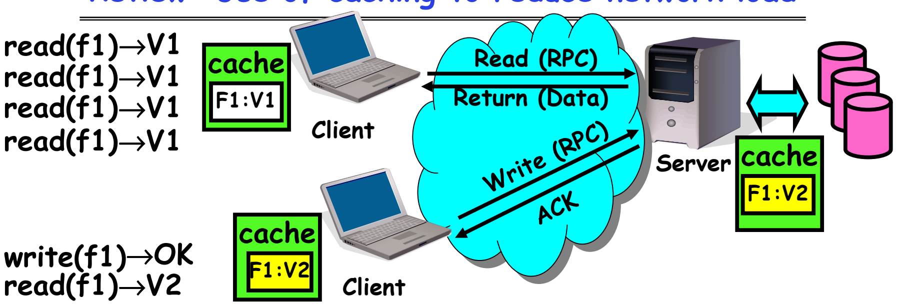
Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

Review: Testing

- Testing Goals
  - Reveal faults
  - Clarify Specification
- Testing Frameworks:
  - Provide mechanism for applying tests (driver), checking results, reporting problems
  - Oracle: simpler version of code for testing outputs
  - Assertions: Documents (and checks) important invariants
- Levels of Tests:
  - Unit testing: per module
  - Integration Testing: tying modules together
  - Code Inspections:
    - » One person explains to others how a piece of code works
    - » Finds 70%-90% of bugs
  - Regression Testing: making sure bugs don't reappear
    - » When you find a bug, Write a test that exhibits the bug,
    - » And always run that test when the code changes

Review: Use of caching to reduce network load



- Idea: Use caching to reduce network load
  - In practice: use buffer cache at source and destination
- Advantage: if open/read/write/close can be done locally, don't need to do any network traffic…fast!
- Problems:
  - Failure:
    - » Client caches have data not committed at server
  - Cache consistency!
    - » Client caches not consistent with server/each other

Goals for Today

- Finish discussing distributed file systems/Caching
- Security Mechanisms
  - Authentication
  - Authorization
  - Enforcement
- Cryptographic Mechanisms

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Network File System (NFS)

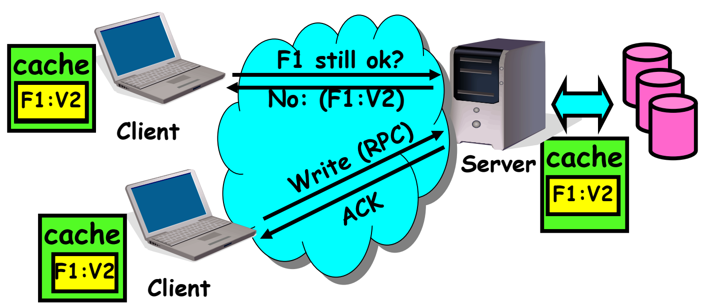
- Three Layers for NFS system
  - UNIX file-system interface: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - » Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - » Implements the NFS protocol
- NFS Protocol: RPC for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes/reading and writing files
- Write-through caching: Modified data committed to server's disk before results are returned to the client
  - lose some of the advantages of caching
  - time to perform write() can be long
  - Need some mechanism for readers to eventually notice changes! (more on this later)

NFS Continued

- NFS servers are stateless; each request provides all arguments require for execution
  - E.g. reads include information for entire operation, such as `ReadAt(inumber,position)`, not `Read(openfile)`
  - No need to perform network open() or close() on file – each operation stands on its own
- Idempotent: Performing requests multiple times has same effect as performing it exactly once
  - Example: Server crashes between disk I/O and message send, client resend read, server does operation again
  - Example: Read and write file blocks: just re-read or re-write file block – no side effects
  - Example: What about "remove"? NFS does operation twice and second time returns an advisory error
- Failure Model: Transparent to client system
  - Is this a good idea? What if you are in the middle of reading a file and server crashes?
  - Options (NFS Provides both):
    - » Hang until server comes back up (next week?)
    - » Return an error. (Of course, most applications don't know they are talking over network)

NFS Cache consistency

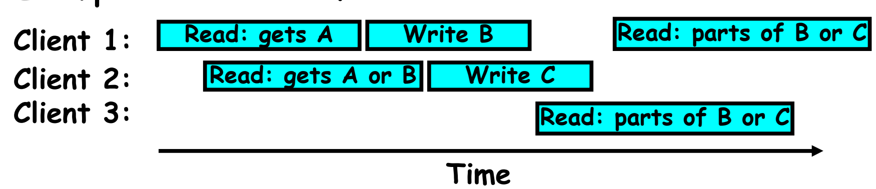
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - » Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.



  - What if multiple clients write to same file?
    - » In NFS, can get either version (or parts of both)
    - » Completely arbitrary!

Sequential Ordering Constraints

- What sort of cache coherence might we expect?
  - i.e. what if one CPU changes file, and before it's done, another CPU reads file?
- Example: Start with file contents = "A"



- What would we actually want?
  - Assume we want distributed system to behave exactly the same as if all processes are running on single system
    - » If read finishes before write starts, get old copy
    - » If read starts after write finishes, get new copy
    - » Otherwise, get either new or old copy
  - For NFS:
    - » If read starts more than 30 seconds after write, get new copy; otherwise, could get partial update

NFS Pros and Cons

- NFS Pros:
  - Simple, Highly portable
- NFS Cons:
  - Sometimes inconsistent!
  - Doesn't scale to large # clients
    - » Must keep checking to see if caches out of date
    - » Server becomes bottleneck due to polling traffic

Andrew File System

- Andrew File System (AFS, late 80's) → DCE DFS (commercial product)
- **Callbacks:** Server records who has copy of file
  - On changes, server immediately tells all with old copy
  - No polling bandwidth (continuous checking) needed
- Write through on close
  - Changes not propagated to server until close()
  - Session semantics: updates visible to other clients only after the file is closed
    - » As a result, do not get partial writes: all or nothing!
    - » Although, for processes on local machine, updates visible immediately to other programs who have file open
- In AFS, everyone who has file open sees old version
  - Don't get newer versions until reopen file

Andrew File System (con't)

- Data cached on local disk of client as well as memory
  - On open with a cache miss (file not on local disk):
    - » Get file from server, set up callback with server
  - On write followed by close:
    - » Send copy to server; tells all clients with copies to fetch new version from server on next open (using callbacks)
- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client: go ask everyone "who has which files cached?"
- AFS Pro: Relative to NFS, less server load:
  - Disk as cache ⇒ more files can be cached locally
  - Callbacks ⇒ server not involved if file is read-only
- For both AFS and NFS: central server is bottleneck!
  - Performance: all writes→server, cache misses→server
  - Availability: Server is single point of failure
  - Cost: server machine's high cost relative to workstation

Administrivia

World Wide Web

- Key idea: graphical front-end to RPC protocol
- What happens when a web server fails?
  - System breaks!
  - Solution: Transport or network-layer redirection
    - » Invisible to applications
    - » Can also help with scalability (load balancers)
    - » Must handle "sessions" (e.g., banking/e-commerce)
- Initial version: no caching
  - Didn't scale well – easy to overload servers

WWW Caching

- Use client-side caching to reduce number of interactions between clients and servers and/or reduce the size of the interactions:
  - Time-to-Live (TTL) fields – HTTP "Expires" header from server
  - Client polling – HTTP "If-Modified-Since" request headers from clients
  - Server refresh – HTML "META Refresh tag" causes periodic client poll
- What is the polling frequency for clients and servers?
  - Could be adaptive based upon a page's age and its rate of change
- Server load is still significant!

WWW Proxy Caches

- Place caches in the network to reduce server load
  - But, increases latency in lightly loaded case
  - Caches near servers called "reverse proxy caches"
    - » Offloads busy server machines
  - Caches at the "edges" of the network called "content distribution networks"
    - » Offloads servers and reduce client latency
- Challenges:
  - Caching static traffic easy, but only ~40% of traffic
  - Dynamic and multimedia is harder
    - » Multimedia is a big win: Megabytes versus Kilobytes
  - Same cache consistency problems as before
- Caching is changing the Internet architecture
  - Places functionality at higher levels of comm. protocols

Protection vs Security

- Protection: one or more mechanisms for controlling the access of programs, processes, or users to resources
  - Page Table Mechanism
  - File Access Mechanism
- Security: use of protection mechanisms to prevent misuse of resources
  - Misuse defined with respect to policy
    - » E.g.: prevent exposure of certain sensitive information
    - » E.g.: prevent unauthorized modification/deletion of data
  - Requires consideration of the external environment within which the system operates
    - » Most well-constructed system cannot protect information if user accidentally reveals password
- What we hope to gain today and next time
  - Conceptual understanding of how to make systems secure
  - Some examples, to illustrate why providing security is really hard in practice

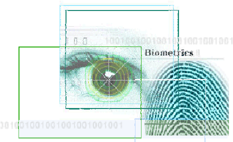
Preventing Misuse

- Types of Misuse:
  - Accidental:
    - » If I delete shell, can't log in to fix it!
    - » Could make it more difficult by asking: "do you really want to delete the shell?"
  - Intentional:
    - » Some high school brat who can't get a date, so instead he transfers $3 billion from B to A.
    - » Doesn't help to ask if they want to do it (of course!)
- Three Pieces to Security
  - Authentication: who the user actually is
  - Authorization: who is allowed to do what
  - Enforcement: make sure people do only what they are supposed to do
- Loopholes in any carefully constructed system:
  - Log in as superuser and you've circumvented authentication
  - Log in as self and can do anything with your resources; for instance: run program that erases all of your files
  - Can you trust software to correctly enforce Authentication and Authorization?????

Authentication: Identifying Users

- How to identify users to the system?
  - Passwords
    - » Shared secret between two parties
    - » Since only user knows password, someone types correct password ⇒ must be user typing it
    - » Very common technique
  - Smart Cards
    - » Electronics embedded in card capable of providing long passwords or satisfying challenge → response queries
    - » May have display to allow reading of password
    - » Or can be plugged in directly; several credit cards now in this category

  - Biometrics
    - » Use of one or more intrinsic physical or behavioral traits to identify someone
    - » Examples: fingerprint reader, palm reader, retinal scan
    - » Becoming quite a bit more common



Passwords: Secrecy

- System must keep copy of secret to check against passwords
  - What if malicious user gains access to list of passwords?
    - » Need to obscure information somehow
  - Mechanism: utilize a transformation that is difficult to reverse without the right key (e.g. encryption)
- Example: UNIX /etc/passwd file
  - passwd→one way transform(hash)→encrypted passwd
  - System stores only encrypted version, so OK even if someone reads the file!
  - When you type in your password, system compares encrypted version
- Problem: Can you trust encryption algorithm?
  - Example: one algorithm thought safe had back door
    - » Governments want back door so they can snoop
  - Also, security through obscurity doesn't work
    - » GSM encryption algorithm was secret; accidentally released; Berkeley grad students cracked in a few hours

Passwords: How easy to guess?

- Ways of Compromising Passwords
  - Password Guessing:
    - » Often people use obvious information like birthday, favorite color, girlfriend's name, etc…
  - Dictionary Attack:
    - » Work way through dictionary and compare encrypted version of dictionary words with entries in /etc/passwd
  - Dumpster Diving:
    - » Find pieces of paper with passwords written on them
    - » (Also used to get social-security numbers, etc)
- Paradox:
  - Short passwords are easy to crack
  - Long ones, people write down!
- Technology means we have to use longer passwords
  - UNIX initially required lowercase, 5-letter passwords: total of 26^5=10million passwords
    - » In 1975, 10ms to check a password→1 day to crack
    - » In 2005, .01μs to check a password→0.1 seconds to crack
  - Takes less time to check for all words in the dictionary!

Passwords: Making harder to crack

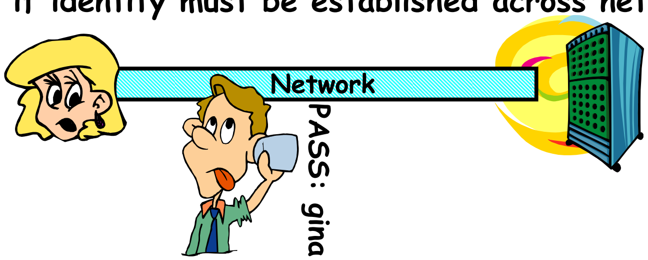
- How can we make passwords harder to crack?
  - Can't make it impossible, but can help
- Technique 1: Extend everyone's password with a unique number (stored in password file)
  - Called "salt". UNIX uses 12-bit "salt", making dictionary attacks 4096 times harder
  - Without salt, would be possible to pre-compute all the words in the dictionary hashed with the UNIX algorithm: would make comparing with /etc/passwd easy!
  - Also, way that salt is combined with password designed to frustrate use of off-the-shelf DES hardware
- Technique 2: Require more complex passwords
  - Make people use at least 8-character passwords with upper-case, lower-case, and numbers
    - » $70^8=6\times10^{14}$=6million seconds=69 days@0.01μs/check
  - Unfortunately, people still pick common patterns
    - » e.g. Capitalize first letter of common word, add one digit

Passwords: Making harder to crack (con't)

- Technique 3: Delay checking of passwords
  - If attacker doesn't have access to /etc/passwd, delay every remote login attempt by 1 second
  - Makes it infeasible for rapid-fire dictionary attack
- Technique 4: Assign very long passwords
  - Long passwords or pass-phrases can have more entropy (randomness→harder to crack)
  - Give everyone a smart card (or ATM card) to carry around to remember password
    - » Requires physical theft to steal password
    - » Can require PIN from user before authenticates self
  - Better: have smartcard generate pseudorandom number
    - » Client and server share initial seed
    - » Each second/login attempt advances to next random number
- Technique 5: "Zero-Knowledge Proof"
  - Require a series of challenge-response questions
    - » Distribute secret algorithm to user
    - » Server presents a number, say "5"; user computes something from the number and returns answer to server
    - » Server never asks same "question" twice
  - Often performed by smartcard plugged into system

Authentication in Distributed Systems

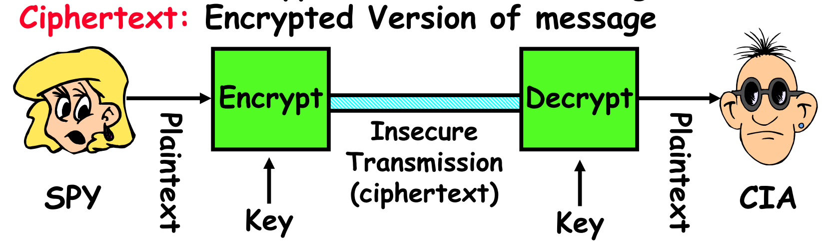
- What if identity must be established across network?



  - Need way to prevent exposure of information while still proving identity to remote system
  - Many of the original UNIX tools sent passwords over the wire "in clear text"
    - » E.g.: telnet, ftp, yp (yellow pages, for distributed login)
    - » Result: Snooping programs widespread
- What do we need? Cannot rely on physical security!
  - Encryption: Privacy, restrict receivers
  - Authentication: Remote Authenticity, restrict senders

Private Key Cryptography

- Private Key (Symmetric) Encryption:
  - Single key used for both encryption and decryption
- Plaintext: Unencrypted Version of message
- Ciphertext: Encrypted Version of message

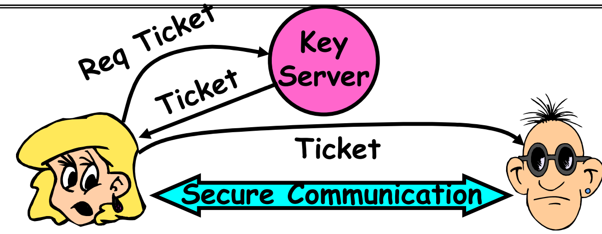


- Important properties
  - Can't derive plain text from ciphertext (decode) without access to key
  - Can't derive key from plain text and ciphertext
  - As long as password stays secret, get both secrecy and authentication
- Symmetric Key Algorithms: DES, Triple-DES, AES

Key Distribution

- How do you get shared secret to both places?
  - For instance: how do you send authenticated, secret mail to someone who you have never met?
  - Must negotiate key over private channel
    - » Exchange code book
    - » Key cards/memory stick/others
- Third Party: Authentication Server (like Kerberos)
  - Notation:
    - » K_{xy} is key for talking between x and y
    - » $(\ldots)^K$ means encrypt message (…) with the key K
    - » Clients: A and B, Authentication server S
  - A asks server for key:
    - » A→S: [Hi! I'd like a key for talking between A and B]
    - » Not encrypted. Others can find out if A and B are talking
  - Server returns *session* key encrypted using B's key
    - » S→A: Message $[\text{ Use } K_{ab} \text{ (This is A! Use } K_{ab})^{Ksb} \text{]}^{Ksa}$
    - » This allows A to know, "S said use this key"
  - Whenever A wants to talk with B
    - » A→B: Ticket $[\text{ This is A! Use } K_{ab} \text{]}^{Ksb}$
    - » Now, B knows that K_{ab} is sanctioned by S

Authentication Server Continued



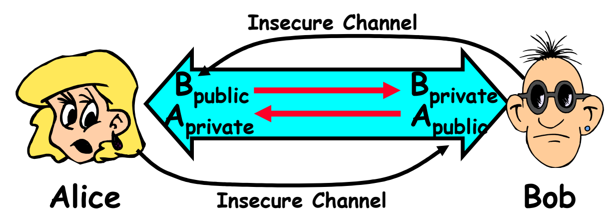
- Details
  - Both A and B use passwords (shared with key server) to decrypt return from key servers
  - Add in timestamps to limit how long tickets will be used to prevent attacker from replaying messages later
  - Also have to include encrypted checksums (hashed version of message) to prevent malicious user from inserting things into messages/changing messages
  - Want to minimize # times A types in password
    - » A→S (Give me temporary secret)
    - » S→A (Use $K_{temp\text{-}sa}$ for next 8 hours)^{Ksa}
    - » Can now use $K_{temp\text{-}sa}$ in place of K_{sa} in prototcol

Public Key Encryption

- Can we perform key distribution without an authentication server?
  - Yes. Use a Public-Key Cryptosystem.
- Public Key Details
  - Don't have one key, have two: K_{public}, $K_{private}$
    - » Two keys are mathematically related to one another
    - » Really hard to derive K_{public} from $K_{private}$ and vice versa
  - Forward encryption:
    - » Encrypt: $(\text{cleartext})^{Kpublic} = \text{ciphertext}_1$
    - » Decrypt: $(\text{ciphertext}_1)^{Kprivate} = \text{cleartext}$
  - Reverse encryption:
    - » Encrypt: $(\text{cleartext})^{Kprivate} = \text{ciphertext}_2$
    - » Decrypt: $(\text{ciphertext}_2)^{Kpublic} = \text{cleartext}$
  - Note that $\text{ciphertext}_1 \neq \text{ciphertext}_2$
    - » Can't derive one from the other!
- Public Key Examples:
  - RSA: Rivest, Shamir, and Adleman
    - » K_{public} of form (k_{public}, N), $K_{private}$ of form $(k_{private}, N)$
    - » N = pq. Can break code if know p and q
  - ECC: Elliptic Curve Cryptography

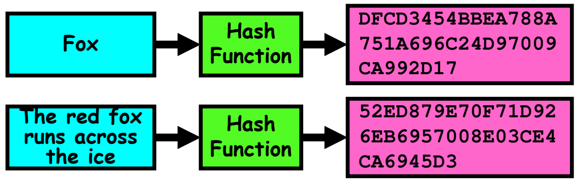
Public Key Encryption Details

- Idea: K_{public} can be made public, keep $K_{private}$ private



- Gives message privacy (restricted receiver):
  - Public keys (secure destination points) can be acquired by anyone/used by anyone
  - Only person with private key can decrypt message
- What about authentication?
  - Use combination of private and public key
  - Alice→Bob: $[(\text{I'm Alice})^{Aprivate} \text{ Rest of message}]^{Bpublic}$
  - Provides restricted sender and receiver
- But: how does Alice know that it was Bob who sent her B_{public}? And vice versa…

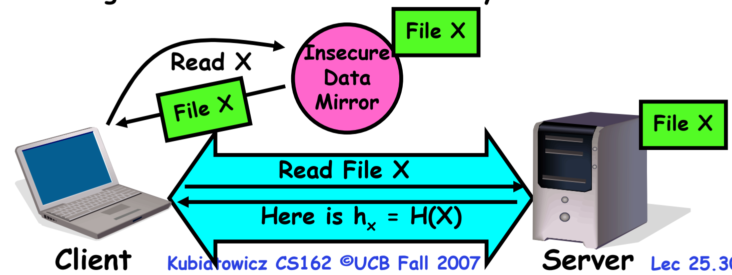
Secure Hash Function



- Hash Function: Short summary of data (message)
  - For instance, $h_1=H(M_1)$ is the hash of message M_1
    - » h_1 fixed length, despite size of message M_1.
    - » Often, h_1 is called the "digest" of M_1.
- Hash function H is considered secure if
  - It is infeasible to find M_2 with $h_1=H(M_2)$; ie. can't easily find other message with same digest as given message.
  - It is infeasible to locate two messages, m_1 and m_2, which "collide", i.e. for which $H(m_1) = H(m_2)$
  - A small change in a message changes many bits of digest/can't tell anything about message given its hash

Use of Hash Functions

- Several Standard Hash Functions:
  - MD5: 128-bit output
  - SHA-1: 160-bit output
- Can we use hashing to securely reduce load on server?
  - Yes. Use a series of insecure mirror servers (caches)
  - First, ask server for digest of desired file
    - » Use secure channel with server
  - Then ask mirror server for file
    - » Can be insecure channel
    - » Check digest of result and catch faulty or malicious mirrors



Signatures/Certificate Authorities

- Can use X_{public} for person X to define their identity
  - Presumably they are the only ones who know $X_{private}$.
  - Often, we think of X_{public} as a "principle" (user)
- Suppose we want X to sign message M?
  - Use private key to encrypt the digest, i.e. $H(M)^{Xprivate}$
  - Send both M and its signature:
    - » Signed message = $[M,H(M)^{Xprivate}]$
  - Now, anyone can verify that M was signed by X
    - » Simply decrypt the digest with X_{public}
    - » Verify that result matches H(M)
- Now: How do we know that the version of X_{public} that we have is really from X???
  - Answer: Certificate Authority
    - » Examples: Verisign, Entrust, Etc.
  - X goes to organization, presents identifying papers
    - » Organization signs X's key: $[X_{public}, H(X_{public})^{CAprivate}]$
    - » Called a "Certificate"
  - Before we use X_{public}, ask X for certificate verifying key
    - » Check that signature over X_{public} produced by trusted authority
- How do we get keys of certificate authority?
  - Compiled into your browser, for instance!

Conclusion

- User Identification
  - Passwords/Smart Cards/Biometrics
- Passwords
  - Encrypt them to help hid them
  - Force them to be longer/not amenable to dictionary attack
  - Use zero-knowledge request-response techniques
- Distributed identity
  - Use cryptography
- Symmetrical (or Private Key) Encryption
  - Single Key used to encode and decode
  - Introduces key-distribution problem
- Public-Key Encryption
  - Two keys: a public key and a private key
    - » Not derivable from one another
- Secure Hash Function
  - Used to summarize data
  - Hard to find another block of data with same hash

CS162
Operating Systems and Systems Programming
Lecture 26

Protection and Security in Distributed Systems II

December 3, 2007

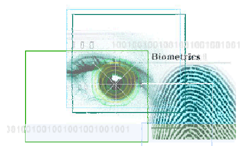
Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

Review: Authentication: Identifying Users

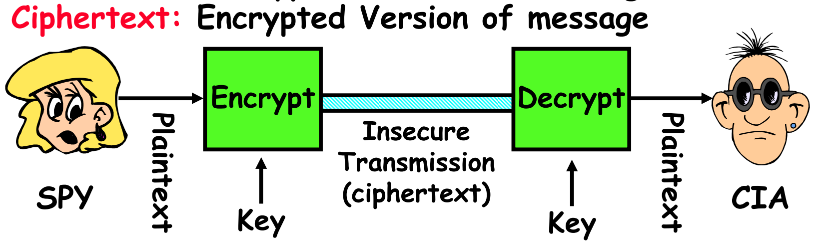
- How to identify users to the system?
  - Passwords
    - » Shared secret between two parties
    - » Since only user knows password, someone types correct password ⇒ must be user typing it
    - » Very common technique
  - Smart Cards
    - » Electronics embedded in card capable of providing long passwords or satisfying challenge → response queries
    - » May have display to allow reading of password
    - » Or can be plugged in directly; several credit cards now in this category

  - Biometrics
    - » Use of one or more intrinsic physical or behavioral traits to identify someone
    - » Examples: fingerprint reader, palm reader, retinal scan
    - » Becoming quite a bit more common



Review: Private Key Cryptography

- Private Key (Symmetric) Encryption:
  - Single key used for both encryption and decryption
- Plaintext: Unencrypted Version of message
- Ciphertext: Encrypted Version of message



- Important properties
  - Can't derive plain text from ciphertext (decode) without access to key
  - Can't derive key from plain text and ciphertext
  - As long as password stays secret, get both secrecy and authentication
- Symmetric Key Algorithms: DES, Triple-DES, AES

Goals for Today

- Use of Cryptographic Mechanisms
- Authorization Mechanisms
- Worms and Viruses

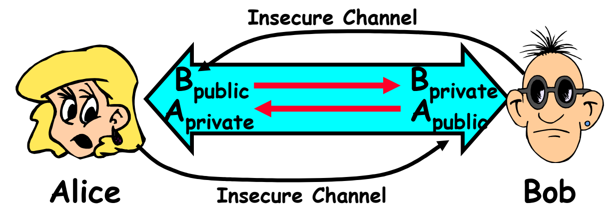
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

Public Key Encryption

- Can we perform key distribution without an authentication server?
  - Yes. Use a Public-Key Cryptosystem.
- Public Key Details
  - Don't have one key, have two: K_{public}, $K_{private}$
    - » Two keys are mathematically related to one another
    - » Really hard to derive K_{public} from $K_{private}$ and vice versa
  - Forward encryption:
    - » Encrypt: $(\text{cleartext})^{K_{public}} = \text{ciphertext}_1$
    - » Decrypt: $(\text{ciphertext}_1)^{K_{private}} = \text{cleartext}$
  - Reverse encryption:
    - » Encrypt: $(\text{cleartext})^{K_{private}} = \text{ciphertext}_2$
    - » Decrypt: $(\text{ciphertext}_2)^{K_{public}} = \text{cleartext}$
  - Note that $\text{ciphertext}_1 \neq \text{ciphertext}_2$
    - » Can't derive one from the other!
- Public Key Examples:
  - RSA: Rivest, Shamir, and Adleman
    - » K_{public} of form (k_{public}, N), $K_{private}$ of form $(k_{private}, N)$
    - » N = pq. Can break code if know p and q
  - ECC: Elliptic Curve Cryptography

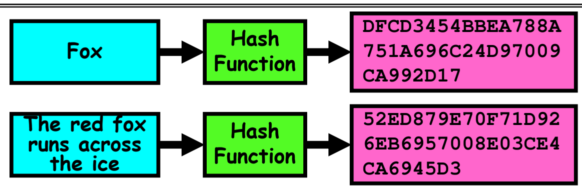
Public Key Encryption Details

- Idea: K_{public} can be made public, keep $K_{private}$ private



- Gives message privacy (restricted receiver):
  - Public keys (secure destination points) can be acquired by anyone/used by anyone
  - Only person with private key can decrypt message
- What about authentication?
  - Use combination of private and public key
  - Alice→Bob: $[(\text{I'm Alice})^{A_{private}} \text{ Rest of message}]^{B_{public}}$
  - Provides restricted sender and receiver
- But: how does Alice know that it was Bob who sent her B_{public}? And vice versa…

Secure Hash Function

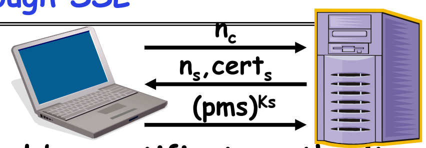


- Hash Function: Short summary of data (message)
  - For instance, $h_1=H(M_1)$ is the hash of message M_1
    - » h_1 fixed length, despite size of message M_1.
    - » Often, h_1 is called the "digest" of M_1.
- Hash function H is considered secure if
  - It is infeasible to find M_2 with $h_1=H(M_2)$; ie. can't easily find other message with same digest as given message.
  - It is infeasible to locate two messages, m_1 and m_2, which "collide", i.e. for which $H(m_1) = H(m_2)$
  - A small change in a message changes many bits of digest/can't tell anything about message given its hash
- Hash function Examples: MD5, SHA-1, SHA-256

Signatures/Certificate Authorities

- Can use X_{public} for person X to define their identity
  - Presumably they are the only ones who know $X_{private}$.
  - Often, we think of X_{public} as a "principle" (user)
- Suppose we want X to sign message M?
  - Use private key to encrypt the digest, i.e. $H(M)^{X_{private}}$
  - Send both M and its signature:
    - » Signed message = $[M, H(M)^{X_{private}}]$
  - Now, anyone can verify that M was signed by X
    - » Simply decrypt the digest with X_{public}
    - » Verify that result matches H(M)
- Now: How do we know that the version of X_{public} that we have is really from X???
  - Answer: Certificate Authority
    - » Examples: Verisign, Entrust, Etc.
  - X goes to organization, presents identifying papers
    - » Organization signs X's key: $[X_{public}, H(X_{public})^{CA_{private}}]$
    - » Called a "Certificate"
  - Before we use X_{public}, ask X for certificate verifying key
    - » Check that signature over X_{public} produced by trusted authority
- How do we get keys of certificate authority?
  - Compiled into your browser, for instance!

Security through SSL

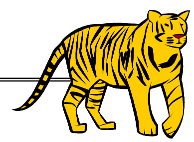


- SSL Web Protocol
  - Port 443: secure http
  - Use public-key encryption for key-distribution
- Server has a certificate signed by certificate authority
  - Contains server info (organization, IP address, etc)
  - Also contains server's public key and expiration date
- Establishment of Shared, 48-byte "master secret"
  - Client sends 28-byte random value n_c to server
  - Server returns its own 28-byte random value n_s, plus its certificate $cert_s$
  - Client verifies certificate by checking with public key of certificate authority compiled into browser
    - » Also check expiration date
  - Client picks 46-byte "premaster" secret (pms), encrypts it with public key of server, and sends to server
  - Now, both server and client have n_c, n_s, and pms
    - » Each can compute 48-byte master secret using one-way and collision-resistant function on three values
    - » Random "nonces" n_c and n_s make sure master secret fresh

SSL Pitfalls

- Netscape claimed to provide secure comm. (SSL)
  - So you could send a credit card # over the Internet
- Three problems (reported in NYT):
  - Algorithm for picking session keys was predictable (used time of day) – brute force key in a few hours
  - Made new version of Netscape to fix #1, available to users over Internet (unencrypted!)
    - » Four byte patch to Netscape executable makes it always use a specific session key
    - » Could insert backdoor by mangling packets containing executable as they fly by on the Internet.
    - » Many mirror sites (including Berkeley) to redistribute new version – anyone with root access to any machine on LAN at mirror site could insert the backdoor
  - Buggy helper applications – can exploit *any* bug in either Netscape, or its helper applications

Cryptographic Summary



- Private Key Encryption (also Symmetric Key)
  - Pros: Very Fast
    - » can encrypt at network speed (even without hardware)
  - Cons: Need to distribute *secret* key to both parties
- Public Key Encryption (also Asymmetric Key)
  - Pros: Can distribute keys in public
    - » Need certificate authority (Public Key Infrastructure)
  - Cons: Very Slow
    - » 100—1000 times slower than private key encryption
- Session Key
  - Randomly generated private key used for single session
  - Often distributed via public key encryption
- Secure Hash
  - Fixed length summary of data that is hard to spoof
- Message Authentication Code (MAC)
  - Technique for using secure hash and session key to verify individual packets (even at the IP level)
  - IPSEC: IP Protocol 50/51, authentic/encrypted IP
- Signature over Document
  - Hash of document encrypted with private key

Administrivia

- MIDTERM II: Monday December 4th!
  - 4:00-7:00pm, 10 Evans
  - All material from last midterm and up to today
  - Includes virtual memory
  - One page of handwritten notes, both sides
- Final Exam
  - December 16th,8:00-11:00, Bechtel Auditorium
  - Covers whole course (except final lecture)
  - Two pages of handwritten notes, both sides
- Last Day of Class – Next Wednesday
  - One more section on Thursday?
- Final Topics suggestions (so far):
  - Google OS
  - Parallel OS
  - Cybersecurity attacks
  - Peer-to-peer systems

Recall: Authorization: Who Can Do What?

- How do we decide who is authorized to do actions in the system?
- Access Control Matrix: contains all permissions in the system
  - Resources across top
    - » Files, Devices, etc…
  - Domains in columns
    - » A domain might be a user or a group of permissions
    - » E.g. above: User D_3 can read F_2 or execute F_3
  - In practice, table would be huge and sparse!

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

- Two approaches to implementation
  - Access Control Lists: store permissions with each object
    - » Still might be lots of users!
    - » UNIX limits each file to: r,w,x for owner, group, world
    - » More recent systems allow definition of groups of users and permissions for each group
  - Capability List: each process tracks objects has permission to touch
    - » Popular in the past, idea out of favor today
    - » Consider page table: Each process has list of pages it has access to, not each page has list of processes …

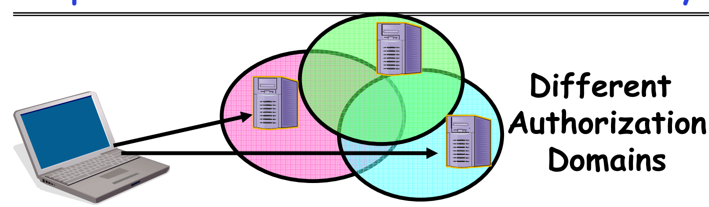
How fine-grained should access control be?

- Example of the problem:
  - Suppose you buy a copy of a new game from "Joe's Game World" and then run it.
  - It's running with your userid
    - » It removes all the files you own, including the project due the next day…
- How can you prevent this?
  - Have to run the program under *some* userid.
    - » Could create a second *games* userid for the user, which has no write privileges.
    - » Like the "nobody" userid in UNIX – can't do much
  - But what if the game needs to write out a file recording scores?
    - » Would need to give write privileges to one particular file (or directory) to your *games* userid.
  - But what about non-game programs you want to use, such as Quicken?
    - » Now you need to create your own private *quicken* userid, if you want to make sure tha the copy of Quicken you bought can't corrupt non-quicken-related files
  - But – how to get this right??? Pretty complex…

Authorization Continued

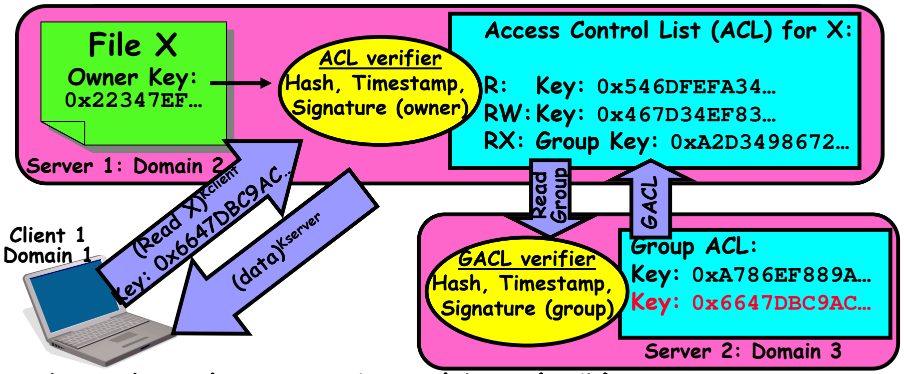
- Principle of least privilege: programs, users, and systems should get only enough privileges to perform their tasks
  - Very hard to do in practice
    - » How do you figure out what the minimum set of privileges is needed to run your programs?
  - People often run at higher privilege then necessary
    - » Such as the "administrator" privilege under windows
- One solution: Signed Software
  - Only use software from sources that you trust, thereby dealing with the problem by means of authentication
  - Fine for big, established firms such as Microsoft, since they can make their signing keys well known and people trust them
    - » Actually, not always fine: recently, one of Microsoft's signing keys was compromised, leading to malicious software that looked valid
  - What about new startups?
    - » Who "validates" them?
    - » How easy is it to fool them?

How to perform Authorization for Distributed Systems?



- Issues: Are all user names in world unique?
  - No! They only have small number of characters
    - » kubi@mit.edu → kubitron@lcs.mit.edu → kubitron@cs.berkeley.edu
    - » However, someone thought their friend was kubi@mit.edu and I got very private email intended for someone else…
  - Need something better, more unique to identify person
- Suppose want to connect with any server at any time?
  - Need an account on every machine! (possibly with different user name for each account)
  - OR: Need to use something more universal as identity
    - » Public Keys! (Called "Principles")
    - » People *are* their public keys

Distributed Access Control



- Distributed Access Control List (ACL)
  - Contains list of attributes (Read, Write, Execute, etc) with attached identities (Here, we show public keys)
    - » ACLs signed by owner of file, only changeable by owner
    - » Group lists signed by group key
  - ACLs can be on different servers than data
    - » Signatures allow us to validate them
    - » ACLs could even be stored separately from verifiers

Analysis of Previous Scheme

- Positive Points:
  - Identities checked via signatures and public keys
    - » Client can't generate request for data unless they have private key to go with their public identity
    - » Server won't use ACLs not properly signed by owner of file
  - No problems with multiple domains, since identities designed to be cross-domain (public keys domain neutral)
- Revocation:
  - What if someone steals your private key?
    - » Need to walk through all ACLs with your key and change…!
    - » This is very expensive
  - Better to have unique string identifying you that people place into ACLs
    - » Then, ask Certificate Authority to give you a certificate matching unique string to your current public key
    - » Client Request: (request + unique ID)^{Cprivate}; give server certificate if they ask for it.
    - » Key compromise⇒must distribute "certificate revocation", since can't wait for previous certificate to expire.
  - What if you remove someone from ACL of a given file?
    - » If server caches old ACL, then person retains access!
    - » Here, cache inconsistency leads to security violations!

Analysis Continued

- Who signs the data?
  - Or: How does the client know they are getting valid data?
  - Signed by server?
    - » What if server compromised? Should client trust server?
  - Signed by owner of file?
    - » Better, but now only owner can update file!
    - » Pretty inconvenient!
  - Signed by group of servers that accepted latest update?
    - » If must have signatures from all servers ⇒ Safe, but one bad server can prevent update from happening
    - » Instead: ask for a threshold number of signatures
    - » Byzantine agreement can help here
- How do you know that data is up-to-date?
  - Valid signature only means data is valid older version
  - Freshness attack:
    - » Malicious server returns old data instead of recent data
    - » Problem with both ACLs and data
    - » E.g.: you just got a raise, but enemy breaks into a server and prevents payroll from seeing latest version of update
  - Hard problem
    - » Needs to be fixed by invalidating old copies or having a trusted group of servers (Byzantine Agrement?)

Involuntary Installation

- What about software loaded without your consent?
  - Macros attached to documents (such as Microsoft Word)
  - Active X controls (programs on web sites with potential access to whole machine)
  - Spyware included with normal products
- Active X controls can have access to the local machine
  - Install software/Launch programs
- Sony Spyware [Sony XCP] (October 2005)
  - About 50 recent CDs from Sony automatically install software when you played them on Windows machines
    - » Called XCP (Extended Copy Protection)
    - » Modify operating system to prevent more than 3 copies and to prevent peer-to-peer sharing
  - Side Effects:
    - » Reporting of private information to Sony
    - » Hiding of generic file names of form $sys_xxx; easy for other virus writers to exploit
    - » Hard to remove (crashes machine if not done carefully)
  - Vendors of virus protection software declare it spyware
    - » Computer Associates, Symantec, even Microsoft

Enforcement

- Enforcer checks passwords, ACLs, etc
  - Makes sure the only authorized actions take place
  - Bugs in enforcer⇒things for malicious users to exploit
- In UNIX, superuser can do anything
  - Because of coarse-grained access control, lots of stuff has to run as superuser in order to work
  - If there is a bug in any one of these programs, you lose!
- Paradox
  - Bullet-proof enforcer
    - » Only known way is to make enforcer as small as possible
    - » Easier to make correct, but simple-minded protection model
  - Fancy protection
    - » Tries to adhere to principle of least privilege
    - » Really hard to get right
- Same argument for Java or C++: What do you make private vs public?
  - Hard to make sure that code is usable but only necessary modules are public
  - Pick something in middle? Get bugs and weak protection!

State of the World

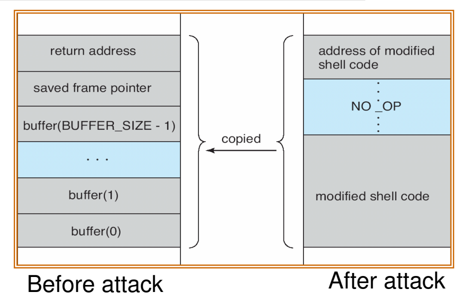
- State of the World in Security
  - Authentication: Encryption
    - » But almost no one encrypts or has public key identity
  - Authorization: Access Control
    - » But many systems only provide very coarse-grained access
    - » In UNIX, need to turn off protection to enable sharing
  - Enforcement: Kernel mode
    - » Hard to write a million line program without bugs
    - » Any bug is a potential security loophole!
- Some types of security problems
  - Abuse of privilege
    - » If the superuser is evil, we're all in trouble/can't do anything
    - » What if sysop in charge of instructional resources went crazy and deleted everybody's files (and backups)???
  - Imposter: Pretend to be someone else
    - » Example: in unix, can set up an .rhosts file to allow logins from one machine to another without retyping password
    - » Allows "rsh" command to do an operation on a remote node
    - » Result: send rsh request, pretending to be from trusted user→install .rhosts file granting you access

Other Security Problems

- Virus:
  - A piece of code that attaches itself to a program or file so it can spread from one computer to another, leaving infections as it travels
  - Most attached to executable files, so don't get activated until the file is actually executed
  - Once caught, can hide in boot tracks, other files, OS
- Worm:
  - Similar to a virus, but capable of traveling on its own
  - Takes advantage of file or information transport features
  - Because it can replicate itself, your computer might send out hundreds or thousands of copies of itself
- Trojan Horse:
  - Named after huge wooden horse in Greek mythology given as gift to enemy; contained army inside
  - At first glance appears to be useful software but does damage once installed or run on your computer

Security Problems: Buffer-overflow Condition

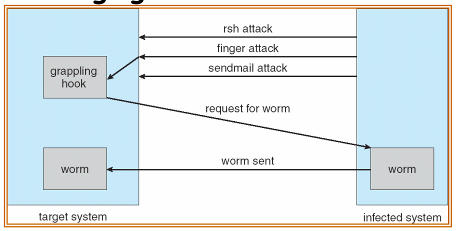
```
#define BUFFER SIZE 256
int process(int argc,
            char *argv[])
{
  char buffer[BUFFER SIZE];
  if (argc < 2)
       return -1;
  else {
       strcpy(buffer,argv[1]);
       return 0;
  }
}
```



- Technique exploited by many network attacks
  - Anytime input comes from network request and is not checked for size
  - Allows execution of code with same privileges as running program – but happens without any action from user!
- How to prevent?
  - Don't code this way! (ok, wishful thinking)
  - New mode bits in Intel, Amd, and Sun processors
    - » Put in page table; says "don't execute code in this page"

The Morris Internet Worm

- Internet worm (Self-reproducing)
  - Author Robert Morris, a first-year Cornell grad student
  - Launched close of Workday on November 2, 1988
  - Within a few hours of release, it consumed resources to the point of bringing down infected machines



- Techniques
  - Exploited UNIX networking features (remote access)
  - Bugs in *finger* (buffer overflow) and *sendmail* programs (debug mode allowed remote login)
  - Dictionary lookup-based password cracking
  - Grappling hook program uploaded main worm program

Some other Attacks

- Trojan Horse Example: Fake Login
  - Construct a program that looks like normal login program
  - Gives "login:" and "password:" prompts
    - » You type information, it sends password to someone, then either logs you in or says "Permission Denied" and exits
  - In Windows, the "ctrl-alt-delete" sequence is supposed to be really hard to change, so you "know" that you are getting official login program
- Is SONY XCP a Trojan horse?
- Salami attack: Slicing things a little at a time
  - Steal or corrupt something a little bit at a time
  - E.g.: What happens to partial pennies from bank interest?
    - » Bank keeps them! Hacker re-programmed system so that partial pennies would go into his account.
    - » Doesn't seem like much, but if you are large bank can be millions of dollars
- Eavesdropping attack
  - Tap into network and see everything typed
  - Catch passwords, etc
  - Lesson: never use unencrypted communication!

Tenex Password Checking

- Tenex – early 70's, BBN
  - Most popular system at universities before UNIX
  - Thought to be very secure, gave "red team" all the source code and documentation (want code to be publicly available, as in UNIX)
  - In 48 hours, they figured out how to get every password in the system
- Here's the code for the password check:

```
for (i = 0; i < 8; i++)
  if (userPasswd[i] != realPasswd[i])
    go to error
```

- How many combinations of passwords?
  - 256^8?
  - Wrong!

Defeating Password Checking

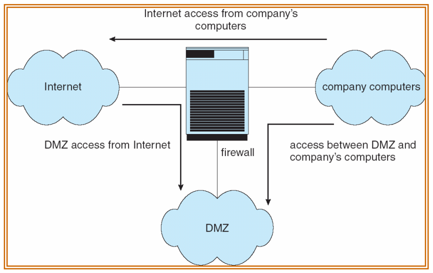
- Tenex used VM, and it interacts badly with the above code
  - Key idea: force page faults at inopportune times to break passwords quickly
- Arrange 1^{st} char in string to be last char in pg, rest on next pg
  - Then arrange for pg with 1^{st} char to be in memory, and rest to be on disk (e.g., ref lots of other pgs, then ref 1^{st} page)

```
              a|aaaaaa
               |
page in memory| page on disk
```

- Time password check to determine if first character is correct!
  - If fast, 1^{st} char is wrong
  - If slow, 1^{st} char is right, pg fault, one of the others wrong
  - So try all first characters, until one is slow
  - Repeat with first two characters in memory, rest on disk
- Only 256 * 8 attempts to crack passwords
  - Fix is easy, don't stop until you look at all the characters

Defense in Depth: Layered Network Security

- How do I minimize the damage when security fails?
  - For instance: I make a mistake in the specification
  - Or: A bug lets something run that shouldn't?
- Firewall: Examines every packet to/from public internet
  - Can disable all traffic to/from certain ports
  - Can route certain traffic to DMZ (De-Militarized Zone)
    - » Semi-secure area separate from critical systems
  - Can do network address translation
    - » Inside network, computers have private IP addresses
    - » Connection from inside→outside is translated
    - » E.g. [10.0.0.2,port 2390] → [169.229.60.38,port 80]
      [12.4.35.2,port 5592] → [169.229.60.38,port 80]



Shrink Wrap Software Woes

- Can I trust software installed by the computer manufacturer?
  - Not really, most major computer manufacturers have shipped computers with viruses
  - How?
    - » Forgot to update virus scanner on "gold" master machine
- Software companies, PR firms, and others routinely release software that contains viruses
- Linux hackers say "Start with the source"
  - Does that work?

Ken Thompson's self-replicating program

- Bury Trojan horse in binaries, so no evidence in source
  - Replicates itself to every UNIX system in the world and even to new UNIX's on new platforms. No visible sign.
  - Gave Ken Thompson ability to log into any UNIX system
- Two steps: Make it possible (easy); Hide it (tricky)
- Step 1: Modify login.c

```
A: if (name == "ken")
       don't check password
       log in as root
```

  - Easy to do but pretty blatant! Anyone looking will see.
- Step 2: Modify C compiler
  - Instead of putting code in login.c, put in compiler:

```
B: if see trigger1
   insert A into input stream
```

  - Whenever compiler sees trigger1 (say /*gobbledygook*/), puts A into input stream of compiler
  - Now, don't need A in login.c, just need trigger1

Self Replicating Program Continued

- Step 3: Modify compiler source code:

```
C: if see trigger2
   insert B+C into input stream
```

  - Now compile this new C compiler to produce binary
- Step 4: Self-replicating code!
  - Simply remove statement C in compiler source code and place "trigger2" into source instead
    - » As long as existing C compiler is used to recompile the C compiler, the code will stay into the C compiler and will compile back door into login.c
    - » But no one can see this from source code!
- When porting to new machine/architecture, use existing C compiler to generate cross-compiler
  - Code will migrate to new architecture!
- Lesson: never underestimate the cleverness of computer hackers for hiding things!

Conclusion

- Distributed identity
  - Use cryptography (Public Key, Signed by PKI)
- Use of Public Key Encryption to get Session Key
  - Can send encrypted random values to server, now share secret with server
  - Used in SSL, for instance
- Authorization
  - Abstract table of users (or domains) vs permissions
  - Implemented either as access-control list or capability list
- Issues with distributed storage example
  - Revocation: How to remove permissions from someone?
  - Integrity: How to know whether data is valid
  - Freshness: How to know whether data is recent
- Buffer-Overrun Attack: exploit bug to execute code