

Introduction

CS 111

Operating System Principles

Peter Reiher

Outline

- Administrative materials
- Introduction to the course
 - Why study operating systems?
 - Basics of operating systems

Administrative Issues

- Instructor and TAs
- Load and prerequisites
- Web site, syllabus, reading, and lectures
- Quizzes, exams, homework, projects
- Grading
- Academic honesty

Instructor: Peter Reiher

- UCLA Computer Science department faculty member
- Long history of research in operating systems
- Email: reiher@cs.ucla.edu
- Office: 3532F Boelter Hall
 - Office hours: TTh 1-2
 - Often available at other times

My OS Background

- My Ph.D. dissertation was on the Locus operating system
- Much research on file systems
 - Ficus, Rumor, Truffles, Conquest
- Research on OS security issues
 - Data Tethers

TAs

- Muhammad Mehdi
 - taqi@cs.ucla.edu
- Diyu Zhou
 - zhoudiyu@cs.ucla.edu
- Zhaoxing Bu
 - zbu@cs.ucla.edu
- Jungbeom Lee
 - jungbeol@cs.ucla.edu
- Lab sessions:
 - Lab 1A, Fridays 10-12 AM, Boelter 9436
 - Lab 1B, Fridays 12- 2 PM, Humans 169
 - Lab 1C, Fridays 2-4 PM, Rolfe 3134
 - Lab 1D, Fridays 4-6 PM, Bunche 2160

• Office hours to be announced

Instructor/TA Division of Responsibilities

- Instructor handles all lectures, readings, and tests
 - Ask me about issues related to these
- TAs handle projects
 - Ask them about issues related to these
- Generally, instructor won't be involved with project issues
 - So direct those questions to the TAs

Web Site

- Some materials found on CCLE web site
 - Like quizzes
- Most materials here:
 - http://www.lasr.cs.ucla.edu/classes/111_fall16
- What's there:
 - Schedules for reading, lectures, exams, projects
 - Copies of lecture slides (Powerpoint and PDF)
 - Announcements
 - Sample midterm and final problems

Prerequisite Subject Knowledge

- CS 32 programming
 - Objects, data structures, queues, stacks, tables, trees
- CS 33 systems programming
 - Assembly language, registers, memory
 - Linkage conventions, stack frames, register saving
- CS 35L Software Construction Laboratory
 - Useful software tools for systems programming
- If you haven't taken these classes, expect to have a hard time in 111

Course Format

- Two weekly reading assignments
 - Mostly from the primary text
 - Some supplementary materials available on web
- Two weekly lectures
 - With online quizzes before the lectures
- Four (10-25 hour) individual projects
 - Exploring and exploiting OS features
 - Plus one warm-up project
- A midterm and a final exam

Course Load

- Reputation: THE hardest undergrad CS class
 - Fast pace through much non-trivial material
- Expectations you should have
 - lectures 4-6 hours/week
 - reading 3-6 hours/week
 - projects 3-20 hours/week
 - exam study 5-15 hours (twice)
- Keeping up (week by week) is critical
 - Catching up is extremely difficult

Primary Text for Course

- Remzi and Andrea Arpaci-Dusseau: *Operating Systems: Three Easy Pieces*
 - Freely available on line at <http://pages.cs.wisc.edu/~remzi/OSTEP/>
- Supplementary readings from Saltzer and Kaashoek: *Principles of Computer Systems Design*
 - Free on line at
<http://www.sciencedirect.com/science/book/9780123749574>
 - Only on-campus or through the UCLA VPN
- Supplemented with web-based materials

Course Grading

- Basis for grading:
 - Quizzes 5%
 - 1 midterm exam 20%
 - Final exam 30%
 - Lab 0 5%
 - Other labs 10% each
- I do look at distribution for final grades
 - But don't use a formal curve
- All scores available on MyUCLA
 - Please check them for accuracy

Quizzes

- 3-5 question quizzes on the assigned reading materials
- Must be taken before the lecture
- Not intended to be hard questions
 - IF you've read the assigned materials

Midterm Examination

- When: Second lecture of the 5th week (in class section, Tuesday, October 25)
- Scope: All lectures up to the exam date
 - Approximately 60% lecture, 40% text
- Format:
 - Closed book
 - 10-15 essay questions, most with short answers
- Goals:
 - Test understanding of key concepts
 - Test ability to apply principles to practical problems

Final Exam

- When: Monday, December 5, 11:30 AM-2:30 PM
- Scope: Entire course
- Format:
 - 6-8 hard multi-part essay questions
 - You get to pick a subset of them to answer
- Goals:
 - Test mastery of key concepts
 - Test ability to apply key concepts to real problems
 - Use key concepts to gain insight into new problems

Lab Projects

- Format:
 - 1 warm-up project
 - 4 regular projects
 - Done individually
- Goals:
 - Develop ability to exploit OS features
 - Develop programming/problem solving ability
 - Practice software project skills
- Lab and lecture are fairly distinct
 - Ask instructor about tests and lectures
 - Ask TAs about labs

Late Assignments & Make-ups

- Quizzes
 - No late quizzes accepted, no make-ups
- Labs
 - Due dates set by TAs
 - TAs also sets policy on late assignments
 - The TAs will handle all issues related to labs
 - Ask them, not me
 - Don't expect me to overrule their decisions
- Exams
 - Alternate times or make-ups only possible with prior consent of the instructor
 - If you miss a test, too bad

Academic Honesty

- It is OK to study with friends
 - Discussing problems helps you to understand them
- It is OK to do independent research on a subject
 - There are many excellent treatments out there
- But all work you submit must be your own
 - Do not write your lab answers with a friend
 - Do not copy another student's work
 - Do not turn in solutions from off the web
 - If you do research on a problem, cite your sources
- I decide when two assignments are too similar
 - And I forward them immediately to the Dean
- If you need help, ask the instructor

Academic Honesty – Projects

- Do your own projects
 - If you need additional help, ask the TA
- You must design and write all your own code
 - Do not ask others how they solved the problem
 - Do not copy solutions from the web, files or listings
 - Cite any research sources you use
- Protect yourself
 - Do not show other people your solutions
 - Be careful with old listings

Academic Honesty and the Internet

- You might be able to find existing answers to some of the assignments on line
- Remember, if you can find it, so can we
 - And we have, before
- It IS NOT OK to copy the answers from other people's old assignments
 - People who tried that have been caught and referred to the Office of the Dean of Students
- ANYTHING you get off the Internet must be treated as reference material
 - If you use it, quote it and reference it

Introduction to the Course

- Purpose of course and relationships to other courses
- Why study operating systems?
- Major themes & lessons in this course

What Will CS 111 Do?

- Build on concepts from other courses
 - Data structures, programming languages, assembly language programming, computer architectures, ...
- Prepare you for advanced courses
 - Data bases and distributed computing
 - Security, fault-tolerance, high availability
 - Network protocols, computer system modeling, queueing theory
- Provide you with foundation concepts
 - Processes, threads, virtual address space, files
 - Capabilities, synchronization, leases, deadlock

Why Study Operating Systems?

- Few of you will actually build OSs
- But many of you will:
 - Set up, configure, manage computer systems
 - Write programs that exploit OS features
 - Work with complex, distributed, parallel software
 - Work with abstracted services and resources
- Many hard problems have been solved in OS context
 - Synchronization, security, integrity, protocols, distributed computing, dynamic resource management, ...
 - In this class, we study these problems and their solutions
 - These approaches can be applied to other areas

Why Are Operating Systems Interesting?

- They are extremely complex
 - But try to appear simple enough for everyone to use
- They are very demanding
 - They require vision, imagination, and insight
 - They must have elegance and generality
 - They demand meticulous attention to detail
- They are held to very high standards
 - Performance, correctness, robustness,
 - Scalability, extensibility, reusability
- They are the base we all work from

Recurring OS Themes

- View services as objects and operations
 - Behind every object there is a data structure
- Separate policy from mechanism
 - Policy determines what can/should be done
 - Mechanism implements basic operations to do it
 - Mechanisms shouldn't dictate or limit policies
 - Policies must be changeable without changing mechanisms
- Parallelism and asynchrony are powerful and vital
 - But dangerous when used carelessly
- Performance and correctness are often at odds

More Recurring Themes

- An interface specification is a contract
 - Specifies responsibilities of producers & consumers
 - Basis for product/release interoperability
- Interface vs. implementation
 - An implementation is not a specification
 - Many compliant implementations are possible
 - Inappropriate dependencies cause problems
- Modularity and functional encapsulation
 - Complexity hiding and appropriate abstraction

Life Lessons From Studying Operating Systems

- There Ain't No Such Thing As A Free Lunch! (TANSTAAFL)
 - Everything has a cost, there are always trade-offs
 - But there are bad, expensive lunches . . .
- Keep It Simple, Stupid!
 - Avoid complex solutions, and being overly clever
 - Both usually create more problems than they solve
- Be very clear what your goals are
 - Make the right trade-offs, focus on the right problems
- Responsible and sustainable living
 - Understand the consequences of your actions
 - Nothing must be lost, everything must be recycled
 - It is all in the details

Moving on To Operating Systems . . .

- What is an operating system?
- What does an OS do?
- How does an OS appear to its clients?
 - Abstracted resources
 - Simplifying, generalizing
 - Serially reusable, partitioned, sharable

What Is An Operating System?

- Many possible definitions
- One is:
 - It is low level software . . .
 - That provides better, more usable abstractions of the hardware below it
 - To allow easy, safe, fair use and sharing of those resources

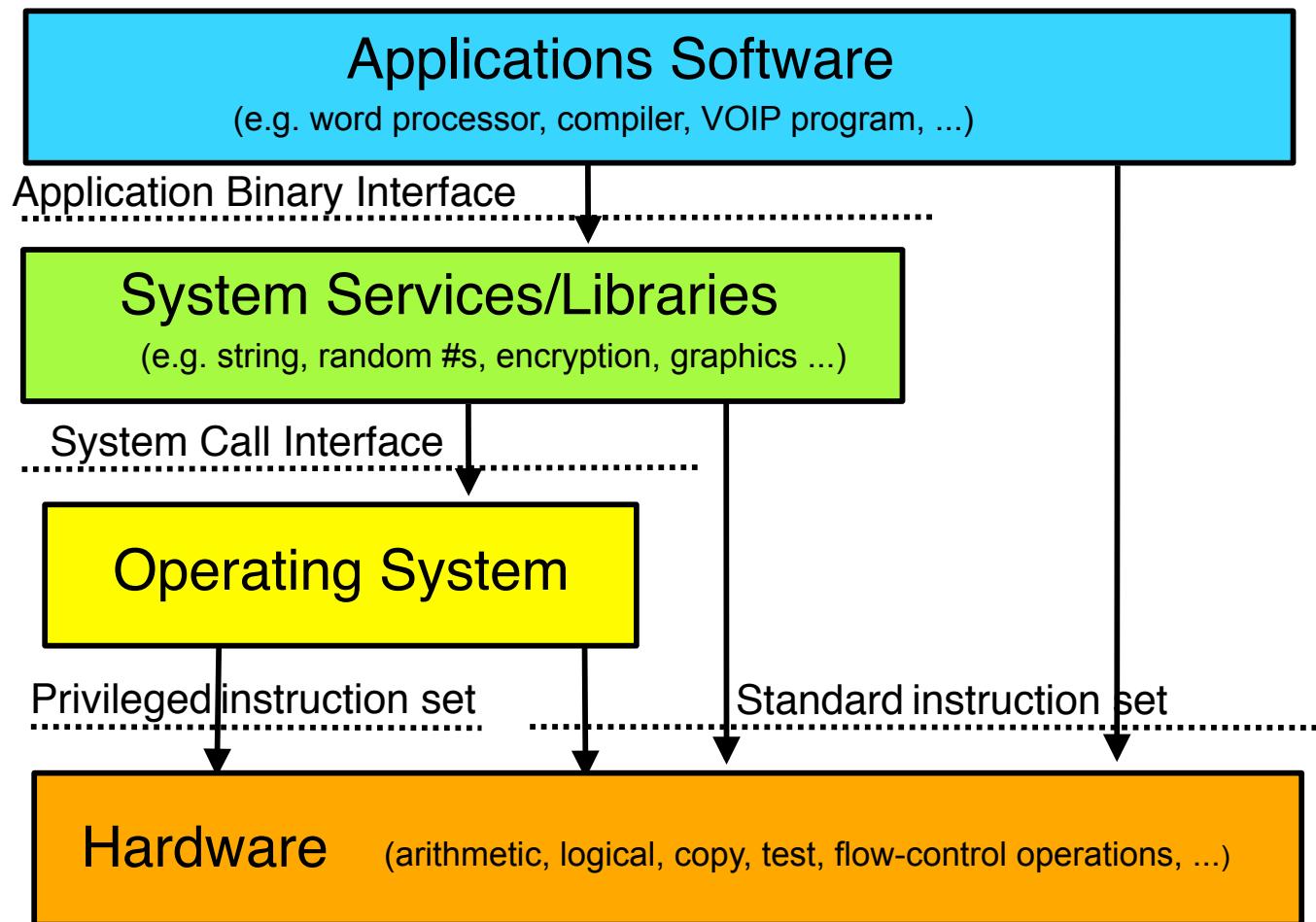
What Does an OS Do?

- It manages hardware for programs
 - Allocates hardware and manages its use
 - Enforces controlled sharing (and privacy)
 - Oversees execution and handles problems
- It abstracts the hardware
 - Makes it easier to use and improves SW portability
 - Optimizes performance
- It provides new abstractions for applications
 - Powerful features beyond the bare hardware

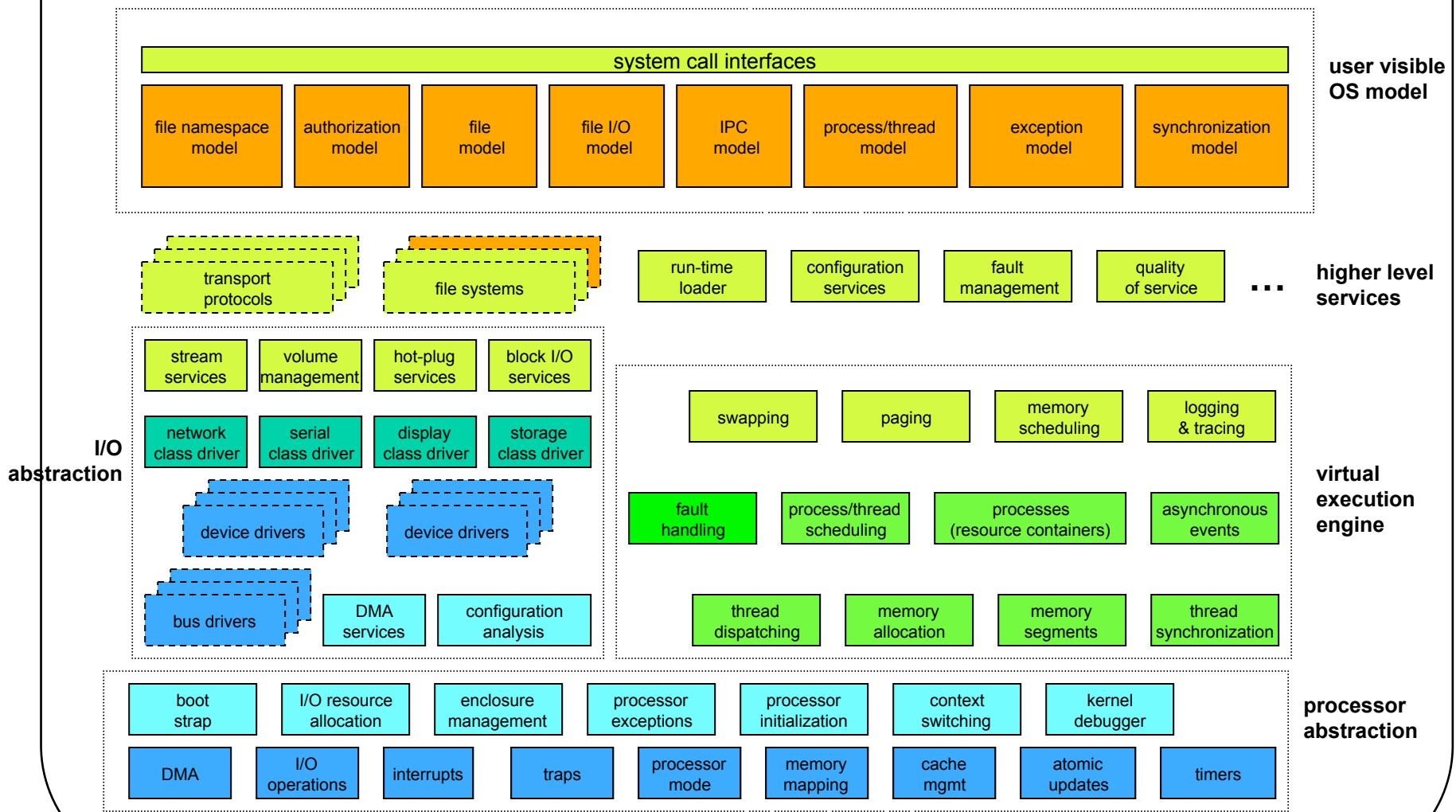
What Does An OS Look Like?

- A set of management & abstraction services
 - Invisible, they happen behind the scenes
- Applications see objects and their services
 - CPU supports data-types and operations
 - bytes, shorts, longs, floats, pointers, ...
 - add, subtract, copy, compare, indirection, ...
 - So does an operating system, but at a higher level
 - files, processes, threads, devices, ports, ...
 - create, destroy, read, write, signal, ...
- An OS extends a computer
 - Creating a much richer virtual computing platform
 - Supporting richer objects, more powerful operations

Where Does the OS Fit In?



Kernel Structure (artists conception)



Instruction Set Architectures (ISAs)

- The set of instructions supported by a computer
 - What bit patterns correspond to what operations
- There are many different ISAs (all incompatible)
 - Different word/bus widths (8, 16, 32, 64 bit)
 - Different features (low power, DSPs, floating point)
 - Different design philosophies (RISC vs. CISC)
 - Competitive reasons (68000, x86, PowerPC)
- They usually come in families
 - Newer models add features (e.g., Pentium vs. 386)
 - But remain upwards-compatible with older models
 - A program written for an ISA will run on any compliant CPU

Privileged vs. General Instructions

- Most modern ISAs divide instruction set into privileged vs. general
- Any code running on the machine can execute general instructions
- Processor must be put into a special mode to execute privileged instructions
 - Usually only in that mode when the OS is running
 - Privileged instructions do things that are “dangerous”

Platforms

- ISA doesn't completely define a computer
 - Functionality beyond user mode instructions
 - Interrupt controllers, DMA controllers
 - Memory management unit, I/O busses
 - BIOS, configuration, diagnostic features
 - Multi-processor & interconnect support
 - I/O devices
 - Display, disk, network, serial device controllers
- These variations are called “platforms”
 - The platform on which the OS must run

Portability to Multiple ISAs

- A successful OS will run on many ISAs
 - Some customers cannot choose their ISA
 - If you don't support it, you can't sell to them
- Minimal assumptions about specific HW
 - General frameworks are HW independent
 - File systems, protocols, processes, etc.
 - HW assumptions isolated to specific modules
 - Context switching, I/O, memory management
 - Careful use of types
 - Word length, sign extension, byte order, alignment

Distributing an OS

- Developers want their OS to run on as many machines as possible
- There are many different ISAs
 - And other platform differences
- Even more types of peripherals
- And vast numbers of different applications and configurations
- How to get wide, effective distribution?

Binary Distribution Model

- Binary is the opposite of source
 - A source distribution must be compiled
 - A binary distribution is ready to run
- OSes usually distributed in binary
- One binary distribution per ISA
 - No need for special per-OEM OS versions
- Binary model for platform support
 - Device drivers can be added, after-market
 - Can be written and distributed by 3rd parties
 - Same driver works with many versions of OS

Why Not a Source Distribution?

- What is wrong with distributing an OS in source form?
 1. *On what are you going to compile it?*
 2. *Are your customers competent to build the OS?
Do they want to build the OS?*
 3. *Do you really want to give all of your customers
the sources to your main product?*
- Open source OSes are available
 - But most users still download the binary versions

Binary Configuration Model

- Good to eliminate manual/static configuration
 - Enable one distribution to serve all users
 - Improve both ease of use and performance
- Automatic hardware discovery
 - Self identifying busses
 - PCI, USB, PCMCIA, EISA, etc.
 - Automatically find and load required drivers
- Automatic resource allocation
 - Eliminate fixed sized resource pools
 - Dynamically (re)allocate resources on demand

Flexibility

- Different customers have different needs
- We cannot anticipate all possible needs
- We must design for flexibility/extension
 - Mechanism/policy separation
 - Allow customers to override default policies
 - Changing policies without having to change the OS
 - Dynamically loadable features
 - Allow new features to be added, after market
 - File systems, protocols, load module formats, etc.
 - Feature independence and orthogonality

Interface Stability

- People want new releases of an OS
 - New features, bug fixes, enhancements
- People also fear new releases of an OS
 - OS changes can break old applications
- How can we prevent such problems?
 - Define well specified Application Interfaces
 - Application programming interfaces (APIs)
 - Application binary interfaces (ABIs)
 - Applications only use committed interfaces
 - OS vendors preserve upwards-compatibility

What's Special About the OS?

- It is always in control of the hardware
 - Automatically loaded when the machine boots
 - First software to have access to hardware
 - Continues running while apps come & go
- It alone has complete access to hardware
 - Privileged instruction set, all of memory & I/O
- It mediates applications' access to hardware
 - Block, permit, or modify application requests
- It is trusted
 - To store and manage critical data
 - To always act in good faith
- If the OS crashes, it takes everything else with it
 - So it better not crash . . .

What Functionality Is In the OS?

- As much as necessary, as little as possible
 - OS code is very expensive to develop and maintain
- Functionality must be in the OS if it ...
 - Requires the use of privileged instructions
 - Requires the manipulation of OS data structures
 - Must maintain security, trust, or resource integrity
- Functions should be in libraries if they ...
 - Are a service commonly needed by applications
 - Do not actually have to be implemented inside OS
- But there is also the performance excuse
 - Some things may be faster if done in the OS

Where To Offer a Service?

- Hardware, OS, library or application?
- Increasing requirements for stability as you move through these options
- Hardware services rarely change
- OS services can change, but it's a big deal
- Libraries are a bit more dynamic
- Applications can change services much more readily

Another Reason For This Choice

- Who uses it?
- Things literally everyone uses belong lower in the hierarchy
 - Particularly if the same service needs to work the same for everyone
- Things used by fewer/more specialized parties belong higher
 - Particularly if each party requires a substantially different version of the service

The OS and Speed

- One reason operating systems get big is based on speed
- It's faster to offer a service in the OS than outside it
- Thus, there's a push to move services with strong performance requirements down to the OS

Why Is the OS Faster?

- Than something at the application level, above it?
 - If it involves processes communicating, working at app level requires scheduling and swapping them
 - The OS has direct access to many pieces of state and system services
 - If an operation requires such things, application has to pay the cost to enter and leave OS, anyway
 - The OS can make direct use of privileged instructions

Is An OS Implementation Always Faster?

- Not always
- Running standard instructions no faster from the OS than from applications
- Entering the OS involves some fairly elaborate state saving and mode changing
- If you don't need special OS services, may be cheaper to manipulate at the app level
 - Maybe by an order of magnitude

The OS and Abstraction

- One major function of an OS is to offer abstract versions of resources
 - As opposed to actual physical resources
- Essentially, the OS implements the abstract resources using the physical resources
 - E.g., processes (an abstraction) are implemented using the CPU and RAM (physical resources)
 - And files (an abstraction) are implemented using disks (a physical resource)

Why Abstract Resources?

- The abstractions are typically simpler and better suited for programmers and users
 - Easier to use than the original resources
 - E.g., don't need to worry about keeping track of disk interrupts
 - Compartmentalize/encapsulate complexity
 - E.g., need not be concerned about what other executing code is doing and how to stay out of its way
 - Eliminate behavior that is irrelevant to user
 - E.g., hide the slow erase cycle of flash memory
 - Create more convenient behavior
 - E.g., make it look like you have the network interface entirely for your own use

Generalizing Abstractions

- Lots of variations in machines' HW and SW
- So make many different types appear to be same
 - So applications can deal with single common class
- Usually involves a common unifying model
 - E.g., portable document format (pdf) for printers
 - Or SCSI standard for disks, CDs and tapes
- Usually involves a federation framework
 - Per sub-type implementations of standard functions
- For example:
 - Printer drivers make different printers look the same
 - Browser plug-ins to handle multi-media data

Why Do We Want This Generality?

- For example, why do we want all printers to look the same?
 - So we could write applications against a single model, and have it “just work” with all printers
- What’s the alternative?
 - Program our application to know about all possible printers
 - Including those that were invented after we had written our application!

Does a General Model Limit Us?

- Does it stick us with the “least common denominator” of a hardware type?
 - Like limiting us to the least-featureful of all printers?
- Not necessarily
 - The model can include “optional features”
 - If present, implemented in a standard way
 - If not present, test for them and do “something” if they’re not there
- Many devices will have features not in the common model
 - There are arguments for and against the value of such features

Common Types of OS Resources

- Serially reusable resources
- Partitionable resources
- Sharable resources

Seriously Reusable Resources

- Used by multiple clients, but only one at a time
 - Time multiplexing
- Require access control to ensure exclusive use
- Require graceful transitions from one user to the next
- Examples: printers, bathroom stalls

What Is A Graceful Transition?

- A switch that totally hides the fact that the resource used to belong to someone else
 - Don't allow the second user to access the resource until the first user is finished with it
 - No incomplete operations that finish after the transition
 - Ensure that each subsequent user finds the resource in “like new” condition
 - No traces of data or state left over from the first user

Partitionable Resources

- Divided into disjoint pieces for multiple clients
 - Spatial multiplexing
- Needs access control to ensure:
 - Containment: *you cannot access resources outside of your partition*
 - Privacy: *nobody else can access resources in your partition*
- Examples: RAM, hotel rooms

Do We Still Need Graceful Transitions?

- Yes
- Most partitionable resources aren't permanently allocated
 - The page frame of RAM you're using now will belong to another process later
- As long as it's “yours,” no transition required
- But sooner or later it's likely to become someone else's

Shareable Resources

- Usable by multiple concurrent clients
 - Clients do not have to “wait” for access to resource
 - Clients don’t “own” a particular subset of resource
- May involve (effectively) limitless resources
 - Air in a room, shared by occupants
 - Copy of the operating system, shared by processes
- May involve under-the-covers multiplexing
 - Cell-phone channel (time and frequency multiplexed)
 - Shared network interface (time multiplexed)

Do We Still Need Graceful Transitions?

- Typically not
- The shareable resource usually doesn't change state
- Or isn't “reused”
 - Like an execute-only copy of the OS
- Or things that disappear after use
 - Like the previous second's bandwidth on a cellular telephone channel
- Shareable resources are great!
 - When you can have them . . .

A Brief History of Operating Systems

- 1950s ... OS? We don't need no stinking OS!
- 1960s batch processing
 - job sequencing, memory allocation, I/O services
- 1970s time sharing
 - multi-user, interactive service, file systems
- 1980s work stations and personal computers
 - graphical user interfaces, productivity tools
- 1990s work groups and the world wide web
 - shared data, standard protocols, domain services
- 2000 large scale distributed systems
 - the network IS the computer

A Certain Irony

- Today's smart phone is immensely more powerful than 1960s mainframes
- But we used the mainframes for the biggest computing tasks we had
- While we use our powerful smart phones to move information around and display stuff
- Which has implications for their operating systems . . .

General OS Trends

- They have grown larger and more sophisticated
- Their role has fundamentally changed
 - From shepherding the use of the hardware
 - To shielding the applications from the hardware
 - To providing powerful application computing platform
 - To becoming a sophisticated “traffic cop”
- They still sit between applications and hardware
- Best understood through services they provide
 - Capabilities they add
 - Applications they enable
 - Problems they eliminate

Why?

- Ultimately because it's what users want
- The OS must provide core services to applications
- Applications have become more complex
 - More complex internal behavior
 - More complex interfaces
 - More interactions with other software
- The OS needs to help with all that complexity

OS Convergence

- There are a handful of widely used Oss
 - And a few special purpose ones (E.g., real time and embedded system OSes)
- New ones come along very rarely
- OSs in the same family (e.g., Windows or Linux) are used for vastly different purposes
 - Making things challenging for the OS designer
- Most OSs are based on pretty old models
 - Linux comes from Unix (1970s vintage)
 - Windows from the early 1980s

Operating Systems for Mobile Devices

- What's down at the bottom for our smart phones and other devices?
- For Apple devices, ultimately XNU
 - Based on Mach (an 80s system), with some features from other 80s systems (like BSD Unix)
- For Android, ultimately Linux
- For Microsoft, ultimately Windows CE
 - Which has its origins in the 1990s
- None of these is all that new, either

Why Have OSes Converged?

- They're expensive to build and maintain
 - So it's a hard business to get into and stay in
- They only succeed if users choose them over other OS options
 - Which can't happen unless you support all the apps the users want (and preferably better)
 - Which requires other parties to do a lot of work
- You need to have some clear advantage over present acceptable alternatives

A Resulting OS Challenge

- We are basing the OS we use today on an architecture designed 20-40 years ago
- We can make some changes in the architecture
- But not too many
 - Due to compatibility
 - And fundamental characteristics of the architecture
- Requires OS designers and builders to shoehorn what's needed today into what made sense yesterday

Conclusion

- Understanding operating systems is critical to understanding how computers work
- Operating systems interact directly with the hardware
- Operating systems provide services via abstractions
- Operating systems are constrained by many non-technical factors

Operating System Principles: Services, Resources, and Interfaces

CS 111

Operating Systems
Peter Reiher

Outline

- Operating systems services
- System service layers and mechanisms
- Service interfaces and standards
- Service and interface abstractions

Key OS Services

- Generally offered as abstractions
- Important basic categories:
 - CPU/Memory abstractions
 - Processes, threads, virtual machines
 - Virtual address spaces, shared segments
 - Persistent storage abstractions
 - Files and file systems
 - Other I/O abstractions
 - Virtual terminal sessions, windows
 - Sockets, pipes, VPNs, signals (as interrupts)

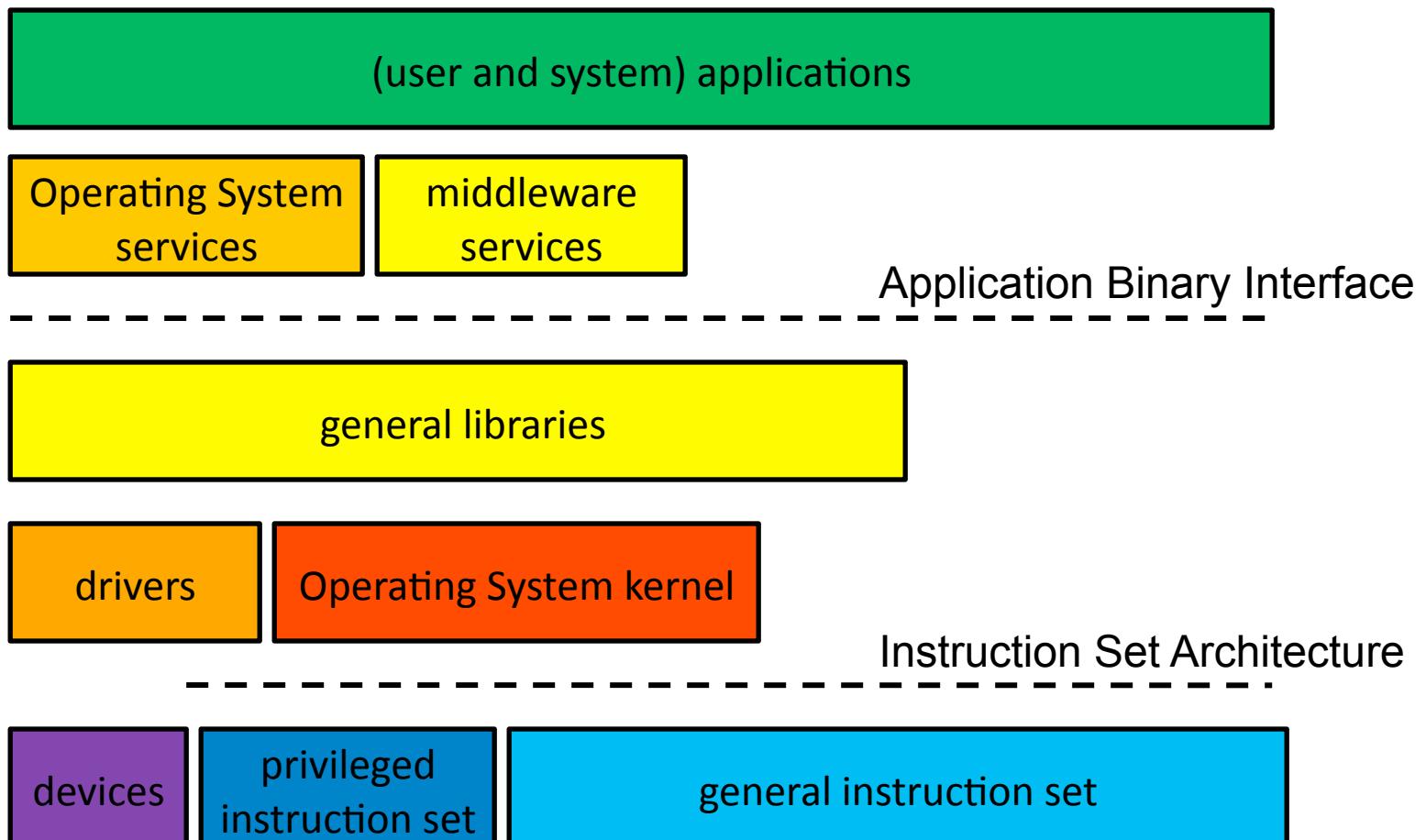
Services: Higher Level Abstractions

- Cooperating parallel processes
 - Locks, condition variables
 - Distributed transactions, leases
- Security
 - User authentication
 - Secure sessions, at-rest encryption
- User interface
 - GUI widgetry, desktop and window management
 - Multi-media

Services: Under the Covers

- Not directly visible to users
- Enclosure management
 - Hot-plug, power, fans, fault handling
- Software updates and configuration registry
- Dynamic resource allocation and scheduling
 - CPU, memory, bus resources, disk, network
- Networks, protocols and domain services
 - USB, BlueTooth
 - TCP/IP, DHCP, LDAP, SNMP
 - iSCSI, CIFS, NFS

Software Layering



How Can the OS Deliver These Services?

- Applications could just call subroutines
- Applications could make system calls
- Applications could send messages to software that performs the services
- At which layer does each of these options work?

Service Delivery via Subroutines

- Access services via direct subroutine calls
 - Push parameters, jump to subroutine, return values in registers on the stack
- Advantages
 - Extremely fast (nano-seconds)
 - DLLs enable run-time implementation binding
- Disadvantages
 - All services implemented in same address space
 - Limited ability to combine different languages
 - Can't usually use privileged instructions

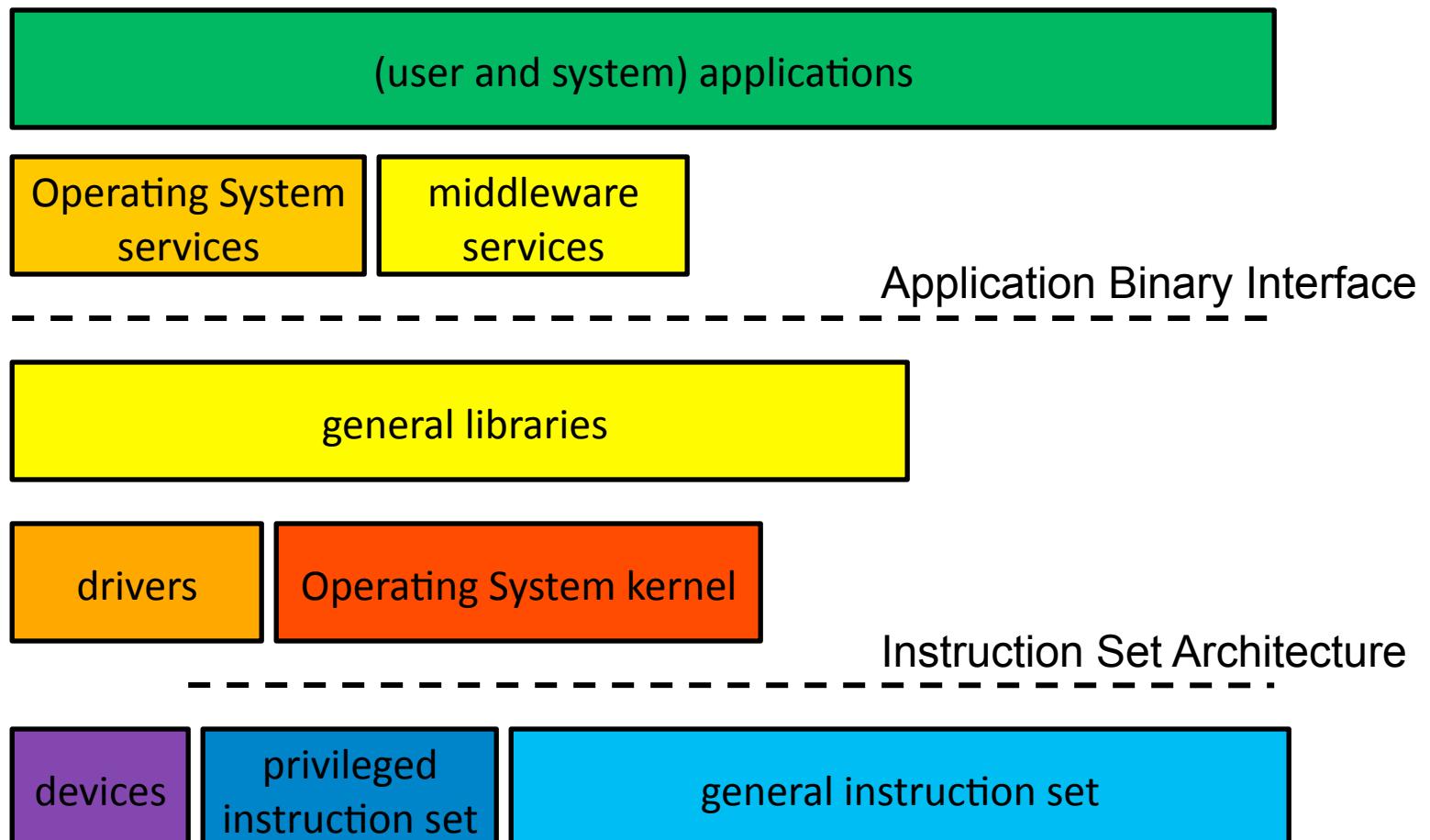
OS Layering

- Modern OSes offer services via layers of software and hardware
- High level abstract services offered at high software layers
- Lower level abstract services offered deeper in the OS
- Ultimately, everything mapped down to relatively simple hardware

Layers: Libraries

- Programmers need not write all code for programs
 - Standard utility functions can be found in libraries
- A library is a collection of object modules
 - A single file that contains many files (like a zip or jar)
 - These modules can be used directly, w/o recompilation
- Most systems come with many standard libraries
 - System services, encryption, statistics, etc.
 - Additional libraries may come with add-on products
- Programmers can build their own libraries
 - Functions commonly needed by parts of a product

The Library Layer



Characteristics of Libraries

- Many advantages
 - Reusable code makes programming easier
 - A single well written/maintained copy
 - Encapsulates complexity ... better building blocks
- Multiple bind-time options
 - Static ... include in load module at link time
 - Shared ... map into address space at exec time
 - Dynamic ... choose and load at run-time
- It is only code ... it has no special privileges

Shared Libraries

- Library modules are usually added to a program's load module
 - Each load module has its own copy of each library
 - This dramatically increases the size of each process
 - Program must be re-linked to incorporate new library
 - Existing load modules don't benefit from bug fixes
- Instead, make each library a sharable code segment
 - One in memory copy, shared by all processes
 - Keep the library separate from the load modules
 - Operating system loads library along with program

Advantages of Shared Libraries

- Reduced memory consumption
 - One copy can be shared by multiple processes/programs
- Faster program start-ups
 - If it's already in memory, it need not be loaded again
- Simplified updates
 - Library modules are not included in program load modules
 - Library can be updated (e.g., a new version with bug fixes)
 - Programs automatically get the newest version when they are restarted

Limitations of Shared Libraries

- Not all modules will work in a shared library
 - They cannot define/include global data storage
- They are read into program memory
 - Whether they are actually needed or not
- Called routines must be known at compile-time
 - Only the fetching of the code is delayed 'til run-time
 - Symbols known at compile time, bound at link time
- Dynamically Loadable Libraries are more general
 - They eliminate all of these limitations ... at a price

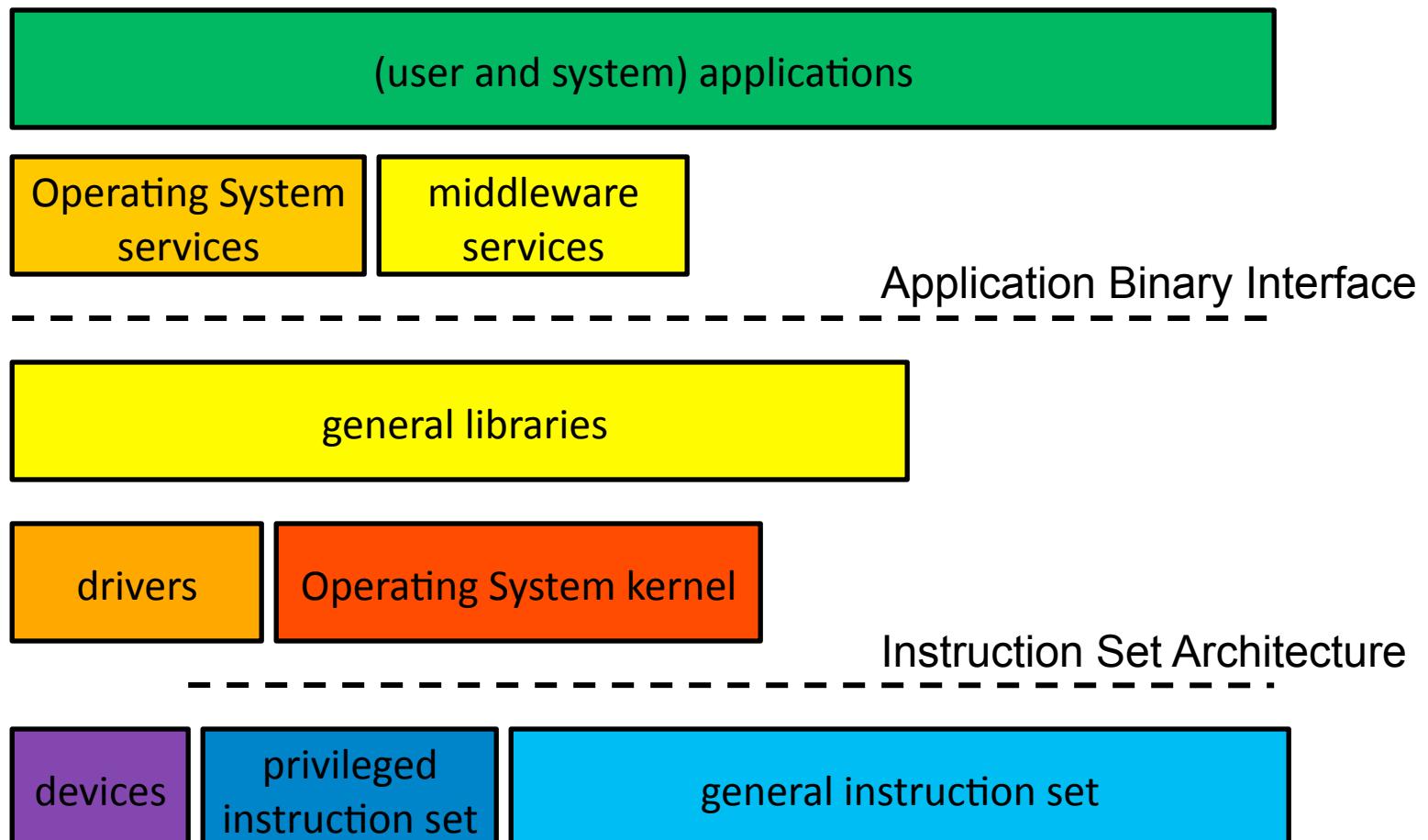
Service Delivery via System Calls

- Force an entry into the operating system
 - Parameters/returns similar to subroutine
 - Implementation is in shared/trusted kernel
- Advantages
 - Able to allocate/use new/privileged resources
 - Able to share/communicate with other processes
- Disadvantages
 - All implemented on the local node
 - 100x-1000x slower than subroutine calls

Layers: The Kernel

- Primarily functions that require privilege
 - Privileged instructions (e.g., interrupts, I/O)
 - Allocation of physical resources (e.g., memory)
 - Ensuring process privacy and containment
 - Ensuring the integrity of critical resources
- Some operations may be out-sourced
 - System daemons, server processes
- Some plug-ins may be less-trusted
 - Device drivers, file systems, network protocols

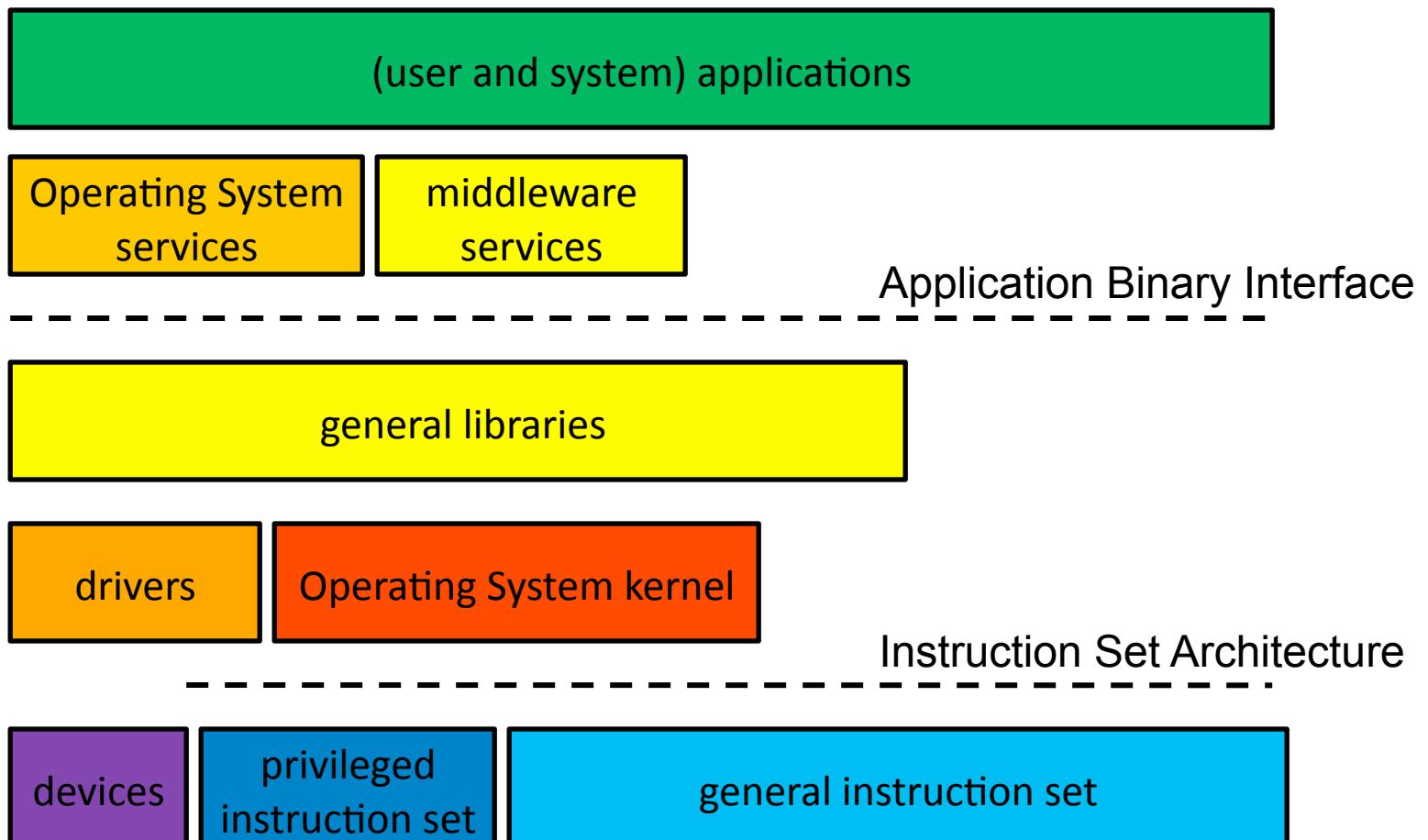
The Kernel Layer



Layers: System Services

- Not all trusted code must be in the kernel
 - It may not need to access kernel data structures
 - It may not need to execute privileged instructions
- Some are actually somewhat privileged processes
 - Login can create/set user credentials
 - Some can directly execute I/O operations
- Some are merely trusted
 - sendmail is trusted to properly label messages
 - NFS server is trusted to honor access control data

System Service Layer



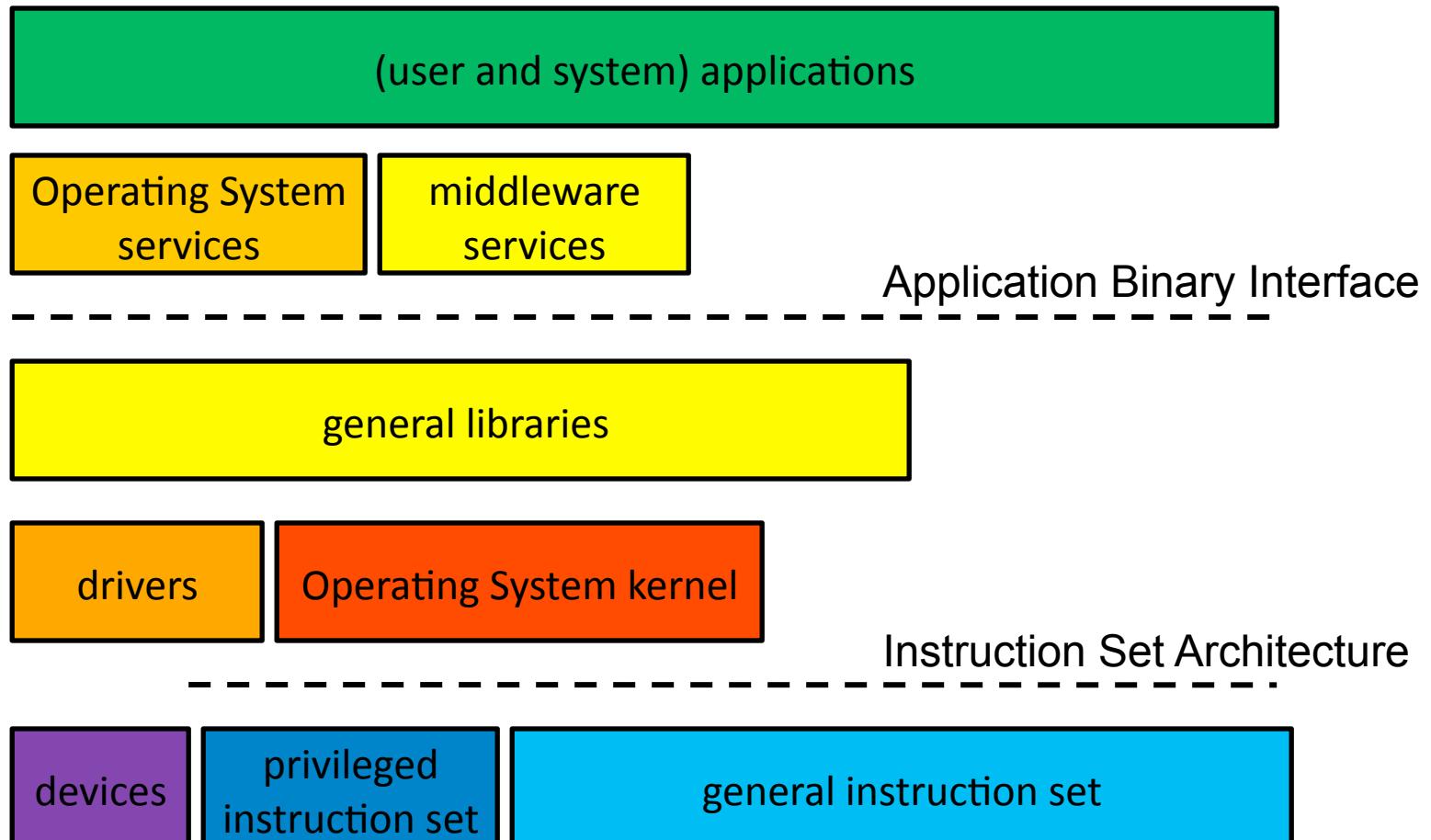
Service Delivery via Messages

- Exchange messages with a server (via syscalls)
 - Parameters in request, returns in response
- Advantages:
 - Server can be anywhere on earth
 - Service can be highly scalable and available
 - Service can be implemented in user-mode code
- Disadvantages:
 - 1,000x-100,000x slower than subroutine
 - Limited ability to operate on process resources

Layers: Middleware

- Software that is a key part of the application or service platform, but not part of the OS
 - Database, pub/sub messaging system
 - Apache, Nginx
 - Hadoop, Zookeeper, Beowulf, OpenStack
 - Cassandra, RAMCloud, Ceph, Gluster
- Kernel code is very expensive and dangerous
 - User-mode code is easier to build, test and debug
 - User-mode code is much more portable
 - User-mode code can crash and be restarted

The Middleware Layer



OS Interfaces

- Nobody buys a computer to run the OS
- The OS is meant to support other programs
 - Via its abstract services
- Usually intended to be very general
 - Supporting many different programs
- Interfaces are required between the OS and other programs to offer general services

Interfaces: APIs

- Application Program Interfaces
 - A source level interface, specifying:
 - Include files, data types, constants
 - Macros, routines and their parameters
- A basis for software portability
 - Recompile program for the desired architecture
 - Linkage edit with OS-specific libraries
 - Resulting binary runs on that architecture and OS
- An API compliant program will compile & run on any compliant system
 - APIs are primarily for programmers

Interfaces: ABIs

- Application Binary Interfaces
 - A binary interface, specifying:
 - Dynamically loadable libraries (DLLs)
 - Data formats, calling sequences, linkage conventions
 - The binding of an API to a hardware architecture
- A basis for binary compatibility
 - One binary serves all customers for that hardware
 - E.g. all x86 Linux/BSD/MacOS/Solaris/...
- An ABI compliant program will run (unmodified) on any compliant system
- ABIs are primarily for users

Why Does My OS Need to Support an ABI?

- Why not just support an API?
- Users would not like that much
- API-only compatibility requires them to obtain and compile their applications' sources
 - If it doesn't build, they have to debug it
- ABI compatibility allows merely loading and running the application (binary)
 - Of course, if it doesn't run, they're out of luck

User Mode Instruction Set vs. ABI

- Why distinguish the user mode instruction set from the Application Binary?
- The user mode instruction set is defined and implemented by hardware
 - It is thus ISA specific
- The Application Binary Interface is defined and implemented by software
 - It is thus OS specific
- Compilers generate code from the user-mode instruction set
- Code that exploits features in the Application Binary Interface is written by people (or higher level tools)

Other Important OS Interfaces

- Data formats and information encodings
 - Multi-media content (e.g. MP3, JPG)
 - Archival (e.g. tar, gzip)
 - File systems (e.g. DOS/FAT, ISO 9660)
- Protocols
 - Networking (e.g. ethernet, WLAN, TCP/IP)
 - Domain services (e.g. IMAP, LPD)
 - System management (e.g. DHCP, SNMP, LDAP)
 - Remote data access (e.g. FTP, HTTP, CIFS, S3)

Interfaces and Interoperability

- Strong, stable interfaces are key to allowing programs to operate together
- Also key to allowing OS evolution
- You don't want an OS upgrade to break your existing programs
- Which means the interface between the OS and those programs better not change

Interoperability Requires Stability

- No program is an island
 - Programs use system calls
 - Programs call library routines
 - Programs operate on external files
 - Programs exchange messages with other software
 - If interfaces change, programs fail
- API requirements are frozen at compile time
 - Execution platform must support those interfaces
 - All partners/services must support those protocols
 - All future upgrades must support older interfaces

Interoperability Requires Compliance

- Complete interoperability testing is impossible
 - Cannot test all applications on all platforms
 - Cannot test interoperability of all implementations
 - New apps and platforms are added continuously
- Instead, we focus on the interfaces
 - Interfaces are completely and rigorously specified
 - Standards bodies manage the interface definitions
 - Compliance suites validate the implementations
- And hope that sampled testing will suffice

Compatibility Taxonomy

- Upwards compatible (with ...)
 - New version still supports previous interfaces
- Backwards compatible (with ...)
 - Will correctly interact with old protocol versions
- Versioned interface, version negotiation
 - Parties negotiate a mutually acceptable version
- Compatibility layer
 - A cross-version translator
- Non-disruptive upgrade

Side Effects

- A *side effect* occurs when an action one object has non-obvious consequences
 - Perhaps even to other objects
 - Effects not specified by interfaces
- Often due to shared state between seemingly independent modules and functions
- Side effects lead to unexpected behaviors
- And the resulting bugs can be hard to find
- In other words, not good

Standards

- Different than interfaces
- Interfaces can differ from OS to OS
 - And machine to machine
- Standards are more global
- Either you follow a standard or you don't
 - If you do, others can work with you
 - If you don't, they can't
- Where did standards come from?

Standards in the Dark Ages (1965)

- No software industry as we now know it
- All the money was made on hardware
 - But hardware is useless without software
 - All software built by hardware suppliers
 - Platforms were distinguished by software
- Software portability was an anti-goal
 - Keep customers captive to your hardware
 - Portability means they could go elsewhere
- Standards were few and weak

The Software Reformation (1985)

- The advent of the "killer application"
 - Desk-top publishing, spreadsheets, ...
 - The rise of the Independent Software Vendor
- Fundamental changes to platform industry
 - The “applications, demand, volume” cycle
 - Application capture became strategic
- Applications portability became strategic
 - Standards are the key to portability
 - Standards compliance became strategic

The Role of Standards Today

- There are many software standards
 - Subroutines, protocols and data formats, ...
 - Both portability and interoperability
 - Some are general (e.g. POSIX 1003, TCP/IP)
 - Some are very domain specific (e.g. MPEG2)
- Key standards are widely required
 - Non-compliance reduces application capture
 - Non-compliance raises price to customers
- Bottom line: if you don't meet the standard, your system isn't used

Where Do Standards Stop?

- Why not just one browser for everyone?
- And just one image format?
- And just one email program?
- Those could be standards themselves
- Why not?
- Why not just bundle everything into the OS?



Abstractions

- Many things an operating system handles are complex
 - Often due to varieties of hardware, software, configurations
- Life is easy for application programmers and users if they work with a simple abstraction
- The operating system creates, manages, and exports such abstractions

Abstractions: An Object-Oriented View

- My execution platform implements objects
 - They may be bytes, longs, and strings
 - They may be processes, files, and sessions
- An *object* is defined by
 - Its properties, methods, and their semantics
- What makes a particular set of objects good?
 - They are powerful enough to do what I need
 - They don't force me to do a lot of extra work
 - They are simple enough for me to understand

Simplifying Abstractions

- Hardware is fast, but complex and limited
 - Using it correctly is extremely complex
 - It may not support the desired functionality
 - It is not a solution, but merely a building block
- Encapsulate implementation details
 - Error handling, performance optimization
 - Eliminate behavior that is irrelevant to the user
- More convenient or powerful behavior
 - Operations better suited to user needs

Critical OS Abstractions

- The OS provides some core abstractions that our computational model relies on
 - And builds others on top of those
- Memory abstractions
- Processor abstractions
- Communications abstractions

Abstractions of Memory

- Many resources used by programs and people relate to data storage
 - Variables
 - Chunks of allocated memory
 - Files
 - Database records
 - Messages to be sent and received
- These all have some similar properties

The Basic Memory Operations

- Regardless of level or type, memory abstractions support a couple of operations
 - `WRITE(name, value)`
 - Put a value into a memory location specified by name
 - `value <- READ(name)`
 - Get a value out of a memory location specified by name
- Seems pretty simple
- But going from a nice abstraction to a physical implementation can be complex

Some Complicating Factors

- Persistent vs. transient memory
- Size of operations
 - Size the user/application wants to work with
 - Size the physical device actually works with
- Coherence and atomicity
- Latency
- Same abstraction might be implemented with many different physical devices
 - Possibly of very different types

Where Do the Complications Come From?

- At the bottom, the OS doesn't have abstract devices with arbitrary properties
- It has particular physical devices
 - With unchangeable, often inconvenient, properties
- The core OS abstraction problem:
 - Creating the abstract device with the desirable properties from the physical device without them

An Example

- A typical file
- We can read or write the file
- We can read or write arbitrary amounts of data
- If we write the file, we expect our next read to reflect the results of the write
 - *Coherence*
- If there are several reads/writes to the file, we expect each to occur in some order
 - With respect to the others

What Is Implementing the File?

- Commonly a hard disk drive
- Disk drives have peculiar characteristics
 - Long, and worse, variable access latencies
 - Accesses performed in chunks of fixed size
 - Atomicity only for accesses of that size
 - Highly variable performance depending on exactly what gets put where
 - Unpleasant failure modes
- So the operating system needs to smooth out these oddities

What Does That Lead To?

- Great effort by file system component of OS to put things in the right place on a disk
- Reordering of disk operations to improve performance
 - Which complicates providing atomicity
- Optimizations based on caching and read-ahead
 - Which complicates maintaining consistency
- Sophisticated organizations to handle failures

Abstractions of Interpreters

- An interpreter is something that performs commands
- Basically, the element of a computer (abstract or physical) that gets things done
- At the physical level, we have a processor
- That level is not easy to use
- The OS provides us with higher level interpreter abstractions

Basic Interpreter Components

- An instruction reference
 - Tells the interpreter which instruction to do next
- A repertoire
 - The set of things the interpreter can do
- An environment reference
 - Describes the current state on which the next instruction should be performed
- Interrupts
 - Situations in which the instruction reference pointer is overridden

An Example

- A process
- The OS maintains a program counter for the process
 - An instruction reference
- Its source code specifies its repertoire
- Its stack, heap, and register contents are its environment
 - With the OS maintaining pointers to all of them
- No other interpreters should be able to mess up the process' resources

Implementing the Process Abstraction in the OS

- Easy if there's only one process
- But there almost always are multiple processes
- The OS has a certain amount of physical memory
 - To hold the environment information
- There is usually only one set of registers
- The process doesn't have exclusive access to the CPU
 - Due to other processes

What Does That Lead To?

- Schedulers to share the CPU among various processes
- Memory management hardware and software
 - To multiplex memory use among the processes
 - Giving each the illusion of full exclusive use of memory
- Access control mechanisms for other memory abstractions
 - So other processes can't fiddle with my files

Abstractions of Communications

- A communication link allows one interpreter to talk to another
 - On the same or different machines
- At the physical level, memory and cables
- At more abstract levels, networks and interprocess communication mechanisms
- Some similarities to memory abstractions
 - But also differences

Basic Communication Link Operations

- SEND(link_name, outgoing_message_buffer)
 - Send some information contained in the buffer on the named link
- RECEIVE(link_name, incoming_message_buffer)
 - Read some information off the named link and put it into the buffer
- Like WRITE and READ, in some respects

Why Are Communication Links Distinct From Memory?

- Highly variable performance
- Often asynchronous
 - And usually issues with synchronizing the parties
- Receiver may only perform the operation because the SEND occurred
 - Unlike a typical READ
- Additional complications when working with a remote machine

An Example Communications Link

- A Unix-style socket
- SEND interface:
 - `send(int sockfd, const void *buf, size_t len, int flags)`
 - The `sockfd` is the link name
 - The `buf` is the outgoing message buffer
- RECEIVE interface:
 - `recv(int sockfd, void *buf, size_t len, int flags)`
 - Same parameters as for send

Implementing the Communications Link Abstraction in the OS

- Easy if both ends are on the same machine
 - Not so easy if they aren't
- On same machine, use memory to perform the transfer
 - Either copy the message from sender's memory to receiver's
 - Or transfer control of memory containing the message from sender to receiver
- Again, more complicated when remote

What Are the Implications?

- Greater uncertainty about the outcome of an operation
 - Things fail for reasons our OS can't see or learn
 - Even on local machine, since OS doesn't control most of receiver's behavior
- Greater asynchrony
 - Even on a single machine
- Higher possibilities for security problems
 - Particularly when receiver is remote
 - You've heard that a lot

What Do We Do About Those Issues?

- OS must be prepared for likely failures
- And high degrees of asynchrony
 - Bad idea to block entire system while waiting for message delivery
- OS shouldn't have complete trust in what comes in from the network
 - But often the OS is in no position to determine its trustworthiness

Generalizing Abstractions

- How can applications deal with many varied resources?
- Make many different things appear the same
 - Applications can all deal with a single class
 - Often Lowest Common Denominator + sub-classes
- Requires a common/unifying model
 - *Portable document format* for printed output
 - SCSI/SATA/SAS standard for disks, CDs, SSDs
- Usually involves a *federation framework*

Federation Frameworks

- A structure that allows many similar, but somewhat different things to be treated uniformly
- By creating one interface that all must meet
- Then plugging in implementations for the particular things you have
- E.g., make all hard disk drives accept the same commands
 - Even though you have 5 different models installed

Are Federation Frameworks Too Limiting?

- Does the common model have to be the “lowest common denominator”?
- Not necessarily
 - The model can include “optional features”,
 - Which (if present) are implemented in a standard way
 - But may not always be present (and can be tested for)
- Many devices will have features that cannot be exploited through the common model
 - There are arguments for and against the value of such features

Abstractions and Layering

- It's common to create increasingly complex services by layering abstractions
 - E.g., a file system layers on top of an abstract disk, which layers on top of a real disk
- Layering allows good modularity
 - Easy to build multiple services on a lower layer
 - E.g., multiple file systems on one disk
 - Easy to use multiple underlying services to support a higher layer
 - E.g., file system can have either a single disk or a RAID below it

A Downside of Layering

- Layers typically add performance penalties
- Often expensive to go from one layer to the next
 - Since it frequently requires changing data structures or representations
 - At least involves extra instructions
- Another downside is that lower layer may limit what the upper layer can do
 - E.g., an abstract disk prevents disk operation reorderings to maximize performance

Layer Bypassing

- Often necessary to allow a high layer to access much lower layers
 - Not going through one or more intermediaries
- Most commonly for performance reasons
- If the higher layer plans to use the very low level layer's services,
 - Why pay the cost of the intermediate layer?
- Layer bypassing has its downsides, too
 - Intermediate layer can't help or understand

Other OS Abstractions

- There are many other abstractions offered by the OS
- Often they provide different ways of achieving similar goals
 - Some higher level, some lower level
- The OS must do work to provide each abstraction
 - The higher level, the more work
- Programmers and users have to choose the right abstractions to work with

Operating System Principles: Processes, Execution, and State

CS 111

Operating Systems

Peter Reiher

Outline

- What are processes?
- How does an operating system handle processes?
- How do we manage the state of processes?

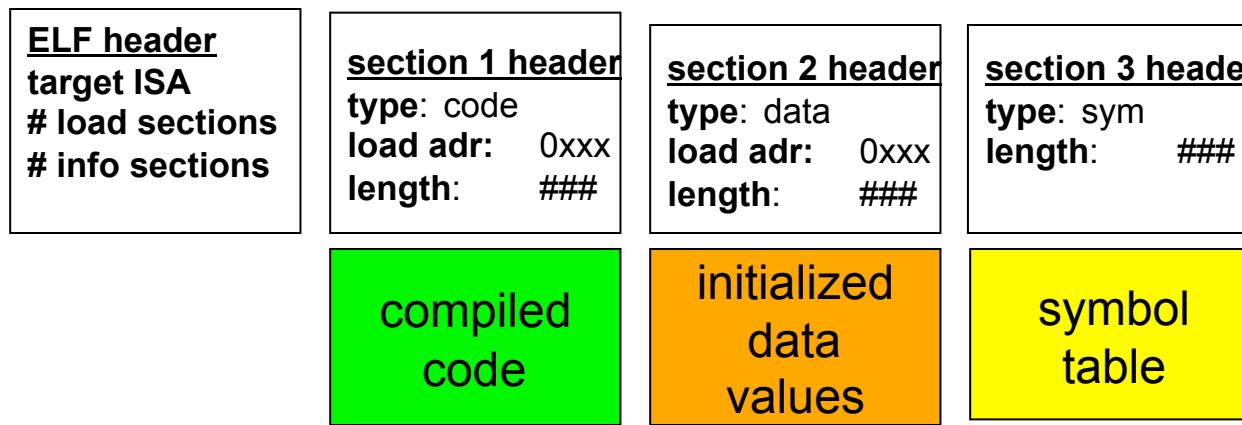
What Is a Process?

- An executing instance of a program
 - How is this different from a program?
- A virtual private computer
 - What does a virtual computer look like?
 - How is a process different from a virtual machine?
- A process is an *object*
 - Characterized by its properties (state)
 - Characterized by its operations

What is “State”?

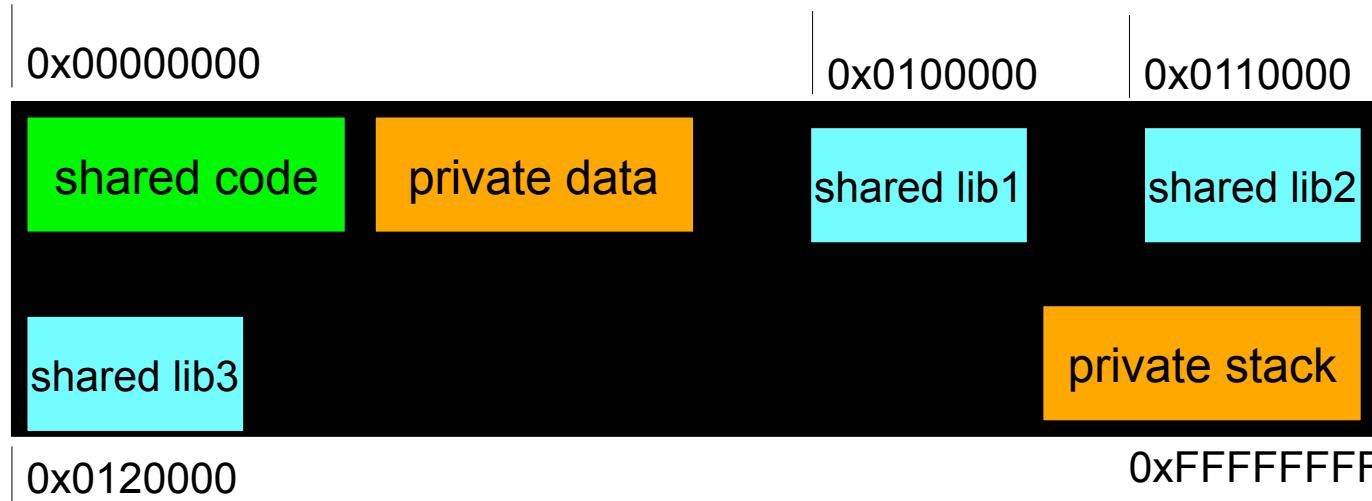
- One dictionary definition of “state” is
 - “A mode or condition of being”
 - An object may have a wide range of possible states
- All persistent objects have “state”
 - Distinguishing it from other objects
 - Characterizing object's current condition
- Contents of state depends on object
 - Complex operations often mean complex state
 - We can save/restore the aggregate/total state
 - We can talk of a subset (e.g., scheduling state)

Program vs. Process Address Space



Program

Process



Process Address Spaces

- Each process has some memory addresses reserved for its private use
- That set of addresses is called its address space
- A process' address space is made up of all memory locations that the process can address
- Modern OSes provide the illusion that the process has all of memory in its address space
 - But that's not true, under the covers

Process Address Space Layout

- All required memory elements for a process must be put somewhere in its address space
- Different types of memory elements have different requirements
 - Code is not writable but must be executable
 - Stacks are readable and writable but not executable
 - Etc.
- Each operating system has some strategy for where to put these process memory segments

Layout of Unix Processes in Memory



- In Unix systems¹,
 - Code segments are statically sized
 - Data segment grows up
 - Stack segment grows down
- They aren't allowed to meet

¹ Linux is one type of Unix system

Address Space: Code Segments

- Load module (output of linkage editor)
 - All external references have been resolved
 - All modules combined into a few segments
 - Includes multiple segments (text, data, BSS)
- Code must be loaded into memory
 - A virtual code segment must be created
 - Code must be read in from the load module
 - Map segment into virtual address space
- Code segments are read/only and sharable
 - Many processes can use the same code segments

Address Space: Data Segments

- Data too must be initialized in address space
 - Process data segment must be created
 - Initial contents must be copied from load module
 - BSS: segments to be initialized to all zeroes
 - Map segment into virtual address space
- Data segments
 - Are read/write, and process private
 - Program can grow or shrink it (using the `sbrk` system call)

Processes and Stack Frames

- Modern programming languages are stack-based
 - Greatly simplified procedure storage management
- Each procedure call allocates a new stack frame
 - Storage for procedure local (vs. global) variables
 - Storage for invocation parameters
 - Save and restore registers
 - Popped off stack when call returns
- Most modern computers also have stack support
 - Stack too must be preserved as part of process state

Address Space: Stack Segment

- Size of stack depends on program activities
 - Grows larger as calls nest more deeply
 - Amount of local storage allocated by each procedure
 - After calls return, their stack frames can be recycled
- OS manages the process's stack segment
 - Stack segment created at same time as data segment
 - Some allocate fixed sized stack at program load time
 - Some dynamically extend stack as program needs it
- Stack segments are read/write and process private

Address Space: Shared Libraries

- Static libraries are added to load module
 - Each load module has its own copy of each library
 - Program must be re-linked to get new version
- Make each library a sharable code segment
 - One in-memory copy, shared by all processes
 - Keep the library separate from the load modules
 - Operating system loads library along with program
- Reduced memory use, faster program loads
- Easier and better library upgrades

Other Process State

- Registers
 - General registers
 - Program counter, processor status
 - Stack pointer, frame pointer
- Processes own OS resources
 - Open files, current working directory, locks
- But also OS-related state information

OS State For a Process

- The state of process's virtual computer
- Registers
 - Program counter, processor status word
 - Stack pointer, general registers
- Address space
 - Text, data, and stack segments
 - Sizes, locations, and contents
- The OS needs some data structure to keep track of a process' state

Process Descriptors

- Basic OS data structure for dealing with processes
- Stores all information relevant to the process
 - State to restore when process is dispatched
 - References to allocated resources
 - Information to support process operations
- Kept in an OS data structure
- Used for scheduling, security decisions, allocation issues

Linux Process Control Block

- The data structure Linux (and other Unix systems) use to handle processes
 - AKA PCB
- An example of a process descriptor
- Keeps track of:
 - Unique process ID
 - State of the process (e.g., running)
 - Parent process ID
 - Address space information
 - And various other things

Other Process State

- Not all process state is stored directly in the process descriptor
- Other process state is in multiple other places
 - Application execution state is on the stack and in registers
 - Linux processes also have a supervisor-mode stack
 - To retain the state of in-progress system calls
 - To save the state of an interrupt preempted process
- A lot of process state is stored in the other memory areas

Handling Processes

- Creating processes
- Destroying processes
- Running processes

Where Do Processes Come From?

- Created by the operating system
 - Using some method to initialize their state
 - In particular, to set up a particular program to run
- At the request of other processes
 - Which specify the program to run
 - And other aspects of their initial state
- Parent processes
 - The process that created your process
- Child processes
 - The processes your process created

Creating a Process Descriptor

- The process descriptor is the OS' basic per-process data structure
- So a new process needs a new descriptor
- What does the OS do with the descriptor?
- Typically puts it into a *process table*
 - The data structure the OS uses to organize all currently active processes

What Else Does a New Process Need?

- An address space
- To hold all of the segments it will need
 - So the OS needs to create one
 - And allocate memory for code, data and stack
 - OS then loads program code and data into new segments
 - Initializes a stack segment
 - Sets up initial registers (PC, PS, SP)

Choices for Process Creation

1. Start with a “blank” process
 - No initial state or resources
 - Have some way of filling in the vital stuff
 - Code
 - Program counter, etc.
 - This is the basic Windows approach
2. Use the calling process as a template
 - Give new process the same stuff as the old one
 - Including code, PC, etc.
 - This is the basic Unix/Linux approach

Starting With a Blank Process

- Basically, create a brand new process
- The system call that creates it obviously needs to provide some information
 - Everything needed to set up the process properly
 - At the minimum, what code is to be run
 - Generally a lot more than that
- Other than bootstrapping, the new process is created by command of an existing process

Windows Process Creation

- The CreateProcess () system call
- A very flexible way to create a new process
 - Many parameters with many possible values
- Generally, the system call includes the name of the program to run
 - In one of a couple of parameter locations
- Different parameters fill out other critical information for the new process
 - Environment information, priorities, etc.

Process Forking

- The way Unix/Linux creates processes
- Essentially clones the existing process
- On assumption that the new process is a lot like the old one
 - Most likely to be true for some kinds of parallel programming
 - Not so likely for more typical user computing

Why Did Unix Use Forking?

- Avoids costs of copying a lot of code
 - If it's the same code as the parents' . . .
- Historical reasons
 - Parallel processing literature used a cloning fork
 - Fork allowed parallelism before threads invented
- Practical reasons
 - Easy to manage shared resources
 - Like stdin, stdout, stderr
 - Easy to set up process pipe-lines (e.g. ls | more)
 - Eases design of command shells

What Happens After a Fork?

- There are now two processes
 - With different IDs
 - But otherwise mostly exactly the same
- How do I profitably use that?
- Program executes a fork
- Now there are two programs
 - With the same code and program counter
- Write code to figure out which is which
 - Usually, parent goes “one way” and child goes “the other”

Forking and the Data Segments

- Forked child shares the parent's code
- But not its stack
 - It has its own stack, initialized to match the parent's
 - Just as if a second process running the same program had reached the same point in its run
- Child should have its own data segment, though
 - Forked processes do not share their data segments

Forking and Copy on Write

- If the parent had a big data area, setting up a separate copy for the child is expensive
 - And fork was supposed to be cheap
- If neither parent nor child write the parent's data area, though, no copy necessary
- So set it up as copy-on-write
- If one of them writes it, then make a copy and let the process write the copy
 - The other process keeps the original

But Fork Isn't What I Usually Want!

- Indeed, you usually don't want another copy of the same process
- You want a process to do something entirely different
- Handled with exec
 - A Unix system call to “remake” a process
 - Changes the code associated with a process
 - Resets much of the rest of its state, too
 - Like open files

The exec Call

- A Linux/Unix system call to handle the common case
- Replaces a process' existing program with a different one
 - New code
 - Different set of other resources
 - Different PC and stack
- Essentially, called after you do a fork

How Does the OS Handle Exec?

- Must get rid of the child's old code
 - And its stack and data areas
 - Latter is easy if you are using copy-on-write
- Must load a brand new set of code for that process
- Must initialize child's stack, PC, and other relevant control structure
 - To start a fresh program run for the child process

Loading Programs Into Processes

- Whether you did a Windows CreateProcess () or a Unix exec ()
 - You need to go from program to runnable process
- To get from the code to the running version, you need to perform the *loading* step
 - Initializing the various memory domains we discussed earlier
 - Code, stack, data segment, etc.

Loading Programs

- You have a load module
 - The output of linkage editor
 - All external references have been resolved
 - All modules combined into a few segments
 - Includes multiple segments (code, data, etc.)
- A computer cannot “execute” a load module
 - Computers execute instructions in memory
 - Memory must be allocated for each segment
 - Code must be copied from load module to memory

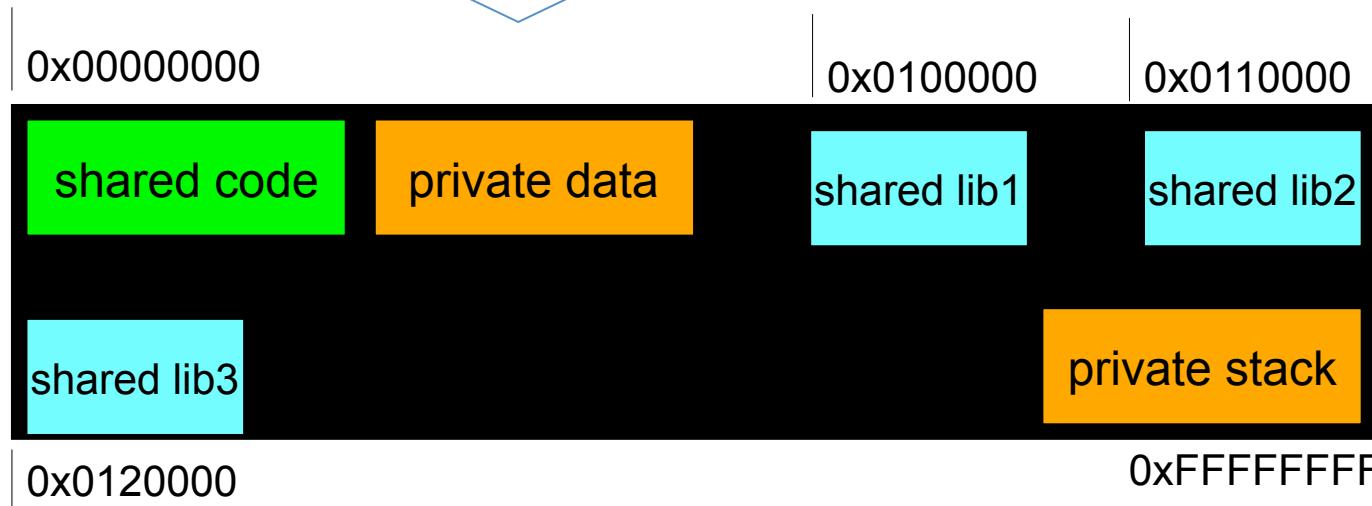
Program to Process Transition

ELF header
target ISA
load sections
info sections

section
type: co
load ad
length:

com
co

This is the job of the
loader and linkage **ram**
editor



Process

Destroying Processes

- Most processes terminate
 - All do, of course, when the machine goes down
 - But most do some work and then exit before that
 - Others are killed by the OS or another process
- When a process terminates, the OS needs to clean it up
 - Essentially, getting rid of all of its resources
 - In a way that allows simple reclamation

What Must the OS Do to Terminate a Process?

- Reclaim any resources it may be holding
 - Memory
 - Locks
 - Access to hardware devices
- Inform any other process that needs to know
 - Those waiting for interprocess communications
 - Parent (and maybe child) processes
- Remove process descriptor from the process table

Running Processes

- Processes must execute code to do their job
- Which means the OS must give them access to a processor core
- But there are usually more processes than cores
- So processes will need to share the cores
 - And they can't all execute instructions at once
- Sooner or later, a process not running on a core needs to be put onto one

Loading a Process

- To run a process on a core, the hardware must be initialized
 - Either to initial state or whatever state it was in the last time it ran
- Must load the core's registers
- Must initialize the stack and set the stack pointer
- Must set up any memory control structures
- Must set the program counter
- Then what?

How a Process Runs on an OS

- It uses an execution model called *limited direct execution*
- Most instructions are executed directly by the process on the core
- Some instructions instead cause a trap to the operating system
 - Privileged instructions that can only execute in supervisor mode
 - The OS takes care of things from there

Limited Direct Execution

- CPU directly executes all application code
 - Punctuated by occasional traps (for system calls)
 - With occasional timer interrupts (for time sharing)
- Maximizing direct execution is always the goal
 - For Linux user mode processes
 - For OS emulation (e.g., Windows on Linux)
 - For virtual machines
- Enter the OS as seldom as possible
 - Get back to the application as quickly as possible

Exceptions

- The technical term for what happens when the process can't (or shouldn't) run an instruction
- Some exceptions are routine
 - End-of-file, arithmetic overflow, conversion error
 - We should check for these after each operation
- Some exceptions occur unpredictably
 - Segmentation fault (e.g. dereferencing NULL)
 - User abort (^C), hang-up, power-failure
 - These are asynchronous exceptions

Asynchronous Exceptions

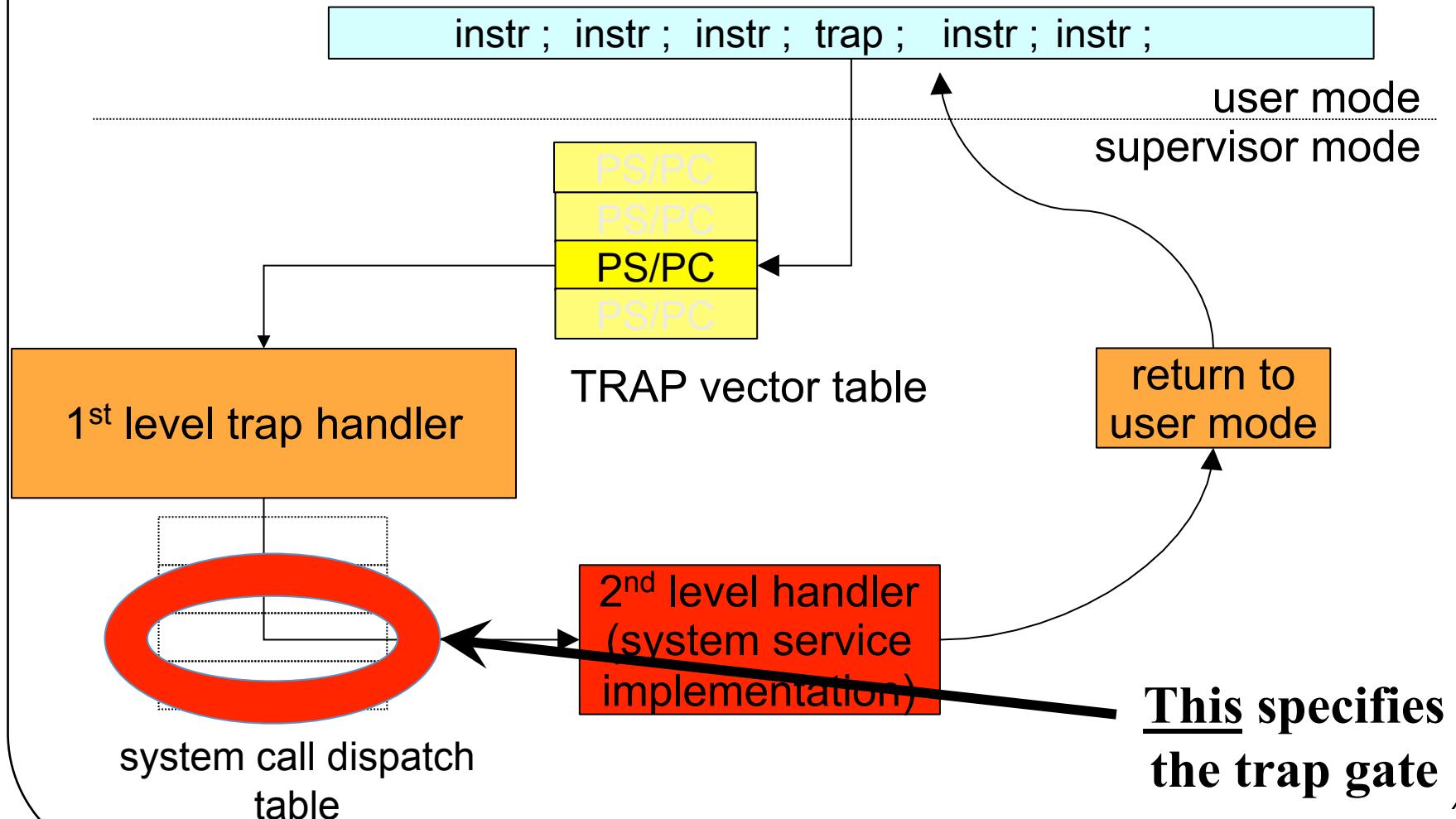
- Inherently unpredictable
- Programs can't check for them, since no way of knowing when and if they happen
- Some languages support try/catch operations
- Hardware and OS support traps
 - Which catch these exceptions and transfer control to the OS
- Operating systems also use these for *system calls*
 - Requests from a program for OS services

Using Traps for System Calls

- Reserve one illegal instruction for system calls
 - Most computers specifically define such instructions
- Define system call linkage conventions
 - Call: r0 = system call number, r1 points to arguments
 - Return: r0 = return code, cc indicates success/failure
- Prepare arguments for the desired system call
- Execute the designated system call instruction
- OS recognizes & performs requested operation
- Returns to instruction after the system call

System Call Trap Gates

Application Program

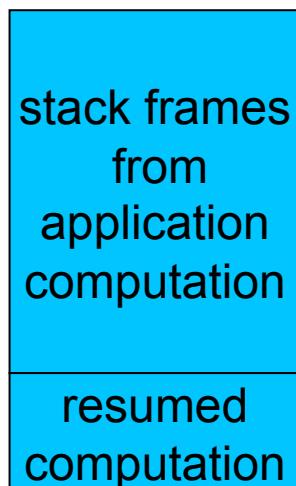


Trap Handling

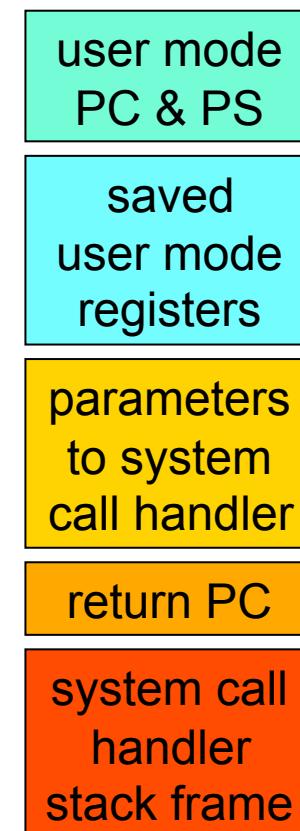
- Hardware portion of trap handling
 - Trap cause as index into trap vector table for PC/PS
 - Load new processor status word, switch to supervisor mode
 - Push PC/PS of program that caused trap onto stack
 - Load PC (with address of 1st level handler)
- Software portion of trap handling
 - 1st level handler pushes all other registers
 - 1st level handler gathers info, selects 2nd level handler
 - 2nd level handler actually deals with the problem
 - Handle the event, kill the process, return ...

Stacking and Unstacking a System Call

User-mode Stack



Supervisor-mode Stack



direction
of growth

Returning to User-Mode

- Return is opposite of interrupt/trap entry
 - 2nd level handler returns to 1st level handler
 - 1st level handler restores all registers from stack
 - Use privileged return instruction to restore PC/PS
 - Resume user-mode execution at next instruction
- Saved registers can be changed before return
 - Change stacked user r0 to reflect return code
 - Change stacked user PS to reflect success/failure

Asynchronous Events

- Some things are worth waiting for
 - When I read(), I want to wait for the data
- Sometimes waiting doesn't make sense
 - I want to do something else while waiting
 - I have multiple operations outstanding
 - Some events demand very prompt attention
- We need event completion call-backs
 - This is a common programming paradigm
 - Computers support interrupts (similar to traps)
 - Commonly associated with I/O devices and timers

User-Mode Signal Handling

- OS defines numerous types of signals
 - Exceptions, operator actions, communication
- Processes can control their handling
 - Ignore this signal (pretend it never happened)
 - Designate a handler for this signal
 - Default action (typically kill or coredump process)
- Analogous to hardware traps/interrupts
 - But implemented by the operating system
 - Delivered to user mode processes

Managing Process State

- A shared responsibility
- The process itself takes care of its own stack
- And the contents of its memory
- The OS keeps track of resources that have been allocated to the process
 - Which memory
 - Open files and devices
 - Supervisor stack
 - And many other things

Blocked Processes

- One important process state element is whether a process is ready to run
 - No point in dispatching it if it isn't
- Why might it not be?
- Perhaps it's waiting for I/O
- Or for some resource request to be satisfied
- The OS keeps track of whether a process is blocked

Blocking and Unblocking Processes

- Why do we block processes?
 - Blocked/unblocked are merely notes to scheduler
 - So the scheduler knows not to choose them
 - And so other parts of OS know if they later need to unblock
- Any part of OS can set blocks, any part can change them
 - And a process can ask to be blocked itself
- Usually happens in a resource manager
 - When process needs an unavailable resource
 - Change process's scheduling state to "blocked"
 - Call the scheduler and yield the CPU
 - When the required resource becomes available
 - Change process's scheduling state to "ready"
 - Notify scheduler that a change has occurred

Swapping Processes

- Processes can only run out of main memory
 - CPU can only execute instructions stored in that memory
- Sometimes we move processes out of main memory to secondary storage
 - E.g., a disk drive
 - Expecting that we'll move them back later
- Usually because of resource shortages
 - Particularly memory

Why We Swap

- To make best use of a limited amount of memory
 - A process can only execute if it is in memory
 - Max number of processes is limited by memory size
 - If it isn't READY, it doesn't need to be in memory
 - Swap it out and make room for some other process
- We don't swap out all blocked processes
 - Swapping is expensive
 - And also expensive to bring them back
 - Typically only done when resources are tight

Basic Mechanics of Swapping

- Process' state is stored in parts of main memory
- Copy them out to secondary storage
 - If you're lucky and careful, some don't need to be copied
- Alter the process descriptor to indicate what you did
- Give the freed resources to another process

Swapping Back

- When whatever blocked the process you swapped is cleared, you can swap back
 - Assuming there's space
- Reallocate required memory and copy state back from secondary storage
 - Both stack and heap
- Unblock the process' descriptor to make it eligible for scheduling
- Ready swapped processes need not be brought back immediately
 - But they won't get any cycles till you do

Operating System Principles: Scheduling

CS 111

Operating Systems
Peter Reiher

Outline

- What is scheduling?
 - What are our scheduling goals?
- What resources should we schedule?
- Example scheduling algorithms and their implications

What Is Scheduling?

- An operating system often has choices about what to do next
- In particular:
 - For a resource that can serve one client at a time
 - When there are multiple potential clients
 - Who gets to use the resource next?
 - And for how long?
- Making those decisions is scheduling

OS Scheduling Examples

- What job to run next on an idle core?
 - How long should we let it run?
- In what order to handle a set of block requests for a disk drive?
- If multiple messages are to be sent over the network, in what order should they be sent?

How Do We Decide How To Schedule?

- Generally, we choose goals we wish to achieve
- And design a scheduling algorithm that is likely to achieve those goals
- Different scheduling algorithms try to optimize different quantities
- So changing our scheduling algorithm can drastically change system behavior

The Process Queue

- The OS typically keeps a queue of processes that are ready to run
 - Ordered by whichever one should run next
 - Which depends on the scheduling algorithm used
- When time comes to schedule a new process, grab the first one on the process queue
- Processes that are not ready to run either:
 - Aren't in that queue
 - Or are at the end
 - Or are ignored by scheduler

Potential Scheduling Goals

- Maximize throughput
 - Get as much work done as possible
- Minimize average waiting time
 - Try to avoid delaying too many for too long
- Ensure some degree of fairness
 - E.g., minimize worst case waiting time
- Meet explicit priority goals
 - Scheduled items tagged with a relative priority
- Real time scheduling
 - Scheduled items tagged with a deadline to be met

Different Kinds of Systems, Different Scheduling Goals

- Time sharing
 - Fast response time to interactive programs
 - Each user gets an equal share of the CPU
- Batch
 - Maximize total system throughput
 - Delays of individual processes are unimportant
- Real-time
 - Critical operations must happen on time
 - Non-critical operations may not happen at all

Preemptive Vs. Non-Preemptive Scheduling

- When we schedule a piece of work, we could let it use the resource until it finishes
- Or we could use virtualization techniques to interrupt it part way through
 - Allowing other pieces of work to run instead
- If scheduled work always runs to completion, the scheduler is non-preemptive
- If the scheduler temporarily halts running jobs to run something else, it's preemptive

Pros and Cons of Non-Preemptive Scheduling

- + Low scheduling overhead
- + Tends to produce high throughput
- + Conceptually very simple
- Poor response time for processes
- Bugs can cause machine to freeze up
 - If process contains infinite loop, e.g.
- Not good fairness (by most definitions)
- May make real time and priority scheduling difficult

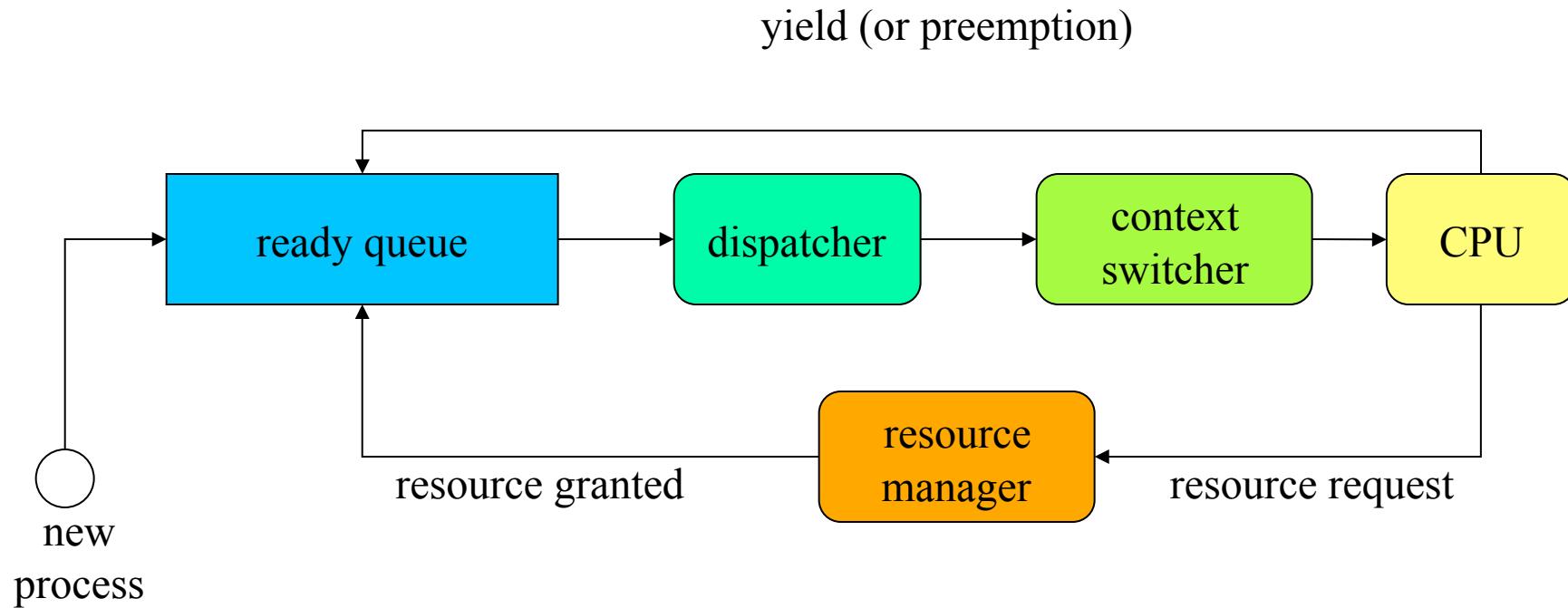
Pros and Cons of Pre-emptive Scheduling

- + Can give good response time
- + Can produce very fair usage
- + Works well with real-time and priority scheduling
- More complex
- Requires ability to cleanly halt process and save its state
- May not get good throughput

Scheduling: Policy and Mechanism

- The scheduler will move jobs into and out of a processor (*dispatching*)
 - Requiring various mechanics to do so
- How dispatching is done should not depend on the policy used to decide who to dispatch
- Desirable to separate the choice of who runs (policy) from the dispatching mechanism
 - Also desirable that OS process queue structure not be policy-dependent

Scheduling the CPU



Scheduling and Performance

- How you schedule important system activities has a major effect on performance
- Performance has different aspects
 - You may not be able to optimize for all of them
- Scheduling performance has very different characteristic under light vs. heavy load
- Important to understand the performance basics regarding scheduling

General Comments on Performance

- Performance goals should be quantitative and measurable
 - If we want “goodness” we must be able to quantify it
 - You cannot optimize what you do not measure
- Metrics ... the way & units in which we measure
 - Choose a characteristic to be measured
 - It must correlate well with goodness/badness of service
 - Find a unit to quantify that characteristic
 - It must a unit that can actually be measured
 - Define a process for measuring the characteristic
- That's enough for now
 - But actually measuring performance is complex

How Should We Quantify Scheduler Performance?

- Candidate metric: throughput (processes/second)
 - But different processes need different run times
 - Process completion time not controlled by scheduler
- Candidate metric: delay (milliseconds)
 - But specifically what delays should we measure?
 - Time to finish a job (turnaround time)?
 - Time to get some response?
 - Some delays are not the scheduler's fault
 - Time to complete a service request
 - Time to wait for a busy resource

Fairness as a Scheduling Metric

- Maybe we want to make sure all processes are treated fairly
- In what dimension?
 - Fairness in delay? Which one?
 - Fairness in time spent processing?
- Many metrics can be used in Jain's fairness equation:

$$J(x_1, x_2, \dots, x_n) = \frac{\left(\sum_{i=1}^n x_i \right)^2}{n \cdot \sum_{i=1}^n x_i^2}$$

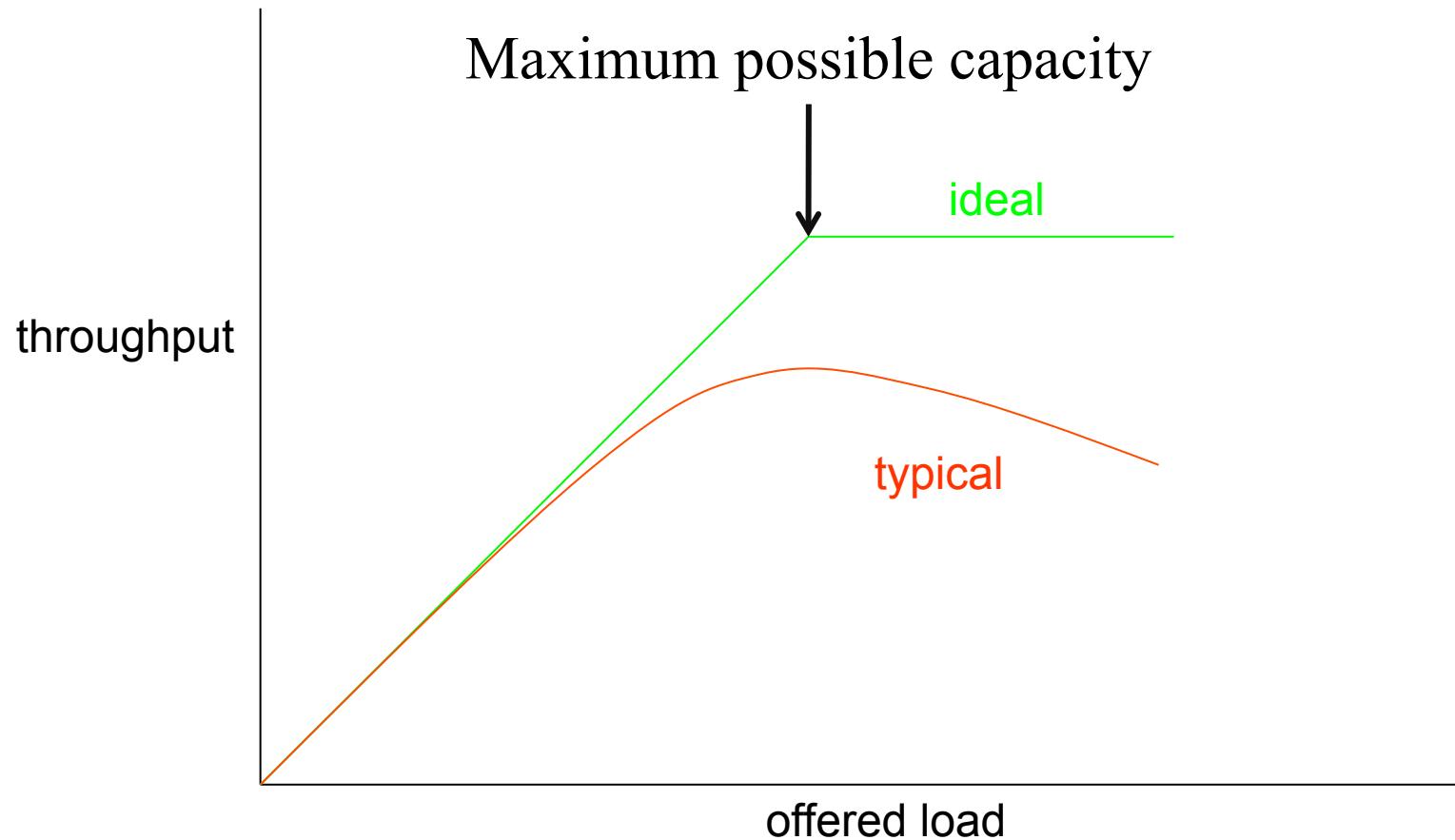
Other Scheduling Metrics

- Mean time to completion (seconds)
 - For a particular job mix (benchmark)
- Throughput (operations per second)
 - For a particular activity or job mix (benchmark)
- Mean response time (milliseconds)
 - Time spent on the ready queue
- Overall “goodness”
 - Requires a customer specific weighting function
 - Often stated in Service Level Agreements

An Example – Measuring CPU Scheduling

- Process execution can be divided into phases
 - Time spent running
 - The process controls how long it needs to run
 - Time spent waiting for resources or completions
 - Resource managers control how long these take
 - Time spent waiting to be run
 - This time is controlled by the scheduler
- Proposed metric:
 - Time that “ready” processes spend waiting for the CPU

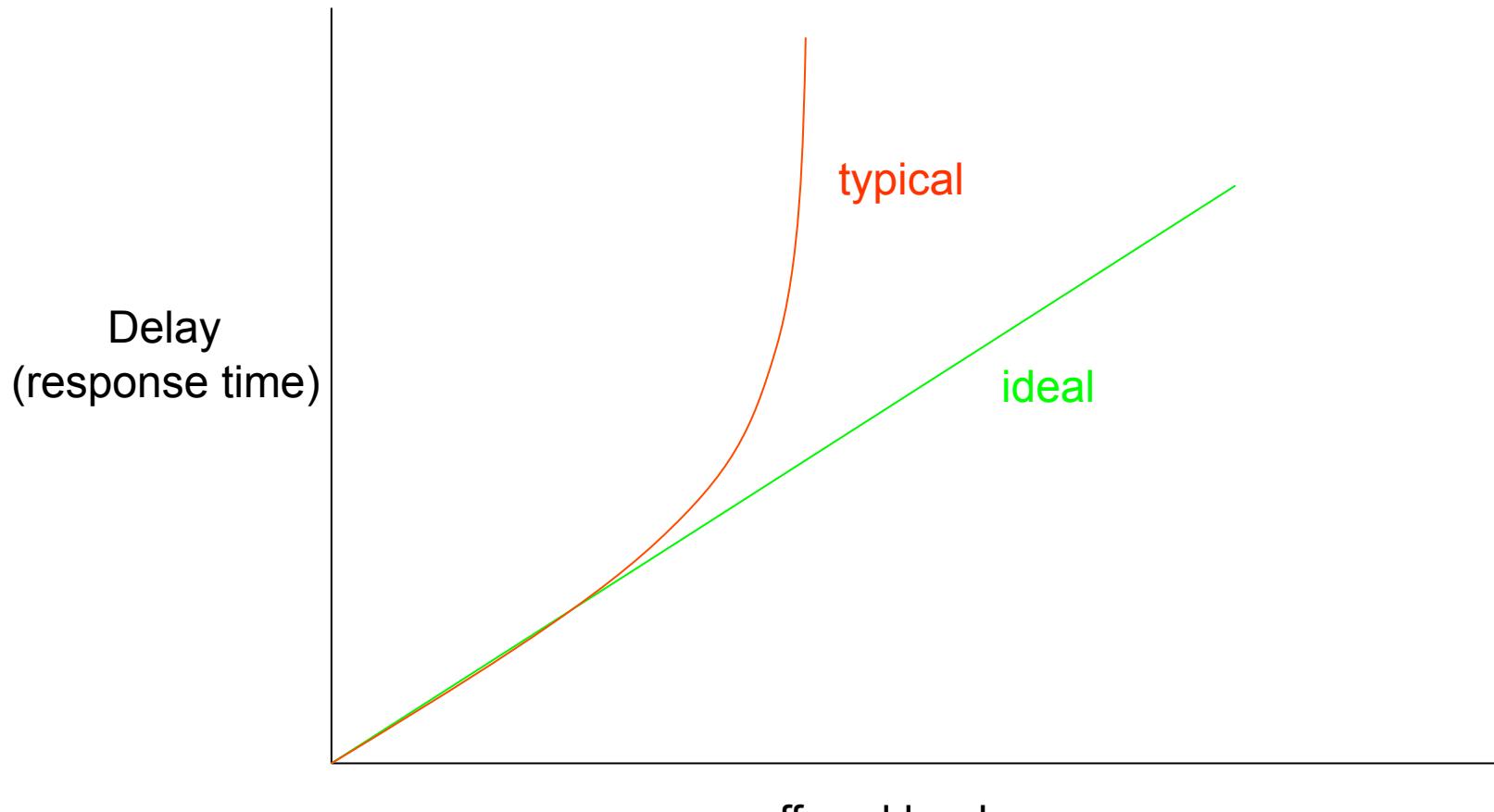
Typical Throughput vs. Load Curve



Why Don't We Achieve Ideal Throughput?

- Scheduling is not free
 - It takes time to dispatch a process (overhead)
 - More dispatches means more overhead (lost time)
 - Less time (per second) is available to run processes
- How to minimize the performance gap
 - Reduce the overhead per dispatch
 - Minimize the number of dispatches (per second)
- This phenomenon is seen in many areas besides process scheduling

Typical Response Time vs. Load Curve



Why Does Response Time Explode?

- Real systems have finite limits
 - Such as queue size
- When limits exceeded, requests are typically dropped
 - Which is an infinite response time, for them
 - There may be automatic retries (e.g., TCP), but they could be dropped, too
- If load arrives a lot faster than it is serviced, lots of stuff gets dropped
- Unless careful, overheads during heavy load explode
- Effects like receive livelock can also hurt in this case

Graceful Degradation

- When is a system “overloaded”?
 - When it is no longer able to meet service goals
- What can we do when overloaded?
 - Continue service, but with degraded performance
 - Maintain performance by rejecting work
 - Resume normal service when load drops to normal
- What should we not do when overloaded?
 - Allow throughput to drop to zero (i.e., stop doing work)
 - Allow response time to grow without limit

Non-Preemptive Scheduling

- Consider in the context of CPU scheduling
- Scheduled process runs until it yields CPU
- Works well for simple systems
 - Small numbers of processes
 - With natural producer consumer relationships
- Good for maximizing throughput
- Depends on each process to voluntarily yield
 - A piggy process can starve others
 - A buggy process can lock up the entire system

Non-Preemptive Scheduling Algorithms

- First come first served
- Shortest job next
 - We won't cover this in detail
- Real time schedulers

First Come First Served

- The simplest of all scheduling algorithms
- Run first process on ready queue
 - Until it completes or yields
- Then run next process on queue
 - Until it completes or yields
- Highly variable delays
 - Depends on process implementations
- All processes will eventually be served

First Come First Served Example

Dispatch Order		0, 1, 2, 3, 4	
Process	Duration	Start Time	End Time
0	350	0	350
1	125	350	475
2	475	475	950
3	250	950	1200
4	75	1200	1275
Total	1275		
Average wait		595	

Note: Average is worse than total/5 because four other processes had to wait for the slow-poke who ran first.

When Would First Come First Served Work Well?

- FCFS scheduling is very simple
- It may deliver very poor response time
- Thus it makes the most sense:
 1. In batch systems, where response time is not important
 2. In embedded (e.g. telephone or set-top box) systems where computations are brief and/or exist in natural producer/consumer relationships

Real Time Schedulers

- For certain systems, some things must happen at particular times
 - E.g., industrial control systems
 - If you don't rivet the widget before the conveyer belt moves, you have a worthless widget
- These systems must schedule on the basis of real-time deadlines
- Can be either *hard* or *soft*

Hard Real Time Schedulers

- The system absolutely must meet its deadlines
- By definition, system fails if a deadline is not met
 - E.g., controlling a nuclear power plant . . .
- How can we ensure no missed deadlines?
- Typically by very, very careful analysis
 - Make sure no possible schedule causes a deadline to be missed
 - By working it out ahead of time
 - Then scheduler rigorously follows deadlines

Ensuring Hard Deadlines

- Must have deep understanding of the code used in each job
 - You know exactly how long it will take
- Vital to avoid non-deterministic timings
 - Even if the non-deterministic mechanism usually speeds things up
 - You're screwed if it ever slows them down
- Typically means you do things like turn off interrupts
- And scheduler is non-preemptive
- Typically you set up a pre-defined schedule
 - No run time decisions

Soft Real Time Schedulers

- Highly desirable to meet your deadlines
- But some (or any) of them can occasionally be missed
- Goal of scheduler is to avoid missing deadlines
 - With the understanding that you might
- May have different classes of deadlines
 - Some “harder” than others
- Need not require quite as much analysis

Soft Real Time Schedulers and Non-Preemption

- Not as vital that tasks run to completion to meet their deadline
 - Also not as predictable, since you probably did less careful analysis
- In particular, a new task with an earlier deadline might arrive
- If you don't pre-empt, you might not be able to meet that deadline

What If You Don't Meet a Deadline?

- Depends on the particular type of system
- Might just drop the job whose deadline you missed
- Might allow system to fall behind
- Might drop some other job in the future
- At any rate, it will be well defined in each particular system

What Algorithms Do You Use For Soft Real Time?

- Most common is Earliest Deadline First
- Each job has a deadline associated with it
 - Based on a common clock
- Keep the job queue sorted by those deadlines
- Whenever one job completes, pick the first one off the queue
- Perhaps prune the queue to remove jobs whose deadlines were missed
- Minimizes *total lateness*

Example of a Soft Real Time Scheduler

- A video playing device
- Frames arrive
 - From disk or network or wherever
- Ideally, each frame should be rendered “on time”
 - To achieve highest user-perceived quality
- If you can’t render a frame on time, might be better to skip it entirely
 - Rather than fall further behind

Preemptive Scheduling

- Again in the context of CPU scheduling
- A thread or process is chosen to run
- It runs until either it yields
- Or the OS decides to interrupt it
- At which point some other process/thread runs
- Typically, the interrupted process/thread is restarted later

Implications of Forcing Preemption

- A process can be forced to yield at any time
 - If a higher priority process becomes ready
 - Perhaps as a result of an I/O completion interrupt
 - If running process's priority is lowered
 - Perhaps as a result of having run for too long
- Interrupted process might not be in a “clean” state
 - Which could complicate saving and restoring its state
- Enables enforced “fair share” scheduling
- Introduces gratuitous context switches
 - Not required by the dynamics of processes
- Creates potential resource sharing problems

Implementing Preemption

- Need a way to get control away from process
 - E.g., process makes a sys call, or clock interrupt
- Consult scheduler before returning to process
 - Has any ready process had its priority raised?
 - Has any process been awakened?
 - Has current process had its priority lowered?
- Scheduler finds highest priority ready process
 - If current process, return as usual
 - If not, yield on behalf of current process and switch to higher priority process

Clock Interrupts

- Modern processors contain a clock
- A peripheral device
 - With limited powers
- Can generate an interrupt at a fixed time interval
- Which temporarily halts any running process
- Good way to ensure that runaway process doesn't keep control forever
- Key technology for preemptive scheduling

Round Robin Scheduling

Algorithm

- Goal - fair share scheduling
 - All processes offered equal shares of CPU
 - All processes experience similar queue delays
- All processes are assigned a nominal time slice
 - Usually the same sized slice for all
- Each process is scheduled in turn
 - Runs until it blocks, or its time slice expires
 - Then put at the end of the process queue
- Then the next process is run
- Eventually, each process reaches front of queue

Properties of Round Robin Scheduling

- All processes get relatively quick chance to do some computation
 - At the cost of not finishing any process as quickly
 - A big win for interactive processes
- Far more context switches
 - Which can be expensive
- Runaway processes do relatively little harm
 - Only take $1/n^{\text{th}}$ of the overall cycles

Round Robin and I/O Interrupts

- Processes get halted by round robin scheduling if their time slice expires
- If they block for I/O (or anything else) on their own, the scheduler doesn't halt them
- Thus, some percentage of the time round robin acts no differently than FIFO
 - When I/O occurs in a process and it blocks

Round Robin Example

Assume a 50 msec time slice (or *quantum*)

Dispatch Order: 0, 1, 2, 3, 4, 0, 1, 2, ...											
Process	Length	1st	2nd	3d	4th	5th	6th	7th	8th	Finish	Switches
0	350	0	250	475	650	800	950	1050		1100	7
1	125	50	300	525						525	3
2	475	100	350	550	700	850	1000	1100	1250	1275	10
3	250	150	400	600	750	900				900	5
4	75	200	450							475	2
Average waiting time: 100 msec										1275	27

First process completed: 475 msec

Comparing Example to Non-Preemptive Examples

- Context switches: 27 vs. 5 (for both FIFO and SJF)
 - Clearly more expensive
- First job completed: 475 msec vs.
 - 75 (shortest job first)
 - 350 (FIFO)
 - Clearly takes longer to complete some process
- Average waiting time: 100 msec vs.
 - 350 (shortest job first)
 - 595 (FIFO)
 - For first opportunity to compute
 - Clearly more responsive

Choosing a Time Slice

- Performance of a preemptive scheduler depends heavily on how long time slice is
- Long time slices avoid too many context switches
 - Which waste cycles
 - So better throughput and utilization
- Short time slices provide better response time to processes
- How to balance?

Costs of a Context Switch

- Entering the OS
 - Taking interrupt, saving registers, calling scheduler
- Cycles to choose who to run
 - The scheduler/dispatcher does work to choose
- Moving OS context to the new process
 - Switch stack, non-resident process description
- Switching process address spaces
 - Map-out old process, map-in new process
- Losing instruction and data caches
 - Greatly slowing down the next hundred instructions

Characterizing Costs of a Time Slice Choice

- What % of CPU use does a process get?
- Depends on workload
 - More processes in queue = fewer slices/second
- CPU share = time_slice * slices/second
 - $2\% = 20\text{ms/sec} = 2\text{ms/slice} * 10 \text{ slices/sec}$
 - $2\% = 20\text{ms/sec} = 5\text{ms/slice} * 4 \text{ slices/sec}$
- Natural rescheduling interval
 - When a typical process blocks for resources or I/O
 - Ideally, fair-share would be based on this period
 - Time-slice ends only if process runs too long

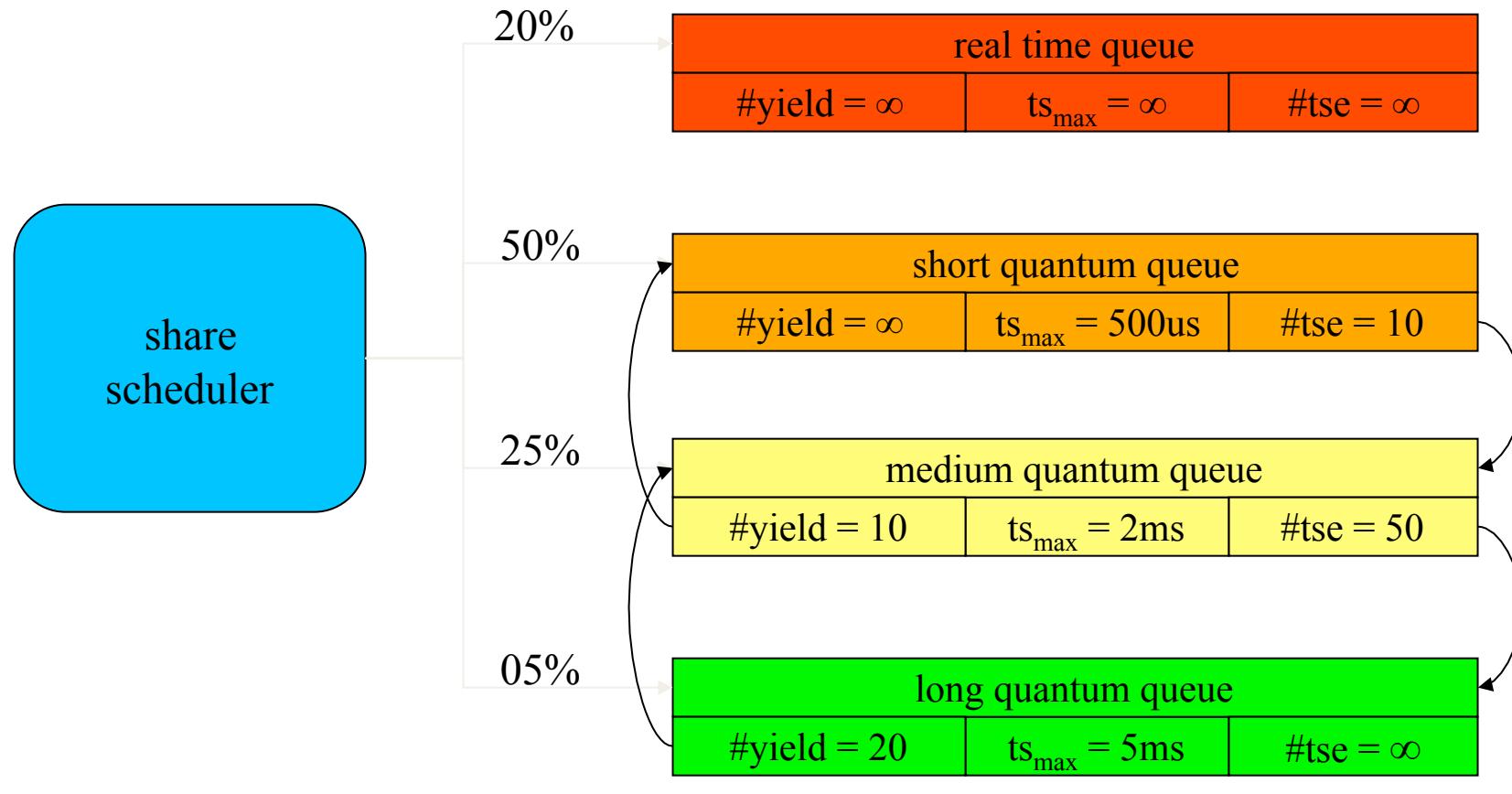
Multi-Level Feedback Queue (MLFQ) Scheduling

- One time slice length may not fit all processes
- Create multiple ready queues
 - Short quantum (foreground) tasks that finish quickly
 - Short but frequent time slices, optimize response time
 - Long quantum (background) tasks that run longer
 - Longer but infrequent time slices, minimize overhead
 - Different queues may get different shares of the CPU
- Finds balance between good response time and good turnaround time

How Do I Know What Queue To Put New Process Into?

- If it's in the wrong queue, its scheduling discipline causes it problems
- Start all processes in short quantum queue
 - Move downwards if too many time-slice ends
 - Move back upwards if too few time slice ends
 - Processes dynamically find the right queue
- If you also have real time tasks, you know what belongs there
 - Start them in real time queue and don't move them

Multiple Queue Scheduling



Priority Scheduling Algorithm

- Sometimes processes aren't all equally important
- We might want to preferentially run the more important processes first
- How would our scheduling algorithm work then?
- Assign each job a priority number
- Run according to priority number

Priority and Preemption

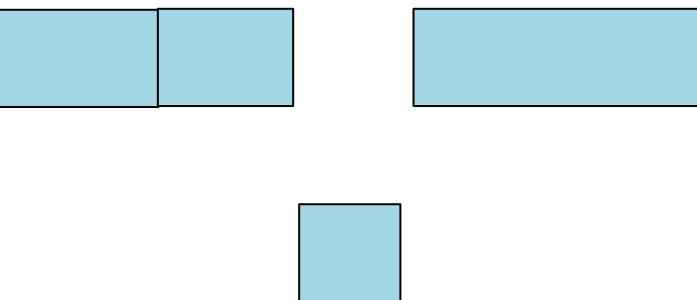
- If non-preemptive, priority scheduling is just about ordering processes
- Much like shortest job first, but ordered by priority instead
- But what if scheduling is preemptive?
- In that case, when new process is created, it might preempt running process
 - If its priority is higher

Priority Scheduling Example

550

Time

Process	Priority	Length
0	10	350
1	30	125
2	40	475
3	20	250
4	50	75



Process 4 completes

So we go back to process 2

Process 3's priority is lower than
running process

Process 4's priority is higher than
running process

Problems With Priority Scheduling

- Possible starvation
- Can a low priority process ever run?
- If not, is that really the effect we wanted?
- May make more sense to adjust priorities
 - Processes that have run for a long time have priority temporarily lowered
 - Processes that have not been able to run have priority temporarily raised

Hard Priorities Vs. Soft Priorities

- What does a priority mean?
- That the higher priority has absolute precedence over the lower?
 - Hard priorities
 - That's what the example showed
- That the higher priority should get a larger share of the resource than the lower?
 - Soft priorities

Priority Scheduling in Linux

- Each process in Linux has a priority
 - Called a *nice* value
 - A soft priority describing share of CPU that a process should get
- Commands can be run to change process priorities
- Anyone can request lower priority for his processes
- Only privileged user can request higher

Priority Scheduling in Windows

- 32 different priority levels
 - Half for regular tasks, half for soft real time
 - Real time scheduling requires special privileges
 - Using a multi-queue approach
- Users can choose from 5 of these priority levels
- Kernel adjusts priorities based on process behavior
 - Goal of improving responsiveness

Operating System Principles: Memory Management

CS 111

Operating Systems
Peter Reiher

Outline

- What is memory management about?
- Memory management strategies:
 - Fixed partition strategies
 - Dynamic partitions
 - Buffer pools
 - Garbage collection
 - Memory compaction

Memory Management

- Memory is one of the key assets used in computing
- In particular, memory abstractions that are usable from a running program
 - Which, in modern machines, typically means RAM
- We have a limited amount of it
- Lots of processes need to use it
- How do we manage it?

Memory Management Goals

1. Transparency

- Process sees only its own address space
- Process is unaware memory is being shared

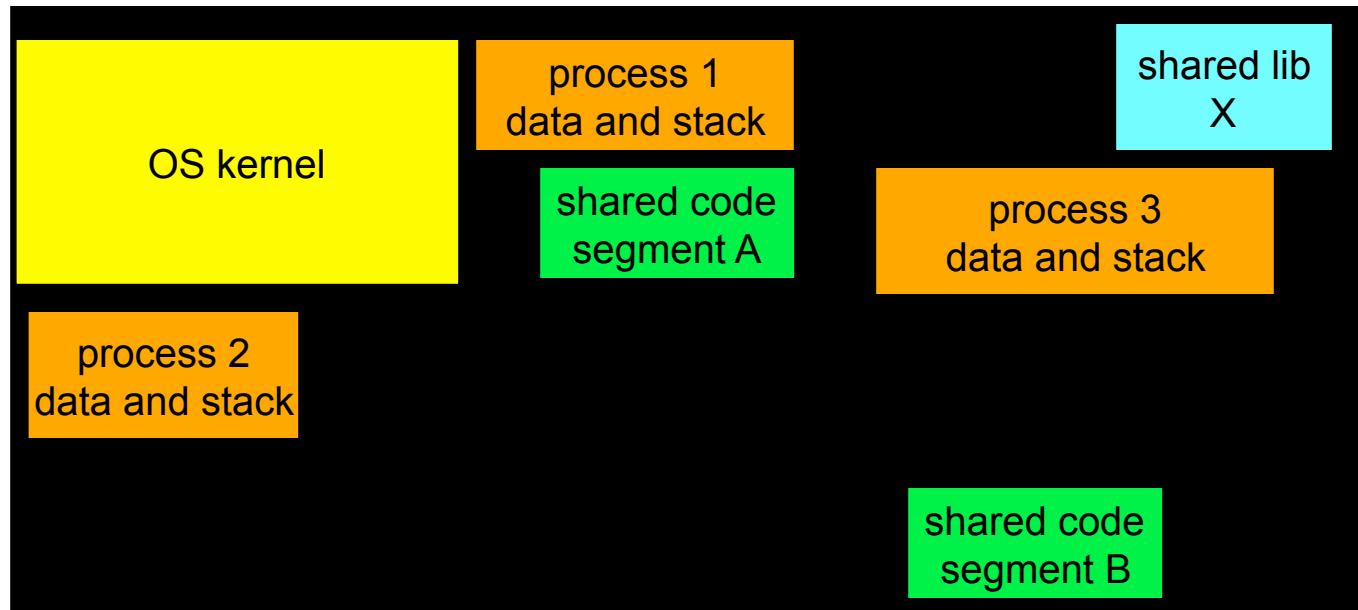
2. Efficiency

- High effective memory utilization
- Low run-time cost for allocation/relocation

3. Protection and isolation

- Private data will not be corrupted
- Private data cannot be seen by other processes

Physical Memory Allocation

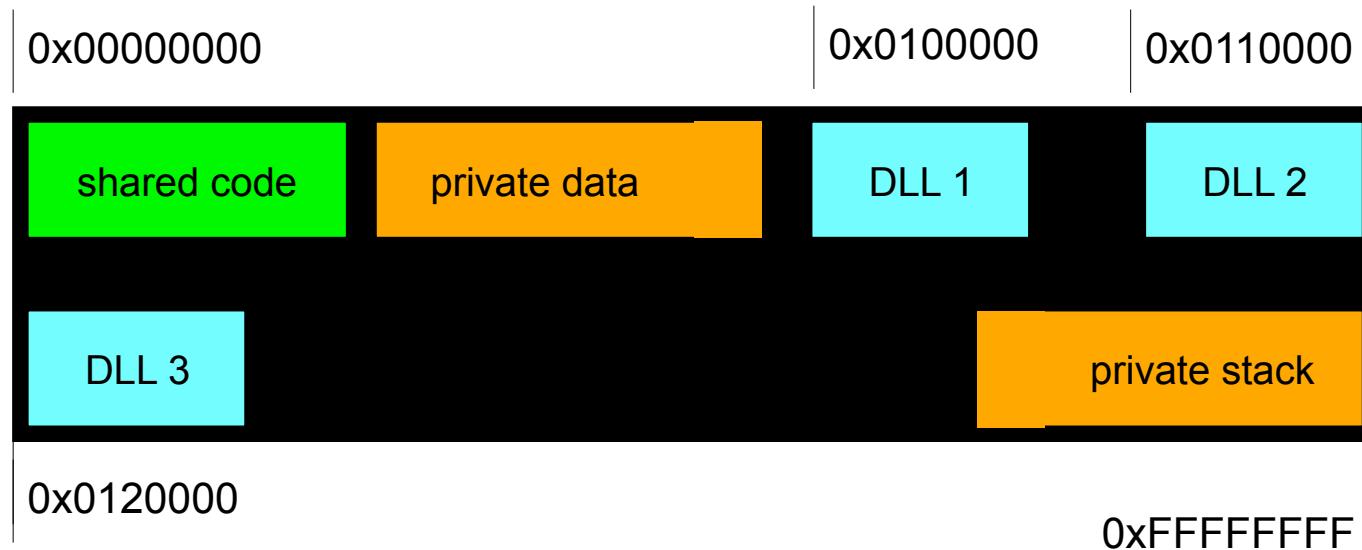


Physical memory is divided between the OS kernel, process private data, and shared code segments.

Physical and Virtual Addresses

- A cell of RAM has a particular physical address
- Years ago, that address was used by processes to name RAM locations
- Instead, we can have processes use virtual addresses
 - Which may not be the same as physical addresses
- More flexibility in memory management, but requires virtual to physical translation

A Linux Process' Virtual Address Space



All of these segments appear to be present in memory whenever the process runs.

Note this virtual address space contains no OS or other process segments

Aspects of the Memory Management Problem

- Most processes can't perfectly predict how much memory they will use
- The processes expect to find their existing data when they need it where they left it
- The entire amount of data required by all processes may exceed amount of available physical memory
- Switching between processes must be fast
 - Can't afford much delay for copying data
- The cost of memory management itself must not be too high

Memory Management Strategies

- Fixed partition allocations
- Dynamic partitions
- Relocation

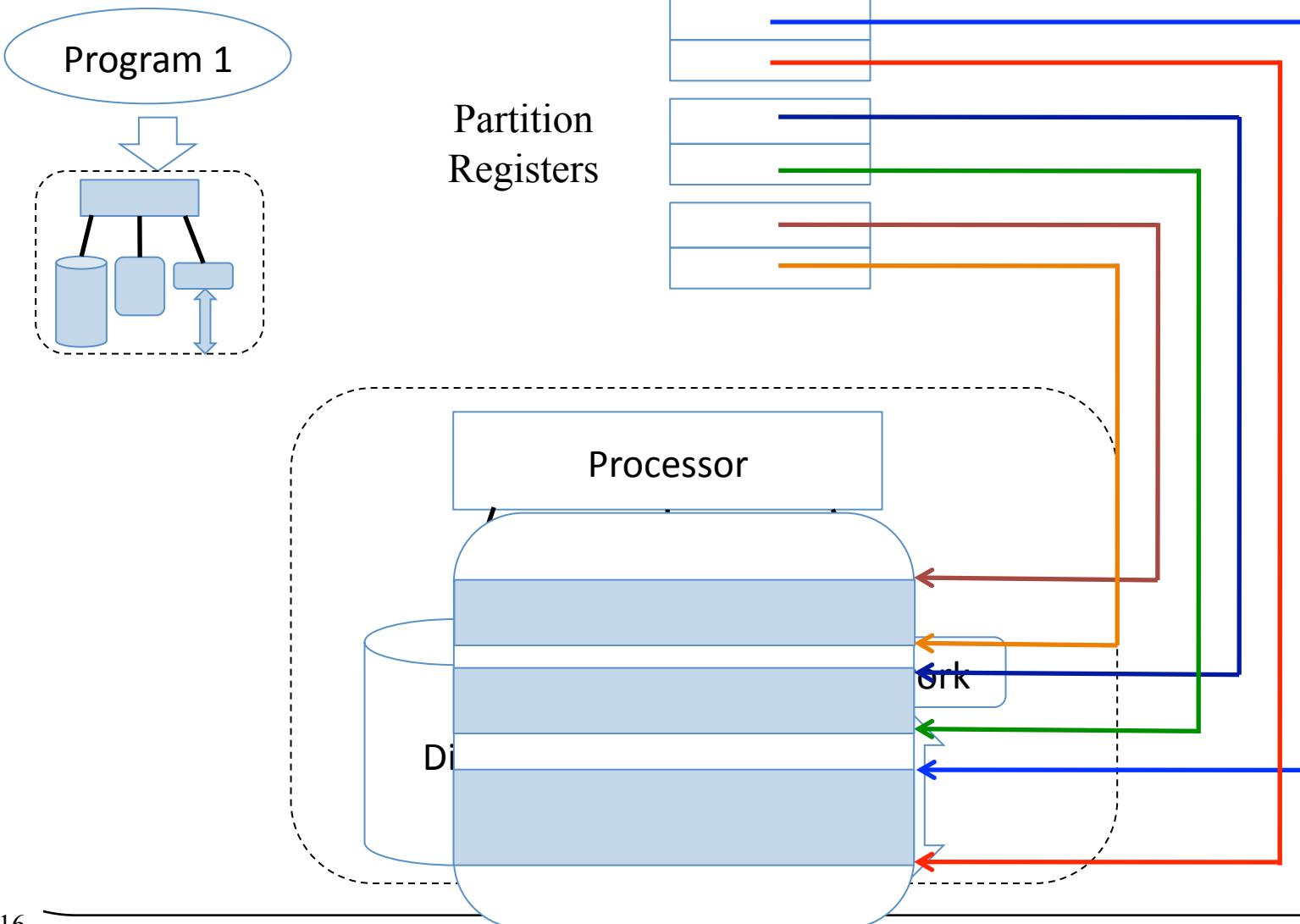
Fixed Partition Allocation

- Pre-allocate partitions for n processes
 - One or more per process
 - Reserving space for largest possible process
- Partitions come in one or a few set sizes
- Very easy to implement
 - Common in old batch processing systems
 - Allocation/deallocation very cheap and easy
- Well suited to well-known job mix

Memory Protection and Fixed Partitions

- Need to enforce partition boundaries
 - To prevent one process from accessing another's memory
- Could use hardware for this purpose
 - Special registers that contain the partition boundaries
 - Only accept addresses within the register values
- Basic scheme doesn't use virtual addresses

The Partition Concept



Problems With Fixed Partition Allocation

- Presumes you know how much memory will be used ahead of time
- Limits the number of processes supported to the total of their memory requirements
- Not great for sharing memory
- *Fragmentation* causes inefficient memory use

Fragmentation

- A problem for all memory management systems
 - Fixed partitions suffer it especially badly
- Based on processes not using all the memory they requested
- As a result, you can't provide memory for as many processes as you theoretically could

Fragmentation Example

Let's say there are three processes, A, B, and C

Their memory requirements:

A: 6 MBytes

B: 3 MBytes

C: 2 MBytes

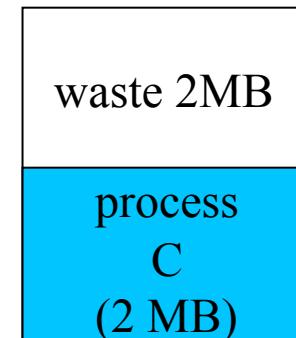
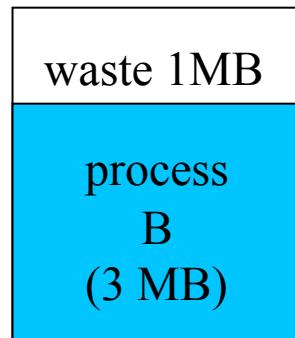
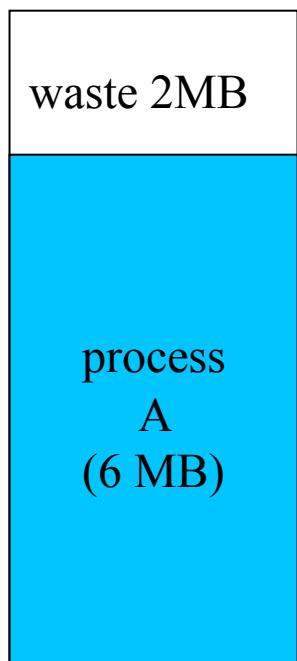
Available partition sizes:

8 Mbytes

4 Mbytes

4 Mbytes

$$\begin{aligned}\text{Total waste} &= 2\text{MB} + 1\text{MB} + 2\text{MB} = \\ &5/16\text{MB} = 31\%\end{aligned}$$



Internal Fragmentation

- Fragmentation comes in two kinds:
 - Internal and external
- This is an example of *internal fragmentation*
 - We'll see external fragmentation later
- Wasted space in fixed sized blocks
 - The requestor was given more than he needed
 - The unused part is wasted and can't be used for others
- Internal fragmentation can occur whenever you force allocation in fixed-sized chunks

More on Internal Fragmentation

- Internal fragmentation is caused by a mismatch between
 - The chosen sizes of a fixed-sized blocks
 - The actual sizes that programs use
- Average waste: 50% of each block
- Overall waste reduced by multiple sizes
 - Suppose blocks come in sizes S_1 and S_2
 - Average waste = $((S_1/2) + (S_2 - S_1)/2)/2$

Summary of Fixed Partition Allocation

- Very simple
- Inflexible
- Subject to a lot of internal fragmentation
- Not used in many modern systems
 - But a possible option for special purpose systems, like embedded systems
 - Where we know exactly what our memory needs will be

Dynamic Partition Allocation

- Like fixed partitions, except
 - Variable sized, usually any size requested
 - Each partition is contiguous in memory addresses
 - Partitions have access permissions for the process
 - Potentially shared between processes
- Each process could have multiple partitions
 - With different sizes and characteristics

Problems With Dynamic Partitions

- Not relocatable
 - Once a process has a partition, you can't easily move its contents elsewhere
- Not easily expandable
- Impossible to support applications with larger address spaces than physical memory
 - Also can't support several applications whose total needs are greater than physical memory
- Also subject to fragmentation

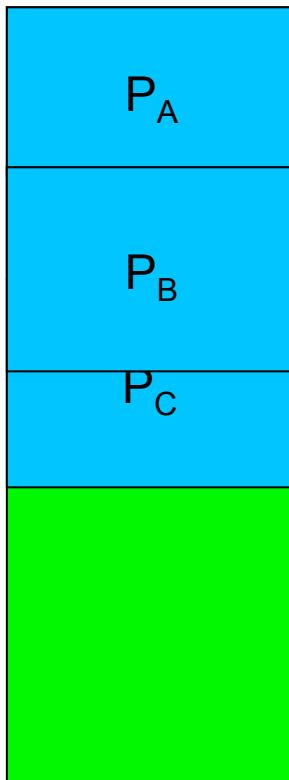
Relocation and Expansion

- Partitions are tied to particular address ranges
 - At least during an execution
- Can't just move the contents of a partition to another set of addresses
 - All the pointers in the contents will be wrong
 - And generally you don't know which memory locations contain pointers
- Hard to expand because there may not be space “nearby”

The Expansion Problem

- Partitions are allocated on request
- Processes may ask for new ones later
- But partitions that have been given can't be moved somewhere else in memory
- Memory management system might have allocated all the space after a given partition
 - In which case, it can't be expanded

Illustrating the Problem



Now Process B wants to
expand its partition size
But if we do that, Process
B steps on Process C's
memory

We can't move C's
partition out of the way
And we can't move B's
partition to a free area

We're stuck, and must deny an expansion request
that we have enough memory to handle

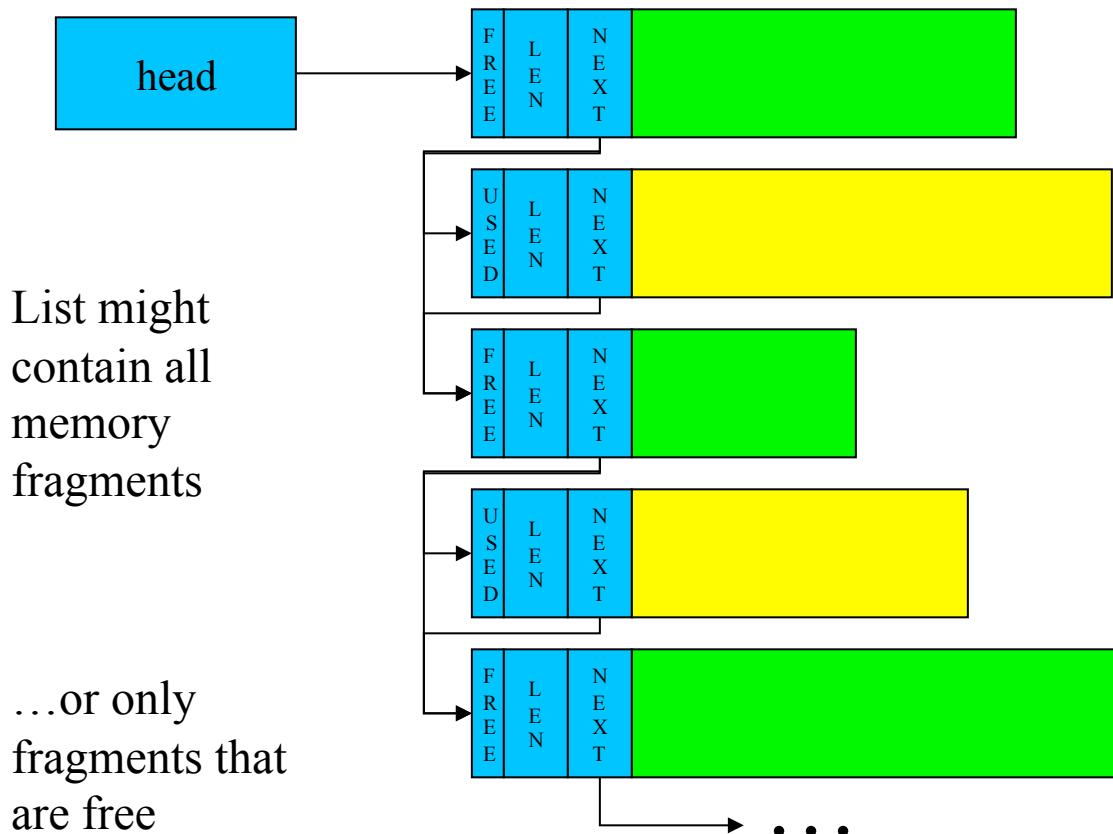
How To Keep Track of Variable Sized Partitions?

- Start with one large “heap” of memory
- Maintain a *free list*
 - Systems data structure to keep track of pieces of unallocated memory
- When a process requests more memory:
 - Find a large enough chunk of memory
 - Carve off a piece of the requested size
 - Put the remainder back on a *free list*
- When a process frees memory
 - Put it back on the free list

Managing the Free List

- Fixed sized blocks are easy to track
 - A bit map indicating which blocks are free
- Variable chunks require more information
 - A linked list of descriptors, one per chunk
 - Each descriptor lists the size of the chunk and whether it is free
 - Each has a pointer to the next chunk on list
 - Descriptors often kept at front of each chunk
- Allocated memory may have descriptors too

The Free List

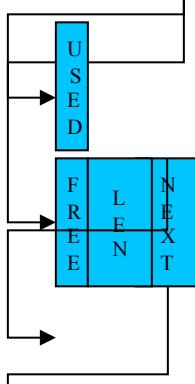


Free Chunk Carving

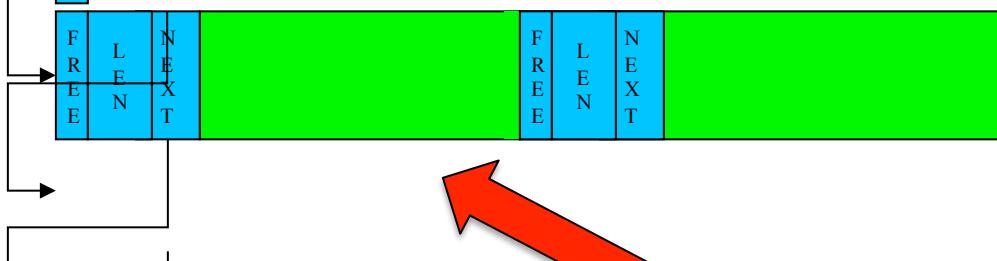
1. Find a large enough free chunk



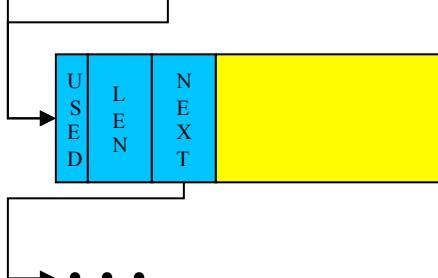
2. Reduce its len to requested size



3. Create a new header for residual chunk



4. Insert the new chunk into the list



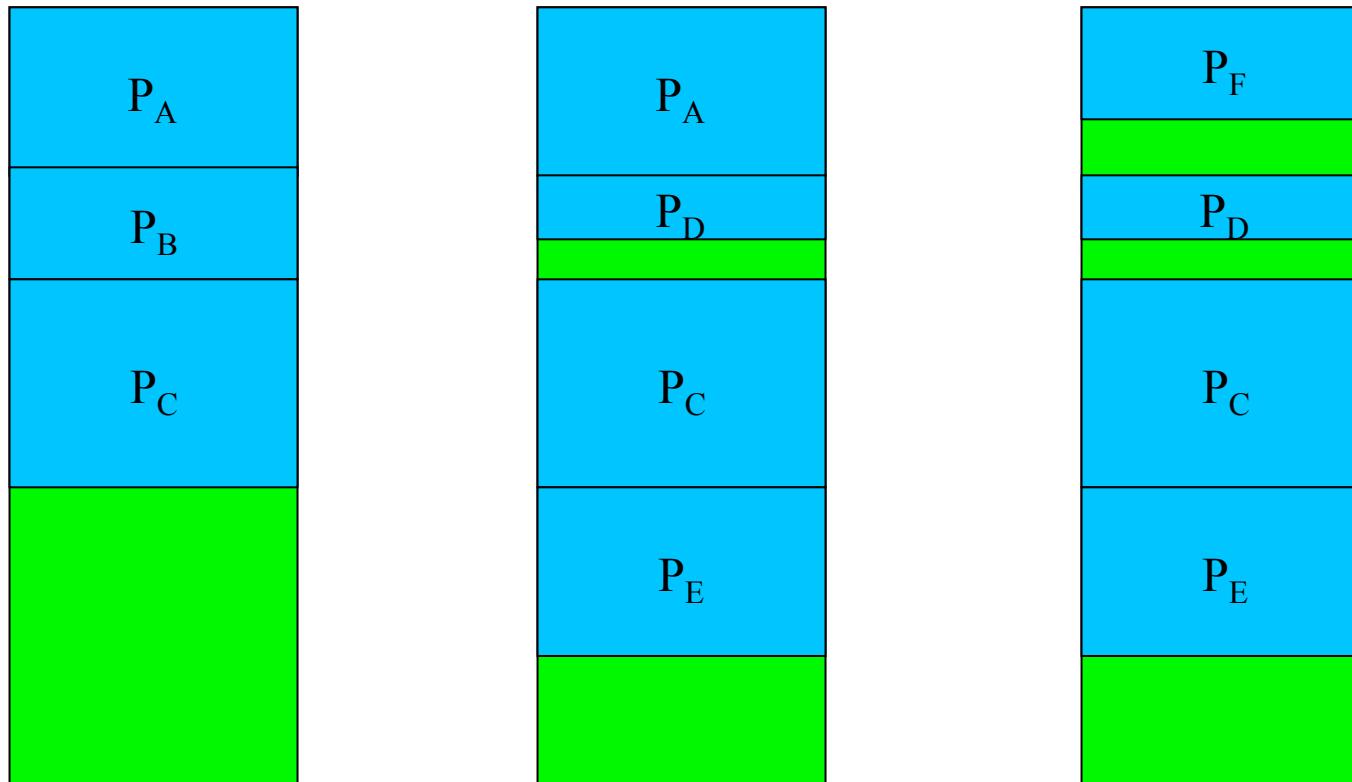
5. Mark the carved piece as in use

...

Variable Partitions and Fragmentation

- Variable sized partitions not as subject to internal fragmentation
 - Unless requestor asked for more than he will use
 - Which is actually pretty common
 - But at least memory manager gave him no more than he requested
- Unlike fixed sized partitions, though, subject to another kind of fragmentation
 - *External fragmentation*

External Fragmentation



We gradually build up small, unusable memory chunks scattered through memory

External Fragmentation: Causes and Effects

- Each allocation creates left-over chunks
 - Over time they become smaller and smaller
- The small left-over fragments are useless
 - They are too small to satisfy any request
 - A second form of fragmentation waste
- Solutions:
 - Try not to create tiny fragments
 - Try to recombine fragments into big chunks

How To Avoid Creating Small Fragments?

- Be smart about which free chunk of memory you use to satisfy a request
- But being smart costs time
- Some choices:
 - Best fit
 - Worst fit
 - First fit
 - Next fit

Best Fit

- Search for the “best fit” chunk
 - Smallest size greater than or equal to requested size
- Advantages:
 - Might find a perfect fit
- Disadvantages:
 - Have to search entire list every time
 - Quickly creates very small fragments

Worst Fit

- Search for the “worst fit” chunk
 - Largest size greater than or equal to requested size
- Advantages:
 - Tends to create very large fragments
... for a while at least
- Disadvantages:
 - Still have to search entire list every time

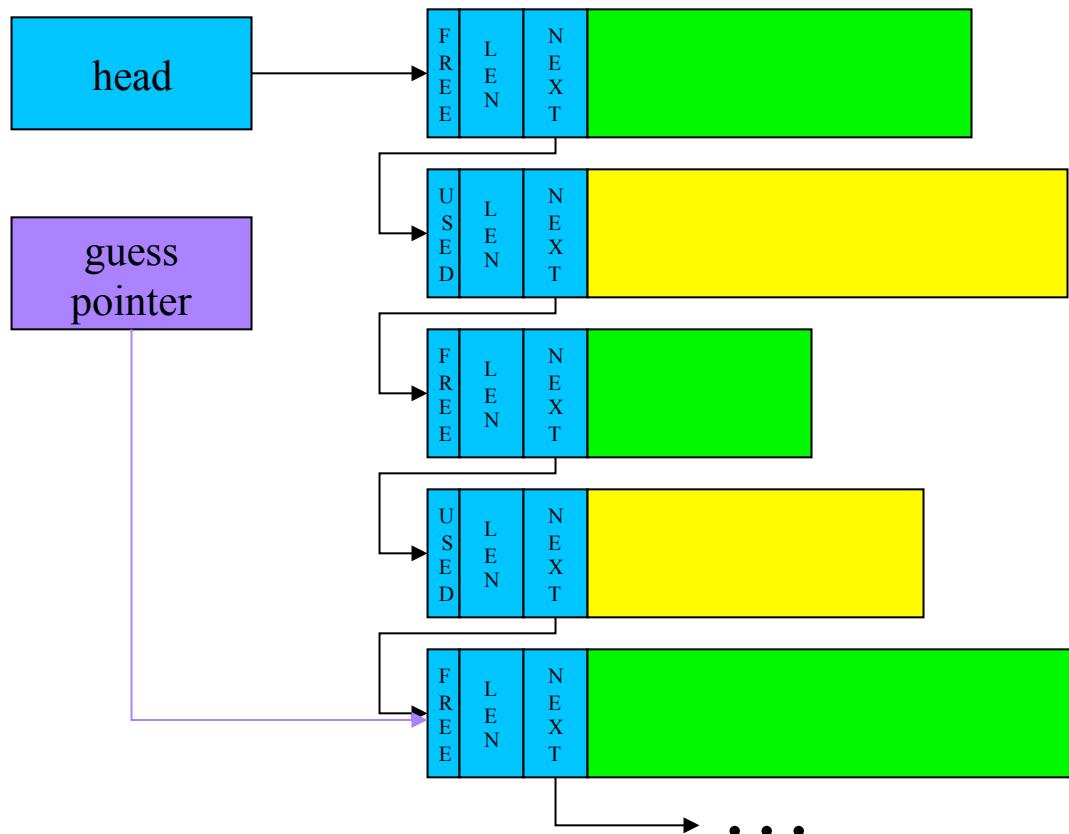
First Fit

- Take first chunk you find that is big enough
- Advantages:
 - Very short searches
 - Creates random sized fragments
- Disadvantages:
 - The first chunks quickly fragment
 - Searches become longer
 - Ultimately it fragments as badly as best fit

Next Fit

After each search, set guess pointer to chunk after the one we chose.

That is the point at which we will begin our next search.



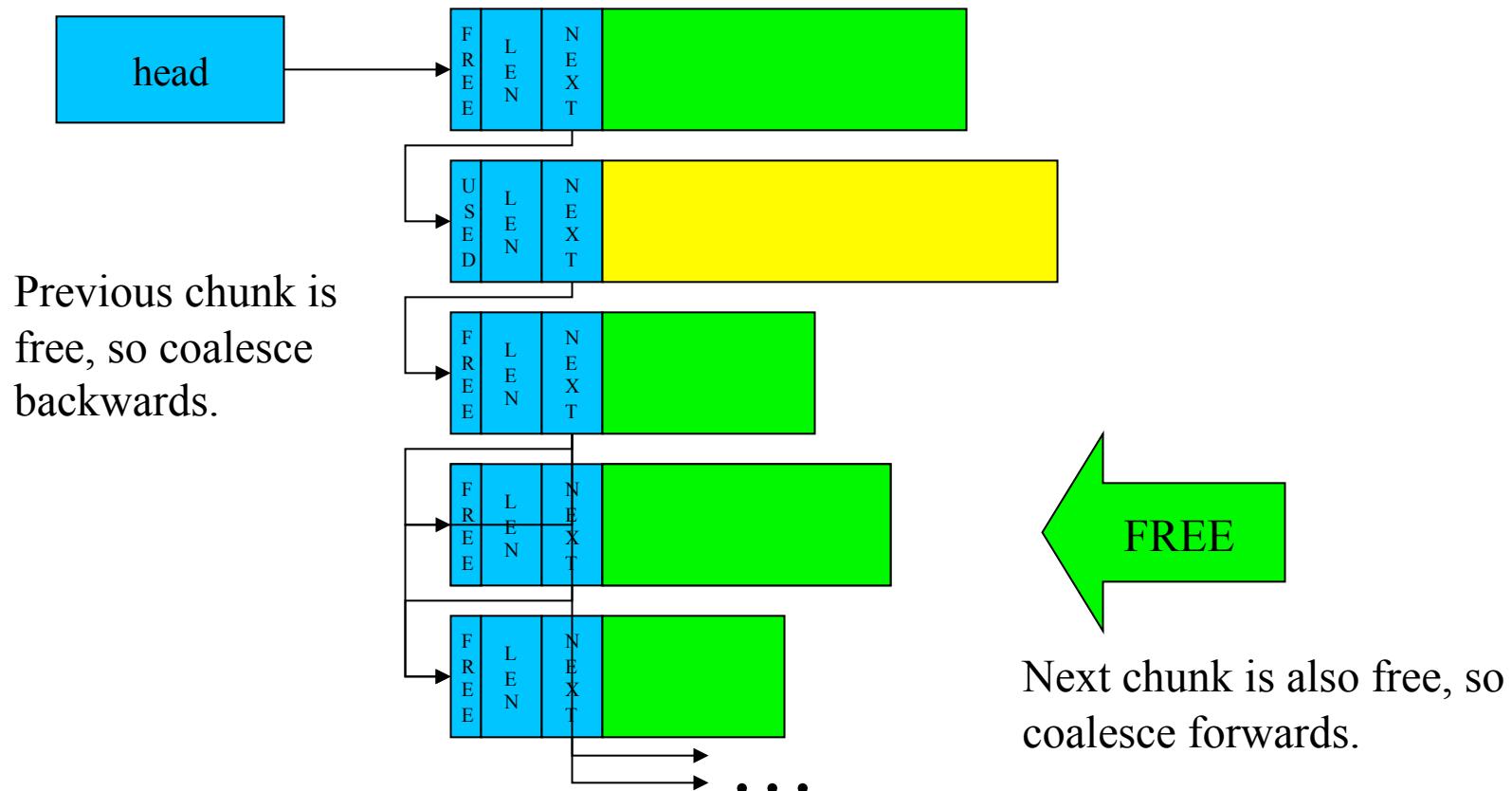
Next Fit Properties

- Tries to get advantages of both first and worst fit
 - Short searches (maybe shorter than first fit)
 - Spreads out fragmentation (like worst fit)
- Guess pointers are a general technique
 - Think of them as a lazy (non-coherent) cache
 - If they are right, they save a lot of time
 - If they are wrong, the algorithm still works
 - They can be used in a wide range of problems

Coalescing Partitions

- All variable sized partition allocation algorithms have external fragmentation
 - Some get it faster, some spread it out
- We need a way to reassemble fragments
 - Check neighbors whenever a chunk is freed
 - Recombine free neighbors whenever possible
 - Free list can be designed to make this easier
 - E.g., where are the neighbors of this chunk?
- Counters forces of external fragmentation

Free Chunk Coalescing



Fragmentation and Coalescing

- Opposing processes that operate in parallel
 - Which of the two processes will dominate?
- What fraction of space is typically allocated?
 - Coalescing works better with more free space
- How fast is allocated memory turned over?
 - Chunks held for long time cannot be coalesced
- How variable are requested chunk sizes?
 - High variability increases fragmentation rate
- How long will the program execute?
 - Fragmentation, like rust, gets worse with time

Coalescing and Free List Implementation

- To coalesce, we must know whether the previous and next chunks are also free
- If the neighbors are guaranteed to be in the free list, we can look at them and see if they are free
- If allocated chunks are not in the free list, we must look at the free chunks before and after us
 - And see if they are our contiguous neighbors
 - This suggests that the free list must be maintained in address order

Variable Sized Partition Summary

- Eliminates internal fragmentation
 - Each chunk is custom-made for requestor
- Implementation is more expensive
 - Long searches of complex free lists
 - Carving and coalescing
- External fragmentation is inevitable
 - Coalescing can counteract the fragmentation
- Must we choose the lesser of two evils?

Another Option

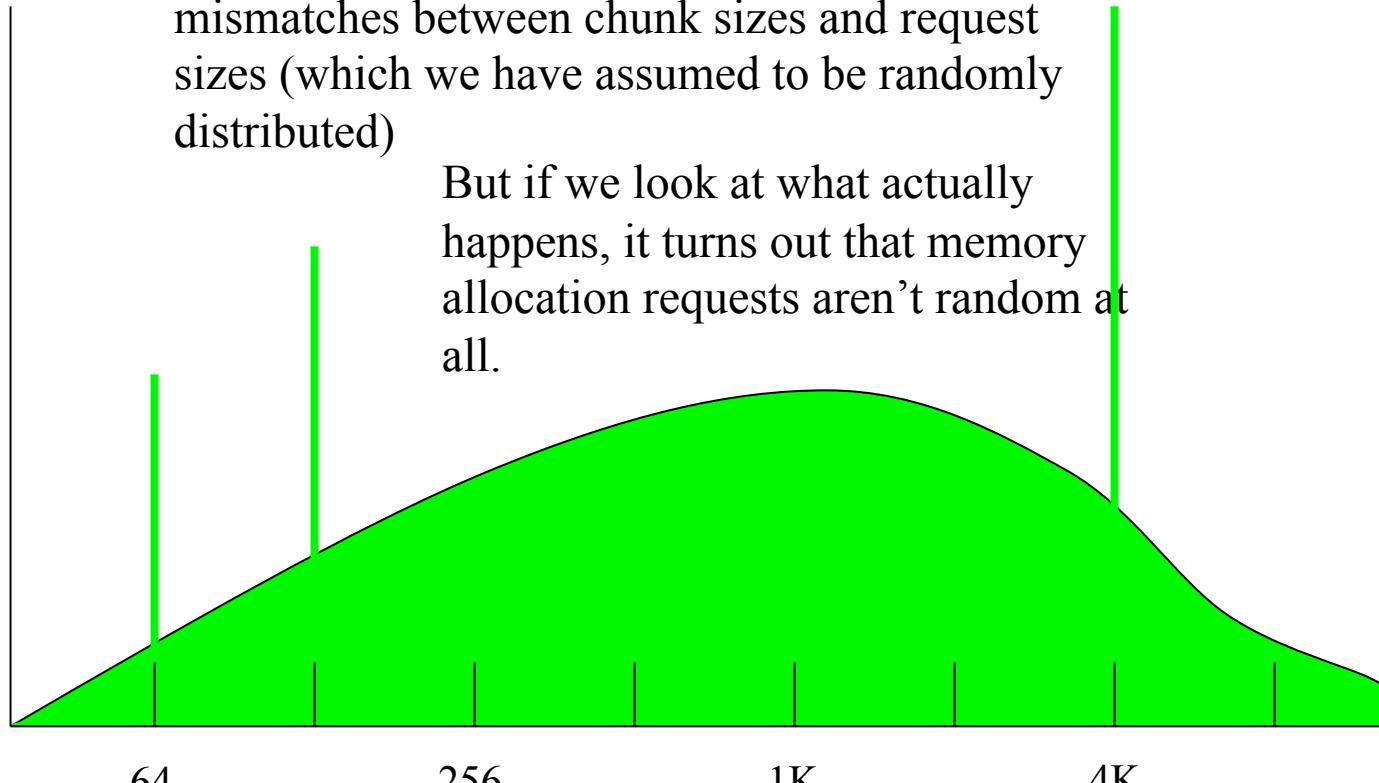
- Fixed partition allocations result in internal fragmentation
 - Processes don't use all of the fixed partition
- Dynamic partition allocations result in external fragmentation
 - The elements on the memory free list get smaller and less useful
- Can we strike a balance in between?

A Special Case for Fixed Allocations

frequency

Internal fragmentation results from mismatches between chunk sizes and request sizes (which we have assumed to be randomly distributed)

But if we look at what actually happens, it turns out that memory allocation requests aren't random at all.



Why Aren't Memory Request Sizes Randomly Distributed?

- In real systems, some sizes are requested much more often than others
- Many key services use fixed-size buffers
 - File systems (for disk I/O)
 - Network protocols (for packet assembly)
 - Standard request descriptors
- These account for much transient use
 - They are continuously allocated and freed
- OS might want to handle them specially

Buffer Pools

- If there are popular sizes,
 - Reserve special pools of fixed size buffers
 - Satisfy matching requests from those pools
- Benefit: improved efficiency
 - Much simpler than variable partition allocation
 - Eliminates searching, carving, coalescing
 - Reduces (or eliminates) external fragmentation
- But we must know how much to reserve
 - Too little, and the buffer pool will become a bottleneck
 - Too much, and we will have a lot of unused buffer space
- Only satisfy perfectly matching requests
 - Otherwise, back to internal fragmentation

How Are Buffer Pools Used?

- Process requests a piece of memory for a special purpose
 - E.g., to send a message
- System supplies one element from buffer pool
- Process uses it, completes, frees memory
 - Maybe explicitly
 - Maybe implicitly, based on how such buffers are used
 - E.g., sending the message will free the buffer “behind the process’ back” once the message is gone

Dynamically Sizing Buffer Pools

- If we run low on fixed sized buffers
 - Get more memory from the free list
 - Carve it up into more fixed sized buffers
- If our free buffer list gets too large
 - Return some buffers to the free list
- If the free list gets dangerously low
 - Ask each major service with a buffer pool to return space
- This can be tuned by a few parameters:
 - Low space (need more) threshold
 - High space (have too much) threshold
 - Nominal allocation (what we free down to)
- Resulting system is highly adaptive to changing loads

Lost Memory

- One problem with buffer pools is memory leaks
 - The process is done with the memory
 - But doesn't free it
- Also a problem when a process manages its own memory space
 - E.g., it allocates a big area and maintains its own free list
- Long running processes with memory leaks can waste huge amounts of memory

Garbage Collection

- One solution to memory leaks
- Don't count on processes to release memory
- Monitor how much free memory we've got
- When we run low, start garbage collection
 - Search data space finding every object pointer
 - Note address/size of all accessible objects
 - Compute the compliment (what is inaccessible)
 - Add all inaccessible memory to the free list

How Do We Find All Accessible Memory?

- Object oriented languages often enable this
 - All object references are tagged
 - All object descriptors include size information
- It is often possible for system resources
 - Where all possible references are known
 - (E.g., we know who has which files open)
- How about for the general case?

General Garbage Collection

- Well, what would you need to do?
- Find all the pointers in allocated memory
- Determine “how much” each points to
- Determine what is and is not still pointed to
- Free what isn’t pointed to
- Why might that be difficult?

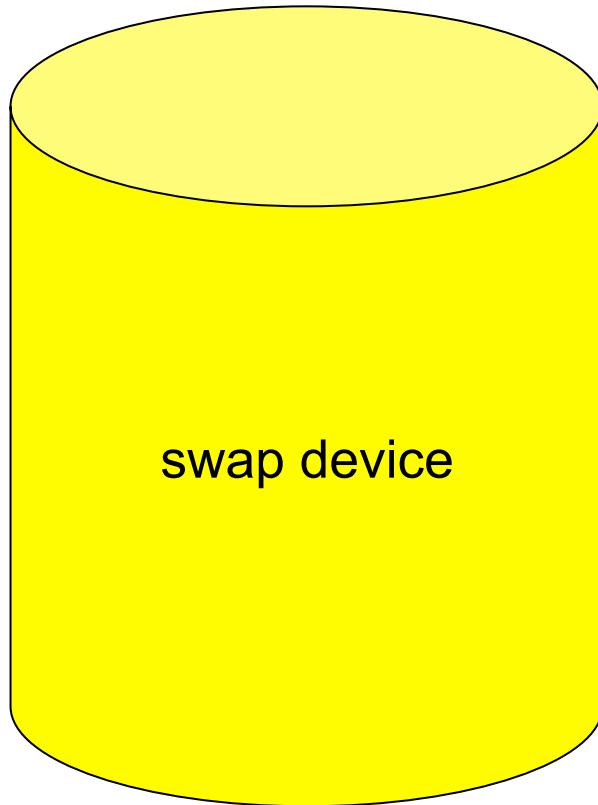
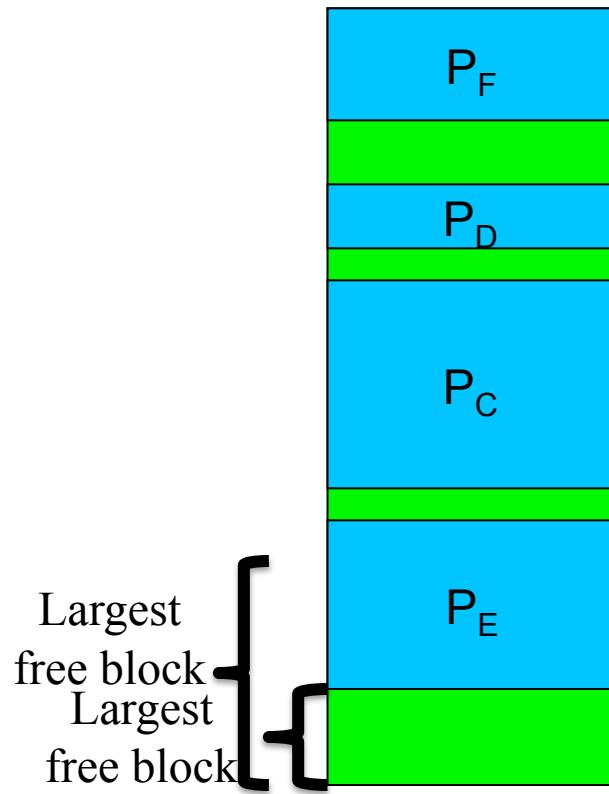
Problems With General Garbage Collection

- A location in the data or stack segments might seem to contain addresses, but ...
 - Are they truly pointers, or might they be other data types whose values happen to resemble addresses?
 - If pointers, are they themselves still accessible?
 - We might be able to infer this (recursively) for pointers in dynamically allocated structures ...
 - But what about pointers in statically allocated (potentially global) areas?
- And how much is “pointed to,” one word or a million?

Compaction and Relocation

- Garbage collection is just another way to free memory
 - Doesn't greatly help or hurt fragmentation
- Ongoing activity can starve coalescing
 - Chunks reallocated before neighbors become free
- We could stop accepting new allocations
 - But resulting convoy on memory manager would trash throughput
- We need a way to rearrange active memory
 - Re-pack all processes in one end of memory
 - Create one big chunk of free space at other end

Memory Compaction



*Now let's
compact!*

*An obvious
improvement!*

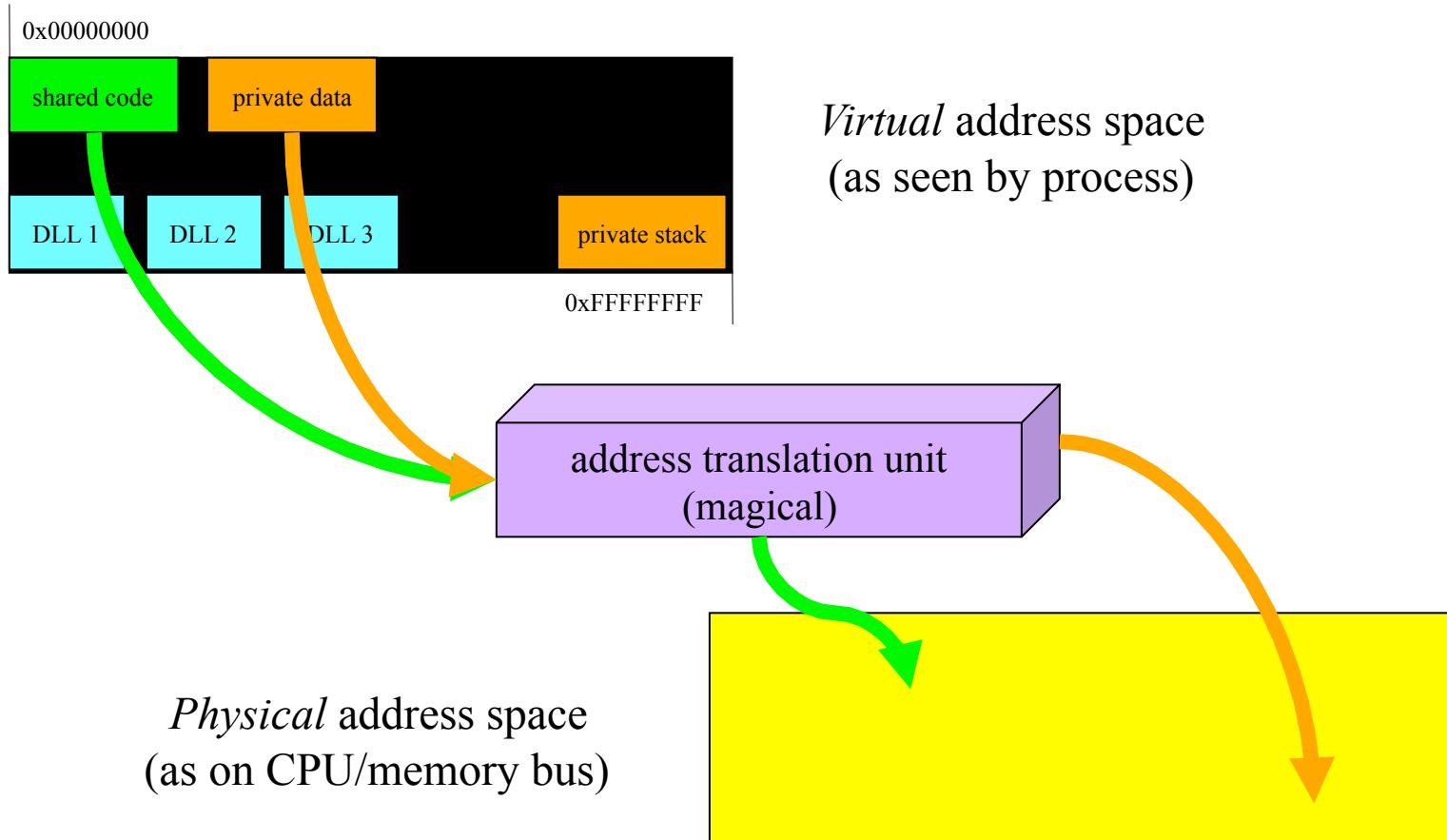
All This Requires Is Relocation . . .

- The ability to move a process
 - From region where it was initially loaded
 - Into a new and different region of memory
- What's so hard about that?
- All addresses in the program will be wrong
 - References in the code segment
 - Calls and branches to other parts of the code
 - References to variables in the data segment
 - Plus new pointers created during execution
 - That point into data and stack segments

The Relocation Problem

- It is not generally feasible to re-relocate a process
 - Maybe we could relocate references to code
 - If we kept the relocation information around
 - But how can we relocate references to data?
 - Pointer values may have been changed
 - New pointers may have been created
- We could never find/fix all address references
 - Like the general case of garbage collection
- Can we make processes location independent?

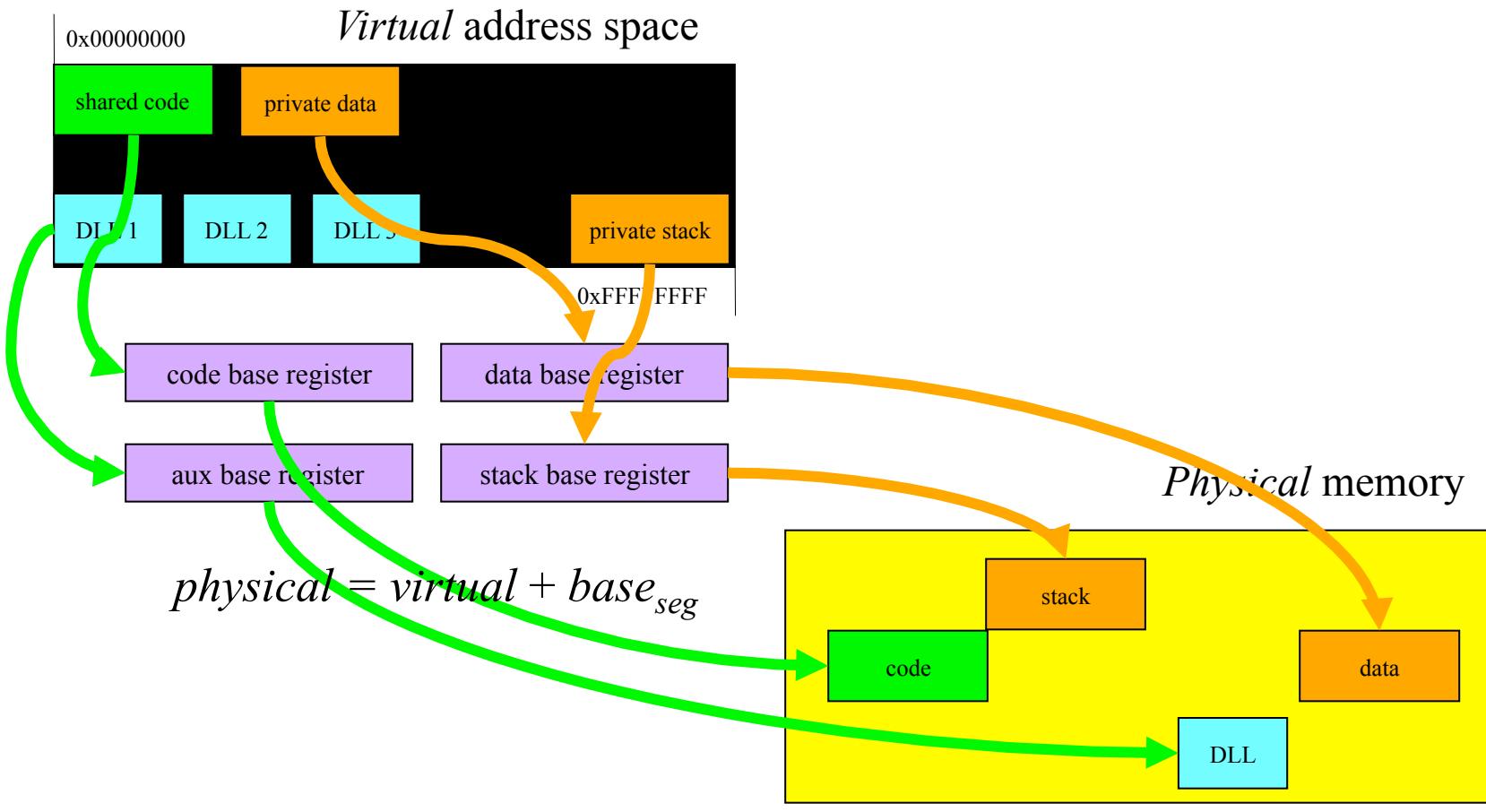
Virtual Address Spaces



Memory Segment Relocation

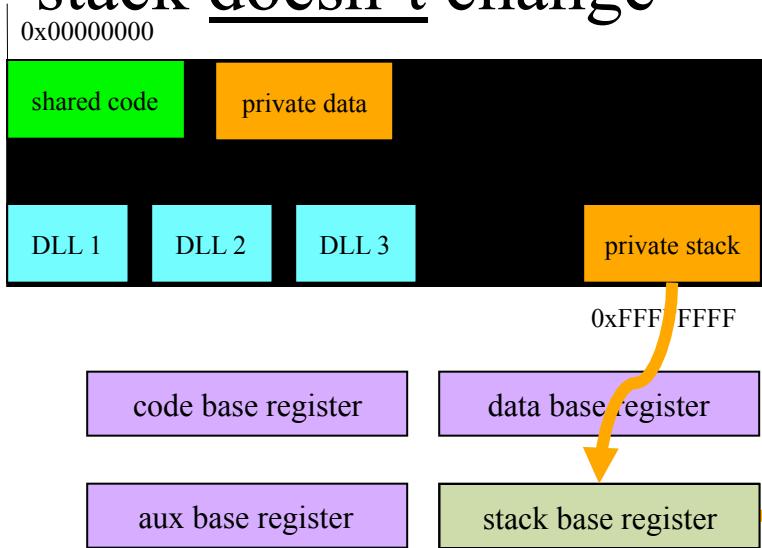
- A natural model
 - Process address space is made up of multiple segments
 - Use the segment as the unit of relocation
 - Long tradition, from the IBM system 360 to Intel x86 architecture
- Computer has special relocation registers
 - They are called segment base registers
 - They point to the start (in physical memory) of each segment
 - CPU automatically adds base register to every address
- OS uses these to perform virtual address translation
 - Set base register to start of region where program is loaded
 - If program is moved, reset base registers to new location
 - Program works no matter where its segments are loaded

How Does Segment Relocation Work?



Relocating a Segment

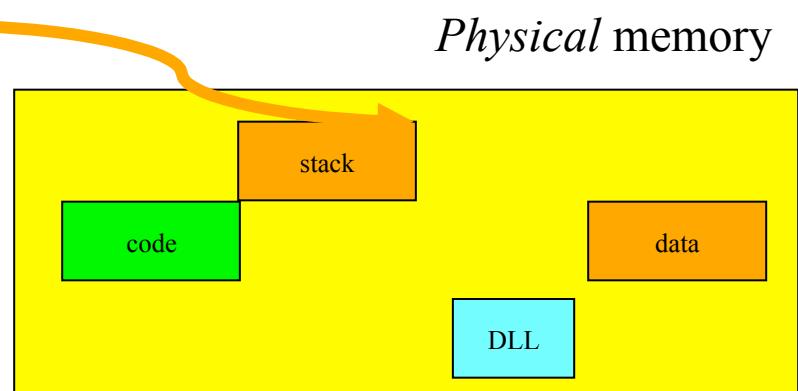
The virtual address of the stack doesn't change



$$\text{physical} = \text{virtual} + \text{base}_{\text{seg}}$$

We just change the value in the stack base register

Let's say we need to move the stack in physical memory



Relocation and Safety

- A relocation mechanism (like base registers) is good
 - It solves the relocation problem
 - Enables us to move process segments in physical memory
 - Such relocation turns out to be insufficient
- We also need protection
 - Prevent process from reaching outside its allocated memory
 - E.g., by overrunning the end of a mapped segment
- Segments also need a length (or limit) register
 - Specifies maximum legal offset (from start of segment)
 - Any address greater than this is illegal (in the hole)
 - CPU should report it via a segmentation exception (trap)

How Much of Our Problem Does Relocation Solve?

- We can use variable sized partitions
 - Cutting down on internal fragmentation
- We can move partitions around
 - Which helps coalescing be more effective
 - But still requires contiguous chunks of data for segments
 - So external fragmentation is still a problem
- We need to get rid of the requirement of contiguous segments

Operating System Principles: Memory Management – Swapping, Paging, and Virtual Memory

CS 111

Operating Systems
Peter Reiher

Outline

- Swapping
- Paging
- Virtual memory

Swapping

- What if we don't have enough RAM?
 - To handle all processes' memory needs
 - Perhaps even to handle one process
- Maybe we can keep some of their memory somewhere other than RAM
- Where?
- Maybe on a disk
- Of course, you can't directly use code or data on a disk . . .

Swapping To Disk

- An obvious strategy to increase effective memory size
- When a process yields, copy its memory to disk
- When it is scheduled, copy it back
- If we have relocation hardware, we can put the memory in different RAM locations
- Each process could see a memory space as big as the total amount of RAM

Downsides To Simple Swapping

- If we actually move everything out, the costs of a context switch are very high
 - Copy all of RAM out to disk
 - And then copy other stuff from disk to RAM
 - Before the newly scheduled process can do anything
- We're still limiting processes to the amount of RAM we actually have

Paging

- What is paging?
 - What problem does it solve?
 - How does it do so?
- Paged address translation
- Paging and fragmentation
- Paging memory management units

Segmentation Revisited

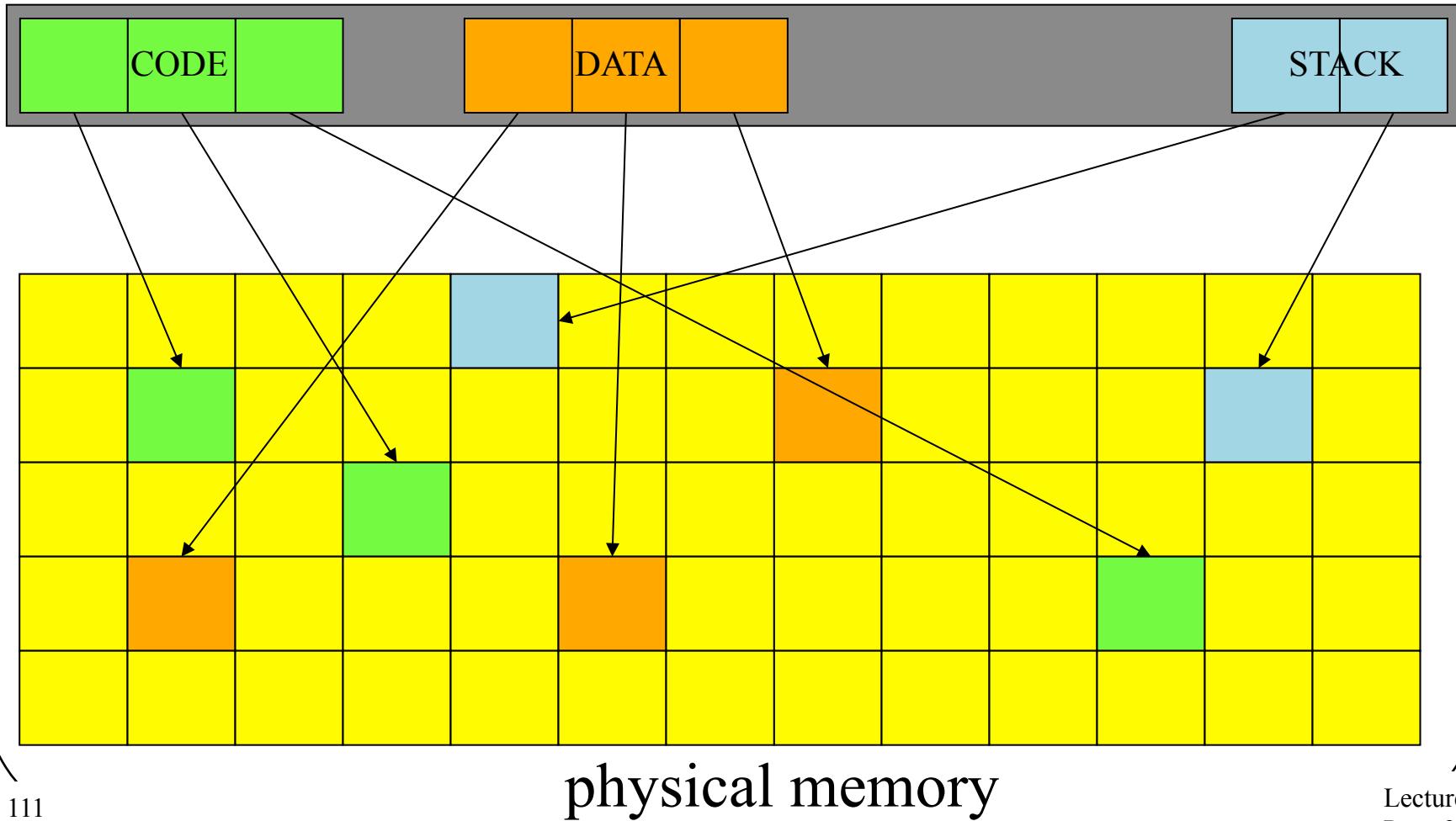
- Segment relocation solved the relocation problem for us
- It used base registers to compute a physical address from a virtual address
 - Allowing us to move data around in physical memory
 - By only updating the base register
- It did nothing about external fragmentation
 - Because segments are still required to be contiguous
- We need to eliminate the “contiguity requirement”

The Paging Approach

- Divide physical memory into units of a single fixed size
 - A pretty small one, like 1-4K bytes or words
 - Typically called a *page frame*
- Treat the virtual address space in the same way
- For each virtual address space page, store its data in one physical address page frame
- Use some magic per-page translation mechanism to convert virtual to physical pages

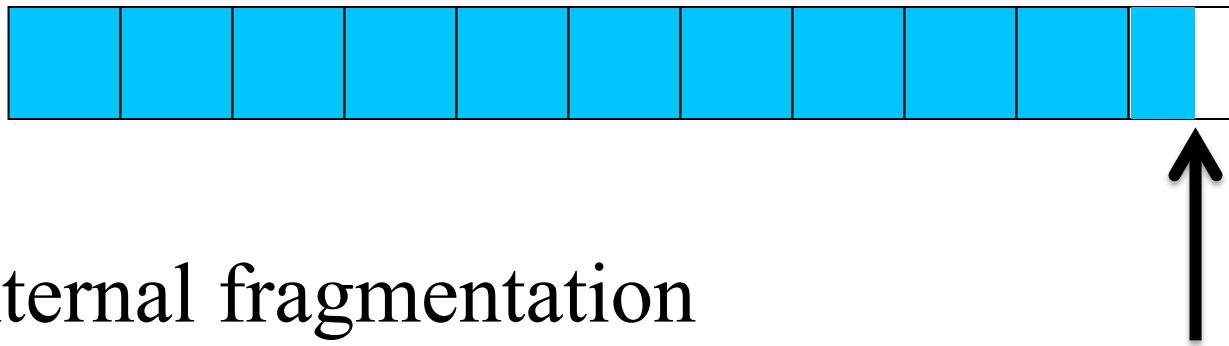
Paged Address Translation

process virtual address space



Paging and Fragmentation

- A segment is implemented as a set of virtual pages



- Internal fragmentation
 - Averages only $\frac{1}{2}$ page (half of the last one)
- External fragmentation
 - Completely non-existent
 - We never carve up pages

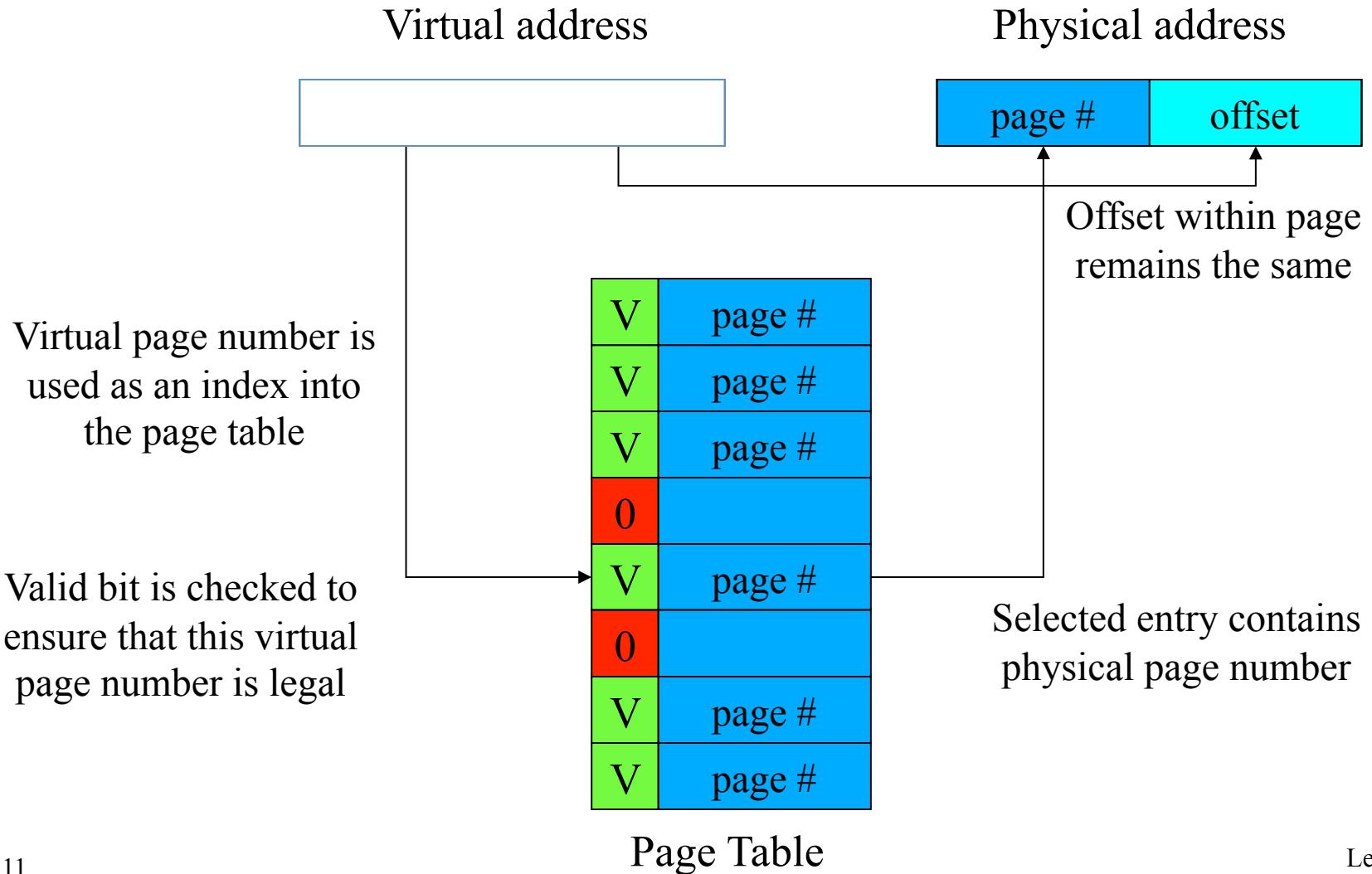
How Does This Compare To Segment Fragmentation?

- Consider this scenario:
 - Average requested allocation is 128K
 - 256K fixed size segments available
 - In the paging system, 4K pages
- For segmentation, average internal fragmentation is 50% (128K of 256K used)
- For paging?
 - Only the last page of an allocation is not full
 - On average, half of it is unused, or 2K
 - So 2K of 128K is wasted, or around 1.5%
- **Segmentation: 50% waste**
- **Paging: 1.5% waste**

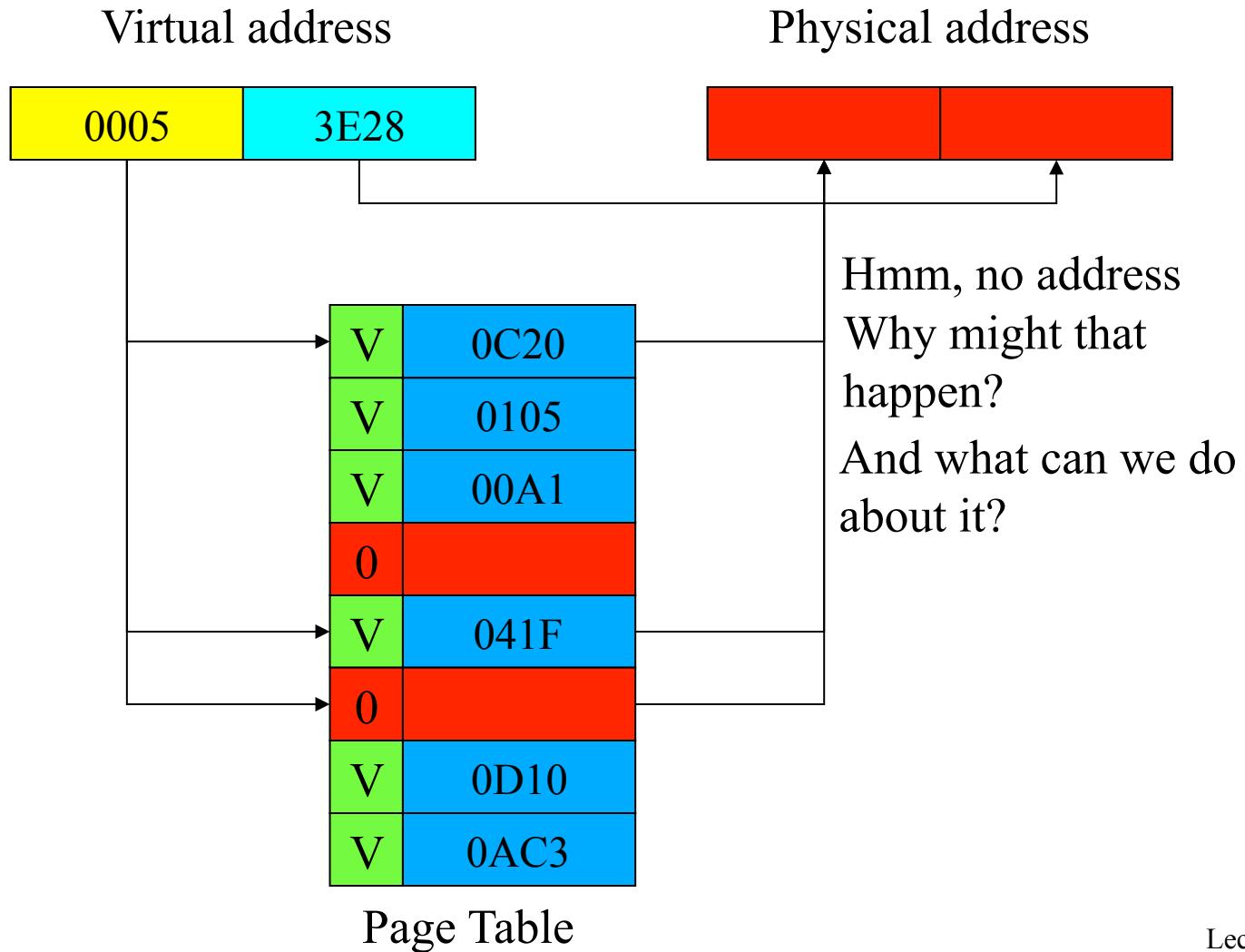
Providing the Magic Translation Mechanism

- On per page basis, we need to change a virtual address to a physical address
- Needs to be fast
 - So we'll use hardware
- The Memory Management Unit (MMU)
 - A piece of hardware designed to perform the magic quickly

Paging and MMUs



Some Examples



The MMU Hardware

- MMUs used to sit between the CPU and bus
 - Now they are typically integrated into the CPU
- What about the page tables?
 - Originally implemented in special fast registers
 - But there's a problem with that today
 - If we have 4K pages, and a 64 Gbyte memory, how many pages are there?
 - $2^{36}/2^{12} = 2^{24}$
 - Or 16 M of pages
 - We can't afford 16 M of fast registers

Handling Big Page Tables

- 16 M entries in a page table means we can't use registers
- So now they are stored in normal memory
- But we can't afford 2 bus cycles for each memory access
 - One to look up the page table entry
 - One to get the actual data
- So we have a very fast set of MMU registers used as a cache
 - Which means we need to worry about hit ratios, cache invalidation, and other nasty issues
 - TANSTAAFL

The MMU and Multiple Processes

- There are several processes running
- Each needs a set of pages
- We can put any page anywhere
- But if they need, in total, more pages than we've physically got,
- Something's got to go
- How do we handle these ongoing paging requirements?

Ongoing MMU Operations

- What if the current process adds or removes pages?
 - Directly update active page table in memory
 - Privileged instruction to flush (stale) cached entries
- What if we switch from one process to another?
 - Maintain separate page tables for each process
 - Privileged instruction loads pointer to new page table
 - A reload instruction flushes previously cached entries
- How to share pages between multiple processes?
 - Make each page table point to same physical page
 - Can be read-only or read/write sharing

Demand Paging

- What is paging?
 - What problem does it solve?
 - How does it do so?
- Locality of reference
- Page faults and performance issues

What Is Demand Paging?

- A process doesn't actually need all its pages in memory to run
- It only needs those it actually references
- So, why bother loading up all the pages when a process is scheduled to run?
- And, perhaps, why get rid of all of a process' pages when it yields?
- Move pages onto and off of disk “on demand”

How To Make Demand Paging Work

- The MMU must support “not present” pages
 - Generates a fault/trap when they are referenced
 - OS can bring in page and retry the faulted reference
- Entire process needn’t be in memory to start running
 - Start each process with a subset of its pages
 - Load additional pages as program demands them
- The big challenge will be performance

Achieving Good Performance for Demand Paging

- Demand paging will perform poorly if most memory references require disk access
 - Worse than bringing in all the pages at once, maybe
- So we need to be sure most don't
- How?
- By ensuring that the page holding the next memory reference is already there
 - Almost always

Demand Paging and Locality of Reference

- How can we predict what pages we need in memory?
 - Since they'd better be there when we ask
- Primarily, rely on *locality of reference*
 - Put simply, the next address you ask for is likely to be close to the last address you asked for
- Do programs typically display locality of reference?
- Fortunately, yes!

Why is Locality of Reference Usually Present?

- Code usually executes sequences of consecutive or nearby instructions
 - Most branches tend to be relatively short distances (into code in the same routine)
- We typically need access to things in the current or previous stack frame
- Many heap references to recently allocated structures
 - E.g., creating or processing a message
- No guarantees, but all three types of memory are likely to show locality of reference

Page Faults

- Page tables no longer necessarily contain points to pages of RAM
- In some cases, the pages are not in RAM, at the moment
 - They're out on disk
- When a program requests an address from such a page, what do we do?
- Generate a *page fault*
 - Which is intended to tell the system to go get it

Handling a Page Fault

- Initialize page table entries to “not present”
- CPU faults if “not present” page is referenced
 - Fault enters kernel, just like any other trap
 - Forwarded to page fault handler
 - Determine which page is required, where it resides
 - Schedule I/O to fetch it, then block the process
 - Make page table point at newly read-in page
 - Back up user-mode PC to retry failed instruction
 - Return to user-mode and try again
- Meanwhile, other processes can run

Page Faults Don't Impact Correctness

- Page faults only slow a process down
- After a fault is handled, the desired page is in RAM
- And the process runs again and can use it
 - Based on the OS ability to save process state and restore it
- Programs never crash because of page faults
- But they might be very slow if there are too many

Pages and Secondary Storage

- When not in memory, pages live on secondary storage
 - Typically a disk
 - In an area called “swap space”
- How do we manage swap space?
 - As a pool of variable length partitions?
 - Allocate a contiguous region for each process
 - As a random collection of pages?
 - Just use a bit-map to keep track of which are free
 - As a file system?
 - Create a file per process (or segment)
 - File offsets correspond to virtual address offsets

Demand Paging Performance

- Page faults may block processes
- Overhead (fault handling, paging in and out)
 - Process is blocked while we are reading in pages
 - Delaying execution and consuming cycles
 - Directly proportional to the number of page faults
- Key is having the “right” pages in memory
 - Right pages -> few faults, little paging activity
 - Wrong pages -> many faults, much paging
- We can't control what pages we read in
 - Key to performance is choosing which to kick out

Virtual Memory

- A generalization of what demand paging allows
- A form of memory where the system provides a useful abstraction
 - A very large quantity of memory
 - For each process
 - All directly accessible via normal addressing
 - At a speed approaching that of actual RAM
- The state of the art in modern memory abstractions

The Basic Concept

- Give each process an address space of immense size
 - Perhaps as big as your hardware's word size allows
- Allow processes to request segments within that space
- Use dynamic paging and swapping to support the abstraction
- The key issue is how to create the abstraction when you don't have that much real memory

The Key VM Technology: Replacement Algorithms

- The goal is to have each page already in memory when a process accesses it
- We can't know ahead of time what pages will be accessed
- We rely on locality of access
 - In particular, to determine what pages to move out of memory and onto disk
- If we make wise choices, the pages we need in memory will still be there

The Basics of Page Replacement

- We keep some set of all possible pages in memory
 - Perhaps not all belonging to the current process
- Under some circumstances, we need to replace one of them with another page that's on disk
 - E.g., when we have a page fault
- Paging hardware and MMU translation allows us to choose any page for ejection to disk
- Which one of them should go?

The Optimal Replacement Algorithm

- Replace the page that will be next referenced furthest in the future
- Why is this the right page?
 - It delays the next page fault as long as possible
 - Fewer page faults per unit time = lower overhead
- A slight problem:
 - We would need an oracle to know which page this algorithm calls for
 - And we don't have one

Do We Require Optimal Algorithms?

- Not absolutely
- What's the consequence being wrong?
 - We take an extra page fault that we shouldn't have
 - Which is a performance penalty, not a program correctness penalty
 - Often an acceptable tradeoff
- The more often we're right, the fewer page faults we take
- For traces, we can run the optimal algorithm, comparing it to what we use when live

Approximating the Optimal

- Rely on locality of reference
- Note which pages have recently been used
 - Perhaps with extra bits in the page tables
 - Updated when the page is accessed
- Use this data to predict future behavior
- If locality of reference holds, the pages we accessed recently will be accessed again soon

Candidate Replacement Algorithms

- Random, FIFO
 - These are dogs, forget ‘em
- Least Frequently Used
 - Sounds better, but it really isn’t
- Least Recently Used
 - Assert that near future will be like the recent past
 - If we haven’t used a page recently, we probably won’t use it soon
 - The computer science equivalent to the “*unseen hand*”

Naïve LRU

- Each time a page is accessed, record the time
- When you need to eject a page, look at all timestamps for pages in memory
- Choose the one with the oldest timestamp
- Will require us to store timestamps somewhere
- And to search all timestamps every time we need to eject a page

True LRU Page Replacement

Reference stream

a	b	c	d	a	b	d	e	f	a	b	c	d	a	e	d
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Page table using true LRU

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
frame 0	a				!				f				d			!
frame 1		b				!				a				!		
frame 2			c					e				c				
frame 3				d			!			b				e		

**Loads 4
Replacements 7**

Maintaining Information for LRU

- Can we keep it in the MMU?
 - MMU notes the time whenever a page is referenced
 - MMU translation must be blindingly fast
 - Getting/storing time on every fetch would be very expensive
 - At best they will maintain a *read* and a *written* bit per page
- Can we maintain this information in software?
 - Mark all pages invalid, even if they are in memory
 - Take a fault first time each page is referenced, note the time
 - Then mark this page valid for the rest of the time slice
 - Causing page faults to reduce the number of page faults???
- We need a cheap software surrogate for LRU
 - No extra page faults
 - Can't scan entire list each time, since it's big

Clock Algorithms

- A surrogate for LRU
- Organize all pages in a circular list
- MMU sets a reference bit for the page on access
- Scan whenever we need another page
 - For each page, ask MMU if page has been referenced
 - If so, reset the reference bit in the MMU & skip this page
 - If not, consider this page to be the least recently used
 - Next search starts from this position, not head of list
- Use position in the scan as a surrogate for age
- No extra page faults, usually scan only a few pages

Clock Algorithm Page Replacement

Reference Stream

	a	b	c	d	a	b	d	e	f	a	b	c	d	a	e	d
LRU clock	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
frame 0	a				!	!	!	!	f				d			!
frame 1		b				!	!	!		a				!	!	
frame 2			c					e			b			e		
frame 3				d				!	!	!		c				
clock pos	0	1	2	3	0	0	0	0	0	1	2	3	0	1	2	3

loads 4, replacements 7

True LRU

frame 0	a				a											d
frame 1		b														a
frame 2			c						e							
frame 3				d					d			b				e
CS 111 Fall 2016																Lecture 6 Page 42

Loads 4
Replacements 7

Comparing True LRU To Clock Algorithm

- Same number of loads and replacements
 - But didn't replace the same pages
- What, if anything, does that mean?
- Both are just approximations to the optimal
- If LRU clock's decisions are 98% as good as true LRU
 - And can be done for 1% of the cost (in hardware and cycles)
 - It's a bargain!

Page Replacement and Multiprogramming

- We don't want to clear out all the page frames on each context switch
- How do we deal with sharing page frames?
- Possible choices:
 - Single global pool
 - Fixed allocation of page frames per process
 - Working set-based page frame allocations

Single Global Page Frame Pool

- Treat the entire set of page frames as a shared resource
- Approximate LRU for the entire set
- Replace whichever process' page is LRU
- Probably a mistake
 - Bad interaction with round-robin scheduling
 - The guy who was last in the scheduling queue will find all his pages swapped out
 - And not because he isn't using them
 - When he gets in, lots of page faults

Per-Process Page Frame Pools

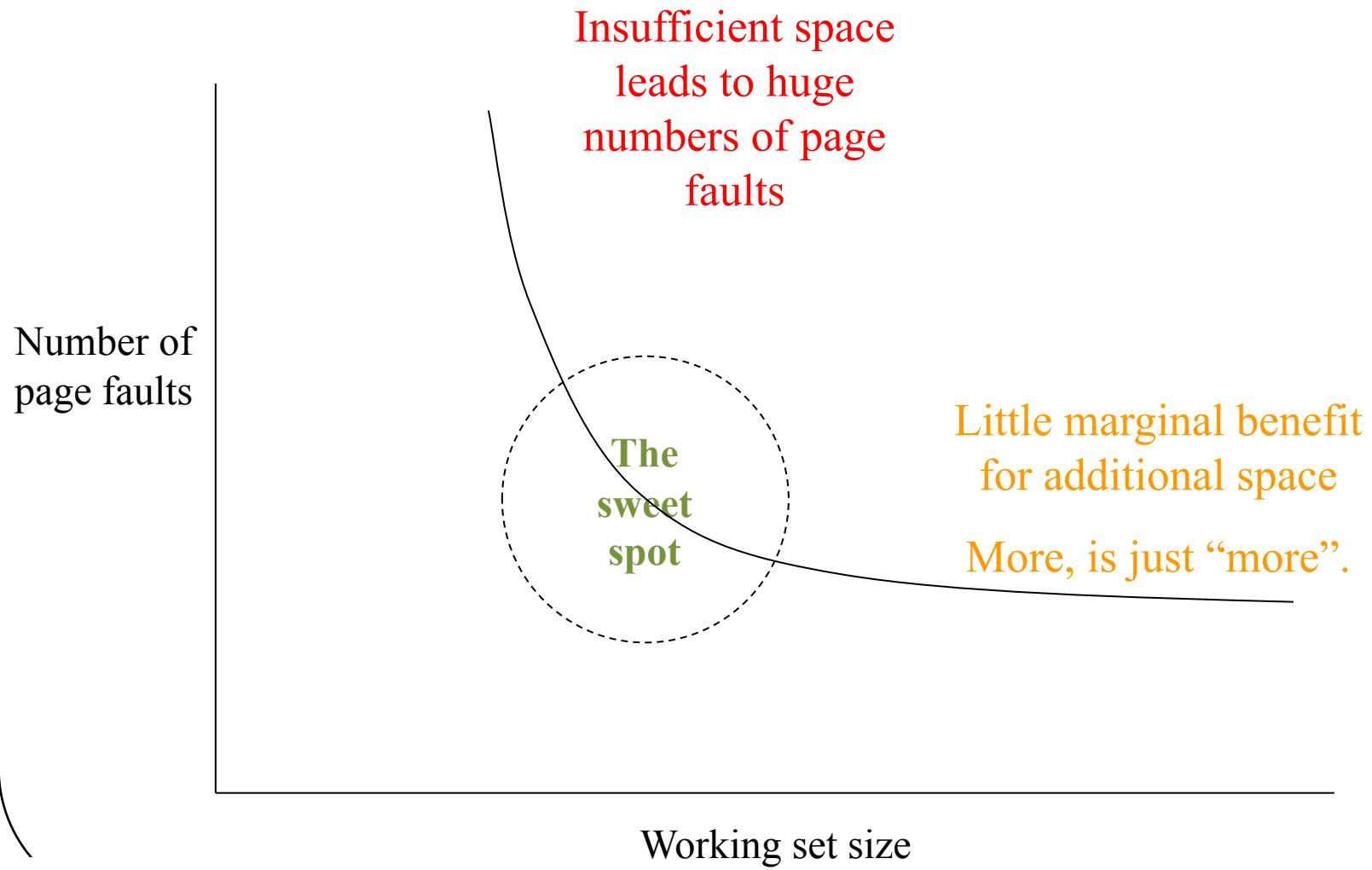
- Set aside some number of page frames for each running process
 - Use an LRU approximation separately for each
- How many page frames per process?
- Fixed number of pages per process is bad
 - Different processes exhibit different locality
 - Which pages are needed changes over time
 - Number of pages needed changes over time
 - Much like different natural scheduling intervals
- We need a dynamic customized allocation

Working Sets

- Give each running process an allocation of page frames matched to its needs
- How do we know what its needs are?
- Use *working sets*
- Set of pages used by a process in a fixed length sampling window in the immediate past¹
- Allocate enough page frames to hold each process' working set
- Each process runs replacement within its own set

¹This definition paraphrased from Peter Denning's definition

The Natural Working Set Size



Optimal Working Sets

- What is optimal working set for a process?
 - Number of pages needed during next time slice
- What if try to run the process in fewer pages?
 - Needed pages will replace one another continuously
 - Process will run very slowly
- How can we know what working set size is?
 - By observing the process' behavior
- Which pages should be in the working-set?
 - No need to guess, the process will fault for them

Implementing Working Sets

- Manage the working set size
 - Assign page frames to each in-memory process
 - Processes page against themselves in working set
 - Observe paging behavior (faults per unit time)
 - Adjust number of assigned page frames accordingly
- Page stealing algorithms
 - E.g., Working Set-Clock
 - Track last use time for each page, for owning process
 - Find page least recently used (by its owner)
 - Processes that need more pages tend to get more
 - Processes that don't use their pages tend to lose them

Thrashing

- Working set size characterizes each process
 - How many pages it needs to run for τ milliseconds
- What if we don't have enough memory?
 - Sum of working sets exceeds available memory
 - No one will have enough pages in memory
 - Whenever anything runs, it will grab a page from someone else
 - So they'll get a page fault soon after they start running
- This behavior is called *thrashing*
- When systems thrash, all processes run slow
- Generally continues till system takes action

Preventing Thrashing

- We cannot squeeze working set sizes
 - This will also cause thrashing
- We can reduce number of competing processes
 - Swap some of the ready processes out
 - To ensure enough memory for the rest to run
- Swapped-out processes won't run for quite a while
- But we can round-robin which are in and which are out

Unswapping a Process

- What happens when a swapped process comes in from disk?
- Pure swapping?
 - Bring in all pages before process is run, no page faults
- Pure demand paging?
 - Pages are only brought in as needed
 - Fewer pages per process, more processes in memory
- What if we pre-loaded the last working set?
 - Far fewer pages to be read in than swapping
 - *Probably* the same disk reads as pure demand paging
 - Far fewer initial page faults than pure demand paging

Clean Vs. Dirty Pages

- Consider a page, recently paged in from disk
 - There are two copies, one on disk, one in memory
- If the in-memory copy has not been modified, there is still a valid copy on disk
 - The in-memory copy is said to be “clean”
 - Clean pages can be replaced without writing them back to disk
- If the in-memory copy has been modified, the copy on disk is no longer up-to-date
 - The in-memory copy is said to be “dirty”
 - If swapped out of memory, must be written to disk

Dirty Pages and Page Replacement

- Clean pages can be replaced at any time
 - The copy on disk is already up to date
- Dirty pages must be written to disk before the frame can be reused
 - A slow operation we don't want to wait for
- Could only kick out clean pages
 - But that would limit flexibility
- How to avoid being hamstrung by too many dirty page frames in memory?

Pre-Emptive Page Laundering

- Clean pages give memory scheduler flexibility
 - Many pages that can, if necessary, be replaced
- We can increase flexibility by converting dirty pages to clean ones
- Ongoing background write-out of dirty pages
 - Find and write out all dirty, non-running pages
 - No point in writing out a page that is actively in use
 - On assumption we will eventually have to page out
 - Make them clean again, available for replacement
- An outgoing equivalent of pre-loading

Paging and Shared Segments

- Some memory segments will be shared
 - Shared memory, executables, DLLs
- Created/managed as mappable segments
 - One copy mapped into multiple processes
 - Demand paging same as with any other pages
 - Secondary home may be in a file system
- Shared pages don't fit working set model
 - May not be associated with just one process
 - Global LRU may be more appropriate
 - Shared pages often need/get special handling

Operating System Principles: Threads, IPC, and Synchronization

CS 111

Operating Systems
Peter Reiher

Outline

- Threads
- Interprocess communications
- Synchronization
 - Critical sections
 - Asynchronous event completions

Threads

- Why not just processes?
- What is a thread?
- How does the operating system deal with threads?

Why Not Just Processes?

- Processes are very expensive
 - To create: they own resources
 - To dispatch: they have address spaces
- Different processes are very distinct
 - They cannot share the same address space
 - They cannot (usually) share resources
- Not all programs require strong separation
 - Multiple activities working cooperatively for a single goal
 - Mutually trusting elements of a system

What Is a Thread?

- Strictly a unit of execution/scheduling
 - Each thread has its own stack, PC, registers
 - But other resources are shared with other threads
- Multiple threads can run in a process
 - They all share the same code and data space
 - They all have access to the same resources
 - This makes it cheaper to create and run
- Sharing the CPU between multiple threads
 - User level threads (with voluntary yielding)
 - Scheduled system threads (with preemption)

When Should You Use Processes?

- To run multiple distinct programs
- When creation/destruction are rare events
- When running agents with distinct privileges
- When there are limited interactions and shared resources
- To prevent interference between executing interpreters
- To firewall one from failures of the other

When Should You Use Threads?

- For parallel activities in a single program
- When there is frequent creation and destruction
- When all can run with same privileges
- When they need to share resources
- When they exchange many messages/signals
- When you don't need to protect them from each other

Processes vs. Threads – Trade-offs

- If you use multiple processes
 - Your application may run much more slowly
 - It may be difficult to share some resources
- If you use multiple threads
 - You will have to create and manage them
 - You will have serialize resource use
 - Your program will be more complex to write
- TANSTAAFL

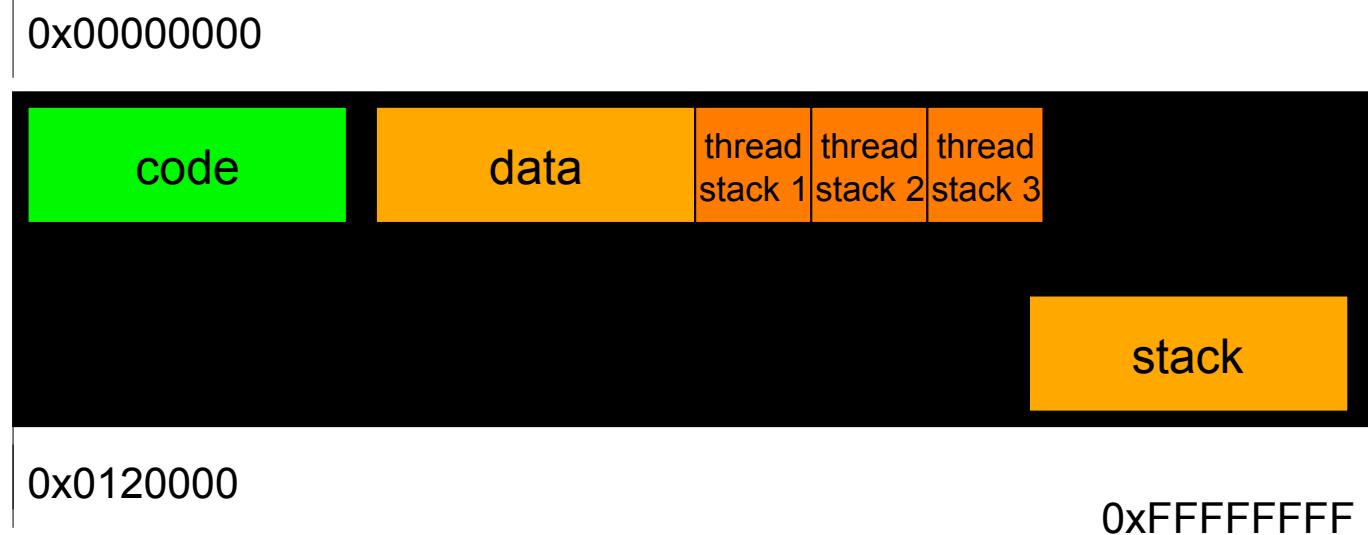
Thread State and Thread Stacks

- Each thread has its own registers, PS, PC
- Each thread must have its own stack area
- Maximum stack size specified when thread is created
 - A process can contain many threads
 - They cannot all grow towards a single hole
 - Thread creator must know max required stack size
 - Stack space must be reclaimed when thread exits
- Procedure linkage conventions are unchanged

UNIX Process Stack Space Management



Thread Stack Allocation



Inter-Process Communication

- Even fairly distinct processes may occasionally need to exchange information
- The OS provides mechanisms to facilitate that
 - As it must, since processes can't normally “touch” each other
- IPC

Goals for IPC Mechanisms

- We look for many things in an IPC mechanism
 - Simplicity
 - Convenience
 - Generality
 - Efficiency
 - Robustness and reliability
- Some of these are contradictory
 - Partially handled by providing multiple different IPC mechanisms

OS Support For IPC

- Provided through system calls
- Typically requiring activity from both communicating processes
 - Usually can't “force” another process to perform IPC
- Usually mediated at each step by the OS
 - To protect both processes
 - And ensure correct behavior

IPC: Synchronous and Asynchronous

- Synchronous IPC
 - Writes block until message sent/delivered/received
 - Reads block until a new message is available
 - Very easy for programmers
- Asynchronous operations
 - Writes return when system accepts message
 - No confirmation of transmission/delivery/reception
 - Requires auxiliary mechanism to learn of errors
 - Reads return promptly if no message available
 - Requires auxiliary mechanism to learn of new messages
 - Often involves "wait for any of these" operation
 - Much more efficient in some circumstances

Typical IPC Operations

- Create/destroy an IPC channel
- Write/send/put
 - Insert data into the channel
- Read/receive/get
 - Extract data from the channel
- Channel content query
 - How much data is currently in the channel?
- Connection establishment and query
 - Control connection of one channel end to another
 - Provide information like:
 - Who are end-points?
 - What is status of connections?

IPC: Messages vs. Streams

- A fundamental dichotomy in IPC mechanisms
- Streams
 - A continuous stream of bytes
 - Read or write a few or many bytes at a time
 - Write and read buffer sizes are unrelated
 - Stream may contain app-specific record delimiters
- Messages (aka datagrams)
 - A sequence of distinct messages
 - Each message has its own length (subject to limits)
 - Each message is typically read/written as a unit
 - Delivery of a message is typically all-or-nothing
- Each style is suited for particular kinds of interactions

IPC and Flow Control

- Flow control: making sure a fast sender doesn't overwhelm a slow receiver
- Queued messages consume system resources
 - Buffered in the OS until the receiver asks for them
- Many things can increase required buffer space
 - Fast sender, non-responsive receiver
- Must be a way to limit required buffer space
 - Sender side: block sender or refuse message
 - Receiving side: stifle sender, flush old messages
 - This is usually handled by network protocols
- Mechanisms for feedback to sender

<https://www.youtube.com/watch?v=HnbNcQlzV-4>

IPC Reliability and Robustness

- Within a single machine, OS won't accidentally "lose" IPC data
- Across a network, requests and responses can be lost
- Even on single machine, though, a sent message may not be processed
 - The receiver is invalid, dead, or not responding
- And how long must the OS be responsible for IPC data?

Reliability Options

- When do we tell the sender “OK”?
 - When it’s queued locally?
 - When it’s Added to receivers input queue?
 - When the receiver has read it?
 - When the receiver has explicitly acknowledged it?
- How persistently does the system attempt delivery?
 - Especially across a network
 - Do we try retransmissions? How many?
 - Do we try different routes or alternate servers?
- Do channel/contents survive receiver restarts?
 - Can a new server instance pick up where the old left off?

Some Styles of IPC

- Pipelines
- Sockets
- Mailboxes and named pipes
- Shared memory

Pipelines

- Data flows through a series of programs
 - ls | grep | sort | mail
 - Macro processor | complier | assembler
- Data is a simple byte stream
 - Buffered in the operating system
 - No need for intermediate temporary files
- There are no security/privacy/trust issues
 - All under control of a single user
- Error conditions
 - Input: End of File
 - Output: next program failed
- *Simple, but very limiting*

Sockets

- Connections between addresses/ports
 - Connect/listen/accept
 - Lookup: registry, DNS, service discovery protocols
- Many data options
 - Reliable or best effort data-grams
 - Streams, messages, remote procedure calls, ...
- Complex flow control and error handling
 - Retransmissions, timeouts, node failures
 - Possibility of reconnection or fail-over
- Trust/security/privacy/integrity
 - We'll discuss these issues later
- *Very general, but more complex*

Mailboxes and Named Pipes

- A compromise between sockets and pipes
- A client/server rendezvous point
 - A name corresponds to a service
 - A server awaits client connections
 - Once open, it may be as simple as a pipe
 - OS may authenticate message sender
- Limited fail-over capability
 - If server dies, another can take its place
 - But what about in-progress requests?
- Client/server must be on same system
- *Some advantages/disadvantages of other options*

Shared Memory

- OS arranges for processes to share read/write memory segments
 - Mapped into multiple process' address spaces
 - Applications must provide their own control of sharing
 - OS is not involved in data transfer
 - Just memory reads and writes via limited direct execution
 - So very fast
- Simple in some ways
 - Terribly complicated in others
 - The cooperating processes must achieve whatever effects they want

Only works on a local machine

Synchronization

- Making things happen in the “right” order
- Easy if only one set of things is happening
- Easy if simultaneously occurring things don’t affect each other
- Hideously complicated otherwise
- Wouldn’t it be nice if we could avoid it?
- Well, we can’t
 - We must have parallelism

The Benefits of Parallelism

- Improved throughput
 - Blocking of one activity does not stop others
- Improved modularity
 - Separating complex activities into simpler pieces
- Improved robustness
 - The failure of one thread does not stop others
- A better fit to emerging paradigms
 - Client server computing, web based services
 - Our universe is cooperating parallel processes

Why Is There a Problem?

- Sequential program execution is easy
 - First instruction one, then instruction two, ...
 - Execution order is obvious and deterministic
- Independent parallel programs are easy
 - If the parallel streams do not interact in any way
- Cooperating parallel programs are hard
 - If the two execution streams are not synchronized
 - Results depend on the order of instruction execution
 - Parallelism makes execution order non-deterministic
 - Results become combinatorially intractable

Synchronization Problems

- Race conditions
- Non-deterministic execution

Race Conditions

- What happens depends on execution order of processes/threads running in parallel
 - Sometimes one way, sometimes another
 - These happen all the time, most don't matter
- But some race conditions affect correctness
 - Conflicting updates (mutual exclusion)
 - Check/act races (sleep/wakeup problem)
 - Multi-object updates (all-or-none transactions)
 - Distributed decisions based on inconsistent views
- Each of these classes can be managed
 - If we recognize the race condition and danger

Non-Deterministic Execution

- Parallel execution reduces predictability of process behavior
 - Processes block for I/O or resources
 - Time-slice end preemption
 - Interrupt service routines
 - Unsynchronized execution on another core
 - Queuing delays
 - Time required to perform I/O operations
 - Message transmission/delivery time
- Which can lead to many problems

What Is “Synchronization”?

- True parallelism is imponderable
 - We’re not smart enough to understand it
 - Pseudo-parallelism may be good enough
 - Mostly ignore it
 - But identify and control key points of interaction
- Actually two interdependent problems
 - *Critical section serialization*
 - *Notification of asynchronous completion*
- They are often discussed as a single problem
 - Many mechanisms simultaneously solve both
 - Solution to either requires solution to the other
- They can be understood and solved separately

The Critical Section Problem

- A *critical section* is a resource that is shared by multiple threads
 - By multiple concurrent threads, processes or CPUs
 - By interrupted code and interrupt handler
- Use of the resource changes its state
 - Contents, properties, relation to other resources
- Correctness depends on execution order
 - When scheduler runs/preempts which threads
 - Relative timing of asynchronous/independent events

Reentrant & MultiThread-safe Code

- Consider a simple recursive routine:
`int factorial(x) { tmp = factorial(x-1); return x*tmp }`
- Consider a possibly multi-threaded routine:
`void debit(amt) {tmp = bal-amt; if (tmp >=0) bal = tmp}`
- Neither would work if tmp was shared/static
 - Must be dynamic, each invocation has own copy
 - This is not a problem with read-only information
- Some variables must be shared
 - And proper sharing often involves critical sections

Critical Section Example 1: Updating a File

Process 1

```
remove("database");
fd = create("database");
write(fd,newdata,length);
close(fd);
```

```
remove("database");
fd = create("database");
fd = open("database",READ);
count = read(fd,buffer,length);
write(fd,newdata,length);
close(fd);
```

Process 2

```
fd = open("database",READ);
count = read(fd,buffer,length);
```

- Process 2 reads an empty database
 - This result could not occur with any sequential execution

Critical Section Example 2: Re-entrant Signals

First signal

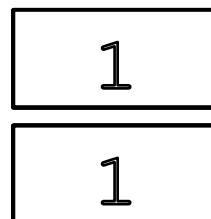
```
load r1,numsigs // = 0  
add r1,=1 // = 1  
store r1,numsigs // =1
```

```
load r1,numsigs // = 0  
add r1,=1 // = 1
```

```
store r1,numsigs // =1
```

So numsigs is 1,
instead of 2

numsigs
r1



Second signal

```
load r1,numsigs // = 0  
add r1,=1 // = 1  
store r1,numsigs // =1
```

```
load r1,numsigs // = 0  
add r1,=1 // = 1  
store r1,numsigs // =1
```

The signal handlers share
numsigs and r1 ...

Critical Section Example 3: Multithreaded Banking Code

Thread 1

```
load r1, balance // = 100  
load r2, amount1 // = 50  
add r1, r2      // = 150  
store r1, balance // = 150
```

```
load r1, t  
load r2, :  
add r1, r_
```

CONTEXT SWITCH!!!

```
store r1, balance // = 150
```

Thread 2

```
load r1, balance // = 100  
load r2, amount2 // = 25  
sub r1, r2      // = 75  
store r1, balance // = 75
```

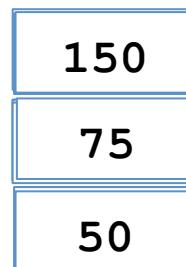
The \$25 debit was lost!!!

```
load r1, balance // = 100  
load r2, amount2 // = 25  
sub r1, r2      // = 75  
store r1, balance // = 75
```

amount1

50

balance



amount2

25

Even A Single Instruction Can Contain a Critical Section

thread #1

counter = counter + 1;

thread #2

counter = counter + 1;

*But what looks like one instruction in
C gets compiled to:*

mov counter, %eax

add \$0x1, %eax

mov %eax, counter

Three instructions . . .

Why Is This a Critical Section?

thread #1

counter = counter + 1;

thread #2

counter = counter + 1;

This could happen:

mov counter, %eax
add \$0x1, %eax

mov counter, %eax
add \$0x1, %eax
mov %eax, counter

mov %eax, counter

If counter started at 1, it should end at 3
In this execution, it ends at 2

These Kinds of Interleavings Seem Pretty Unlikely

- To cause problems, things have to happen exactly wrong
- Indeed, that's true
- But you're executing a billion instructions per second
- So even very low probability events can happen with frightening frequency
- Often, one problem blows up everything that follows

Critical Sections and Mutual Exclusion

- Critical sections can cause trouble when more than one thread executes them at a time
 - Each thread doing part of the critical section before any of them do all of it
- Preventable if we ensure that only one thread can execute a critical section at a time
- We need to achieve *mutual exclusion* of the critical section
- How?

One Solution: Interrupt Disables

- Temporarily block some or all interrupts
 - Can be done with a privileged instruction
 - Side-effect of loading new Processor Status Word
- Abilities
 - Prevent Time-Slice End (timer interrupts)
 - Prevent re-entry of device driver code
- Dangers
 - May delay important operations
 - A bug may leave them permanently disabled

What Happens During an Interrupt?

- What we discussed before
- The hardware traps to stop whatever is executing
- A trap table is consulted
- An Interrupt Service Routine (ISR) is consulted
- The ISR handles the interrupt and restores the CPU to its earlier state
 - Generally, interrupted code continues

Preventing Preemption

```
DLL_insert(DLL *head, DLL*element) {  
    int save = disableInterrupts();  
    DLL *last = head->prev;  
    element->prev = last;  
    element->next = head;  
    last->next = element;  
    head->prev = element;  
}  
  
restoreInterrupts(save);
```

```
DLL_insert(DLL *head, DLL*element) {  
    DLL *last = head->prev;  
    element->prev = last;  
    element->next = head;  
    last->next = element;  
    head->prev = element;  
}
```

Preventing Driver Reentrancy

```
zz_io_startup( struct iorq *bp ) {
    ...
    save = intr_enable( ZZ_DISABLE );

    /* program the DMA request */
    zzSetReg(ZZ_R_ADDR, bp->buffer_start );
    zzSetReg(ZZ_R_LEN, bp->buffer_length);
    zzSetReg(ZZ_R_BLOCK, bp->blocknum);
    zzSetReg(ZZ_R_CMD, bp->write?
        ZZ_C_WRITE : ZZ_C_READ );
    zzSetReg(ZZ_R_CTRL, ZZ_INTR+ZZ_GO);

    /* reenable interrupts      */
    intr_enable( save );
}

zz_intr_handler() {
    ...
    /* update data read count */
    resid = zzGetReg(ZZ_R_LEN);

    /* turn off device ability to interrupt */
    zzSetReg(ZZ_R_CTRL, ZZ_NOINTR);
    ...
}
```

Serious consequences could result if the interrupt handler was called while we were half-way through programming the DMA operation.

Preventing Driver Reentrancy

- Interrupts are usually self-disabling
 - CPU may not deliver #2 until #1 is *acknowledged*
 - Interrupt vector PS usually disables causing interrupts
- They are restored after servicing is complete
 - ISR may explicitly *acknowledge* the interrupt
 - Return from ISR will restore previous (enabled) PS
- Drivers usually disable during critical sections
 - Updating registers used by interrupt handlers
 - Updating resources used by interrupt handlers

Downsides of Disabling Interrupts

- Not an option in user mode
 - Requires use of privileged instructions
- Dangerous if improperly used
 - Could disable preemptive scheduling, disk I/O, etc.
- Delays system response to important interrupts
 - Received data isn't processed until interrupt serviced
 - Device will sit idle until next operation is initiated
- May prevent safe concurrency

Interrupts and Resource Allocation

- Interrupt handlers are not allowed to block
 - Only a scheduled process/thread can block
 - Interrupts are disabled until call completes
- Ideally they should never need to wait
 - Needed resources are already allocated
 - Operations implemented with lock-free code
- Brief spins may be acceptable
 - Wait for hardware to acknowledge a command
 - Wait for a co-processor to release a lock

Interrupts – When To Disable Them

- In situations that involve shared resources
 - Used by both synchronous and interrupt code
 - Hardware registers (e.g., in a device or clock)
 - Communications queues and data structures
- That also involve non-atomic updates
 - Operations that require multiple instructions
 - Where pre-emption in mid-operation could lead to data corruption or a deadlock.
- Must disable interrupts in these critical sections
 - Disable them as seldom and as briefly as possible

Be Careful With Interrupts

- Be very sparing in your use of disables
 - Interrupt service time is very costly
 - Scheduled processes have been preempted
 - Devices may be idle, awaiting new instructions
 - The system will be less responsive
 - Disable as few interrupts as possible
 - Disable them as briefly as possible
- Interrupt routines cannot block or yield the CPU
 - They are not a scheduled thread that can block/run
 - Cannot do resource allocations that might block
 - Cannot do synchronization operations that might block

Evaluating Interrupt Disables

- Effectiveness/Correctness
 - Ineffective against multiprocessor/device parallelism
 - Only usable by kernel mode code
- Progress
 - Deadlock risk (if handler can block for resources)
- Fairness
 - Pretty good (assuming disables are brief)
- Performance
 - One instruction, much cheaper than system call
 - Long disables may impact system performance

Other Possible Solutions

- Avoid shared data whenever possible
- Eliminate critical sections with atomic instructions
 - Atomic (uninterruptable) read/modify/write operations
 - Can be applied to 1-8 contiguous bytes
 - Simple: increment/decrement, and/or/xor
 - Complex: test-and-set, exchange, compare-and-swap
- Use atomic instructions to implement locks
 - Use the lock operations to protect critical sections
- We'll cover this in the next class

Operating System Principles: Mutual Exclusion and Asynchronous Completion

CS 111

Operating Systems
Peter Reiher

Outline

- Mutual Exclusion
- Asynchronous Completions

Mutual Exclusion

- Critical sections can cause trouble when more than one thread executes them at a time
 - Each thread doing part of the critical section before any of them do all of it
- Preventable if we ensure that only one thread can execute a critical section at a time
- We need to achieve *mutual exclusion* of the critical section

Critical Sections in Operating System

- Operating systems are loaded with internal critical sections
- Shared data used by concurrent threads
 - Process state variables
 - Resource pools
 - Device driver state
- Logical parallelism
 - Created by preemptive scheduling and asynchronous interrupts
- Physical parallelism
 - Shared memory, symmetric multi-processors
- OSes extensively use locks to avoid these problems
 - Without any user-visible effects

Critical Sections in Applications

- Most common for multithreaded applications
 - Which frequently share data structures
- Can also happen with processes
 - Which share operating system resources
 - Like files
- Avoidable if you don't share resources of any kind
 - But that's not always feasible

Recognizing Critical Sections

- Generally involves updates to object state
 - May be updates to a single object
 - May be related updates to multiple objects
- Generally involves multi-step operations
 - Object state inconsistent until operation finishes
 - Pre-emption compromises object or operation
- Correct operation requires mutual exclusion
 - Only one thread at a time has access to object(s)
 - Client 1 completes before client 2 starts

Critical Sections and Atomicity

- Using mutual exclusion allows us to achieve *atomicity* of a critical section
- Atomicity has two aspects:
 1. Before or After atomicity
 - A enters critical section before B starts
 - B enters critical section after A completes
 - There is no overlap
 2. All or None atomicity
 - An update that starts will complete
 - An uncompleted update has no effect
- Correctness generally requires both

Options for Protecting Critical Sections

- Turn off interrupts
 - We covered that in the last class
 - Prevents concurrency
- Avoid shared data whenever possible
- Protect critical sections using hardware mutual exclusion
 - In particular, atomic CPU instructions

Avoiding Shared Data

- A good design choice when feasible
- Don't share things you don't need to share
- But not always an option
- Even if possible, may lead to inefficient resource use
- Sharing read only data also avoids problems
 - If no writes, the order of reads doesn't matter
 - But a single write can blow everything out of the water

Atomic Instructions

- CPU instructions are uninterruptable
- What can they do?
 - Read/modify/write operations
 - Can be applied to 1-8 contiguous bytes
 - Simple: increment/decrement, and/or/xor
 - Complex: test-and-set, exchange, compare-and-swap
- Either do entire critical section in one atomic instruction
- Or use atomic instructions to implement locks
 - Use the lock operations to protect critical sections

Atomic Instructions – Test and Set

A C description of a machine language instruction

```
bool TS( char *p) {  
    bool rc;  
    rc = *p;          /* note the current value */  
    *p = TRUE;        /* set the value to be TRUE */  
    return rc;         /* return the value before we set it */  
}  
  
if !TS(flag) {  
    /* We have control of the critical section! */  
}
```

Atomic Instructions – Compare and Swap

Again, a C description of machine instruction

```
bool compare_and_swap( int *p, int old, int new ) {  
    if (*p == old) { /* see if value has been changed */  
        *p = new; /* if not, set it to new value */  
        return( TRUE); /* tell caller he succeeded */  
    } else /* value has been changed */  
        return( FALSE); /* tell caller he failed */  
  
}  
  
if (compare_and_swap(flag,UNUSED,IN_USE) {  
    /* I got the critical section! */  
} else {  
    /* I didn't get it. */  
}
```

Preventing Concurrency Via Atomic Instructions

- CPU instructions are hardware-atomic
 - So if you can squeeze a critical section into one instruction, no concurrency problems
- What can you do in one instruction?
 - Simple operations like read/write
 - Some slightly more complex operations
 - With careful design, some data structures can be implemented this way
- Limitations
 - Unusable for complex critical sections
 - Unusable as a waiting mechanism

Lock-Free Operations

- Multi-thread safe data structures and operations
 - An alternative to locking or disabling interrupts
- How do they work?
 - Carefully program data structure to perform critical operations with one instruction
- Allows:
 - Single reader/writer with ordinary instructions
 - Multi-reader/writer with atomic instructions
 - All-or-none and before-or-after semantics
- Limitations
 - Unusable for complex critical sections
 - Unusable as a waiting mechanism

An Example

```
// push an element on to a singly linked LIFO list
void SLL_push(SLL *head, SLL *element) {
    do {
        SLL *prev = head->next;
        element->next = prev;
    } while ( CompareAndSwap(&head->next, prev, element) != prev);
}
```

Evaluating Lock-Free Operations

- Effectiveness/Correctness
 - Effective against all conflicting updates
 - Cannot be used for complex critical sections
- Progress
 - No possibility of deadlock or convoy
- Fairness
 - Small possibility of brief spins
 - Like the compare-and-swap while loop in example
- Performance
 - Expensive instructions, but cheaper than syscalls

Locking

- Protect critical sections with a data structure
 - Use atomic instructions to implement that structure
- Locks
 - The party holding a lock can access the critical section
 - Parties not holding the lock cannot access it
- A party needing to use the critical section tries to acquire the lock
 - If it succeeds, it goes ahead
 - If not . . . ?
- When finished with critical section, release the lock
 - Which someone else can then acquire

Using Locks

- Remember this example?

thread #1	thread #2
counter = counter + 1;	counter = counter + 1;

*What looks like one instruction in C
gets compiled to:*

```
mov counter, %eax  
add $0x1, %eax  
mov %eax, counter
```

Three instructions . . .

- How can we solve this with locks?

Using Locks For Mutual Exclusion

```
pthread_mutex_t lock;  
pthread_mutex_init(&lock, NULL);  
...  
if (pthread_mutex_lock(&lock) == 0) {  
    counter = counter + 1;  
    pthread_mutex_unlock(&lock);  
}
```

Now the three assembly instructions are mutually exclusive

What Happens When You Don't Get the Lock?

- You could just give up
 - But then you'll never execute your critical section
- You could try to get it again
- But it still might not be available
- So you could try to get it again . . .

Locks and Interrupts: A Dangerous Combination

Synchronous Code

Interrupt Handler

Infinite Loop!

```
while( TS(lockp) );
/* critical section */
...
*lockp = 0;
```

```
while( TS(lockp) );
/* critical section */
...
```



Interrupt handler will loop
Interrupts disabled when handler entered
Interrupt handler can't get the lock
Interrupts will remain disabled

Synchronous code will never complete

So lock will never be released

Spin Waiting



- The computer science equivalent
- Check if the event occurred
- If not, check again
- And again
- And again
- . . .

Spin Locks: Pluses and Minuses

- Good points
 - Properly enforces access to critical sections
 - Assuming properly implemented locks
 - Simple to program
- Dangers
 - Wasteful
 - Spinning uses processor cycles
 - Likely to delay freeing of desired resource
 - Spinning uses processor cycles
 - Bug may lead to infinite spin-waits

How Do We Build Locks?

- The very operation of locking and unlocking a lock is itself a critical section
 - If we don't protect it, two threads might acquire the same lock
- Sounds like a chicken-and-egg problem
- But we can solve it with hardware assistance
- Individual CPU instructions are atomic
 - So if we can implement a lock with one instruction . . .

Single Instruction Locks

- Sounds tricky
- The core operation of acquiring a lock (when it's free) requires:
 1. Check that no one else has it
 2. Change something so others know we have it
- Sounds like we need to do two things in one instruction
- No problem – hardware designers have provided for that

Building Locks From Single Instructions

- Requires a complex atomic instruction
 - Test and set
 - Compare and swap
- Instruction must atomically:
 - Determine if someone already has the lock
 - Grant it if no one has it
 - Return something that lets the caller know what happened
- Caller must honor the lock . . .

Using Atomic Instructions to Implement a Lock

- Assuming C implementation of test and set

```
bool getlock( lock *lockp) {  
    if (TS(lockp) == 0 )  
        return( TRUE);  
    else  
        return( FALSE);  
}  
void freelock( lock *lockp ) {  
    *lockp = 0;  
}
```

Locks Come in Many Flavors

- Lock and wait
 - Block until resource becomes available
- Non-blocking
 - Return an error if resource is unavailable
- Timed wait
 - Block a specified maximum time, then fail
- Spin and wait (futex)
 - Spin briefly, and then join a waiting list
- Strict FIFO
 - Join a FIFO queue of those waiting on the lock
 - Other wait options might not guarantee FIFO

The Asynchronous Completion Problem

- Parallel activities move at different speeds
- One activity may need to wait for another to complete
- The *asynchronous completion problem* is how to perform such waits without killing performance
- Examples of asynchronous completions
 - Waiting for an I/O operation to complete
 - Waiting for a response to a network request
 - Delaying execution for a fixed period of real time

How Can We Wait?

- Spin locking/busy waiting
- Yield and spin ...
- Either spin option may still require mutual exclusion
- Completion events

Spin Waiting For Asynchronous Completions

- Wastes CPU, memory, bus bandwidth
 - Each path through the loop costs instructions
- May actually delay the desired event
 - One of your cores is busy spinning
 - Maybe it could be doing the work required to complete the event instead
 - But it's spinning . . .

Spinning Sometimes Makes Sense

1. When awaited operation proceeds in parallel
 - A hardware device accepts a command
 - Another CPU releases a briefly held spin-lock
2. When awaited operation is guaranteed to be soon
 - Spinning is less expensive than sleep/wakeup
3. When spinning does not delay awaited operation
 - Burning CPU delays running another process
 - Burning memory bandwidth slows I/O
4. When contention is expected to be rare
 - Multiple waiters greatly increase the cost

A Classic “spin-wait”

```
/* set a specified register in the ZZ controller to a specified value */  
zzSetReg( struct zzcontrol *dp, short reg, long value ) {  
    while( (dp->zz_status & ZZ_CMD_READY) == 0)  
        ;  
    dp->zz_value = value;  
    dp->zz_reg = reg;  
    dp->zz_cmd = ZZ_SET_REG;  
}  
  
/* program the ZZ for a specified DMA read or write operation */  
zzStartIO( struct zzcontrol *dp, struct ioreq *bp ) {  
    zzSetReg(dp, ZZ_R_ADDR, bp->buffer_start);  
    zzSetReg(dp, ZZ_R_LEN, bp->buffer_length);  
    zzSetReg(dp, ZZ_R_CMD, bp->write ? ZZ_C_WRITE : ZZ_C_READ );  
    zzSetReg(dp, ZZ_R_CTRL, ZZ_INTR + ZZ_GO);  
}
```

No guarantee
that hardware
is ready when
this routine
returns.

Yield and Spin

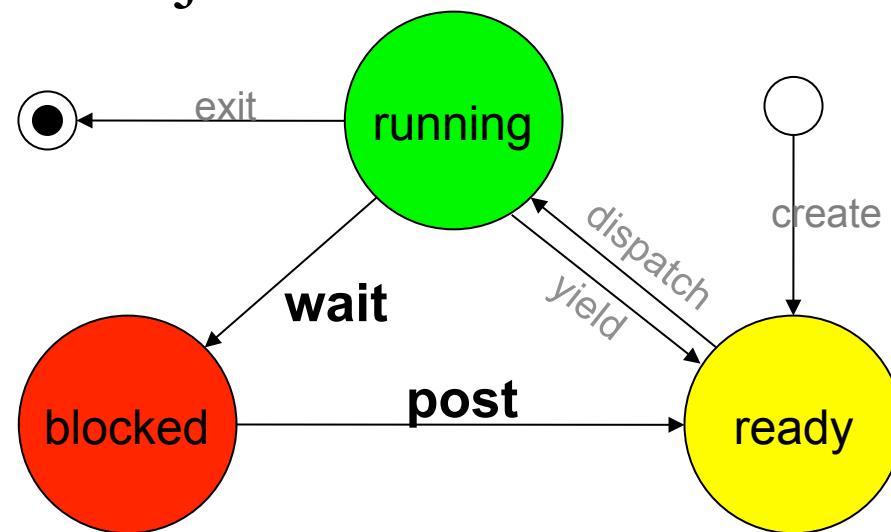
- Check if your event occurred
- Maybe check a few more times
- But then yield
- Sooner or later you get rescheduled
- And then you check again
- Repeat checking and yielding until your event is ready

Problems With Yield and Spin

- Extra context switches
 - Which are expensive
- Still wastes cycles if you spin each time you're scheduled
- You might not get scheduled to check until long after event occurs
- Works very poorly with multiple waiters

Another Approach: Condition Variables

- Create a synchronization object associated with a resource or request
 - Requester blocks awaiting event on that object
 - Upon completion, the event is “posted”
 - Posting event to object unblocks the waiter



Condition Variables and the OS

- Generally the OS provides condition variables
 - Or library code that implements threads does
- It blocks a process or thread when condition variable is used
 - Moving it out of the ready queue
- It observes when the desired event occurs
- It then unblocks the blocked process or thread
 - Putting it back in the ready queue
 - Possibly preempting the running process

Waiting Lists

- Likely to have threads waiting on several different things
- Pointless to wake up everyone on every event
 - Each should wake up when his event happens
- Suggests all events need a waiting list
 - When posting an event, look up who to awaken
 - Wake up everyone on the list?
 - One-at-a-time in FIFO order?
 - One-at-a-time in priority order (possible starvation)?
 - Choice depends on event and application

Who To Wake Up?

- Who wakes up when a condition variable is signaled?
 - `pthread_cond_wait` ... at least one blocked thread
 - `pthread_cond_broadcast` ... all blocked threads
- The broadcast approach may be wasteful
 - If the event can only be consumed once
 - Potentially unbounded waiting times
- A waiting queue would solve these problems
 - Each post wakes up the first client on the queue

Evaluating Waiting List Options

- Effectiveness/Correctness
 - Should be very good
- Progress
 - There is a trade-off involving *cutting* in line
- Fairness
 - Should be very good
- Performance
 - Should be very efficient
 - Depends on frequency of spurious wakeups

Locking and Waiting Lists

- Spinning for a lock is usually a bad thing
 - Locks should probably have waiting lists
- A waiting list is a (shared) data structure
 - Implementation will likely have critical sections
 - Which may need to be protected by a lock
- This seems to be a circular dependency
 - Locks have waiting lists
 - Which must be protected by locks
 - What if we must wait for the waiting list lock?

A Possible Problem

- The sleep/wakeup race condition

Consider this sleep code:

```
void sleep( eventp *e ) {  
    while(e->posted == FALSE) {  
        add_to_queue( &e->queue,  
                      myproc );  
        myproc->runstate |= BLOCKED;  
        yield();  
    }  
}
```

And this wakeup code:

```
void wakeup( eventp *e) {  
    struct proce *p;  
  
    e->posted = TRUE;  
    p = get_from_queue(&e->  
                      queue);  
    if (p) {  
        p->runstate &= ~BLOCKED;  
        resched();  
    } /* if !p, nobody's  
        waiting */  
}
```

What's the problem with this?

A Sleep/Wakeup Race

- Let's say thread B is using a resource and thread A needs to get it
- So thread A will call `sleep()`
- Meanwhile, thread B finishes using the resource
 - So thread B will call `wakeup()`
- No other threads are waiting for the resource

The Race At Work

Thread A

```
void sleep( eventp *e ) {  
    while(e->posted == FALSE) {  
  
        CONTEXT SWITCH!  
  
        Nope, nobody's in the queue!  
  
        CONTEXT SWITCH!  
  
        add_to_queue( &e->queue, myproc );  
        myproc->runstate |= BLOCKED;  
        yield();  
    }  
}
```

Thread A is sleeping

Thread B

Yep, somebody's locked it!

```
void wakeup( eventp *e) {  
    struct proce *p;  
  
    e->posted = TRUE;  
    p = get_from_queue(&e-> queue);  
    if (p) {  
        } /* if !p, nobody's waiting */  
    }
```

The effect?

But there's no one to
wake him up

Solving the Problem

- There is clearly a critical section in `sleep()`
 - Starting before we test the posted flag
 - Ending after we put ourselves on the notify list
- During this section, we need to prevent
 - Wakeups of the event
 - Other people waiting on the event
- This is a mutual-exclusion problem
 - Fortunately, we already know how to solve those

Progress vs. Fairness

- Consider ...
 - P1: lock(), park()
 - P2: unlock(), unpark()
 - P3: lock()
- Progress says:
 - It is available, so P3 gets it
 - Spurious wakeup of P1
- Fairness says:
 - FIFO, P3 gets in line
 - And a convoy forms

```
void lock(lock_t *m) {
    while(true) {
        while (TestAndSet(&m->guard, 1) == 1);
        if (!m->locked) {
            m->locked = 1;
            m->guard = 0;
            return;
        }
        queue_add(m->q, me);
        m->guard = 0;
        park();
    }
}
```

```
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1);
    m->locked = 0;
    if (!queue_empty(m->q))
        unpark(queue_remove(m->q));
    m->guard = 0;
}
```

Spin-Waits Revisited

- Spin-waits await asynchronous completions
 - But they do so by busy-waiting
while (event_not_ready) ;
- Sleep/wake-up is almost always better
 - Fewer wasted cycles and faster response
 - But these are software completion mechanisms
 - There are hardware-related situations where they don't work (or don't make sense)
- There are cases where it makes sense to spin
 - Very briefly for events originating outside our CPU

Spin-waits: when to use them

- When the event does not come from our CPU
 - So spinning will not delay the completion
- And waiting time guaranteed to be very brief
 - Fewer cycles than would be required to go to sleep
- Examples:
 - Waiting a few μ -seconds for hardware to come ready
 - **IF** it is guaranteed to be come back promptly
 - Waiting for another CPU to release a lock
 - **IF** critical section is very short (e.g. 1 digit # of instructions)
 - **IF** interrupts are disabled so preemption is impossible
- **Almost never** appropriate in user-mode code

Operating System Principles: Semaphores and Locks for Synchronization

CS 111

Operating Systems
Peter Reiher

Outline

- Locks
- Semaphores
- Mutexes and object locking
- Getting good performance with locking

Our Synchronization Choices

- To repeat:
 1. Don't share resources
 2. Turn off interrupts to prevent concurrency
 3. Always access resources with atomic instructions
 4. Use locks to synchronize access to resources
- If we use locks,
 1. Use spin loops when your resource is locked
 2. Use primitives that block you when your resource is locked and wake you later

Concentrating on Locking

- Locks are necessary for many synchronization problems
- How do we implement locks?
 - It had better be correct, always
- How do we ensure that locks are used in ways that don't kill performance?

Basic Locking Operations

- When possible concurrency problems,
 1. Obtain a lock related to the shared resource
 - Block or spin if you don't get it
 2. Once you have the lock, use the shared resource
 3. Release the lock
- Whoever implements the locks ensures no concurrency problems in the lock itself
 - Using atomic instructions
 - Or disabling interrupts

Semaphores

- A theoretically sound way to implement locks
 - With important extra functionality critical to use in computer synchronization problems
- Thoroughly studied and precisely specified
 - Not necessarily so usable, however
- Like any theoretically sound mechanism, could be gaps between theory and implementation

Semaphores – A Historical Perspective

When direct communication was not an option



E.g., between villages, ships, trains



The Semaphores We're Studying

- Concept introduced in 1968 by Edsger Dijkstra
 - Cooperating sequential processes
- THE classic synchronization mechanism
 - Behavior is well specified and universally accepted
 - A foundation for most synchronization studies
 - A standard reference for all other mechanisms
- More powerful than simple locks
 - They incorporate a FIFO waiting queue
 - They have a counter rather than a binary flag

Semaphores - Operations

- Semaphore has two parts:
 - An integer counter (initial value unspecified)
 - A FIFO waiting queue
- P (proberen/test) ... “wait”
 - Decrement counter, if count ≥ 0 , return
 - If counter < 0 , add process to waiting queue
- V (verhogen/raise) ... “post” or “signal”
 - Increment counter
 - If counter ≥ 0 & queue non-empty, wake 1st process

Using Semaphores for Exclusion

- Initialize semaphore count to one
 - Count reflects # threads allowed to hold lock
- Use P/wait operation to take the lock
 - The first will succeed
 - Subsequent attempts will block
- Use V/post operation to release the lock
 - Restore semaphore count to non-negative
 - If any threads are waiting, unblock the first in line

Using Semaphores for Notifications

- Initialize semaphore count to zero
 - Count reflects # of completed events
- Use P/wait operation to await completion
 - If already posted, it will return immediately
 - Else all callers will block until V/post is called
- Use V/post operation to signal completion
 - Increment the count
 - If any threads are waiting, unblock the first in line
- One signal per wait: no broadcasts

Counting Semaphores

- Initialize semaphore count to ...
 - Count reflects # of available resources
- Use P/wait operation to consume a resource
 - If available, it will return immediately
 - Else all callers will block until V/post is called
- Use V/post operation to produce a resource
 - Increment the count
 - If any threads are waiting, unblock the first in line
- One signal per wait: no broadcasts

Semaphores For Mutual Exclusion

```
struct account {  
    struct semaphore s;      /* initialize count to 1, queue empty, lock 0 */  
    int balance;  
    ...  
};  
  
int write_check( struct account *a, int amount ) {  
    int ret;  
    p( &a->semaphore );          /* get exclusive access to the account */  
  
    if ( a->balance >= amount ) { /* check for adequate funds */  
        amount -= balance;  
        ret = amount;  
    } else {  
        ret = -1;  
    }  
  
    v( &a->semaphore );          /* release access to the account */  
    return( ret );  
}
```

Semaphores for Completion Events

```
struct semaphore pipe_semaphore = { 0, 0, 0 }; /* count = 0; pipe empty */
char buffer[BUFSIZE]; int read_ptr = 0, write_ptr = 0;

char pipe_read_char() {
    p(&pipe_semaphore);                      /* wait for input available */
    c = buffer[read_ptr++];                  /* get next input character */
    if (read_ptr >= BUFSIZE)                /* circular buffer wrap */
        read_ptr -= BUFSIZE;
    return(c);
}

void pipe_write_string( char *buf, int count ) {
    while( count-- > 0 ) {
        buffer[write_ptr++] = *buf++;          /* store next character */
        if (write_ptr >= BUFSIZE) /* circular buffer wrap */
            write_ptr -= BUFSIZE;
        v( &pipe_semaphore );                 /* signal char available */
    }
}
```

Implementing Semaphores

```
void sem_wait(sem_t *s) {  
    pthread_mutex_lock(&s->lock);  
    while (s->value <= 0)  
        pthread_cond_wait(&s->cond, &s->lock);  
    s->value--;  
    pthread_mutex_unlock(&s->lock);  
}  
  
void sem_post(sem_t *s) {  
    pthread_mutex_lock(&s->lock);  
    s->value++;  
    pthread_cond_signal(&s->cond);  
    pthread_mutex_unlock(&s->lock)  
}
```

Implementing Semaphores in OS

```
void sem_wait(sem_t *s) {
    for (;;) {
        save = intr_enable( ALL_DISABLE );
        while( TestAndSet( &s->lock ) );
        if (s->value > 0) {
            s->value--;
            s->sem_lock = 0;
            intr_enable( save );
            return;
        }
        add_to_queue( &s->queue, myproc );
        myproc->runstate |= PROC_BLOCKED;
        s->lock = 0;
        intr_enable( save );
        yield();
    }
}
```

```
}
```

```
void sem_post(struct sem_t *s) {
    struct proc_desc *p = 0;
    save = intr_enable( ALL_DISABLE );
    while ( TestAndSet( &s->lock ) );
    s->value++;
    if (p = get_from_queue( &s->queue )) {
        p->runstate &= ~PROC_BLOCKED;
    }
    s->lock = 0;
    intr_enable( save );
    if (p)
        reschedule( p );
}
```

Limitations of Semaphores

- Semaphores are a very spartan mechanism
 - They are simple, and have few features
 - More designed for proofs than synchronization
- They lack many practical synchronization features
 - It is easy to deadlock with semaphores
 - One cannot check the lock without blocking
 - They do not support reader/writer shared access
 - No way to recover from a wedged V operation
 - No way to deal with priority inheritance
- Nonetheless, most OSs support them

Locking to Solve High Level Synchronization Problems

- Mutexes and object level locking
- Problems with locking
- Solving the problems

Mutexes

- A Linux/Unix locking mechanism
- Intended to lock sections of code
 - Locks expected to be held briefly
- Typically for multiple threads of the same process
- Low overhead and very general

Object Level Locking

- Mutexes protect code critical sections
 - Brief durations (e.g. nanoseconds, milliseconds)
 - Other threads operating on the same data
 - All operating in a single address space
- Persistent objects are more difficult
 - Critical sections are likely to last much longer
 - Many different programs can operate on them
 - May not even be running on a single computer
- Solution: lock objects (rather than code)
 - Typically somewhat specific to object type

Linux File Descriptor Locking

int flock(fd, *operation*)

- Supported *operations*:
 - LOCK_SH ... shared lock (multiple allowed)
 - LOCK_EX ... exclusive lock (one at a time)
 - LOCK_UN ... release a lock
- Lock applies to open instances of same *fd*
 - Distinct opens are not affected
- Locking is purely advisory
 - Does not prevent reads, writes, unlinks

Advisory vs Enforced Locking

- Enforced locking
 - Done within the implementation of object methods
 - Guaranteed to happen, whether or not user wants it
 - May sometimes be too conservative
- Advisory locking
 - A convention that “good guys” are expected to follow
 - Users expected to lock object before calling methods
 - Gives users flexibility in what to lock, when
 - Gives users more freedom to do it wrong (or not at all)
 - Mutexes are advisory locks

Linux Ranged File Locking

int lockf(*fd, cmd, offset, len*)

- Supported *cmds*:
 - F_LOCK ... get/wait for an exclusive lock
 - F_ULOCK ... release a lock
 - F_TEST/F_TLOCK ... test, or non-blocking request
 - *offset/len* specifies portion of file to be locked
- Lock applies to file (not the open instance)
 - Distinct opens are not affected
- Locking may be enforced
 - Depending on the underlying file system

Locking Problems

- Performance and overhead
- Contention
 - Convoy formation
 - Priority inversion

Performance of Locking

- Locking typically performed as an OS system call
 - Particularly for enforced locking
- Typical system call overheads for lock operations
- If they are called frequently, high overheads
- Even if not in OS, extra instructions run to lock and unlock

Locking Costs

- Locking called when you need to protect critical sections to ensure correctness
- Many critical sections are very brief
 - In and out in a matter of nano-seconds
- Overhead of the locking operation may be much higher than time spent in critical section

What If You Don't Get Your Lock?

- Then you block
- Blocking is much more expensive than getting a lock
 - E.g., 1000x
 - Micro-seconds to yield, context switch
 - Milliseconds if swapped-out or a queue forms
- Performance depends on conflict probability

$$C_{\text{expected}} = (C_{\text{block}} * P_{\text{conflict}}) + (C_{\text{get}} * (1 - P_{\text{conflict}}))$$

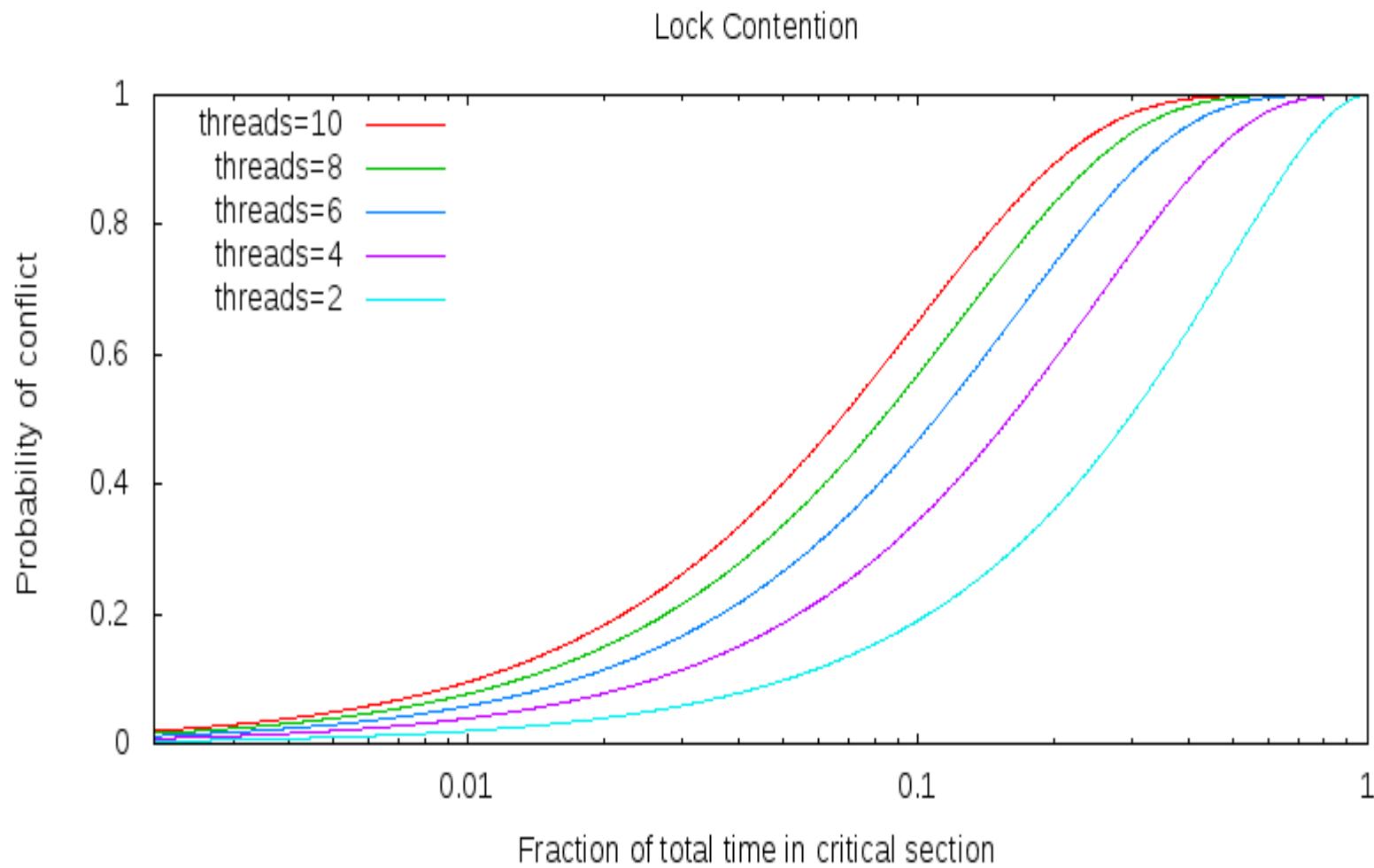
The Riddle of Parallelism

- Parallelism allows better overall performance
 - If one task is blocked, CPU runs another
 - So you must be able to run another
- But concurrent use of shared resources is difficult
 - So we protect critical sections for those resources by locking
- But critical sections serialize tasks
 - Meaning other tasks are blocked
- Which eliminates parallelism

What If Everyone Needs One Resource?

- One process gets the resource
- Other processes get in line behind him
 - Forming a *convoy*
 - Processes in a convoy are all blocked waiting for the resource
- Parallelism is eliminated
 - B runs after A finishes
 - C after B
 - And so on, with only one running at a time
- That resource becomes a *bottleneck*

Probability of Conflict



Convoy Formation

- In general

$$P_{\text{conflict}} = 1 - (1 - (T_{\text{critical}} / T_{\text{total}}))^{\text{threads}}$$

(nobody else in critical section at the same time)

- Unless a FIFO queue forms

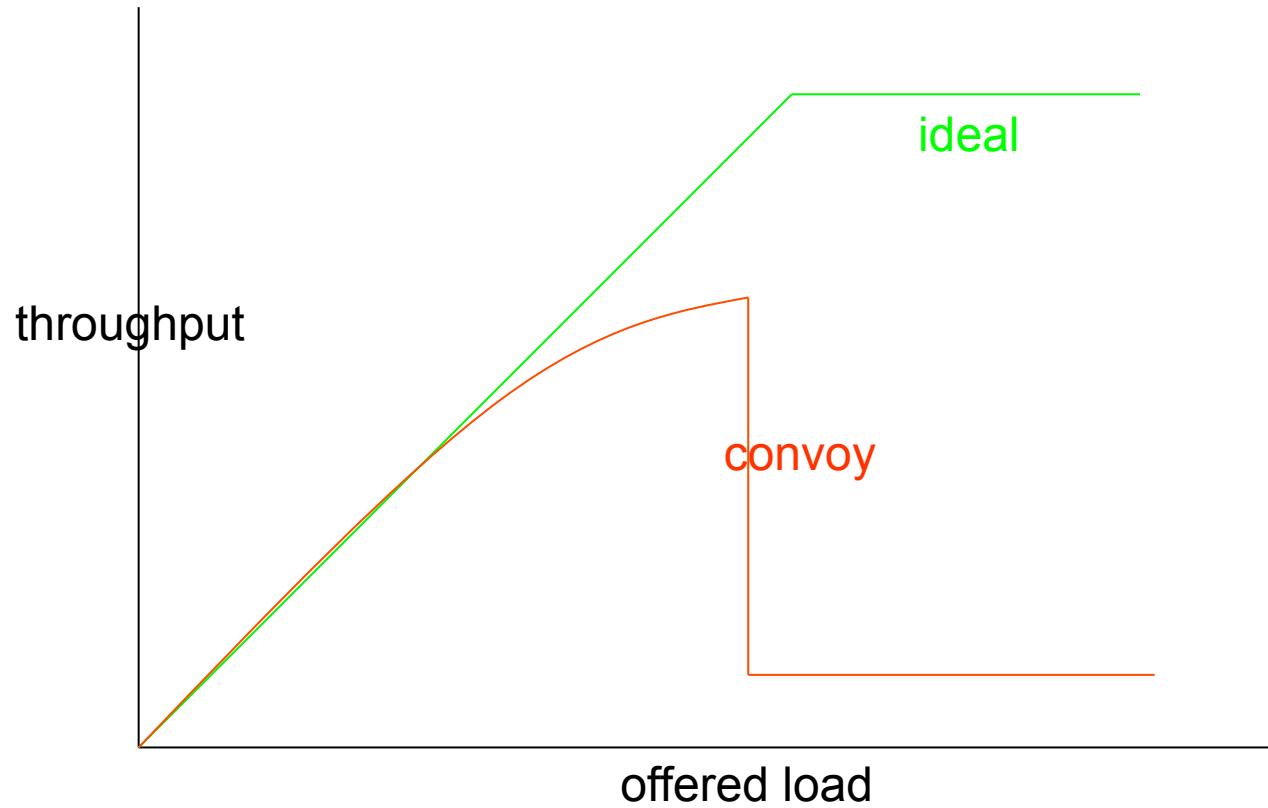
$$P_{\text{conflict}} = 1 - (1 - ((T_{\text{wait}} + T_{\text{critical}}) / T_{\text{total}}))^{\text{threads}}$$

Newcomers have to get into line

And an (already huge) T_{wait} gets even longer

- If T_{wait} reaches the mean inter-arrival time
The line becomes permanent, parallelism ceases

Performance: Resource Convoys



Priority Inversion

- Priority inversion can happen in priority scheduling systems that use locks
 - A low priority process P1 has mutex M1 and is preempted
 - A high priority process P2 blocks for mutex M1
 - Process P2 is effectively reduced to priority of P1
- Depending on specifics, results could be anywhere from inconvenient to fatal

Priority Inversion on Mars



- A real priority inversion problem occurred on the Mars Pathfinder rover
- Caused serious problems with system resets
- Difficult to find

The Pathfinder Priority Inversion

- Special purpose hardware running VxWorks real time OS
- Used preemptive priority scheduling
 - So a high priority task should get the processor
- Multiple components shared an “information bus”
 - Used to communicate between components
 - Essentially a shared memory region
 - Protected by a mutex

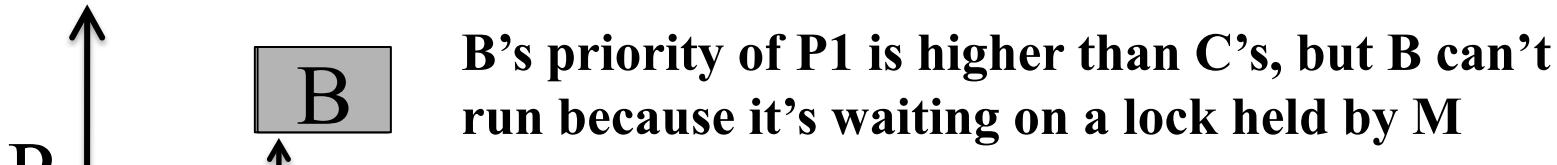
A Tale of Three Tasks

- A high priority bus management task (at P1) needed to run frequently
 - For brief periods, during which it locked the bus
- A low priority meteorological task (at P3) ran occasionally
 - Also for brief periods, during which it locked the bus
- A medium priority communications task (at P2) ran rarely
 - But for a long time when it ran
 - But it didn't use the bus, so it didn't need the lock
- P1>P2>P3

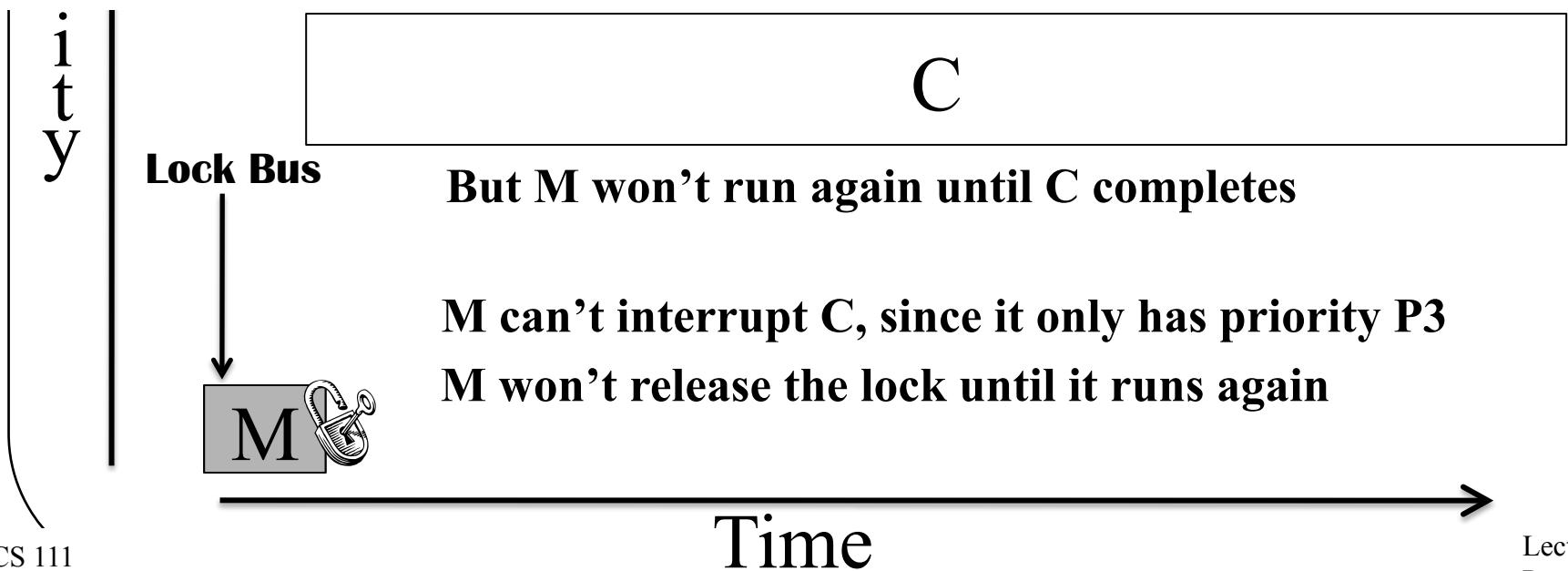
What Went Wrong?

- Rarely, the following happened:
 - The meteorological task ran and acquired the lock
 - And then the bus management task would run
 - It would block waiting for the lock
 - Don't pre-empt low priority if you're blocked anyway
- Since meteorological task was short, usually not a problem
- But if the long communications task woke up in that short interval, what would happen?

The Priority Inversion at Work



*A HIGH PRIORITY TASK DOESN'T RUN
AND A LOW PRIORITY TASK DOES*



The Ultimate Effect

- A watchdog timer would go off every so often
 - At a high priority
 - It didn't need the bus
 - A health monitoring mechanism
- If the bus management task hadn't run for a long time, something was wrong
- So the watchdog code reset the system
- Every so often, the system would reboot
- We'll get to the solution a bit later

Solving Locking Problems

- Reducing overhead
- Reducing contention
- Handling priority inversion

Reducing Overhead of Locking

- Not much more to be done here
- Locking code in operating systems is usually highly optimized
- Certainly typical users can't do better

Reducing Contention

- Eliminate the critical section entirely
 - Eliminate shared resource, use atomic instructions
- Eliminate preemption during critical section
- Reduce time spent in critical section
- Reduce frequency of entering critical section
- Reduce exclusive use of the serialized resource
- Spread requests out over more resources

Eliminating Critical Sections

- Eliminate shared resource
 - Give everyone their own copy
 - Find a way to do your work without it
- Use atomic instructions
 - Only possible for simple operations
- Great when you can do it
- But often you can't

Eliminate Preemption in Critical Section

- If your critical section cannot be preempted, no synchronization problems
- May require disabling interrupts
 - As previously discussed, not always an option

Reducing Time in Critical Section

- Eliminate potentially blocking operations
 - Allocate required memory before taking lock
 - Do I/O before taking or after releasing lock
- Minimize code inside the critical section
 - Only code that is subject to destructive races
 - Move all other code out of the critical section
 - Especially calls to other routines
- Cost: this may complicate the code
 - Unnaturally separating parts of a single operation

Reducing Time in Critical Section

```
int List_Insert(list_t *l, int key) {  
    pthread_mutex_lock(&l->lock);  
    node_t new = (node_t*) malloc(sizeof(node_t));  
    if (new == NULL) {  
        perror("malloc");  
        pthread_mutex_unlock(&l->lock);  
        return(-1);  
    }  
    new->key = key;  
    new->next = l->head;  
    l->head = new;  
    pthread_mutex_unlock(&l->lock);  
    return 0;  
}
```

```
int List_Insert(list_t *l, int key) {  
    node_t new = (node_t*) malloc(sizeof(node_t));  
    if (new == NULL) {  
        perror("malloc");  
        return(-1);  
    }  
    new->key = key;  
    pthread_mutex_lock(&l->lock);  
    new->next = l->head;  
    l->head = new;  
    pthread_mutex_unlock(&l->lock);  
    return 0;  
}
```

Reduced Frequency of Entering Critical Section

- Can we use critical section less often?
 - Less use of high-contention resource/operations
 - Batch operations
- Consider “sloppy counters”
 - Move most updates to a private resource
 - Costs:
 - Global counter is not always up-to-date
 - Thread failure could lose many updates
 - Alternative:
 - Sum single-writer private counters when needed

Remove Requirement for Full Exclusivity

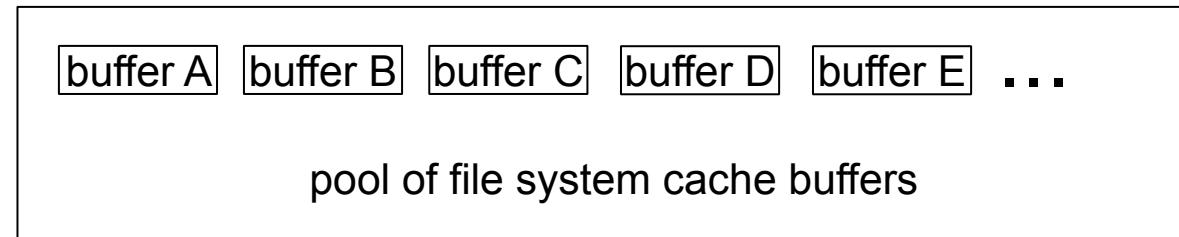
- Read/write locks
- Reads and writes are not equally common
 - File read/write: reads/writes > 50
 - Directory search/create: reads/writes > 1000
- Only writers require exclusive access
- Read/write locks
 - Allow many readers to share a resource
 - Only enforce exclusivity when a writer is active
 - Policy: when are writers allowed in?
 - Potential starvation if writers must wait for readers

Spread Requests Over More Resources

- Change lock granularity
- Coarse grained - one lock for many objects
 - Simpler, and more idiot-proof
 - Greater resource contention (threads/resource)
- Fine grained - one lock per object (or sub-pool)
 - Spreading activity over many locks reduces contention
 - Dividing resources into pools shortens searches
 - A few operations may lock multiple objects/pools
- TANSTAAFL
 - Time/space overhead, more locks, more gets/releases
 - Error-prone: harder to decide what to lock when

Lock Granularity – Pools vs. Elements

- Consider a pool of objects, each with its own lock



- Most operations lock only one buffer within the pool
- But some operations require locking the entire pool
 - Two threads both try to add block AA to the cache
 - Thread 1 looks for block B while thread 2 is deleting it
- The pool lock could become a bottle-neck, so
 - Minimize its use
 - Reader/writer locking
 - Sub-pools ...

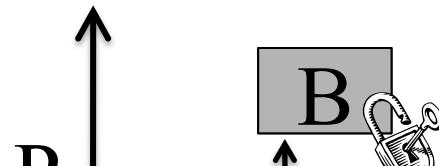
Handling Priority Inversion Problems

- In a priority inversion, lower priority task runs because of a lock held elsewhere
 - Preventing the higher priority task from running
- In the Mars Rover case, the meteorological task held a lock
 - A higher priority bus management task couldn't get the lock
 - A medium priority, but long, communications task preempted the meteorological task
 - So the medium priority communications task ran instead of the high priority bus management task

Solving Priority Inversion

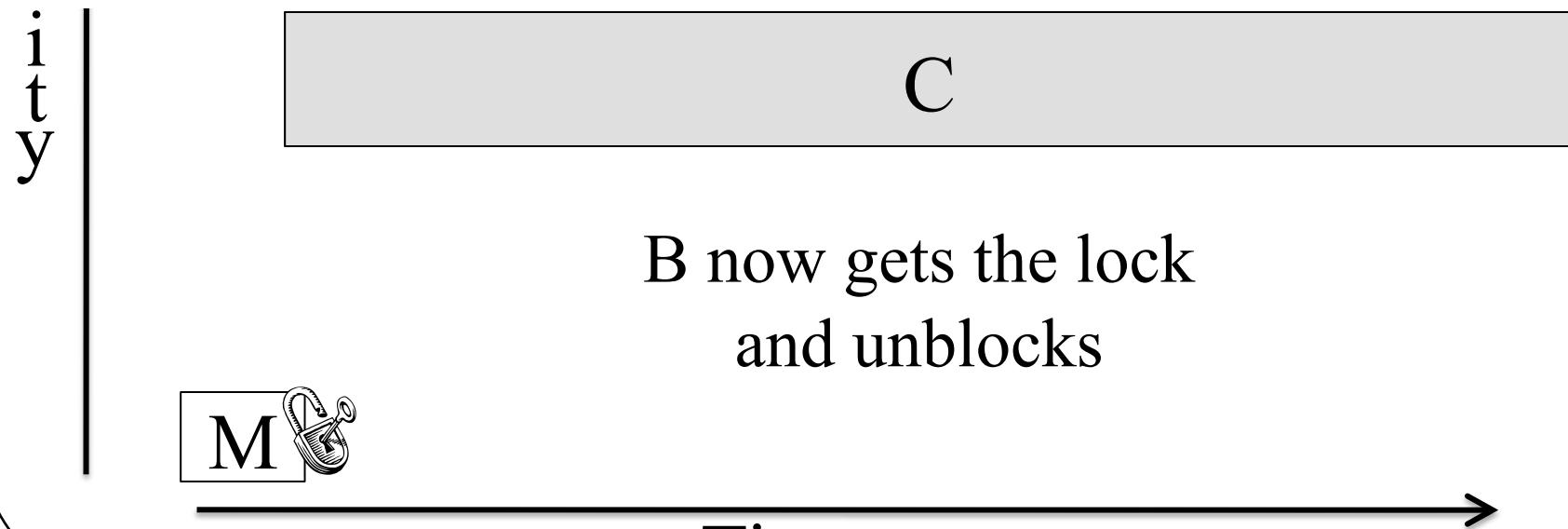
- Temporarily increase the priority of the meteorological task
 - While the high priority bus management task was blocked by it
 - So the communications task wouldn't preempt it
 - When lock is released, drop meteorological task's priority back to normal
- *Priority inheritance*: a general solution to this kind of problem

The Fix in Action



When M releases the
lock it loses high

*Tasks run in proper priority order and
Pathfinder can keep looking around!*



The Snake in the Garden

- Locking is great for preventing improper concurrence
- With care locks can be made to perform very well
- But that can lead to problems
- If we are not careful locking can lead to ou
- Deadlock



Operating System Principles: Deadlocks – Problems and Solutions

CS 111

Operating Systems
Peter Reiher

Outline

- The deadlock problem
 - Approaches to handling the problem
- Handling general synchronization bugs
- Simplifying synchronization

Deadlock

- What is a deadlock?
- A situation where two entities have each locked some resource
- Each needs the other's locked resource to continue
- Neither will unlock till they lock both resources
- Hence, neither can ever make progress

The Dining Philosophers Problem

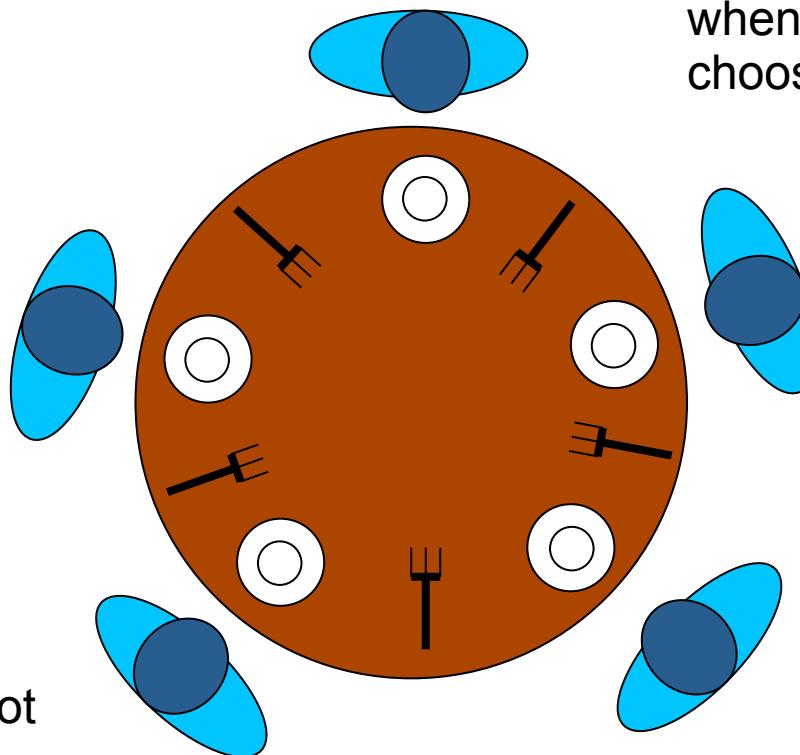
Five philosophers
five plates of pasta
five forks

A philosopher needs
two forks to eat
pasta, but must pick
them up one at a time

Philosophers will not
negotiate with
one-another

Philosophers eat
whenever they
choose to

The problem
demands an
absolute solution



Dining Philosophers and Deadlock

- This problem is the classical illustration of deadlock
- It was created to illustrate deadlock problems
- It is a very artificial problem
 - It was carefully designed to cause deadlocks
 - Changing the rules eliminate deadlocks
 - But then it couldn't be used to illustrate deadlocks
 - Actually, one point of it is to see how changing the rules solves the problem

Why Are Deadlocks Important?

- A major peril in cooperating parallel processes
 - They are relatively common in complex applications
 - They result in catastrophic system failures
- Finding them through debugging is very difficult
 - They happen intermittently and are hard to diagnose
 - They are much easier to prevent at design time
- Once you understand them, you can avoid them
 - Most deadlocks result from careless/ignorant design
 - An ounce of prevention is worth a pound of cure

Deadlocks May Not Be Obvious

- Process resource needs are ever-changing
 - Depending on what data they are operating on
 - Depending on where in computation they are
 - Depending on what errors have happened
- Modern software depends on many services
 - Most of which are ignorant of one-another
 - Each of which requires numerous resources
- Services encapsulate much complexity
 - We do not know what resources they require
 - We do not know when/how they are serialized

Deadlocks and Different Resource Types

- Commodity Resources
 - Clients need an amount of it (e.g. memory)
 - Deadlocks result from over-commitment
 - Avoidance can be done in resource manager
- General Resources
 - Clients need a specific instance of something
 - A particular file or semaphore
 - A particular message or request completion
 - Deadlocks result from specific dependency relationships
 - Prevention is usually done at design time

Four Basic Conditions For Deadlocks

- For a deadlock to occur, these conditions must hold:
 1. Mutual exclusion
 2. Incremental allocation
 3. No pre-emption
 4. Circular waiting

Deadlock Conditions: 1. Mutual Exclusion

- The resources in question can each only be used by one entity at a time
- If multiple entities can use a resource, then just give it to all of them
- If only one can use it, once you've given it to one, no one else gets it
 - Until the resource holder releases it

Deadlock Condition 2: Incremental Allocation

- Processes/threads are allowed to ask for resources whenever they want
 - As opposed to getting everything they need before they start
- If they must pre-allocate all resources, either:
 - They get all they need and run to completion
 - They don't get all they need and abort
- In either case, no deadlock

Deadlock Condition 3: No Pre-emption

- When an entity has reserved a resource, you can't take it away from him
 - Not even temporarily
- If you can, deadlocks are simply resolved by taking someone's resource away
 - To give to someone else
- But if you can't take it away from anyone, you're stuck

Deadlock Condition 4: Circular Waiting

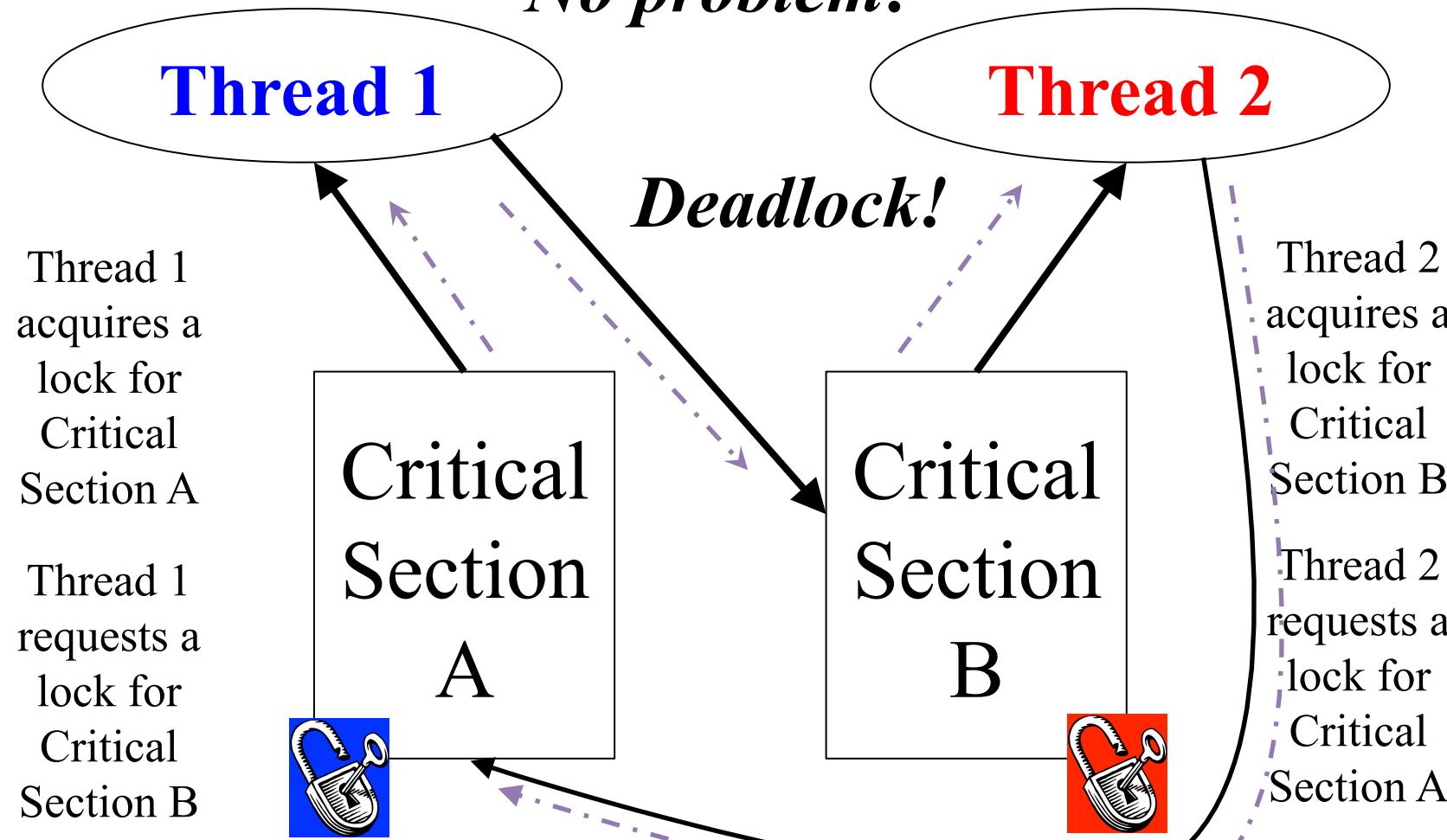
- A waits on B which waits on A
- In graph terms, there's a cycle in a graph of resource requests
- Could involve a lot more than two entities
- But if there is no such cycle, someone can complete without anyone releasing a resource
 - Allowing even a long chain of dependencies to eventually unwind
 - Maybe not very fast, though . . .

We can't give him
the lock right now,
but . . .

A Wait-For Graph

Hmmmm . . .

No problem!



Deadlock Avoidance

- Use methods that guarantee that no deadlock can occur, by their nature
- Advance reservations
 - The problems of under/over-booking
 - The Bankers' Algorithm
- Practical commodity resource management
- Dealing with rejection
- Reserving critical resources

Avoiding Deadlock Using Reservations

- Advance reservations for commodity resources
 - Resource manager tracks outstanding reservations
 - Only grants reservations if resources are available
- Over-subscriptions are detected early
 - Before processes ever get the resources
- Client must be prepared to deal with failures
 - But these do not result in deadlocks
- Dilemma: over-booking vs. under-utilization

Overbooking Vs. Under Utilization

- Processes generally cannot perfectly predict their resource needs
- To ensure they have enough, they tend to ask for more than they will ever need
- Either the OS:
 - Grants requests till everything's reserved
 - In which case most of it won't be used
 - Or grants requests beyond the available amount
 - In which case sometimes someone won't get a resource he reserved

Handling Reservation Problems

- Clients seldom need all resources all the time
- All clients won't need max allocation at the same time
- Question: can one safely over-book resources?
 - For example, seats on an airplane
- What is a “safe” resource allocation?
 - One where everyone will be able to complete
 - Some people may have to wait for others to complete
 - We must be sure there are no deadlocks

Commodity Resource Management in Real Systems

- Advanced reservation mechanisms are common
 - Memory reservations
 - Disk quotas, Quality of Service contracts
- Once granted, system must guarantee reservations
 - Allocation failures only happen at reservation time
 - Hopefully before the new computation has begun
 - Failures will not happen at request time
 - System behavior more predictable, easier to handle
- But clients must deal with reservation failures

Dealing With Reservation Failures

- Resource reservation eliminates deadlock
- Apps must still deal with reservation failures
 - Application design should handle failures gracefully
 - E.g., refuse to perform new request, but continue running
 - App must have a way of reporting failure to requester
 - E.g., error messages or return codes
 - App must be able to continue running
 - All critical resources must be reserved at start-up time

Isn't Rejecting App Requests Bad?

- It's not great, but it's better than failing later
- With advance notice, app may be able to adjust service not to need the unavailable resource
- If app is in the middle of servicing a request, we may have other resources allocated
 - And the request half-performed
 - If we fail then, all of this will have to be unwound
 - Could be complex, or even impossible

System Services and Reservations

- System services must never deadlock for memory
- Potential deadlock: swap manager
 - Invoked to swap out processes to free up memory
 - May need to allocate memory to build I/O request
 - If no memory available, unable to swap out processes
 - So it can't free up memory, and system wedges
- Solution:
 - Pre-allocate and hoard a few request buffers
 - Keep reusing the same ones over and over again
 - Little bit of hoarded memory is a small price to pay to avoid deadlock
- That's just one example system service, of course

Deadlock Prevention

- Deadlock avoidance tries to ensure no lock ever causes deadlock
- Deadlock prevention tries to assure that a particular lock doesn't cause deadlock
- By attacking one of the four necessary conditions for deadlock
- If any one of these conditions doesn't hold, no deadlock

Four Basic Conditions For Deadlocks

- For a deadlock to occur, these conditions must hold:
 1. Mutual exclusion
 2. Incremental allocation
 3. No pre-emption
 4. Circular waiting

1. Mutual Exclusion

- Deadlock requires mutual exclusion
 - P1 having the resource precludes P2 from getting it
- You can't deadlock over a shareable resource
 - Perhaps maintained with atomic instructions
 - Even reader/writer locking can help
 - Readers can share, writers may be handled other ways
- You can't deadlock on your private resources
 - Can we give each process its own private resource?

2. Incremental Allocation

- Deadlock requires you to block holding resources while you ask for others
 - 1. Allocate all of your resources in a single operation
 - If you can't get everything, system returns failure and locks nothing
 - When you return, you have all or nothing
 - 2. Non-blocking requests
 - A request that can't be satisfied immediately will fail
 - 3. Disallow blocking while holding resources
 - You must release all held locks prior to blocking
 - Reacquire them again after you return

Releasing Locks Before Blocking

- Could be blocking for a reason not related to resource locking
- How can releasing locks before you block help?
- Won't the deadlock just occur when you attempt to reacquire them?
 - When you reacquire them, you will be required to do so in a single all-or-none transaction
 - Such a transaction does not involve hold-and-block, and so cannot result in a deadlock

3. No Pre-emption

- Deadlock can be broken by resource confiscation
 - Resource “leases” with time-outs and “lock breaking”
 - Resource can be seized & reallocated to new client
- Revocation must be enforced
 - Invalidate previous owner's resource handle
 - If revocation is not possible, kill previous owner
- Some resources may be damaged by lock breaking
 - Previous owner was in the middle of critical section
 - May need mechanisms to audit/repair resource
- Resources must be designed with revocation in mind

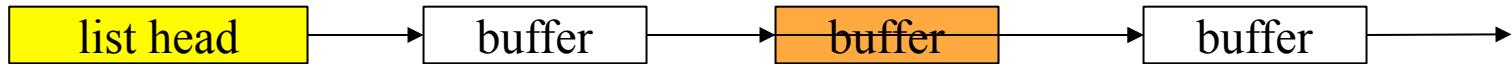
When Can The OS “Seize” a Resource?

- When it can revoke access by invalidating a process' resource handle
 - If process has to use a system service to access the resource, that service can no longer honor requests
- When is it not possible to revoke a process' access to a resource?
 - If the process has direct access to the object
 - E.g., the object is part of the process' address space
 - Revoking access requires destroying the address space
 - Usually killing the process.

4. Circular Dependencies

- Use *total resource ordering*
 - All requesters allocate resources in same order
 - First allocate R1 and then R2 afterwards
 - Someone else may have R2 but he doesn't need R1
- Assumes we know how to order the resources
 - Order by resource type (e.g. groups before members)
 - Order by relationship (e.g. parents before children)
- May require a *lock dance*
 - Release R2, allocate R1, reacquire R2

Lock Dances



list head must be locked for searching, adding & deleting

individual buffers must be locked to perform I/O & other operations

To avoid deadlock, we must always lock the list head before we lock an individual buffer.

To find a desired buffer:

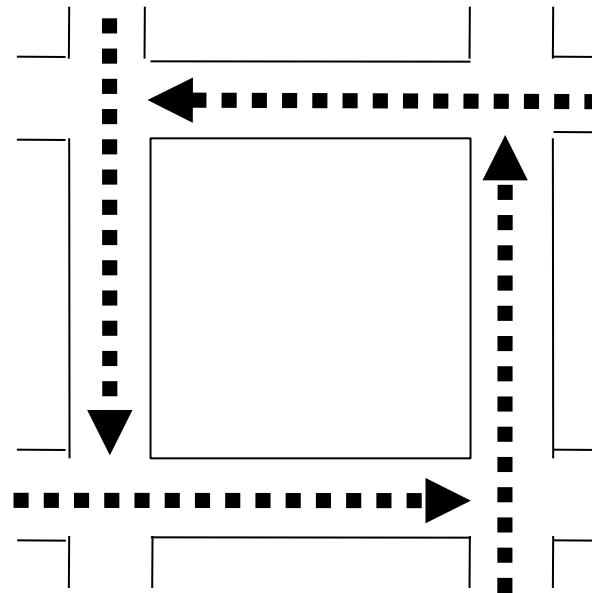
- read lock list head
- search for desired buffer
- lock desired buffer
- unlock list head
- return (locked) buffer

To delete a (locked) buffer from list

- unlock buffer**
- write lock list head
- search for desired buffer**
- lock desired buffer**
- remove from list
- unlock list head

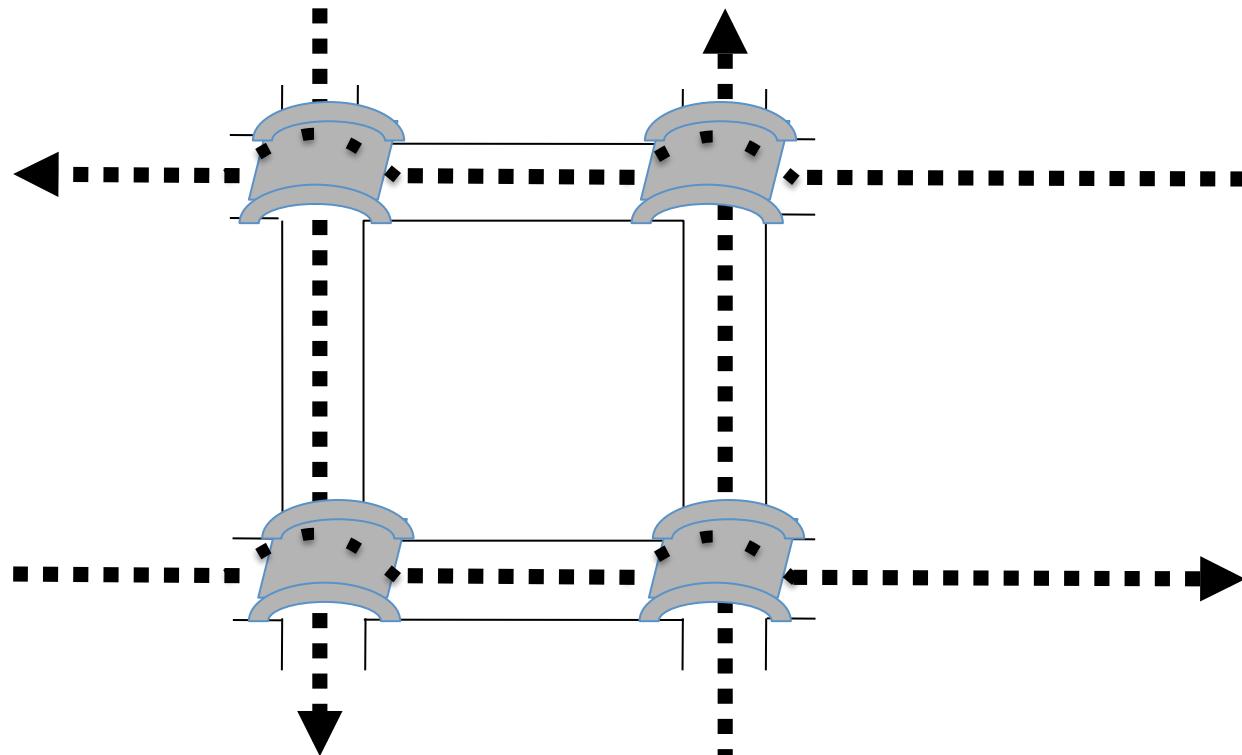
An Example of Breaking Deadlocks

- The problem – urban traffic gridlock
 - “Resource” is the ability to pass through intersection
 - Deadlock happens when nobody can get through



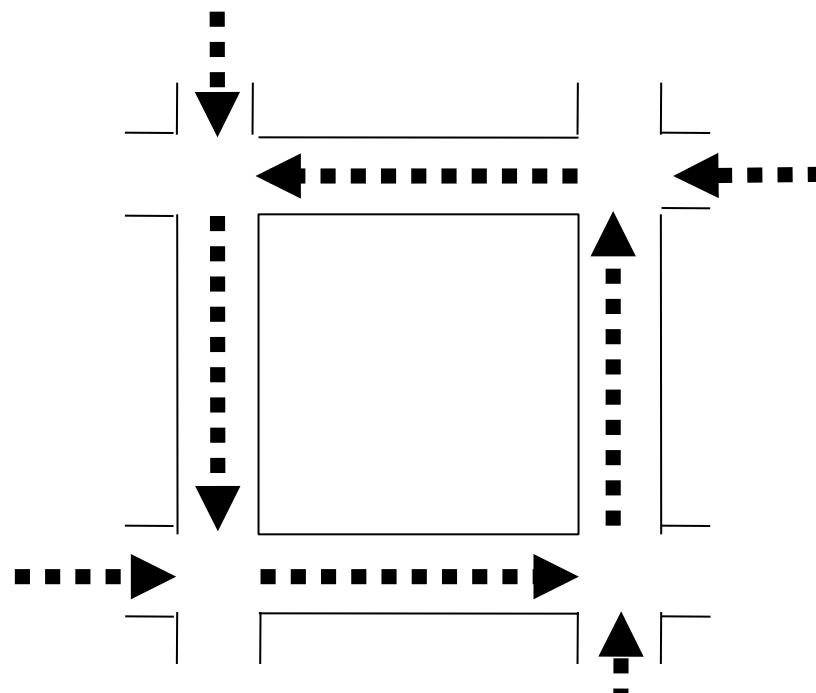
Using Attack Approach 1 To Prevent Deadlock

- Avoid mutual exclusion
- Build overpass bridges for east/west traffic



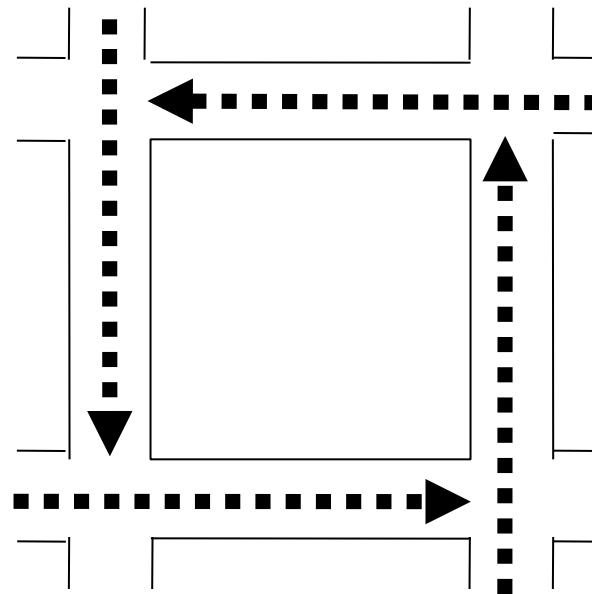
Using Attack Approach 2 To Prevent Deadlock

- Make it illegal to enter the intersection if you can't exit it
 - Thus, preventing “holding” of the intersection



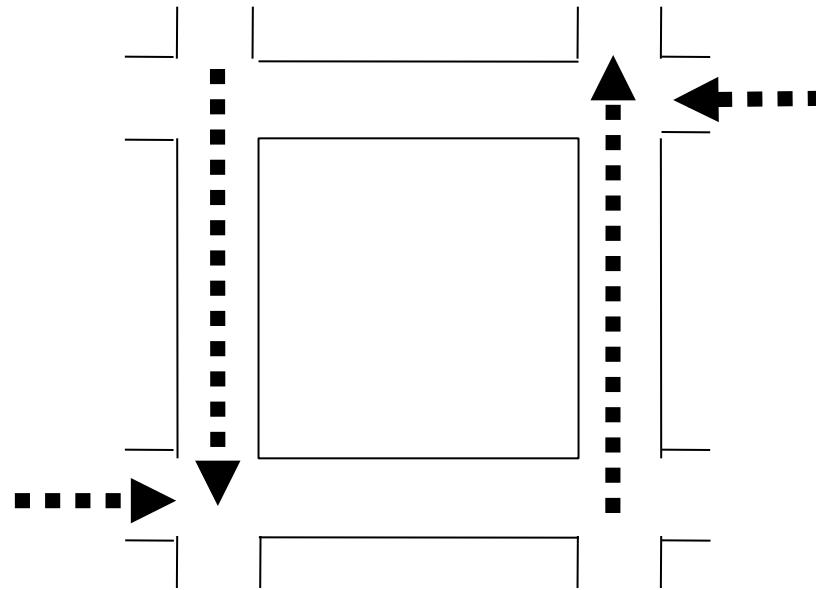
Using Attack Approach 3 To Prevent Deadlock

- Allow preemption
 - Force some car to pull over to the side



Using Attack Approach 4 To Prevent Deadlock

- Avoid circular dependencies by decreeing a totally ordered right of way
 - E.g., North beats West beats South beats East



Which Approach Should You Use?

- There is no one universal solution to all deadlocks
 - Fortunately, we don't need one solution for all resources
 - We only need a solution for each resource
- Solve each individual problem any way you can
 - Make resources sharable wherever possible
 - Use reservations for commodity resources
 - Ordered locking or no hold-and-block where possible
 - As a last resort, leases and lock breaking
- OS must prevent deadlocks in all system services
 - Applications are responsible for their own behavior

One More Deadlock “Solution”

- Ignore the problem
- In many cases, deadlocks are very improbable
- Doing anything to avoid or prevent them might be very expensive
- So just forget about them and hope for the best
- But what if the best doesn’t happen?

Deadlock Detection and Recovery

- Allow deadlocks to occur
- Detect them once they have happened
 - Preferably as soon as possible after they occur
- Do something to break the deadlock and allow someone to make progress
- Is this a good approach?
 - Either in general or when you don't want to avoid or prevent deadlocks?

Implementing Deadlock Detection

- Need to identify all resources that can be locked
- Need to maintain wait-for graph or equivalent structure
- When lock requested, structure is updated and checked for deadlock
 - In which case, might it not be better just to reject the lock request?
 - And not let the requester block?

Dealing With General Synchronization Bugs

- Deadlock detection seldom makes sense
 - It is extremely complex to implement
 - Only detects true deadlocks for a known resource
 - Not always clear cut what you should do if you detect one
- Service/application *health monitoring* is better
 - Monitor application progress/submit test transactions
 - If response takes too long, declare service “hung”
- Health monitoring is easy to implement
- It can detect a wide range of problems
 - Deadlocks, live-locks, infinite loops & waits, crashes

Related Problems Health Monitoring Can Handle

- Live-lock
 - Process is running, but won't free R1 until it gets message
 - Process that will send the message is blocked for R1
- Sleeping Beauty, waiting for “Prince Charming”
 - A process is blocked, awaiting some completion that will never happen
- Priority inversion hangs
 - Which we talked about before
- None of these is a true deadlock
 - Wouldn't be found by deadlock detection algorithm
 - All leave the system just as hung as a deadlock
- Health monitoring handles them

How To Monitor Process Health

- Look for obvious failures
 - Process exits or core dumps
- Passive observation to detect hangs
 - Is process consuming CPU time, or is it blocked?
 - Is process doing network and/or disk I/O?
- External health monitoring
 - “Pings”, null requests, standard test requests
- Internal instrumentation
 - White box audits, exercisers, and monitoring

What To Do With “Unhealthy” Processes?

- Kill and restart “all of the affected software”
- How many and which processes to kill?
 - As many as necessary, but as few as possible
 - The hung processes may not be the ones that are broken
- How will kills and restarts affect current clients?
 - That depends on the service APIs and/or protocols
 - Apps must be designed for cold/warm/partial restarts
- Highly available systems define restart groups
 - Groups of processes to be started/killed as a group
 - Define inter-group dependencies (restart B after A)

Failure Recovery Methodology

- Retry if possible ... but not forever
 - Client should not be kept waiting indefinitely
 - Resources are being held while waiting to retry
- Roll-back failed operations and return an error
- Continue with reduced capacity or functionality
 - Accept requests you can handle, reject those you can't
- Automatic restarts (cold, warm, partial)
- Escalation mechanisms for failed recoveries
 - Restart more groups, reboot more machines

Making Synchronization Easier

- Locks, semaphores, mutexes are hard to use correctly
 - Might not be used when needed
 - Might be used incorrectly
 - Might lead to deadlock, livelock, etc.
- We need to make synchronization easier for programmers
 - But how?

One Approach

- We identify shared resources
 - Objects whose methods may require serialization
- We write code to operate on those objects
 - Just write the code
 - Assume all critical sections will be serialized
- Complier generates the serialization
 - Automatically generated locks and releases
 - Using appropriate mechanisms
 - Correct code in all required places

Monitors – Protected Classes

- Each monitor class has a semaphore
 - Automatically acquired on method invocation
 - Automatically released on method return
 - Automatically released/acquired around CV waits
- Good encapsulation
 - Developers need not identify critical sections
 - Clients need not be concerned with locking
 - Protection is completely automatic
- High confidence of adequate protection

Monitors: Use

```
monitor CheckBook {  
    // class is locked when any method is invoked  
    private int balance;  
    public int balance() {  
        return(balance);  
    }  
    public int debit(int amount) {  
        balance -= amount;  
        return( balance)  
    }  
}
```

Monitors: Simplicity vs. Performance

- Monitor locking is very conservative
 - Lock the entire class (not merely a specific object)
 - Lock for entire duration of any method invocations
- This can create performance problems
 - They eliminate conflicts by eliminating parallelism
 - If a thread blocks in a monitor a convoy can form
- TANSTAAFL
 - Fine-grained locking is difficult and error prone
 - Coarse-grained locking creates bottle-necks

Evaluating Monitors

- Correctness
 - Complete mutual exclusion is assured
- Fairness
 - Semaphore queue prevents starvation
- Progress
 - Inter-class dependencies can cause deadlocks
- Performance
 - Coarse grained locking is not scalable

Java Synchronized Methods

- Each object has an associated mutex
 - Acquired before calling a synchronized method
 - Nested calls (by same thread) do not reacquire
 - Automatically released upon final return
- Static synchronized methods lock class mutex
- Advantages
 - Finer lock granularity, reduced deadlock risk
- Costs
 - Developer must identify serialized methods

Using Java Synchronized Methods

```
class CheckBook {  
    private int balance;  
    public int balance() {  
        return(balance);  
    }  
    // object is locked when this method is invoked  
    public synchronized int debit(int amount) {  
        balance -= amount;  
        return( balance)  
    }  
}
```

Evaluating Java Synchronized Methods

- Correctness
 - Correct if developer chose the right methods
- Fairness
 - Priority thread scheduling (potential starvation)
- Progress
 - Safe from single thread deadlocks
- Performance
 - Fine grained (per object) locking
 - Selecting which methods to synchronize

Operating System Principles: Performance Measurement and Analysis

CS 111

Operating Systems
Peter Reiher

Outline

- Introduction to performance measurement
- Issues in performance measurement
- A performance measurement example

Performance Measurement

- Performance is almost always a key issue in software
- Especially in system software like operating systems
- Everyone wants the best possible performance
 - But achieving it is not always easy
 - And sometimes involves trading off other desirable qualities
- How can we know what performance we've achieved?
 - Especially given that we must do some work to learn that

Performance Analysis Goals

- Quantify the system performance
 - For competitive positioning
 - To assess the efficacy of previous work
 - To identify future opportunities for improvement
- Understand the system performance
 - What factors are limiting our current performance
 - What choices make us subject to these limitations
- Predict system performance

An Overarching Goal

- This applies to any performance analysis you ever do:
- **We seek wisdom, not numbers!**
- The point is never to produce a spreadsheet full of data
- The point is to understand critical performance issues

Why Are You Measuring Performance?

- Sometimes to understand your system's behavior
- Sometimes to compare to other systems
- Sometimes to investigate alternatives
 - In how you can configure or manage your system
- Sometimes to determine how your system will (or won't) scale up
- Sometimes to find the cause of performance problems

Why Is It Hard?

- Components operate in a complex system
 - Many steps/components in every process
 - Ongoing competition for all resources
 - Difficulty of making clear/simple assertions
 - Systems may be too large to replicate in laboratory
 - Or have other non-reproducible properties
- Lack of clear/rigorous requirements
 - Performance is highly dependent on specifics
 - What we measure, how we measure it
 - Ask the wrong question, get the wrong answer

Performance Analysis

- Can you characterize latency and throughput?
 - Of the system?
 - Of each major component?
- Can you account for all the end-to-end time?
 - Processing, transmission, queuing delays
- Can you explain how these vary with load?
- Are there any significant unexplained results?
- Can you predict the performance of a system?
 - As a function of its configuration/parameters

Design For Performance Measurement

- Successful systems will need to have their performance measured
- Becoming a successful system will generally require that you improve its performance
 - Which implies measuring it
- It's best to assume your system will need to be measured
- So put some forethought into making it easy

How To Design for Performance

- Establish performance requirements early
- Anticipate bottlenecks
 - Frequent operations (interrupts, copies, updates)
 - Limiting resources (network/disk bandwidth)
 - Traffic concentration points (resource locks)
- Design to minimize problems
 - Eliminate, reduce use, add resources
- Include performance measurement in design
 - What will be measured, and how

Issues in Performance Measurement

- Performance measurement terminology
- Types of performance problems

Some Important Measurement Terminology

- Metrics
 - Indices of tendency and dispersion
- Factors and levels
- Workloads

Metrics

- A metric is a measurable quantity
 - Measurable: we can observe it in situations of interest
 - Quantifiable: time/rate, size/capacity, effectiveness/reliability ...
- A metric's value should describe an important phenomenon in a system
 - Relevant to the questions we are addressing
- Much of performance evaluation is about properly evaluating metrics

Common Types of System Metrics

- Duration/ response time
 - How long did the program run?
- Processing rate
 - How many web requests handled per second?
- Resource consumption
 - How much disk is currently used?
- Reliability
 - How many messages were delivered without error?

Choosing Your Metrics

- Core question in any performance study
- Pick metrics based on:
 - Completeness: will my metrics cover everything I need to know?
 - (Non-)redundancy: does each metric provide information not provided by others?
 - Variability: will this metric show meaningful variation?
 - Feasibility: can I accurately measure this metric?

Variability in Metrics

- Performance of a system is often complex
- Perhaps not fully explainable
- One result is variability in many metric readings
 - You measure it twice/thrice/more and get different results every time
- Good performance measurement takes this into account

An Example

- 11 pings from UCLA to MIT in one night
- Each took a different amount of time
(expressed in msec):

149.1	28.1	28.1	28.5	28.6	28.2
28.4	187.8	74.3	46.1	155.8	

- How do we understand what this says about how long a packet takes to get from LA to Boston and back?

Where Does Variation Come From?

- Inconsistent test conditions
 - Varying platforms, operations, injection rates
 - Background activity on test platform
 - Start-up, accumulation, cache effects
- Flawed measurement choices/techniques
 - Measurement artifact, sampling errors
 - Measuring indirect/aggregate effects
- Non-deterministic factors
 - Queuing of processes, network and disk I/O
 - Where (on disk) files are allocated

Tendency and Dispersion

- Given variability in metric readings, how do we understand what they tell us?
- Tendency
 - What is common or characteristic of all readings?
- Dispersion
 - How much do the various measurements of the metric vary?
- Good performance experiments capture and report both

Indices of Tendency

- What can we compactly say that sheds light on all of the values observed?
- Some example indices of tendency:
 - Mean ... the average of all samples
 - Median ... the value of the middle sample
 - Mode ... the most commonly occurring value
- Each of these tells us something different, so which we use depends on our goals

Applied to Our Example Ping Data

- Mean: 71.2
 - Median: 28.6
 - Mode: 28.1
 - Which of these best expresses the delay we saw?
 - Depends on what you care about
- | | | | | | |
|-------|-------|------|------|-------|------|
| 149.1 | 28.1 | 28.1 | 28.5 | 28.6 | 28.2 |
| 28.4 | 187.8 | 74.3 | 46.1 | 155.8 | |

Indices of Dispersion

- Compact descriptions of how much variation we observed in our measurements
 - Among the values of particular metrics under supposedly identical conditions
- Some examples:
 - Range – the high and low values observed
 - Standard deviation – statistical measure of common deviations from a mean
 - Coefficient of variance – ratio of standard deviation to mean
- Again, choose the index that describes what's important for the goal under examination

Applied to Our Ping Data Example

- Range: 28.1,188
- Standard deviation: 62.0
- Coefficient of variation: .87

149.1	28.1	28.1	28.5	28.6	28.2
28.4	187.8	74.3	46.1	155.8	

Capturing Variation

- Generally requires repetition of the same experiment
- Ideally, sufficient repetitions to capture all likely outcomes
 - How do you know how many repetitions that is?
 - You don't
- Design your performance measurements bearing this in mind

Meaningful Measurements

- Measure under controlled conditions
 - On a specified platform
 - Under a controlled and calibrated load
 - Removing as many extraneous external influences as possible
- Measure the right things
 - Direct measurements of key characteristics
- Ensure quality of results
 - Competing measurements we can cross-compare
 - Measure/correct for artifacts
 - Quantify repeatability/variability of results

Factors and Levels

- Sometimes we only want to measure one thing
- More commonly, we are interested in several alternatives
 - What if I doubled the memory?
 - What if work came in twice as fast?
 - What if I used a different file system?
- Such controlled variations for comparative purposes are called *factors*

Factors in Experiments

- Choose factors related to your experiment goals
- If you care about web server scaling, factors probably related to amount of work offered
- If you want to know which file system works best for you, factor is likely to be different file systems
- If you're deciding how to partition a disk, factor is likely to be different partitionings

Levels

- Factors vary (by definition)
- Levels describe which values you test for each factor
- Levels can thus be numerical
 - Number of web requests applied per second
 - Amount of memory devoted to I/O buffers
- Or they can be categorical
 - Btrfs vs. Ext3 vs. XFS

Choosing Factors and Levels

- Your experiment should look at all vital factors
- Each factor should be examined at important levels
- But . . .
- The effort involved in the experiment is related to (number of factors) X (number of levels)
- If you're not careful, this can cause your effort to explode
 - Especially if you repeat runs to capture variation

Measurement Workloads

- Most measurement programs require the use of a *workload*
- Some kind of work applied to the system you are testing
 - Preferably similar to the work you care about
- Can be of several different forms
 - Simulated workloads
 - Replayed trace
 - Live workload
 - Standard benchmarks

Simulated Work Loads

- Artificial load generation
 - On-demand generation of a specified load
- Strengths
 - Controllable operation rates, parameters, mixes
 - Scalable to produce arbitrarily large loads
 - Can collect excellent performance data
- Weaknesses
 - Random traffic is not a usage scenario
 - Simulation may not create all realistic situations
 - Wrong parameter choices yield unrealistic loads

Replayed Workloads

- Captured operations from real systems
- Strengths
 - Represent real usage scenarios
 - Can be analyzed and replayed over and over
- Weakness
 - Often hard to obtain
 - Not necessarily scalable
 - Multiple instances not equivalent to more users
 - Represent a limited set of possible behaviors
 - Limited ability to exercise little-used features
 - They are kept around forever, and become stale

Testing Under Live Loads

- Instrumented systems serving clients
- Strengths
 - Real combinations of real scenarios
 - Measured against realistic background loads
 - Enables collection of data on real usage
- Weakness
 - Demands good performance and reliability
 - Potentially limited testing opportunities
 - Load cannot be repeated or scaled on demand

Standard Benchmarks

- Carefully crafted/reviewed simulators
- Strengths
 - Heavily reviewed by developers and customers
 - Believed to be representative of real usage
 - Standardized and widely available
 - Well maintained (bugs, currency, improvements)
 - Allows comparison of competing products
- Weakness
 - Inertia, used where they are not applicable

Types of Performance Problems

- Non-scalable solutions
 - Cost per operation becomes prohibitive at scale
 - Worse-than-linear overheads and algorithms
 - Queuing delays associated with high utilization
- Bottlenecks
 - One component that limits system throughput
- Accumulated costs
 - Layers of calls, data copies, message exchanges
 - Redundant or unnecessary work

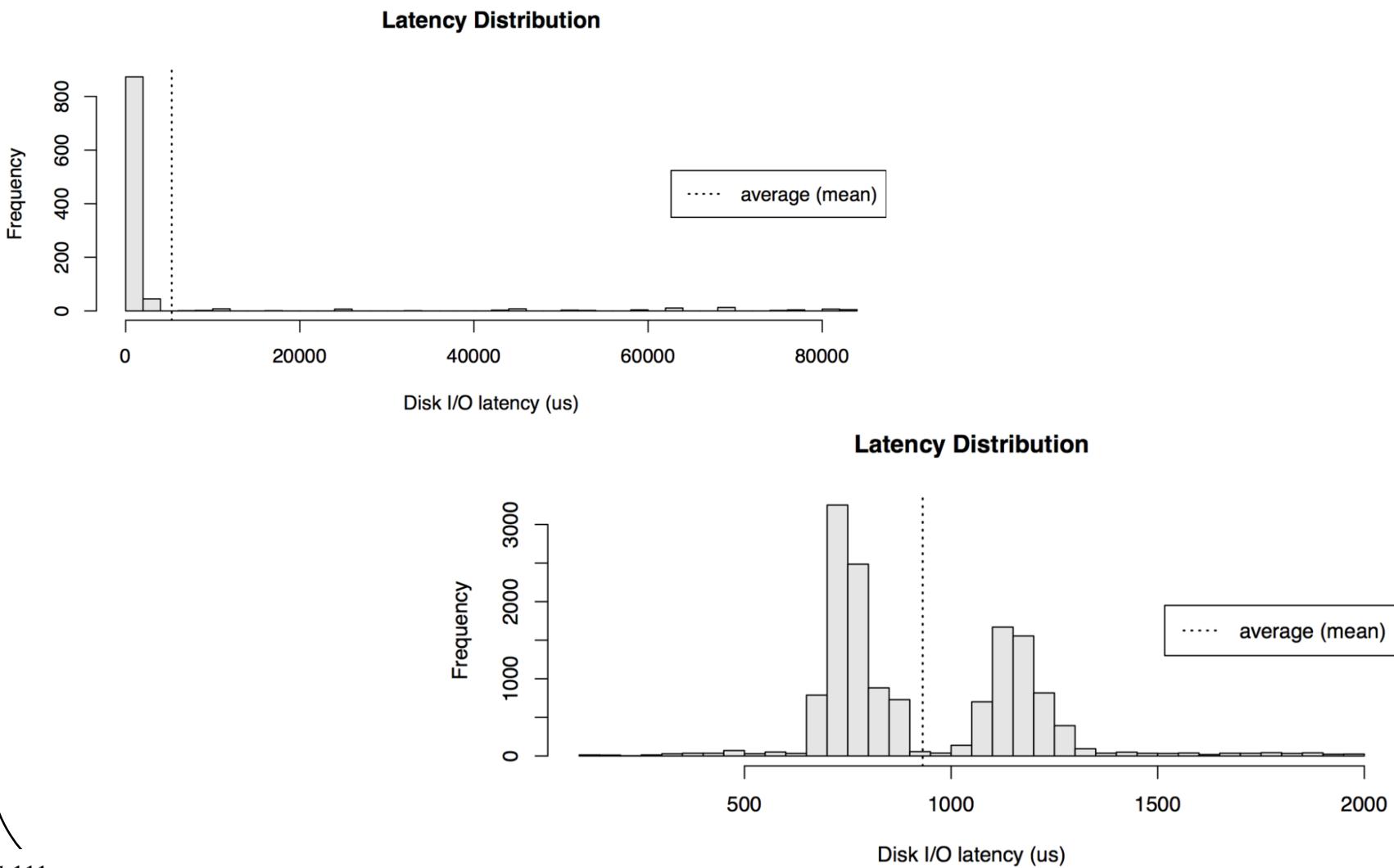
Dealing With Performance Problems

- A lot like finding and fixing a bug
 - Formulate a hypothesis
 - Gather data to verify your hypothesis
 - Be sure you understand underlying problem
 - Review proposed solutions
 - For effectiveness
 - For potential side effects
 - Make simple changes, one at a time
 - Re-measure to confirm effectiveness of each
- Only harder

Common Measurement Mistakes

- Measuring time but not utilization
 - Everything is fast on a lightly loaded system
- Capturing averages rather than distributions
 - Outliers are usually interesting
- Ignoring start-up, accumulation, cache effects
 - Not measuring what we thought
- Ignoring instrumentation artifacts
 - They may greatly distort both times and loads

Averages Don't Tell the Story



Cache, Accumulation Start-up Effects

- Cached results may accelerate some runs
 - Random requests that are unlikely to be in cache
 - Overwhelm cache with new data between tests
 - Disable or bypass cache entirely
- Start-up costs distort total cost of computation
 - Do all start-up ops prior to starting actual test
 - Long test runs to amortize start-up effects
 - Measure and subtract start-up costs
- System performance may degrade with age
 - Reestablish base condition for each test

Measurement Artifacts

- Costs of instrumentation code
 - Additional calls, instructions, cache misses
 - Additional memory consumption and paging
 - Costs of logging results
 - May dwarf the costs of instrumentation
 - Increased disk load/latency may slow everything
- Minimize frequency and costs of measuring
- Don't measure everything always
 - Counters/accumulators instead of individual records
 - In-memory circular buffer, reduce before writing to files
 - Probabilistic methods that don't execute on each occurrence

Measurement Tools

- Execution profiling
- Event logs
- End-to-end testing

Execution Profiling

- Automated measurement tools
 - Compiler options for routine call counting
 - One counter per routine, incremented on entry
 - Statistical execution sampling
 - Timer interrupts execution at regular intervals
 - Increment a counter in table based on PC value
 - May have configurable time/space granularity
 - Tools to extract data and prepare reports
 - Number of calls, time per call, percentage of time
- Very useful in identifying the bottlenecks

Time Stamped Event Logs

- Application instrumentation technique
- Create a log buffer and routine
 - Call log routine for all interesting events
 - Routine stores time and event in a buffer
 - Requires a cheap, very high resolution timer
- Extract buffer, archive, mine the data
 - Time required for particular operations
 - Frequency of operations
 - Combinations of operations
 - Also useful for post-mortem analysis

Time Stamping

Dump of simple trace log

datetime	event	sub-type	
-----	-----	-----	-----
05/11/06 09:02:31.207408	packet_rcv	0x20749329	
05/11/06 09:02:31.209301	packet_route	0x20749329	
05/11/06 09:02:31.305208	wakeup	0x4D8C2042	
05/11/06 09:02:31.401106	read_packet	0x033C2DA0	
05/11/06 09:02:31.401223	read_packet	0x033C2DA0	
05/11/06 09:02:31.402110	sleep	0x4D8C2042	
05/11/06 09:02:31.614209	interrupt	0x00000003	
05/11/06 09:02:31.614209	dispatch	0x1B0324C0	
05/11/06 09:02:31.614210	intr_return	0x00000003	
05/11/06 09:02:31.652303	check_queue	0x2D3F2040	
05/11/06 09:02:31.652306	packet_rcv	0x20749329	

End-to-End Testing

- Client-side throughput/latency measurements
 - Elapsed time for X operations of type Y
 - Instrumented clients to collect detailed timings
- Strengths
 - Easy tests to run, easy data to analyze
 - Results reflect client experienced performance
- Weaknesses
 - No information about why it took that long
 - No information about resources consumed

A Performance Measurement Example

- The Conquest file system
 - A research system built by one of my students
- Using persistent RAM to store many files
 - Which allowed him to get rid of a lot of OS code related to disk drives
- Stored some files on disk
 - Which we won't worry about here
- Expectation was better performance than disk-based file systems

How Did We Measure Conquest?

- What were the metrics?
- What were the factors?
- What was the workload?
- What were the results?

Choosing the Metrics

- Core claim was better speed
- So metrics should be speed-related
- Speeding up overall file system operations was the goal
 - Not speeding up an isolated operation
- So we needed metrics capturing that
- We used several “operations per second” metrics
 - Reads, writes, creates, also bandwidth

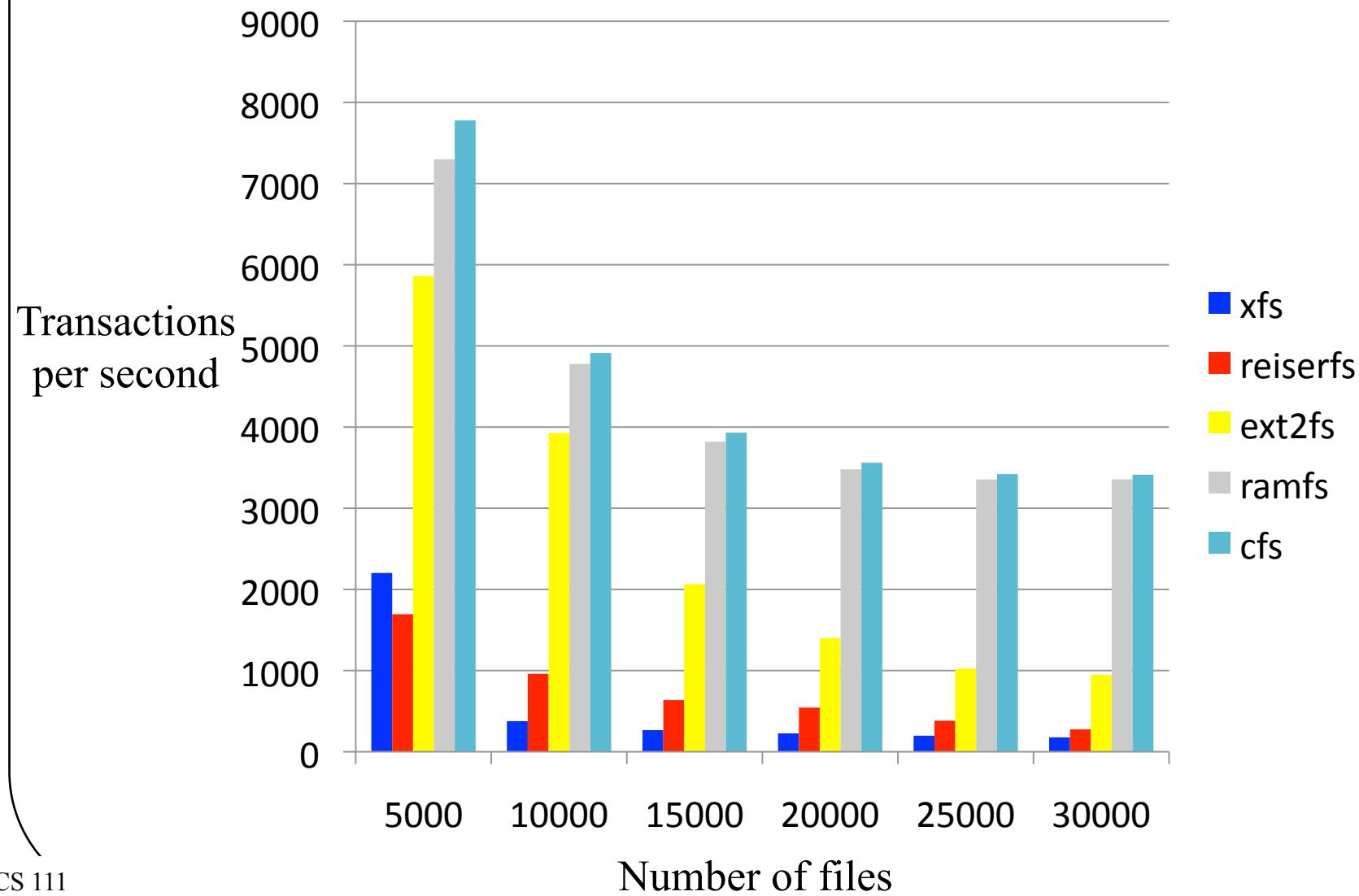
Choosing the Factors

- We were claiming better performance than other file systems
- So one factor was which file system we tested
- We also wanted to show scaling effects
 - It didn't perform well just for tiny systems
- So another factor chosen was number of files in the file system

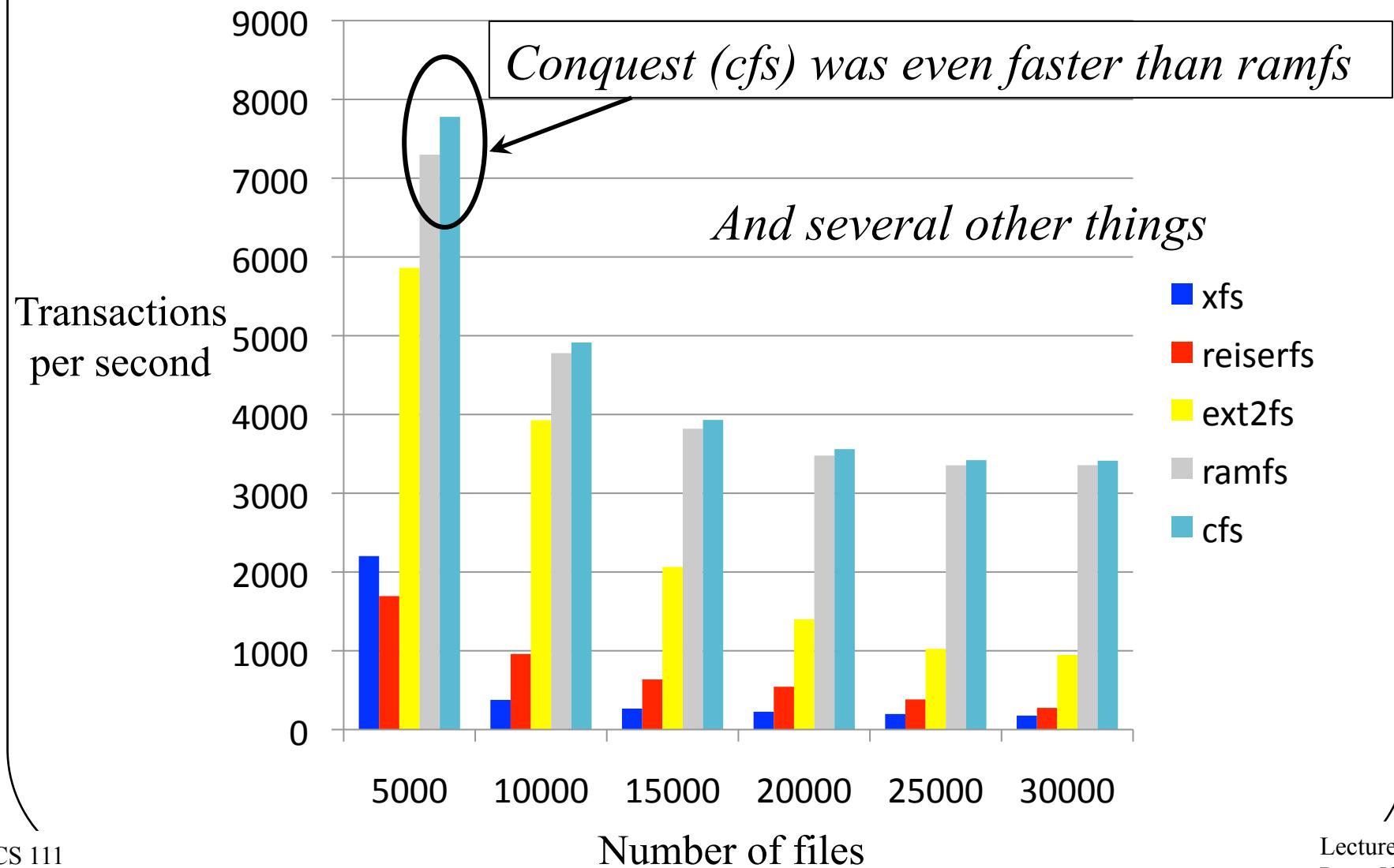
Choosing the Workload

- File systems are traditionally tested against standard benchmarks
- We tested against several of those
- One benchmark we used is called Postmark
- Postmark performs various “transactions” related to file operations
- The metric we’ll show is Postmark transactions per second

One Set of Results



Which Showed What?



A Couple of Words on Presentation

- Always consider these questions:
 1. To whom am I speaking?
 - What they know and not know?
 - What are they prepared to absorb, and what not?
 2. Why are they listening to me?
 - How might this help them achieve their goals?
 - How might this address their concerns?
 3. What do I want them to leave with?
 - What conclusions do I want them to draw?
 - What actions do I want them to take?

Performance Presentation

- Highlight the key results
 - Answers to the basic questions
 - Identified problems, risks and opportunities
- Why should they believe these results
 - Methodology employed, relation to other results
 - Back-up details
- Not just numbers, but explanations
 - How do we now better understand the system
 - How does this affect our plans and intentions

Operating System Principles: Devices, Device Drivers, and I/O

CS 111

Operating Systems

Peter Reiher

Outline

- Devices and device drivers
- I/O performance issues
- Device driver abstractions

So You've Got Your Computer . . .

It's got memory, a bus,
a CPU or two

But there's usually a lot
more to it than that



Welcome to the Wonderful World of Peripheral Devices!

- Our computers typically have lots of devices attached to them
- Each device needs to have some code associated with it
 - To perform whatever operations it does
 - To integrate it with the rest of the system
- In modern commodity OSes, the code that handles these devices dwarfs the rest

Peripheral Device Code and the OS

- Why are peripheral devices the OS' problem, anyway?
- Why can't they be handled in user-level code?
- Maybe they sometimes can, but . . .
- Some of them are critical for system correctness
 - E.g., the disk drive holding swap space
- Some of them must be shared among multiple processes
 - Which is often rather complex
- Some of them are security-sensitive
- Perhaps more appropriate to put the code in the OS

Where the Device Driver Fits in

- At one end you have an application
 - Like a web browser
- At the other end you have a very specific piece of hardware
 - Like an Intel Gigabit CT PCI-E Network Adapter
- In between is the OS
- When the application sends a packet, the OS needs to invoke the proper device driver
- Which feeds detailed instructions to the hardware

Device Drivers

- Generally, the code for these devices is pretty specific to them
- It's basically code that *drives* the device
 - Makes the device perform the operations it's designed for
- So typically each system device is represented by its own piece of code
- The *device driver*
- A Linux 2.6 kernel came with over 3200 of them . . .

Typical Properties of Device Drivers

- Highly specific to the particular device
 - System only needs drivers for devices it hosts
- Inherently modular
- Usually interacts with the rest of the system in limited, well defined ways
- Their correctness is critical
 - Device behavior correctness and overall correctness
- Generally written by programmers who understand the device well
 - But are not necessarily experts on systems issues

Abstractions and Device Drivers

- OS defines idealized device classes
 - Disk, display, printer, tape, network, serial ports
- Classes define expected interfaces/behavior
 - All drivers in class support standard methods
- Device drivers implement standard behavior
 - Make diverse devices fit into a common mold
 - Protect applications from device eccentricities
- Abstractions regularize and simplify the chaos of the world of devices

What Can Driver Abstractions Help With?

- Encapsulate knowledge of how to use the device
 - Map standard operations into operations on device
 - Map device states into standard object behavior
 - Hide irrelevant behavior from users
 - Correctly coordinate device and application behavior
- Encapsulate knowledge of optimization
 - Efficiently perform standard operations on a device
- Encapsulate fault handling
 - Understanding how to handle recoverable faults
 - Prevent device faults from becoming OS faults

How Do Device Drivers Fit Into a Modern OS?

- There may be a lot of them
- They are each pretty independent
- You may need to add new ones later
- So a pluggable model is typical
- OS provides capabilities to plug in particular drivers in well defined ways
- Then plug in the ones a given machine needs
- Making it easy to change or augment later

Layering Device Drivers

- The interactions with the bus, down at the bottom, are pretty standard
 - How you address devices on the bus, coordination of signaling and data transfers, etc.
 - Not too dependent on the device itself
- The interactions with the applications, up at the top, are also pretty standard
 - Typically using some file-oriented approach
- In between are some very device specific things

A Pictorial View

User space

System Call

App 1

App 2

App 3

Kernel space

Device Call

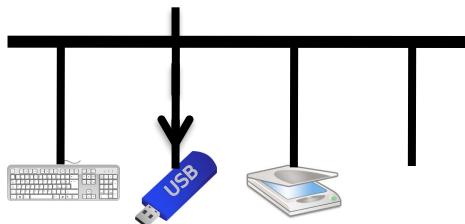
USB bus controller

Device Drivers

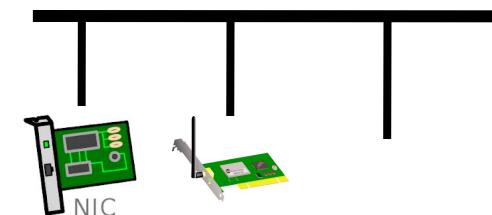
PCI bus controller

Hardware

USB bus



PCI bus



Device Drivers Vs. Core OS Code

- Device driver code is in the OS, but . . .
- What belongs in core OS vs. a device driver?
- Common functionality belongs in the OS
 - Caching
 - File systems code not tied to a specific device
 - Network protocols above physical/link layers
- Specialized functionality belongs in the drivers
 - Things that differ in different pieces of hardware
 - Things that only pertain to the particular piece of hardware

Devices and Interrupts

- Devices are primarily interrupt-driven
 - Drivers aren't schedulable processes
- They work at different speed than the CPU
 - Typically slower
- They can do their own work while CPU does something else
- They use interrupts to get the CPU's attention

Devices and Busses

- Devices are not connected directly to the CPU
- Both CPU and devices are connected to a bus
- Sometimes the same bus, sometimes a different bus
- Devices communicate with CPU across the bus
- Bus used both to send/receive interrupts and to transfer data and commands
 - Devices signal controller when they are done/ready
 - When device finishes, controller puts interrupt on bus
 - Bus then transfers interrupt to the CPU
 - Perhaps leading to movement of data

CPUs and Interrupts

- Interrupts look very much like traps
 - Traps come from CPU
 - Interrupts are caused externally to CPU
- Unlike traps, interrupts can be enabled/disabled by special CPU instructions
 - Device can be told when they may generate interrupts
 - Interrupt may be held *pending* until software is ready for it

The Changing I/O Landscape

- To quote a recent Nobel Prize winner, “the times they are a’changing”
- Storage paradigms
 - **Old**: swapping, paging, file systems, data bases
 - **New**: NAS, distributed object/key-value stores
- I/O traffic
 - **Old**: most I/O was disk I/O
 - **New**: network and video dominate many systems
- Performance goals:
 - **Old**: maximize throughput, IOPS
 - **New**: low latency, scalability, reliability, availability

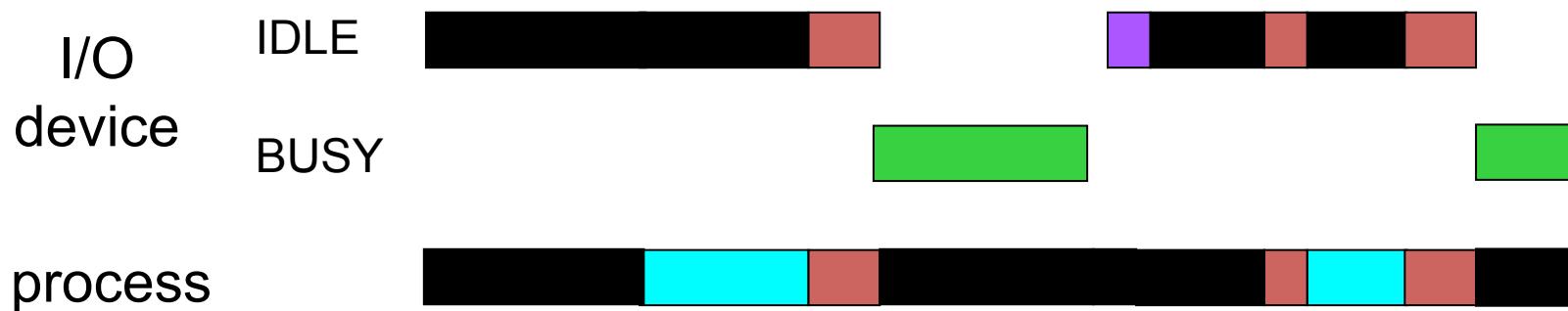
Device Performance

- The importance of good device utilization
- How to achieve good utilization

Good Device Utilization

- Key system devices limit system performance
 - File system I/O, swapping, network communication
- If device sits idle, its throughput drops
 - This may result in lower system throughput
 - Longer service queues, slower response times
- Delays can disrupt real-time data flows
 - Resulting in unacceptable performance
 - Possible loss of irreplaceable data
- It is very important to keep key devices busy
 - Start request $n+1$ immediately when n finishes

Poor I/O Device Utilization



1. process waits to run
2. process does computation in preparation for I/O operation
3. process issues read system call, blocks awaiting completion
4. device performs requested operation
5. completion interrupt awakens blocked process
6. process runs again, finishes read system call
7. process does more computation
8. Process issues read system call, blocks awaiting completion

How To Do Better

- The usual way:
 - Exploit parallelism
- Devices operate independently of the CPU
- So a device and the CPU can operate in parallel
- But often devices need to access RAM
 - As does the CPU
- How to handle that?

What's Really Happening on the CPU?

- Modern CPUs try to avoid going to RAM
 - Working with registers
 - Caching on the CPU chip itself
- If things go well, the CPU doesn't use the memory bus that much
 - If not, life will be slow, anyway
- So one way to parallelize activities is to let a device use the bus instead of the CPU

Direct Memory Access (DMA)

- Allows any two devices attached to the memory bus to move data directly
 - Without passing it through the CPU first
- Bus can only be used for one thing at a time
- So if it's doing DMA, it's not servicing CPU requests
- But often the CPU doesn't need it, anyway
- With DMA, data moves from device to memory at bus/device/memory speed

Keeping Key Devices Busy

- Allow multiple requests to be pending at a time
 - Queue them, just like processes in the ready queue
 - Requesters block to await eventual completions
- Use DMA to perform the actual data transfers
 - Data transferred, with no delay, at device speed
 - Minimal overhead imposed on CPU
- When the currently active request completes
 - Device controller generates a completion interrupt
 - OS accepts interrupt and calls appropriate handler
 - Interrupt handler posts completion to requester
 - Interrupt handler selects and initiates next transfer

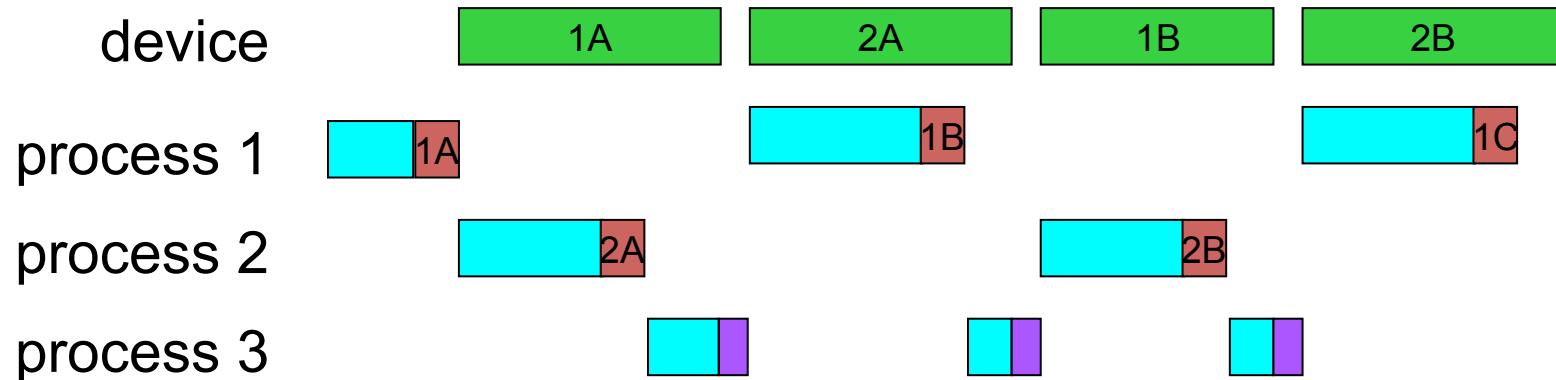
Interrupt Driven Chain Scheduled I/O

```
xx_read/write() {
    allocate a new request descriptor
    fill in type, address, count, location
    insert request into service queue
    if (device is idle) {
        disable_device_interrupt();
        xx_start();
        enable_device_interrupt();
    }
    await completion of request
    extract completion info for caller
}
```

```
xx_start() {
    get next request from queue
    get address, count, disk address
    load request parameters into controller
    start the DMA operation
    mark device busy
}
```

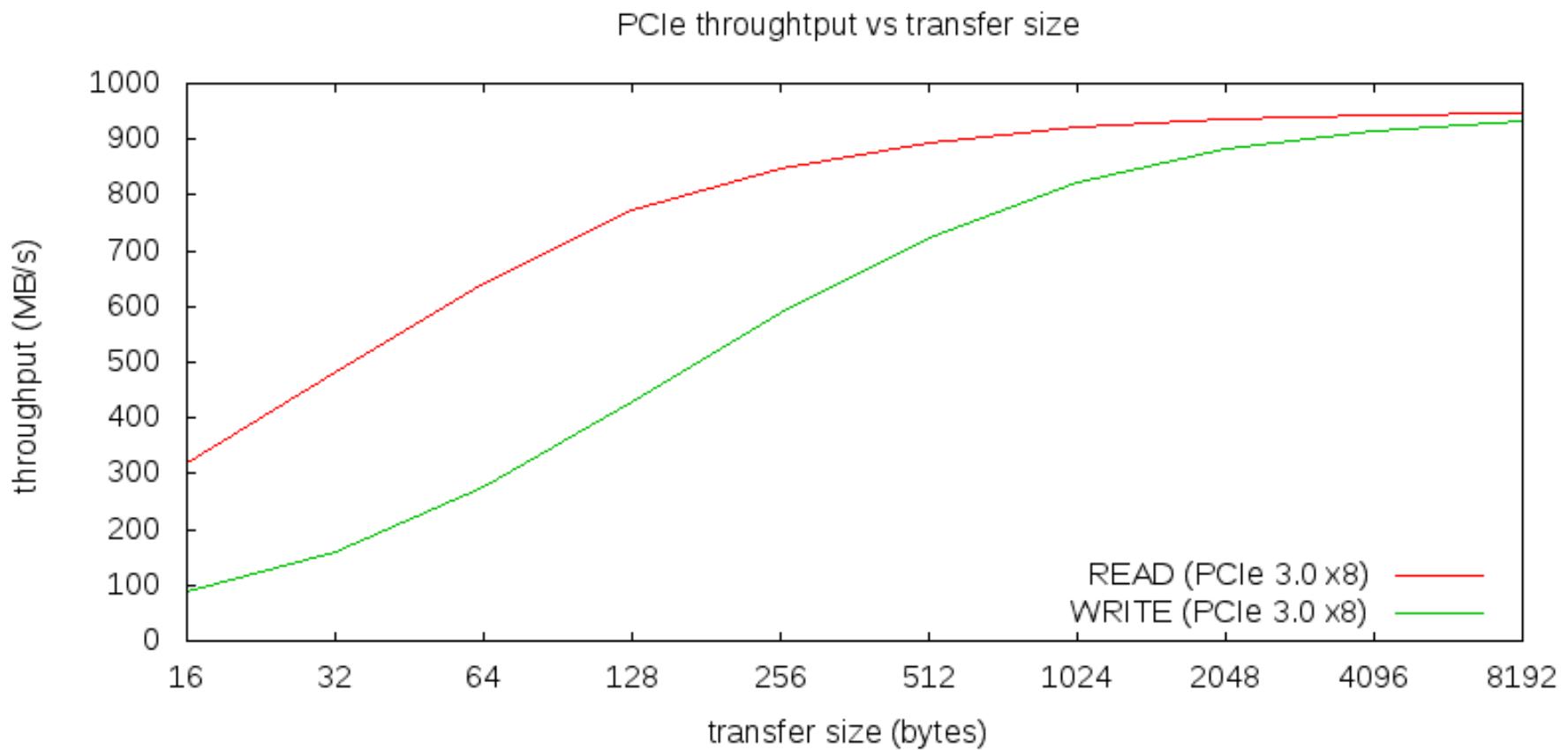
```
xx_intr() {
    extract completion info from controller
    update completion info in current req
    wakeup current request
    if (more requests in queue)
        xx_start()
    else
        mark device idle
}
```

Multi-Tasking & Interrupt Driven I/O



1. P₁ runs, requests a read, and blocks
2. P₂ runs, requests a read, and blocks
3. P₃ runs until interrupted
4. Awaken P₁ and start next read operation
5. P₁ runs, requests a read, and blocks
6. P₃ runs until interrupted
7. Awaken P₂ and start next read operation
8. P₂ runs, requests a read, and blocks
9. P₃ runs until interrupted
10. Awaken P₁ and start next read operation
11. P₁ runs, requests a read, and blocks

Bigger Transfers are Better



(Bigger Transfers are Better)

- Disks have high seek/rotation overheads
 - Larger transfers amortize down the cost/byte
- All transfers have per-operation overhead
 - Instructions to set up operation
 - Device time to start new operation
 - Time and cycles to service completion interrupt
- Larger transfers have lower overhead/byte
 - This is not limited to software implementations

I/O and Buffering

- Most I/O requests cause data to come into the memory or to be copied to a device
- That data requires a place in memory
 - Commonly called a buffer
- Data in buffers is ready to send to a device
- An existing empty buffer is ready to receive data from a device
- OS needs to make sure buffers are available when devices are ready to use them

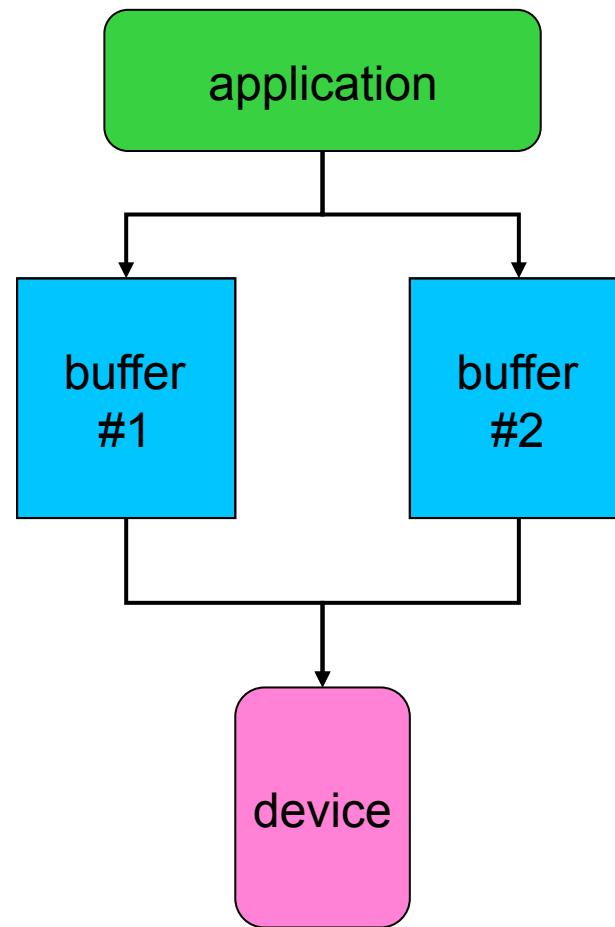
OS Buffering Issues

- Fewer/larger transfers are more efficient
 - They may not be convenient for applications
 - Natural record sizes tend to be relatively small
- Operating system can consolidate I/O requests
 - Maintain a cache of recently used disk blocks
 - Accumulate small writes, flush out as blocks fill
 - Read whole blocks, deliver data as requested
- Enables read-ahead
 - OS reads/caches blocks not yet requested

Deep Request Queues

- Having many I/O operations queued is good
 - Maintains high device utilization (little idle time)
 - Reduces mean seek distance/rotational delay
 - May be possible to combine adjacent requests
 - Can sometimes avoid performing a write at all
- Ways to achieve deep queues:
 - Many processes making requests
 - Individual processes making parallel requests
 - Read-ahead for expected data requests
 - Write-back cache flushing

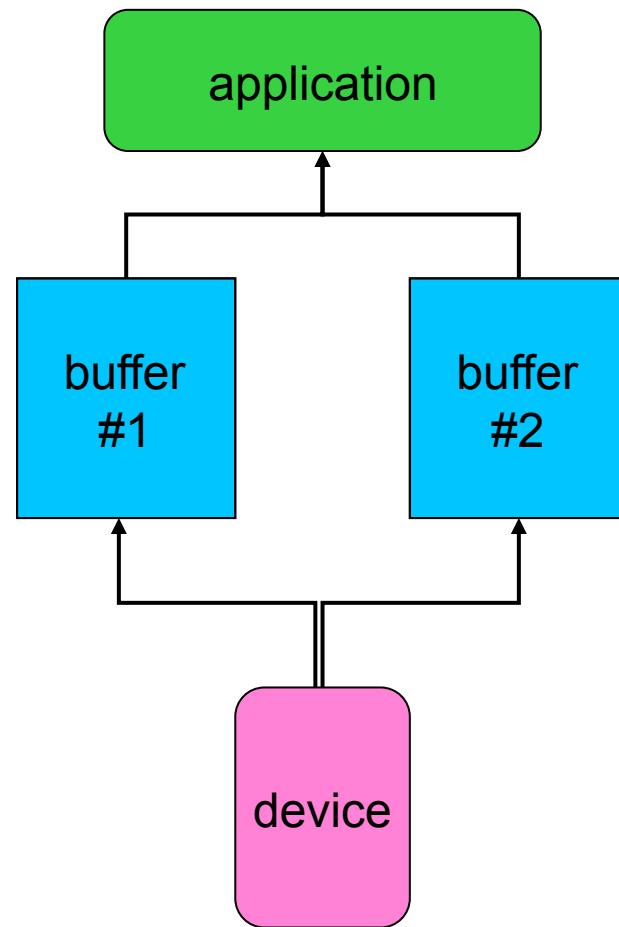
Double-Buffered Output



Performing Double-Buffered Output

- Have multiple buffers queued up, ready to write
 - Each write completion interrupt starts the next write
- Application and device I/O proceed in parallel
 - Application queues successive writes
 - Don't bother waiting for previous operation to finish
 - Device picks up next buffer as soon as it is ready
- If we're CPU-bound (more CPU than output)
 - Application speeds up because it doesn't wait for I/O
- If we're I/O-bound (more output than CPU)
 - Device is kept busy, which improves throughput
 - But eventually we may have to block the process

Double-Buffered Input



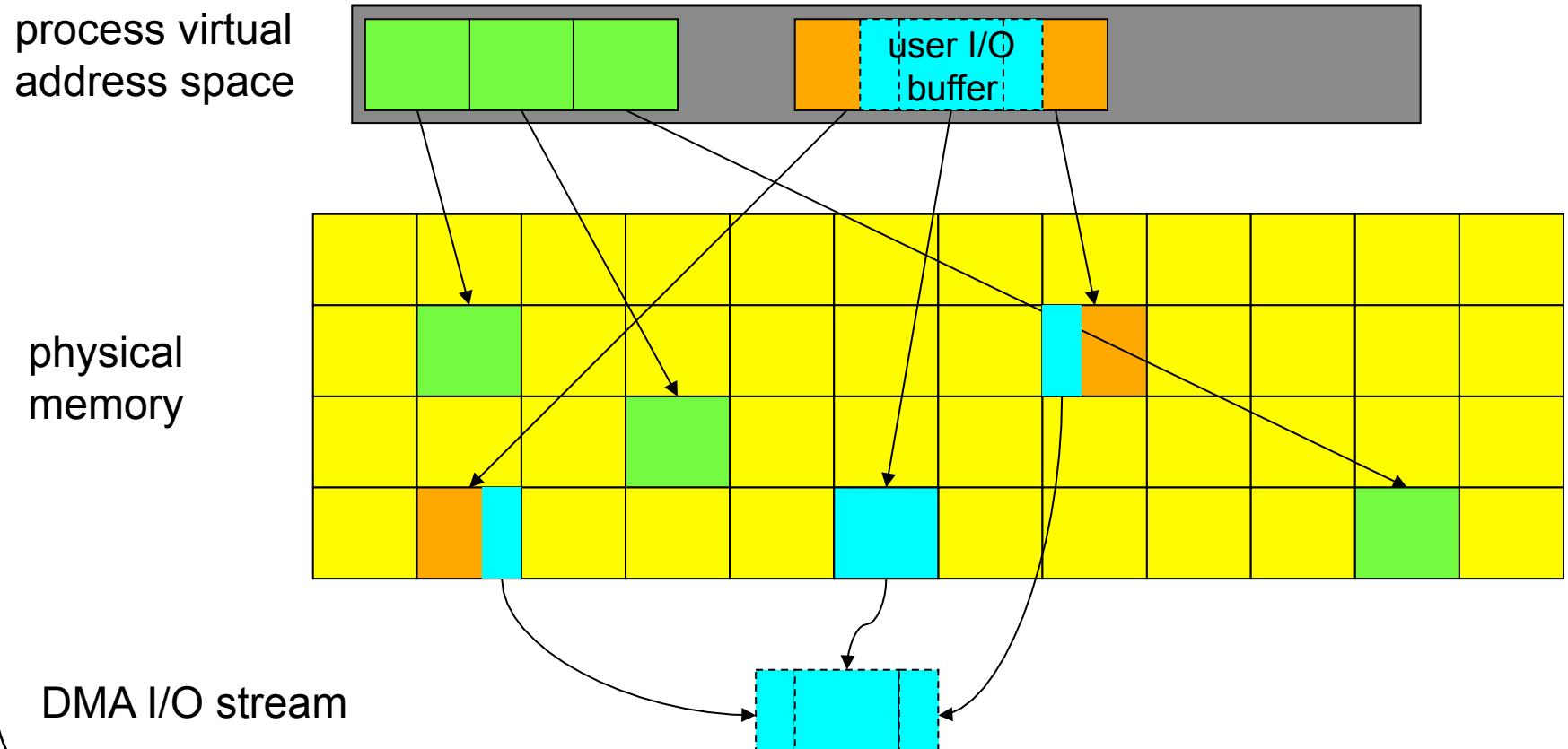
Performing Double Buffered Input

- Have multiple reads queued up, ready to go
 - Read completion interrupt starts read into next buffer
- Filled buffers wait until application asks for them
 - Application doesn't have to wait for data to be read
- When can we do chain-scheduled reads?
 - Each app will probably block until its read completes
 - So we won't get multiple reads from one application
 - We can queue reads from multiple processes
 - We can do predictive read-ahead

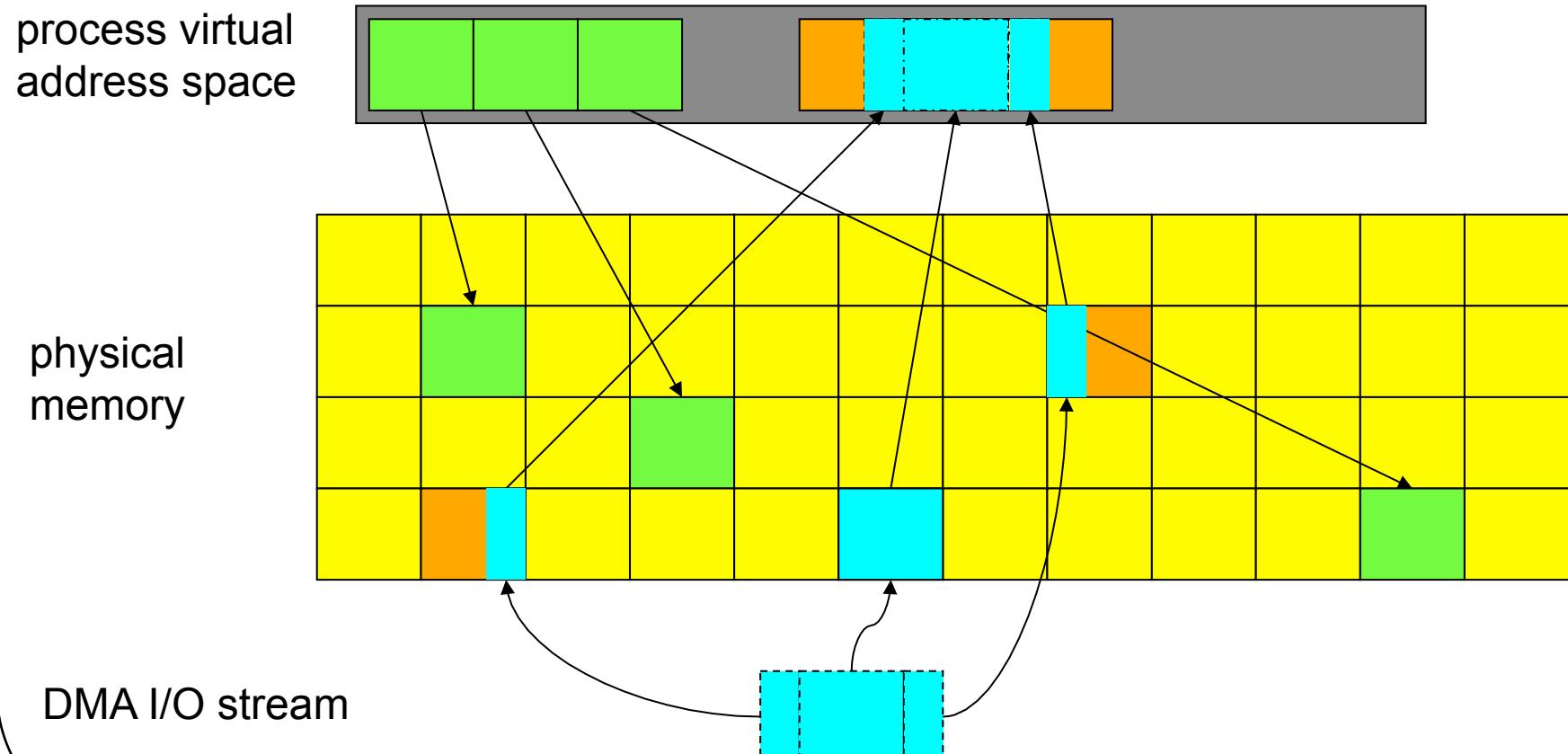
Scatter/Gather I/O

- Many device controllers support DMA transfers
 - Entire transfer must be contiguous in physical memory
- User buffers are in paged virtual memory
 - User buffers may be spread all over physical memory
 - *Scatter*: read from device to multiple pages
 - *Gather*: writing from multiple pages to device
- Three basic approaches apply
 1. Copy all user data into contiguous physical buffer
 2. Split logical request into chain-scheduled page requests
 3. I/O MMU may automatically handle scatter/gather

“Gather” Writes From Paged Memory



“Scatter” Reads Into Paged Memory



Memory Mapped I/O

- DMA may not be the best way to do I/O
 - Designed for large contiguous transfers
 - Some devices have many small sparse transfers
 - E.g., consider a video game display adaptor
- Instead, treat registers/memory in device as part of the regular memory space
 - Accessed by reading/writing those locations
- For example, a bit-mapped display adaptor
 - 1Mpixel display controller, on the CPU memory bus
 - Each word of memory corresponds to one pixel
 - Application uses ordinary stores to update display
- Low overhead per update, no interrupts to service
- Relatively easy to program

Trade-off: Memory Mapping vs. DMA

- DMA performs large transfers efficiently
 - Better utilization of both the devices and the CPU
 - Device doesn't have to wait for CPU to do transfers
 - But there is considerable per transfer overhead
 - Setting up the operation, processing completion interrupt
- Memory-mapped I/O has no per-op overhead
 - But every byte is transferred by a CPU instruction
 - No waiting because device accepts data at memory speed
- DMA better for occasional large transfers
- Memory-mapped better frequent small transfers
- Memory-mapped devices more difficult to share

Generalizing Abstractions for Device Drivers

- Every device type is unique
 - To some extent, at least in hardware details
- Implying each requires its own unique device driver
- But there are many commonalities
- Particularly among classes of devices
 - All disk drives, all network cards, all graphics cards, etc.
- Can we simplify the OS by leveraging these commonalities?
- By defining simplifying abstractions?

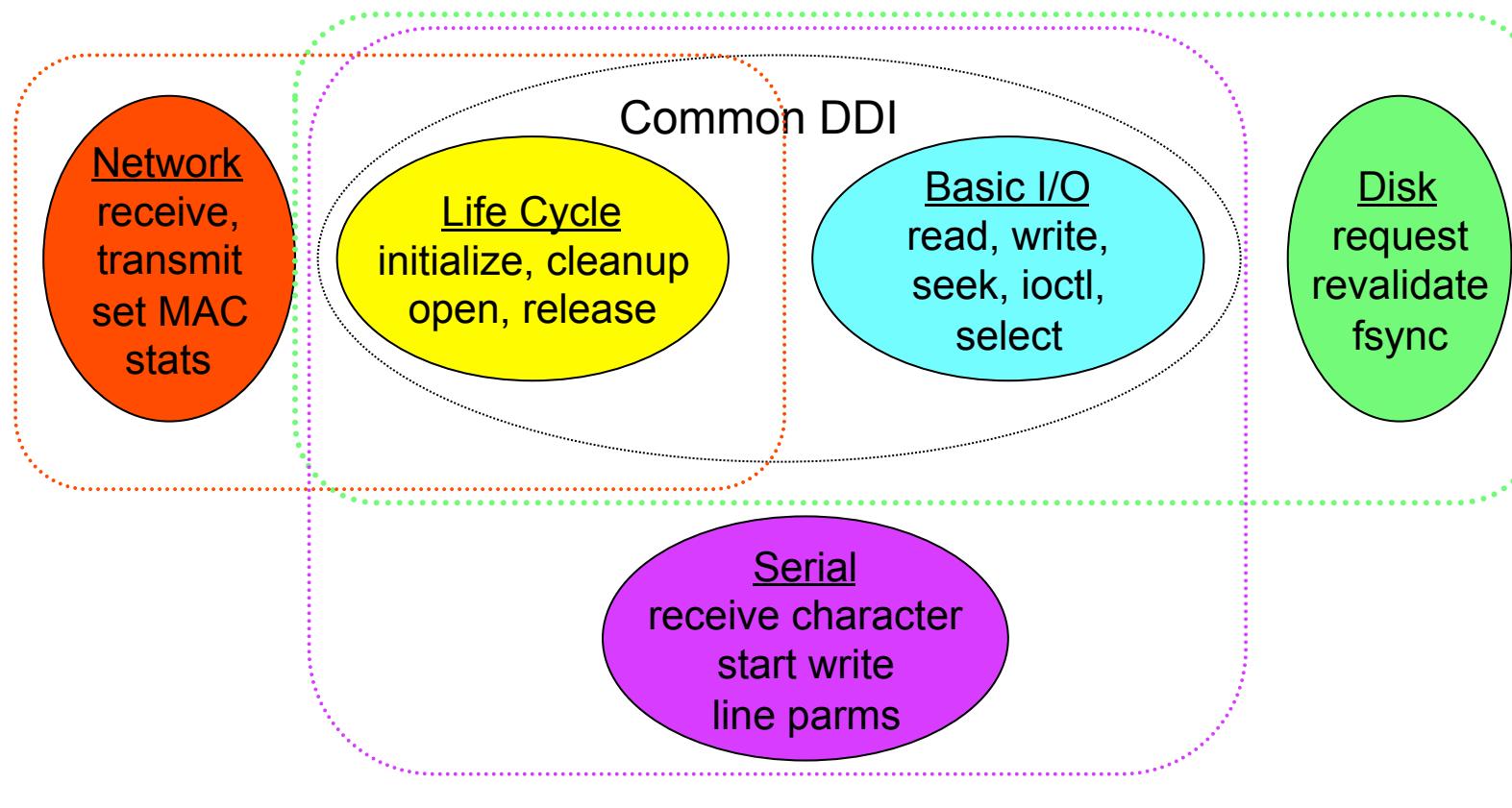
Providing the Abstractions

- The OS defines idealized device classes
 - Disk, display, printer, tape, network, serial ports
- Classes define expected interfaces/behavior
 - All drivers in class support standard methods
- Device drivers implement standard behavior
 - Make diverse devices fit into a common mold
 - Protect applications from device eccentricities
- Interfaces (as usual) are key to providing abstractions

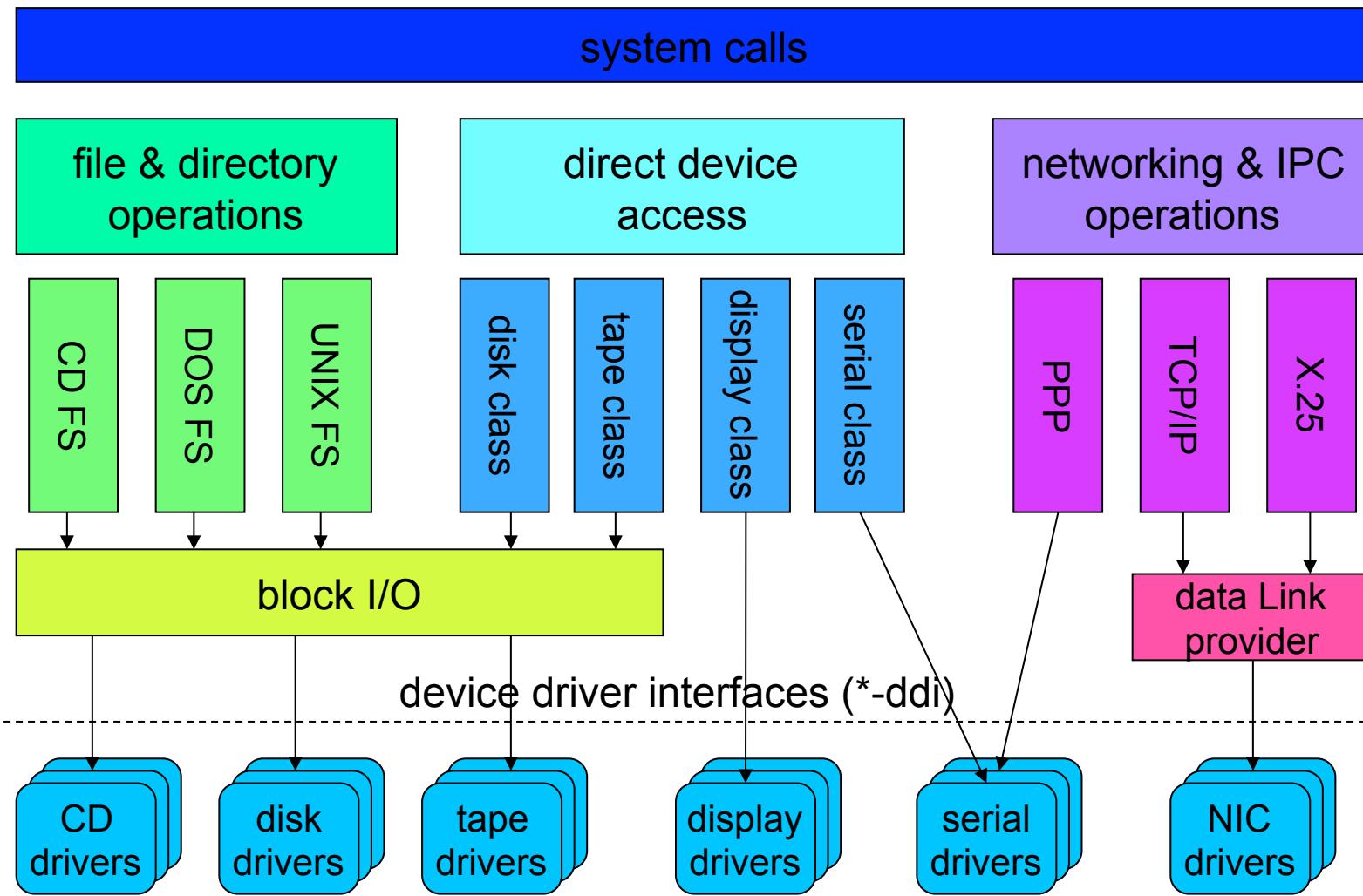
Device Driver Interface (DDI)

- Standard (top-end) device driver entry-points
 - “Top-end” – from the OS to the driver
 - Basis for device-independent applications
 - Enables system to exploit new devices
 - A critical interface contract for 3rd party developers
- Some entry points correspond directly to system calls
 - E.g., open, close, read, write
- Some are associated with OS frameworks
 - Disk drivers are meant to be called by block I/O
 - Network drivers are meant to be called by protocols

DDIs and sub-DDIs



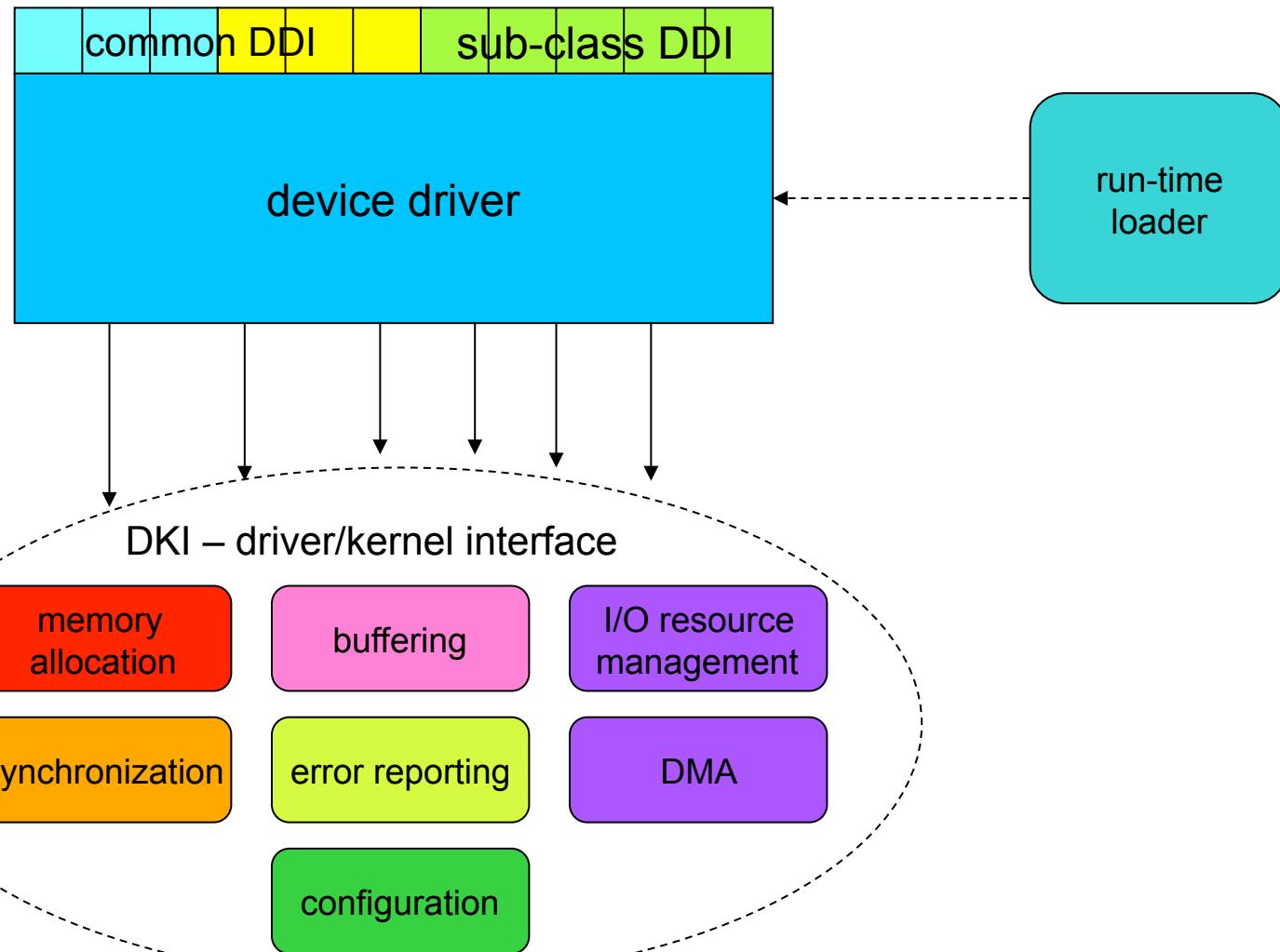
Standard Driver Classes & Clients



Drivers – Simplifying Abstractions

- Encapsulate knowledge of how to use a device
 - Map standard operations into operations onto device
 - Map device states into standard object behavior
 - Hide irrelevant behavior from users
 - Correctly coordinate device and application behavior
- Encapsulate knowledge of optimization
 - Efficiently perform standard operations on a device
- Encapsulation of fault handling
 - Knowledge of how to handle recoverable faults
 - Prevent device faults from becoming OS faults

Kernel Services for device drivers



Driver/Kernel Interface

- Specifies bottom-end services OS provides to drivers
 - Things drivers can ask the kernel to do
 - Analogous to an ABI for device driver writers
- Must be very well-defined and stable
 - To enable 3rd party driver writers to build drivers
 - So old drivers continue to work on new OS versions
- Each OS has its own DKI, but they are all similar
 - Memory allocation, data transfer and buffering
 - I/O resource (e.g. ports, interrupts) mgt, DMA
 - Synchronization, error reporting
 - Dynamic module support, configuration, plumbing

Criticality of Stable Interfaces

- Drivers are largely independent from the OS
 - They are built by different organizations
 - They might not be co-packaged with the OS
- OS and drivers have interface dependencies
 - OS depends on driver implementations of DDI
 - Drivers depends on kernel DKI implementations
- These interfaces must be carefully managed
 - Well defined and well tested
 - Upwards-compatible evolution

Linux Device Driver Abstractions

- An example of how an OS handles device drivers
- Basically inherited from earlier Unix systems
- A class-based system
- Several super-classes
 - Block devices
 - Character devices
 - Some regard network devices as a third major class
- Other divisions within each super-class

Why Classes of Drivers?

- Classes provide a good organization for abstraction
- They provide a common framework to reduce amount of code required for each new device
- The framework ensure all devices in class provide certain minimal functionality
- But a lot of driver functionality is very specific to the device
 - Implying that class abstractions don't cover everything

Character Device Superclass

- Devices that read/write one byte at a time
 - “Character” means byte, not ASCII
- May be either stream or record structured
- May be sequential or random access
- Support direct, synchronous reads and writes
- Common examples:
 - Keyboards
 - Monitors
 - Most other devices

Block Device Superclass

- Devices that deal with a block of data at a time
- Usually a fixed size block
- Most common example is a disk drive
- Reads or writes a single sized block (e.g., 4K bytes) of data at a time
- Random access devices, accessible one block at a time
- Support queued, asynchronous reads and writes

Why a Separate Superclass for Block Devices?

- Block devices span all forms of block-addressable random access storage
 - Hard disks, CDs, flash, and even some tapes
- Such devices require some very elaborate services
 - Buffer allocation, LRU management of a buffer cache, data copying services for those buffers, scheduled I/O, asynchronous completion, etc.
- Important system functionality (file systems and swapping/paging) implemented on top of block I/O
- Block I/O services are designed to provide very high performance for critical functions

Network Device Superclass

- Devices that send/receive data in packets
- Originally treated as character devices
- But sufficiently different from other character devices that some regard as distinct
- Only used in the context of network protocols
 - Unlike other devices
 - Which leads to special characteristics
- Typical examples are Ethernet cards, 802.11 cards, Bluetooth devices

Identifying Device Drivers

- The major device number specifies which device driver to use for it
- Might have several distinct devices using the same drivers
 - E.g., multiple disk drives of the same type
 - Or one disk drive divided into logically distinct pieces
- Minor device number distinguishes between those

Accessing Linux Device Drivers

- Done through the file system
- Special files
 - Files that are associated with a device instance
 - UNIX/LINUX uses <block/character, major, minor>

A block
special
device

- Major number corresponds to a particular device driver
- Minor number identifies an instance under that driver

```
brw-r----- 1 root operator 14, 0 Apr 11 18:03 disk0
brw-r----- 1 root operator 14, 1 Apr 11 18:03 disk0s1
brw-r----- 1 root operator 14, 2 Apr 11 18:03 disk0s2
br--r----- 1 reiher reiher 14, 3 Apr 15 16:19 disk2
br--r----- 1 reiher reiher 14, 4 Apr 15 16:19 disk2s1
br--r----- 1 reiher reiher 14, 5 Apr 15 16:19 disk2s2
```

- Opening a special file opens the associated device
 - Open/close/read/write/etc. calls map to calls to appropriate entry-points of the selected driver

Operating System Principles: File Systems

CS 111

Operating Systems
Peter Reiher

Outline

- File systems:
 - Why do we need them?
 - Why are they challenging?
- Basic elements of file system design
- Designing file systems for disks
 - Basic issues
 - Free space, allocation, and deallocation

Introduction

- Most systems need to store data persistently
 - So it's still there after reboot, or even power down
- Typically a core piece of functionality for the system
 - Which is going to be used all the time
- Even the operating system itself needs to be stored this way
- So we must store some data persistently

Our Persistent Data Options

- Use raw storage blocks to store the data
 - On a hard disk, flash drive, whatever
 - Those make no sense to users
 - Not even easy for OS developers to work with
- Use a database to store the data
 - Probably more structure (and possibly overhead) than we need or can afford
- Use a file system
 - Some organized way of structuring persistent data
 - Which makes sense to users and programmers

File Systems

- Originally the computer equivalent of a physical filing cabinet
- Put related sets of data into individual containers
- Put them all into an overall storage unit
- Organized by some simple principle
 - E.g., alphabetically by title
 - Or chronologically by date
- Goal is to provide:
 - Persistence
 - Ease of access
 - Good performance

The Basic File System Concept

- Organize data into natural coherent units
 - Like a paper, a spreadsheet, a message, a program
- Store each unit as its own self-contained entity
 - A *file*
 - Store each file in a way allowing efficient access
- Provide some simple, powerful organizing principle for the collection of files
 - Making it easy to find them
 - And easy to organize them

File Systems and Hardware

- File systems are typically stored on hardware providing persistent memory
 - Disks, tapes, flash memory, etc.
- With the expectation that a file put in one “place” will be there when we look again
- Performance considerations will require us to match the implementation to the hardware
- But ideally, the same user-visible file system should work on any reasonable hardware

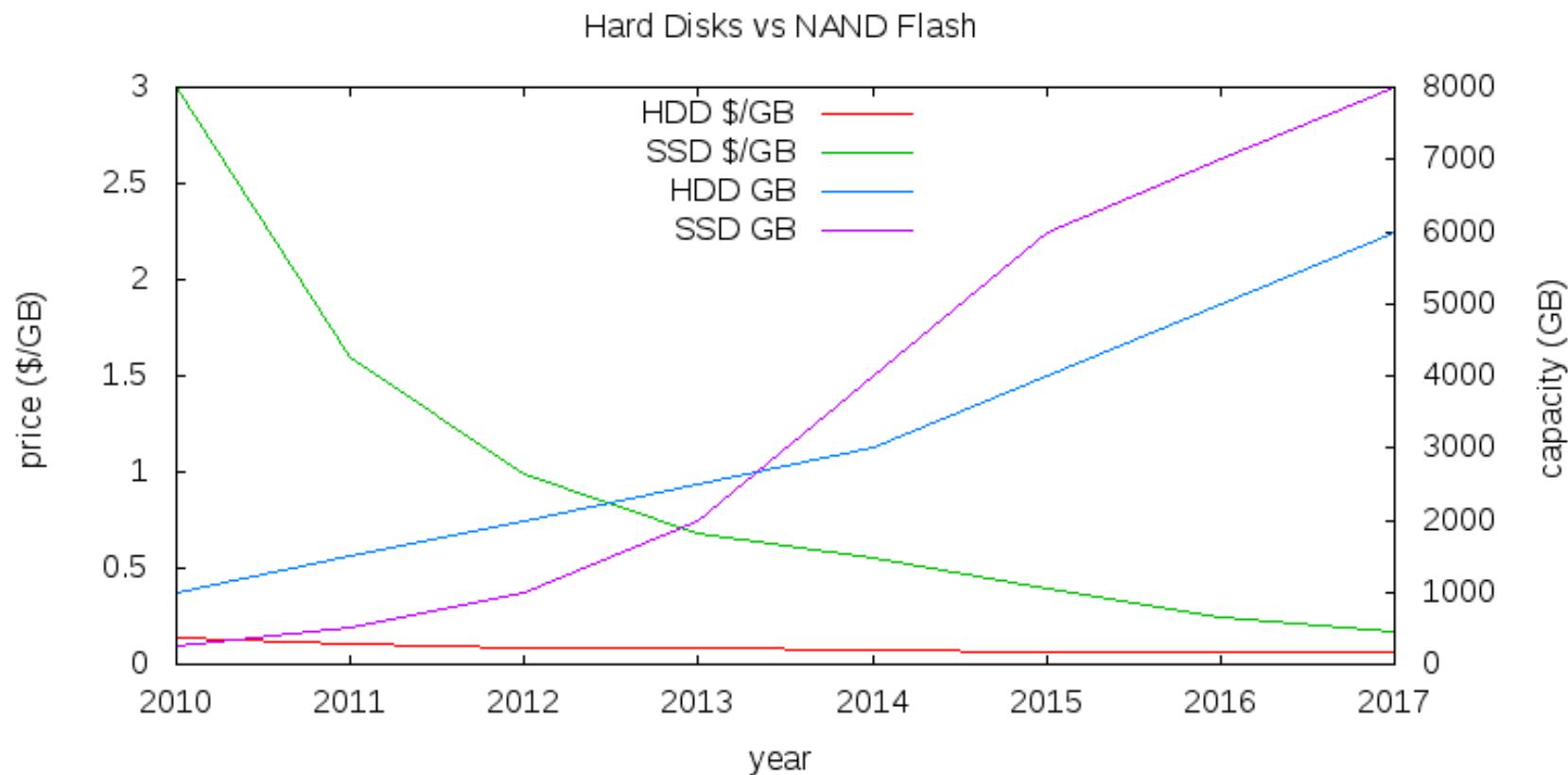
What Hardware Do We Use?

- Until recently, file systems were designed for disks
- Which required many optimizations based on particular disk characteristics
 - To minimize seek overhead
 - To minimize rotational latency delays
- Generally, the disk provided cheap persistent storage at the cost of high latency
 - File system design had to hide as much of the latency as possible

Disk vs SSD Performance

	Cheetah (archival)	Barracuda (high perf)	Extreme/Pro (SSD)
RPM	7,000	15,000	n/a
average latency	4.3ms	2ms	n/a
average seek	9ms	4ms	n/a
transfer speed	105MB/s	125MB/s	540MB/s
sequential 4KB read	39us	33us	10us
sequential 4KB write	39us	33us	11us
random 4KB read	13.2ms	6ms	10us
random 4KB write	13.2ms	6ms	11us

Random Access: Game Over



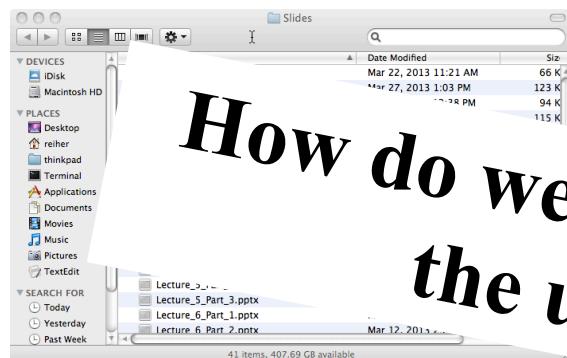
- Hard disks will still be cheaper and offer more capacity
- But not by that much
- And SSDs have all the other advantages

Data and Metadata

- File systems deal with two kinds of information
- *Data* – the information that the file is actually supposed to store
 - E.g., the instructions of the program or the words in the letter
- *Metadata* – Information about the information the file stores
 - E.g., how many bytes are there and when was it created
 - Sometimes called *attributes*
- Ultimately, both data and metadata must be stored persistently
 - And usually on the same piece of hardware

Bridging the Gap

We want something like . . .



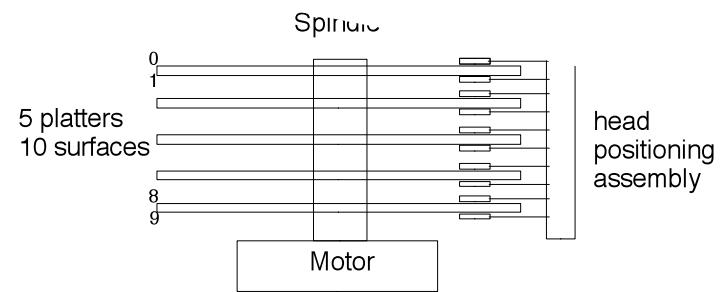
Or . . .



Or at least

```
drwxr-xr-x  8 root  wheel  272 May  4  2010 X11
lrwxr-xr-x  1 root  wheel    3 May  4  2010 X11R6 -> X11
drwxr-xr-x 913 root  wheel 31042 Apr 21 12:21 bin
drwxr-xr-x 336 root  wheel 11424 Mar 17 09:13 lib
drwxr-xr-x 103 root  wheel  3502 Apr 21 12:23 libexec
drwxr-xr-x   7 root  wheel   238 Jan 16 23:00 local
drwxr-xr-x 238 root  wheel  8092 Mar 17 09:13 sbin
drwxr-xr-x   59 root  wheel 2006 Apr 21 12:21 share
drwxr-xr-x    4 root  wheel   136 May  4  2010 standalone
```

But we've got something like . . .



A Further Wrinkle

- We want our file system to be agnostic to the storage medium
- Same program should access the file system the same way, regardless of medium
 - Otherwise it's hard to write portable programs
- Should work the same for disks of different types
- Or if we use a RAID instead of one disk
- Or if we use flash instead of disks
- Or if even we don't use persistent memory at all
 - E.g., RAM file systems

Desirable File System Properties

- What are we looking for from our file system?
 - Persistence
 - Easy use model
 - For accessing one file
 - For organizing collections of files
 - Flexibility
 - No limit on number of files
 - No limit on file size, type, contents
 - Portability across hardware device types
 - Performance
 - Reliability
 - Suitable security

The Performance Issue

- How fast does our file system need to be?
- Ideally, as fast as everything else
 - Like CPU, memory, and the bus
 - So it doesn't provide a bottleneck
- But these other devices operate today at nanosecond speeds
- Disk drives operate at millisecond speeds
 - Flash drives are faster, but not processor or RAM speeds
- Suggesting we'll need to do some serious work to hide the mismatch

The Reliability Issue

- Persistence implies reliability
- We want our files to be there when we check, no matter what
- Not just on a good day
- So our file systems must be free of errors
 - Hardware or software
- Remember our discussion of concurrency, race conditions, etc.?
 - Might we have some challenges here?

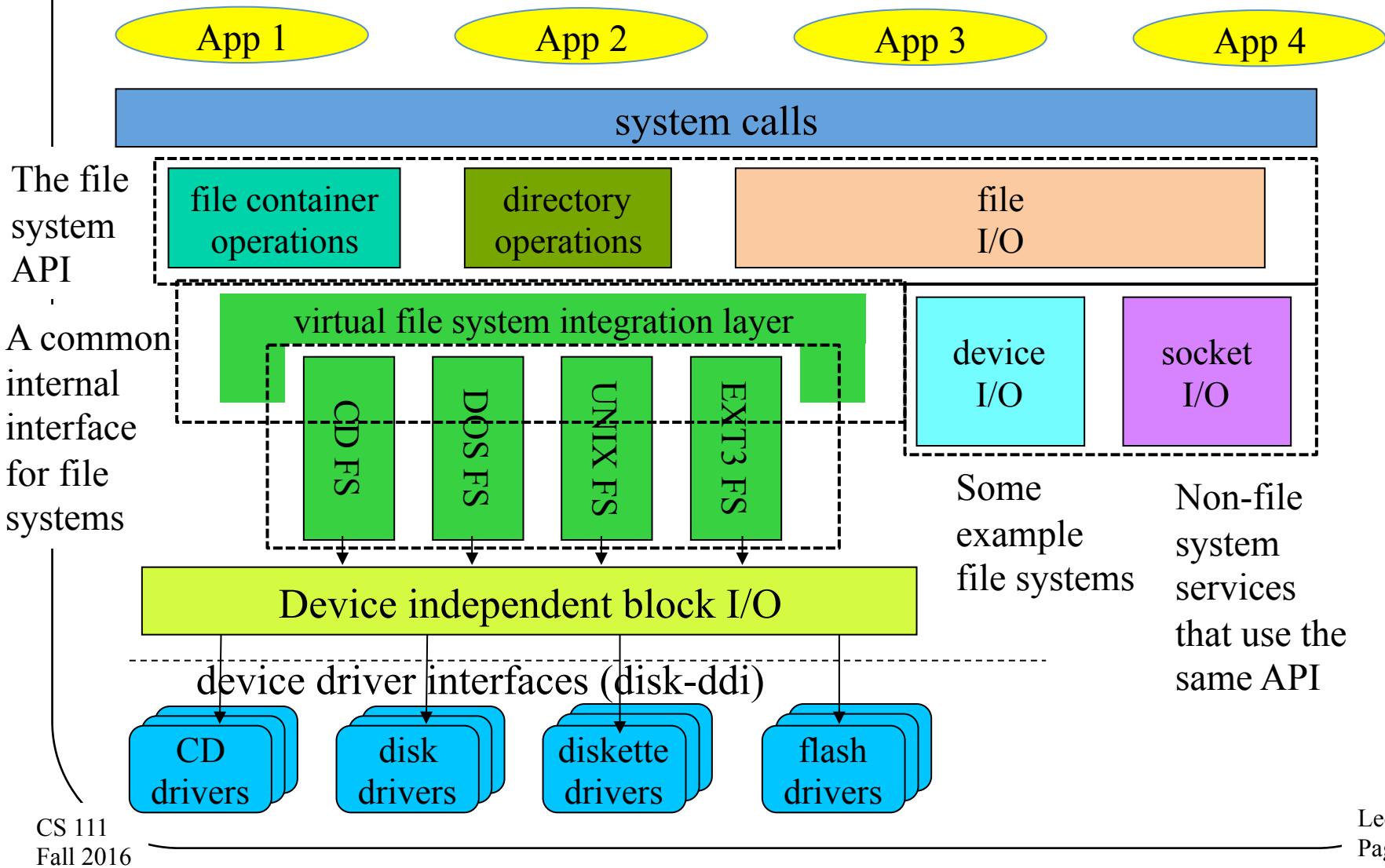
“Suitable” Security

- What does that mean?
- Whoever owns the data should be able to control who accesses it
 - Using some well-defined access control model and mechanism
- With strong guarantees that the system will enforce his desired controls
 - Implying we'll apply complete mediation
 - To the extent performance allows

Basics of File System Design

- Where do file systems fit in the OS?
- File control data structures

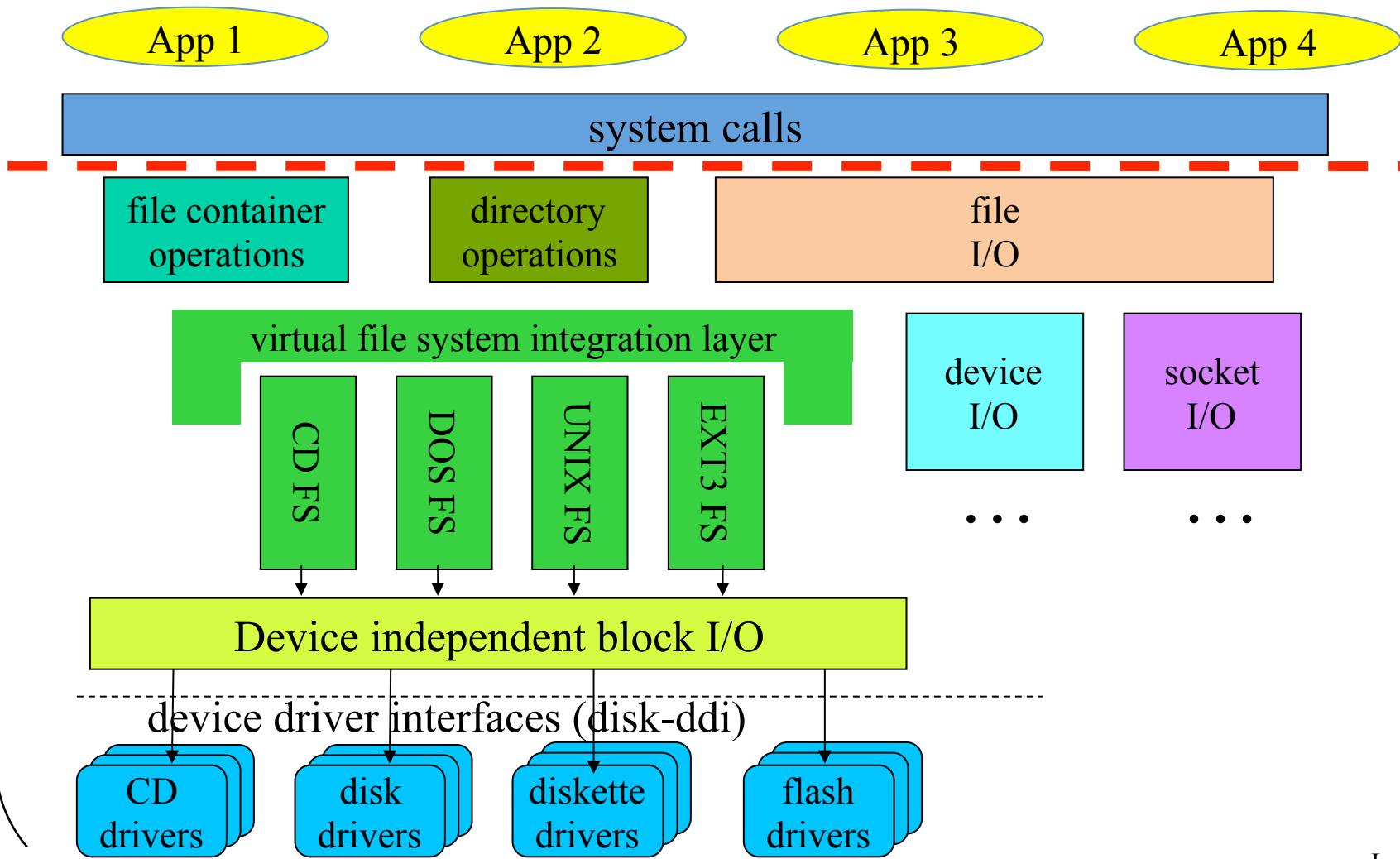
File Systems and the OS



File Systems and Layered Abstractions

- At the top, apps think they are accessing files
- At the bottom, various block devices are reading and writing blocks
- There are multiple layers of abstraction in between
- Why?
- Why not translate directly from application file operations to devices' block operations?

The File System API



The File System API

- Highly desirable to provide a single API to programmers and users for all files
- Regardless of how the file system underneath is actually implemented
- A requirement if one wants program portability
 - Very bad if a program won't work because there's a different file system underneath
- Three categories of system calls here
 1. File container operations
 2. Directory operations
 3. File I/O operations

File Container Operations

- Standard file management system calls
 - Manipulate files as objects
 - These operations ignore the contents of the file
- Implemented with standard file system methods
 - Get/set attributes, ownership, protection ...
 - Create/destroy files and directories
 - Create/destroy links
- Real work happens in file system implementation

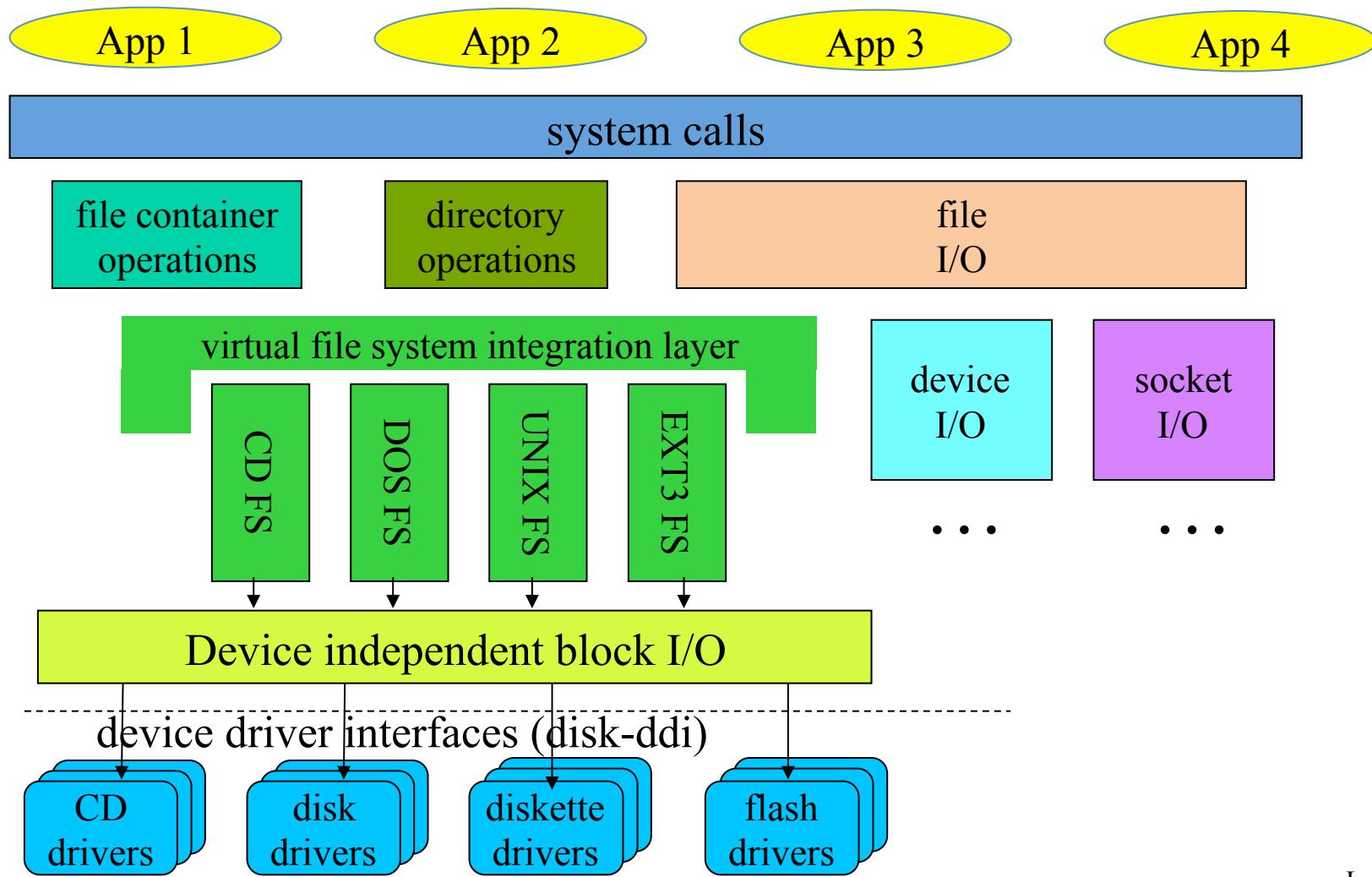
Directory Operations

- Directories provide the organization of a file system
 - Typically hierarchical
 - Sometimes with some extra wrinkles
- At the core, directories translate a name to a lower-level file pointer
- Operations tend to be related to that
 - Find a file by name
 - Create new name/file mapping
 - List a set of known names

File I/O Operations

- Open – use name to set up an open instance
- Read data from file and write data to file
 - Implemented using logical block fetches
 - Copy data between user space and file buffer
 - Request file system to write back block when done
- Seek
 - Change logical offset associated with open instance
- Map file into address space
 - File block buffers are just pages of physical memory
 - Map into address space, page it to and from file system

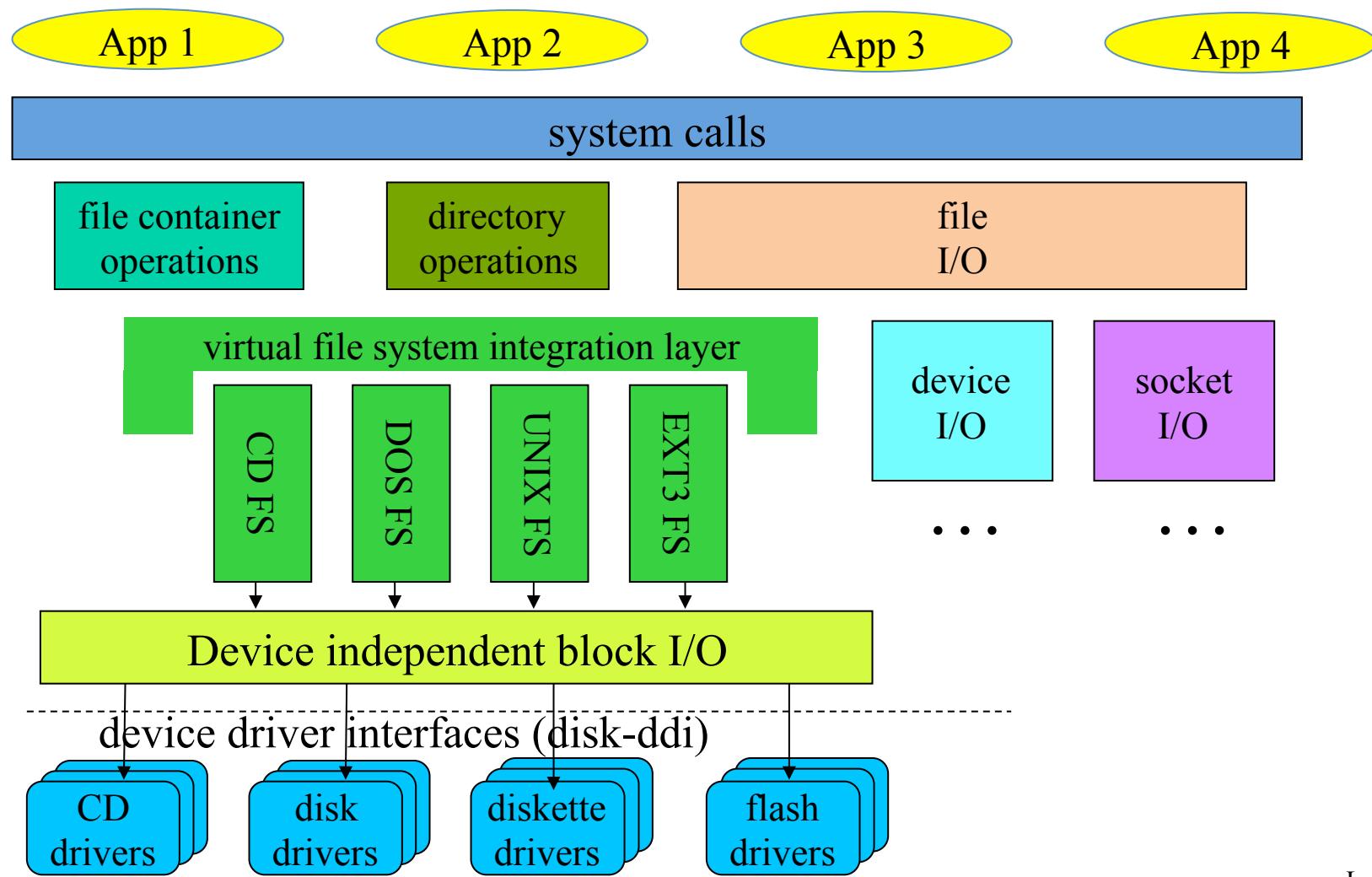
The Virtual File System Layer



The Virtual File System (VFS) Layer

- Federation layer to generalize file systems
 - Permits rest of OS to treat all file systems as the same
 - Support dynamic addition of new file systems
- Plug-in interface or file system implementations
 - DOS FAT, Unix, EXT3, ISO 9660, network, etc.
 - Each file system implemented by a plug-in module
 - All implement same basic methods
 - Create, delete, open, close, link, unlink,
 - Get/put block, get/set attributes, read directory, etc.
- Implementation is hidden from higher level clients
 - All clients see are the standard methods and properties

The File System Layer



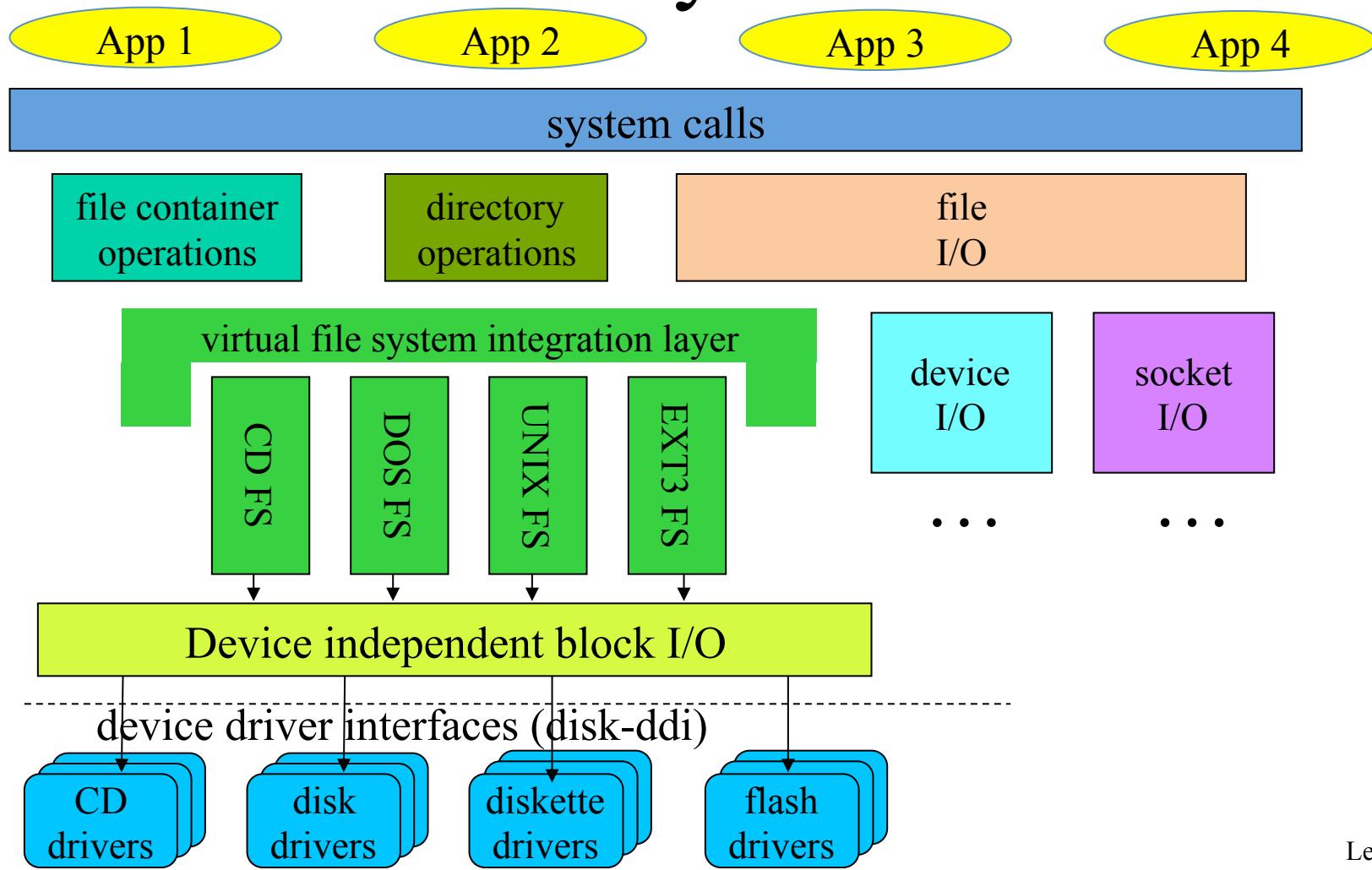
The File Systems Layer

- Desirable to support multiple different file systems
- All implemented on top of block I/O
 - Should be independent of underlying devices
- All file systems perform same basic functions
 - Map names to files
 - Map <file, offset> into <device, block>
 - Manage free space and allocate it to files
 - Create and destroy files
 - Get and set file attributes
 - Manipulate the file name space

Why Multiple File Systems?

- Why not instead choose one “good” one?
- There may be multiple storage devices
 - E.g., hard disk and flash drive
 - They might benefit from very different file systems
- Different file systems provide different services, despite the same interface
 - Differing reliability guarantees
 - Differing performance
 - Read-only vs. read/write
- Different file systems used for different purposes
 - E.g., a temporary file system

Device Independent Block I/O Layer



File Systems and Block I/O Devices

- File systems typically sit on a general block I/O layer
- A generalizing abstraction – make all disks look same
- Implements standard operations on each block device
 - Asynchronous read (physical block #, buffer, bytecount)
 - Asynchronous write (physical block #, buffer, bytecount)
- Map logical block numbers to device addresses
 - E.g., logical block number to <cylinder, head, sector>
- Encapsulate all the particulars of device support
 - I/O scheduling, initiation, completion, error handlings
 - Size and alignment limitations

Why Device Independent Block I/O?

- A better abstraction than generic disks
- Allows unified LRU buffer cache for disk data
 - Hold frequently used data until it is needed again
 - Hold pre-fetched read-ahead data until it is requested
- Provides buffers for data re-blocking
 - Adapting file system block size to device block size
 - Adapting file system block size to user request sizes
- Handles automatic buffer management
 - Allocation, deallocation
 - Automatic write-back of changed buffers

Why Do We Need That Cache?

- File access exhibits a high degree of reference locality at multiple levels:
 - Users often read and write a single block in small operations, reusing that block
 - Users read and write the same files over and over
 - Users often open files from the same directory
 - OS regularly consults the same meta-data blocks
- Having common cache eliminates many disk accesses, which are slow

File Systems Control Structures

- A file is a named collection of information
- Primary roles of file system:
 - To store and retrieve data
 - To manage the media/space where data is stored
- Typical operations:
 - Where is the first block of this file?
 - Where is the next block of this file?
 - Where is block 35 of this file?
 - Allocate a new block to the end of this file
 - Free all blocks associated with this file

Finding Data On Disks

- Essentially a question of how you managed the space on your disk
- Space management on disk is complex
 - There are millions of blocks and thousands of files
 - Files are continuously created and destroyed
 - Files can be extended after they have been written
 - Data placement on disk has performance effects
 - Poor management leads to poor performance
- Must track the space assigned to each file
 - On-disk, master data structure for each file

On-Disk File Control Structures

- On-disk description of important attributes of a file
 - Particularly where its data is located
- Virtually all file systems have such data structures
 - Different implementations, performance & abilities
 - Implementation can have profound effects on what the file system can do (well or at all)
- A core design element of a file system
- Paired with some kind of in-memory representation of the same information

The Basic File Control Structure Problem

- A file typically consists of multiple data blocks
- The control structure must be able to find them
- Preferably able to find any of them quickly
 - I.e., shouldn't need to read the entire file to find a block near the end
- Blocks can be changed
- New data can be added to the file
 - Or old data deleted
- Files can be sparsely populated

The In-Memory Representation

- There is an on-disk structure pointing to disk blocks (and holding other information)
- When file is opened, an in-memory structure is created
- Not an exact copy of the disk version
 - The disk version points to disk blocks
 - The in-memory version points to RAM pages
 - Or indicates that the block isn't in memory
 - Also keeps track of which blocks are dirty and which aren't

In-Memory Structures and Processes

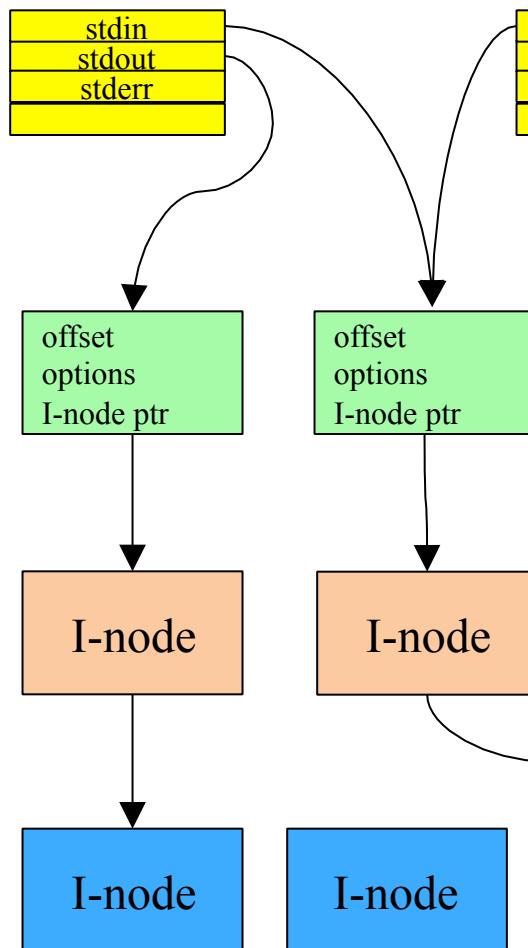
- What if multiple processes have a given file open?
- Should they share one control structure or have one each?
- In-memory structures typically contain a cursor pointer
 - Indicating how far into the file data has been read/written
- Sounds like that should be per-process . . .

Per-Process or Not?

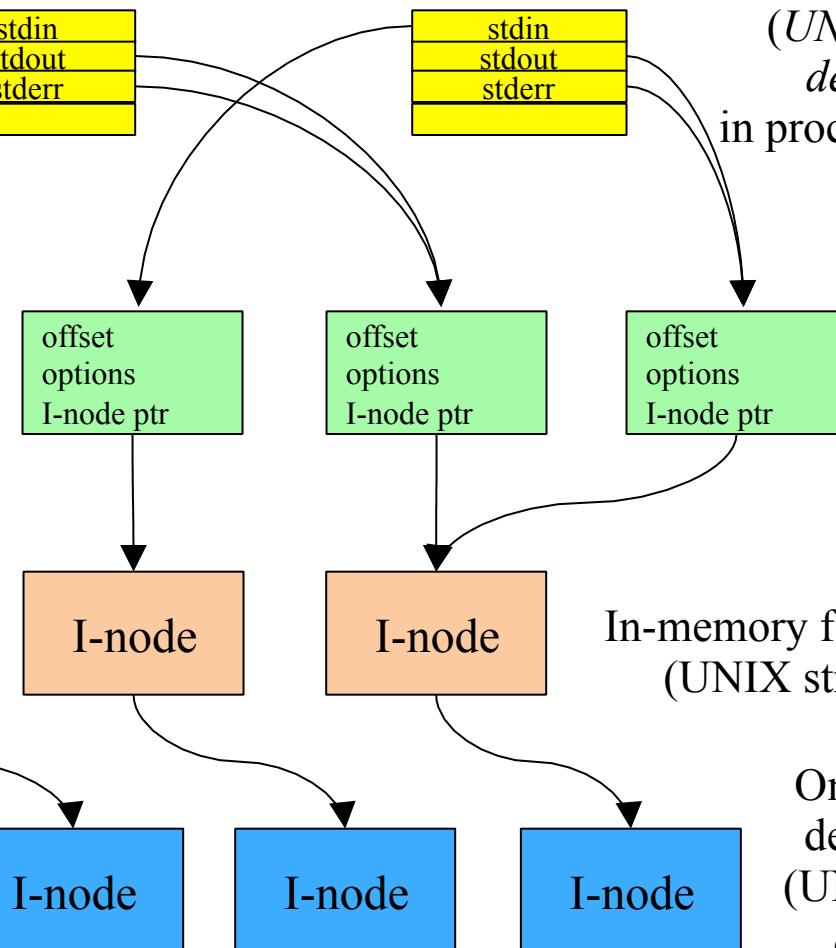
- What if cooperating processes are working with the same file?
 - They might want to share a cursor
- And how can we know when all processes are finished with an open file?
 - So we can reclaim space used for its in-memory descriptor
- Implies a two-level solution
 1. A structure shared by all
 2. A structure shared by cooperating processes

The Unix Approach

Two processes can share one descriptor



Two descriptors can share one inode



Open-file references
(UNIX user file descriptor)
in process descriptor

Open file instance descriptors

In-memory file descriptors
(UNIX struct *inode*)

On-disk file descriptors
(UNIX struct *dinode*)

File System Structure

- How do I organize a disk into a file system?
 - Linked extents
 - The DOS FAT file system
 - File index blocks
 - Unix System V file system

Basics of File System Structure

- Most file systems live on disks
- Disk volumes are divided into fixed-sized blocks
 - Many sizes are used: 512, 1024, 2048, 4096, 8192 ...
- Most blocks will be used to store user data
- Some will be used to store organizing “meta-data”
 - Description of the file system (e.g., layout and state)
 - File control blocks to describe individual files
 - Lists of free blocks (not yet allocated to any file)
- All operating systems have such data structures
 - Different OSes and file systems have very different goals
 - These result in very different implementations

The Boot Block

- The 0th block of a disk is usually reserved for the boot block
 - Code allowing the machine to boot an OS
- Not usually under the control of a file system
 - It typically ignores the boot block entirely
- Not all disks are bootable
 - But the 0th block is usually reserved, “just in case”
- So file systems start work at block 1

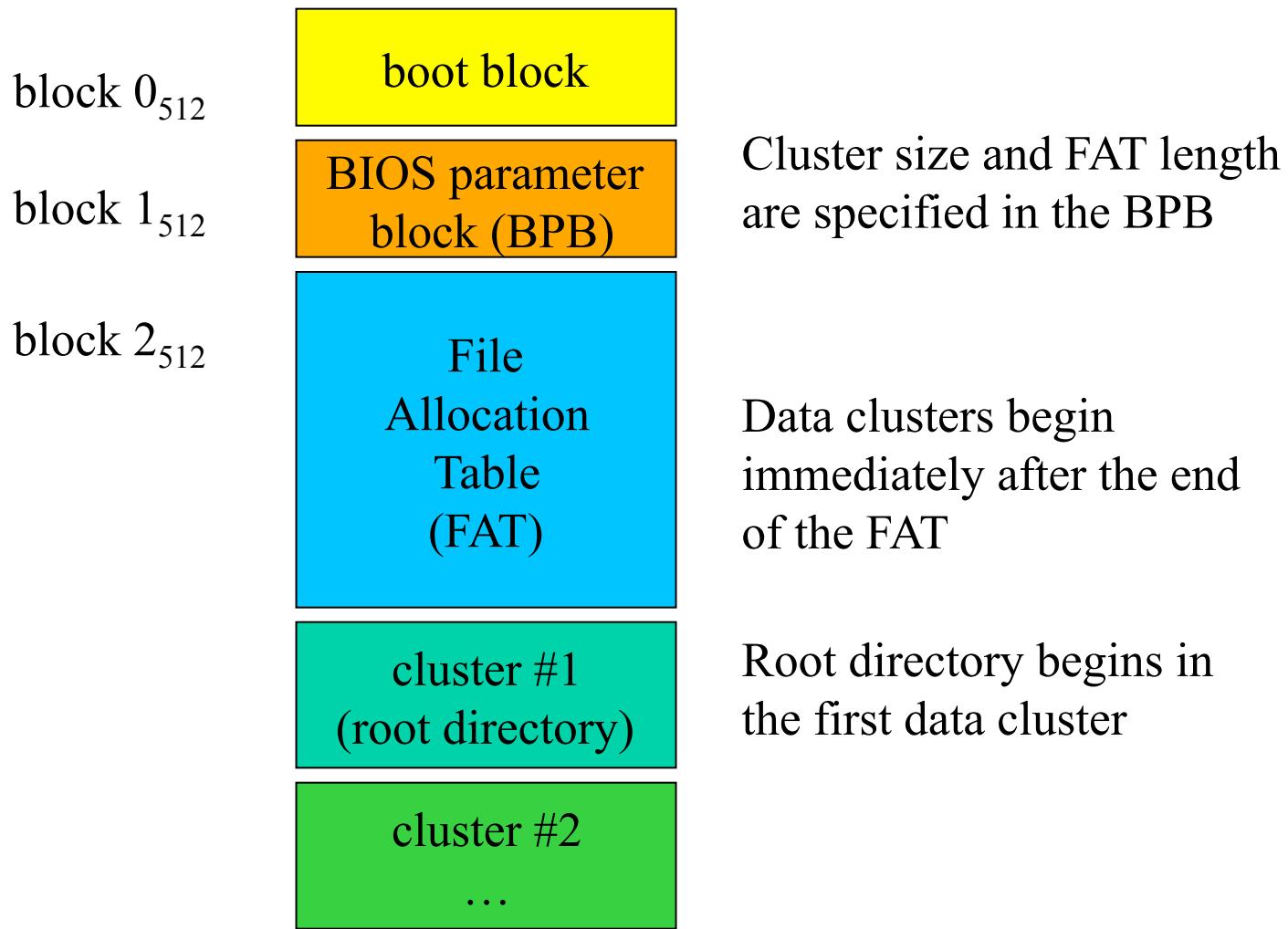
Managing Allocated Space

- A core activity for a file system, with various choices
- What if we give each file same amount of space?
 - Internal fragmentation ... just like memory
- What if we allocate just as much as file needs?
 - External fragmentation, compaction ... just like memory
- Perhaps we should allocate space in “pages”
 - How many chunks can a file contain?
- The file control data structure determines this
 - It only has room for so many pointers, then file is “full”
- So how do we want to organize the space in a file?

Linked Extents

- A simple answer
- File control block contains exactly one pointer
 - To the first chunk of the file
 - Each chunk contains a pointer to the next chunk
 - Allows us to add arbitrarily many chunks to each file
- Pointers can be in the chunks themselves
 - This takes away a little of every chunk
 - To find chunk N, you have to read the first N-1 chunks
- Pointers can be in auxiliary “chunk linkage” table
 - Faster searches, especially if table kept in memory

The DOS File System



DOS File System Overview

- DOS file systems divide space into “clusters”
 - Cluster size (multiple of 512) fixed for each file system
 - Clusters are numbered 1 though N
- File control structure points to first cluster of a file
- File Allocation Table (FAT), one entry per cluster
 - Contains the number of the next cluster in file
 - A 0 entry means that the cluster is not allocated
 - A -1 entry means “end of file”
- File system is sometimes called “FAT,” after the name of this key data structure

DOS FAT Clusters

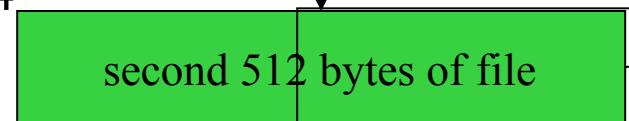
directory entry

name:	myfile.txt
length:	1500 bytes
1 st cluster:	3

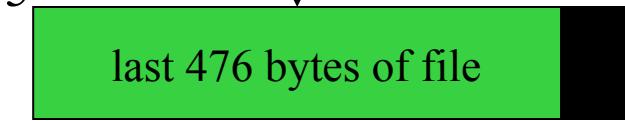
cluster #3



cluster #4



cluster #5



File Allocation Table

1	x
2	x
3	4
4	-
5	5
6	-1
	0

Each FAT entry corresponds to a cluster, and contains the number of the next cluster.

-1 = End of File

0 = free cluster

DOS File System Characteristics

- To find a particular block of a file
 - Get number of first cluster from directory entry
 - Follow chain of pointers through File Allocation Table
- Entire File Allocation Table is kept in memory
 - No disk I/O is required to find a cluster
 - For very large files the search can still be long
- No support for “sparse” files
 - Of a file has a block n , it must have all blocks $< n$
- Width of FAT determines max file system size
 - How many bits describe a cluster address?
 - Originally 8 bits, eventually expanded to 32

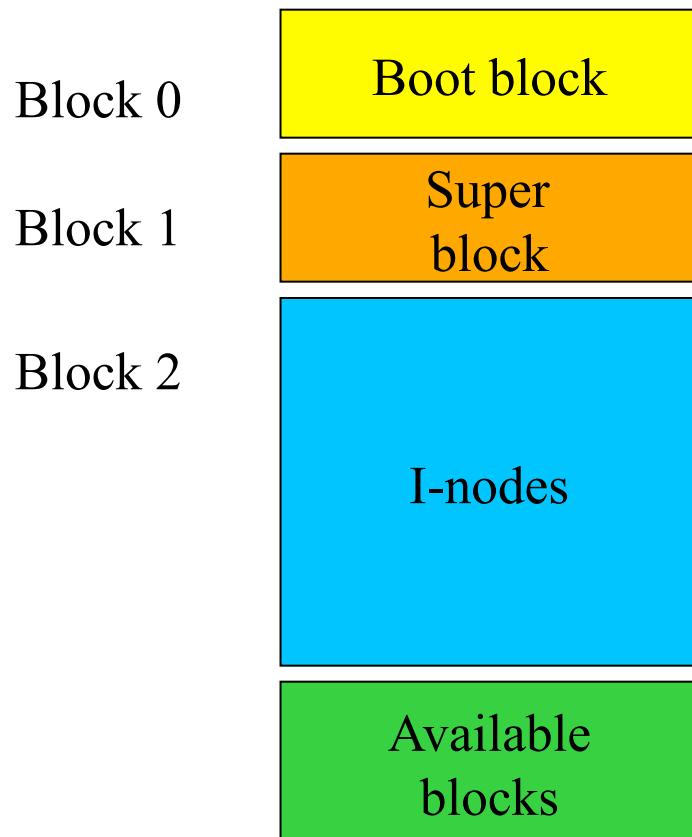
File Index Blocks

- A different way to keep track of where a file's data blocks are on the disk
- A file control block points to all blocks in file
 - Very fast access to any desired block
 - But how many pointers can the file control block hold?
- File control block could point at extent descriptors
 - But this still gives us a fixed number of extents

Hierarchically Structured File Index Blocks

- To solve the problem of file size being limited by entries in file index block
- The basic file index block points to blocks
- Some of those contain pointers which in turn point to blocks
- Can point to many extents, but still a limit to how many
 - But that limit might be a very large number
 - Has potential to adapt to wide range of file sizes

Unix System V File System

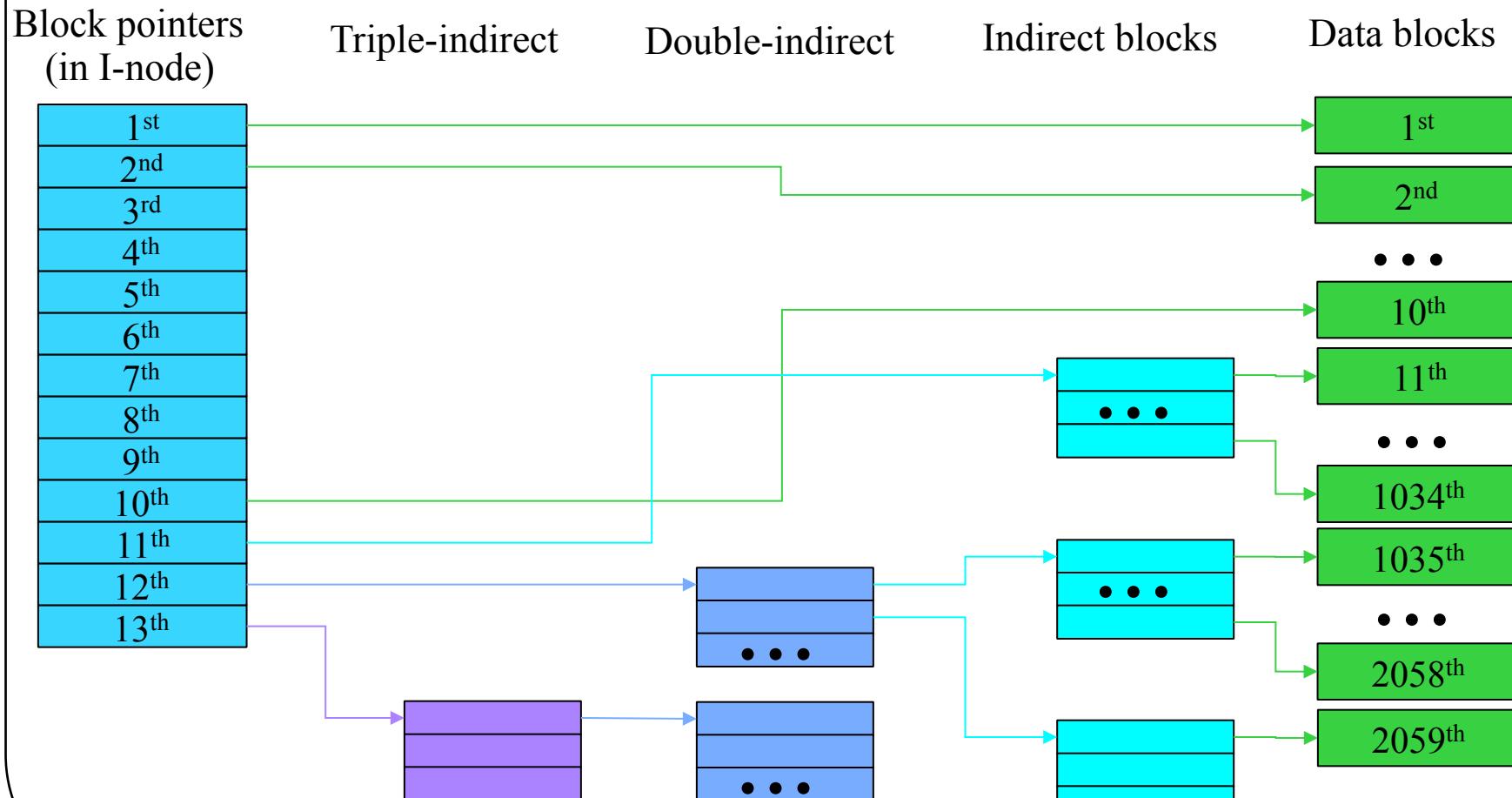


Block size and number of I-nodes are specified in super block

I-node #1 (traditionally) describes the root directory

Data blocks begin immediately after the end of the I-nodes.

Unix Inodes and Block Pointers



Why Is This a Good Idea?

- The UNIX pointer structure seems ad hoc and complicated
- Why not something simpler?
 - E.g., all block pointers are triple indirect
- File sizes are not random
 - The majority of files are only a few thousand bytes long
- Unix approach allows us to access up to 40Kbytes (assuming 4K blocks) without extra I/Os
 - Remember, the double and triple indirect blocks must themselves be fetched off disk

How Big a File Can Unix Handle?

- The on-disk inode contains 13 block pointers
 - First 10 point to first 10 blocks of file
 - 11th points to an indirect block (which contains pointers to 1024 blocks)
 - 12th points to a double indirect block (pointing to 1024 indirect blocks)
 - 13th points to a triple indirect block (pointing to 1024 double indirect blocks)
- Assuming 4k bytes per block and 4 bytes per pointer
 - 10 direct blocks = $10 * 4K$ bytes = 40K bytes
 - Indirect block = $1K * 4K$ = 4M bytes
 - Double indirect = $1K * 4M$ = 4G bytes
 - Triple indirect = $1K * 4G$ = 4T bytes
 - At the time system was designed, that seemed impossibly large
 - But . . .

Unix Inode Performance Issues

- The inode is in memory whenever file is open
- So the first ten blocks can be found with no extra I/O
- After that, we must read indirect blocks
 - The real pointers are in the indirect blocks
 - Sequential file processing will keep referencing it
 - Block I/O will keep it in the buffer cache
- 1-3 extra I/O operations per thousand pages
 - Any block can be found with 3 or fewer reads
- Index blocks can support “sparse” files
 - Not unlike page tables for sparse address spaces

Operating System Principles: File Systems – Allocation, Naming, Performance, and Reliability

CS 111

Operating Systems
Peter Reiher

Outline

- Allocating and managing file system free space
- Other performance improvement strategies
- File naming and directories
- File system reliability issues

Free Space and Allocation Issues

- How do I keep track of a file system's free space?
- How do I allocate new disk blocks when needed?
 - And how do I handle deallocation?

The Allocation/Deallocation Problem

- File systems usually aren't static
- You create and destroy files
- You change the contents of files
 - Sometimes extending their length in the process
- Such changes convert unused disk blocks to used blocks (or visa versa)
- Need correct, efficient ways to do that
- Typically implies a need to maintain a free list of unused disk blocks

Creating a New File

- Allocate a free file control block
 - For UNIX
 - Search the super-block free I-node list
 - Take the first free I-node
 - For DOS
 - Search the parent directory for an unused directory entry
- Initialize the new file control block
 - With file type, protection, ownership, ...
- Give new file a name
 - Naming issues will be discussed in the next lecture

Extending a File

- Application requests new data be assigned to a file
 - May be an explicit allocation/extension request
 - May be implicit (e.g., write to a currently non-existent block – remember sparse files?)
- Find a free chunk of space
 - Traverse the free list to find an appropriate chunk
 - Remove the chosen chunk from the free list
- Associate it with the appropriate address in the file
 - Go to appropriate place in the file or extent descriptor
 - Update it to point to the newly allocated chunk

Deleting a File

- Release all the space that is allocated to the file
 - For UNIX, return each block to the free block list
 - DOS does not free space
 - It uses garbage collection
 - So it will search out deallocated blocks and add them to the free list at some future time
- Deallocate the file control lock
 - For UNIX, zero inode and return it to free list
 - For DOS, zero the first byte of the name in the parent directory
 - Indicating that the directory entry is no longer in use

Free Space Maintenance

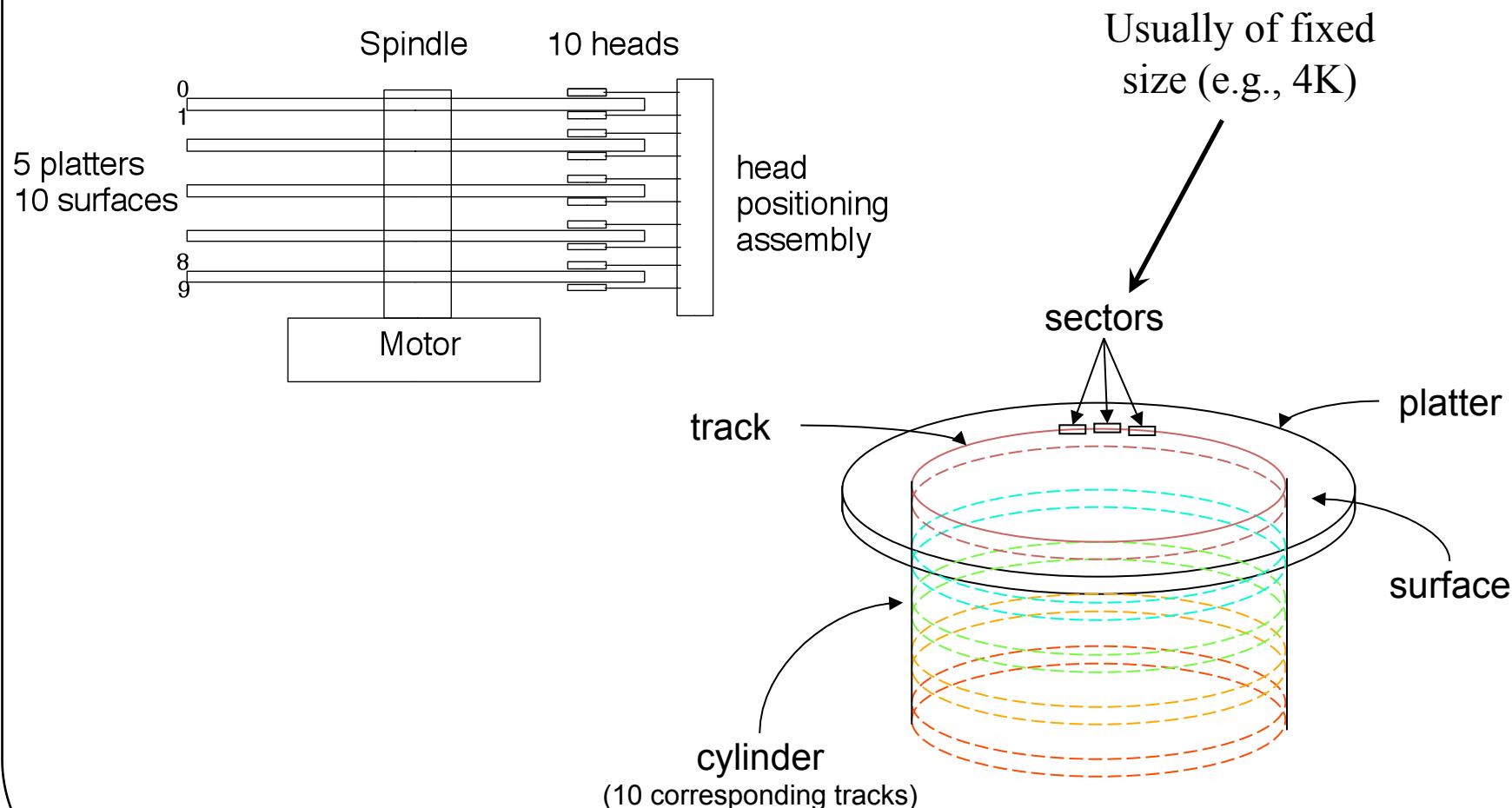
- File system manager manages the free space
- Getting/releasing blocks should be fast operations
 - They are extremely frequent
 - We'd like to avoid doing I/O as much as possible
- Unlike memory, it matters what block we choose
 - Best to allocate new space in same cylinder as file's existing space
 - User may ask for contiguous storage
- Free-list organization must address both concerns
 - Speed of allocation and deallocation
 - Ability to allocate contiguous or near-by space

The BSD File System

Free Space Management

- BSD is another version of Unix
- The details of its inodes are similar to those of Unix System V
 - As previously discussed
- Other aspects are somewhat different
 - Including free space management
 - Typically more advanced
- Uses bit map approach to managing free space
 - Keeping cylinder issues in mind

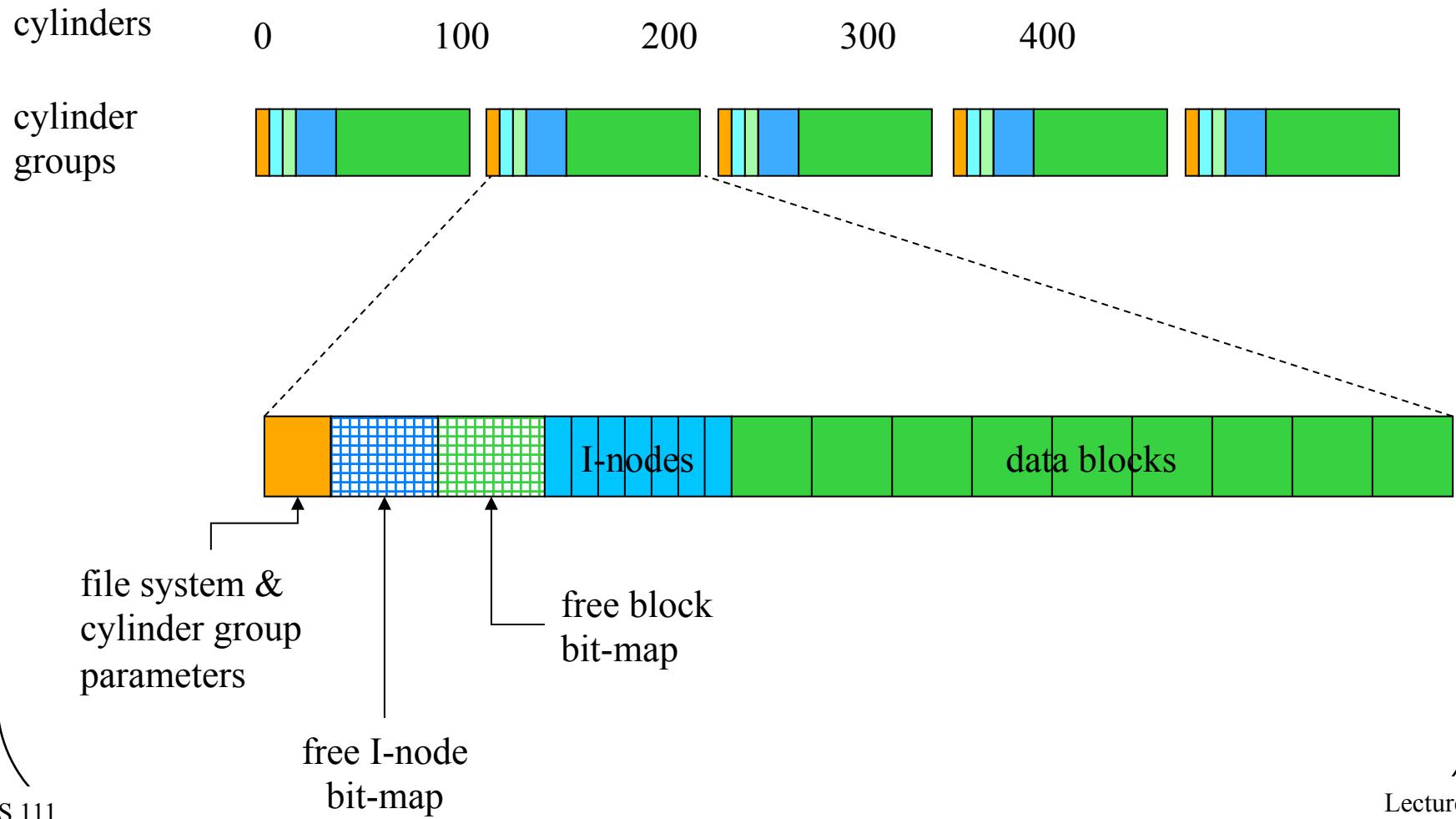
Disk Drives and Geometry



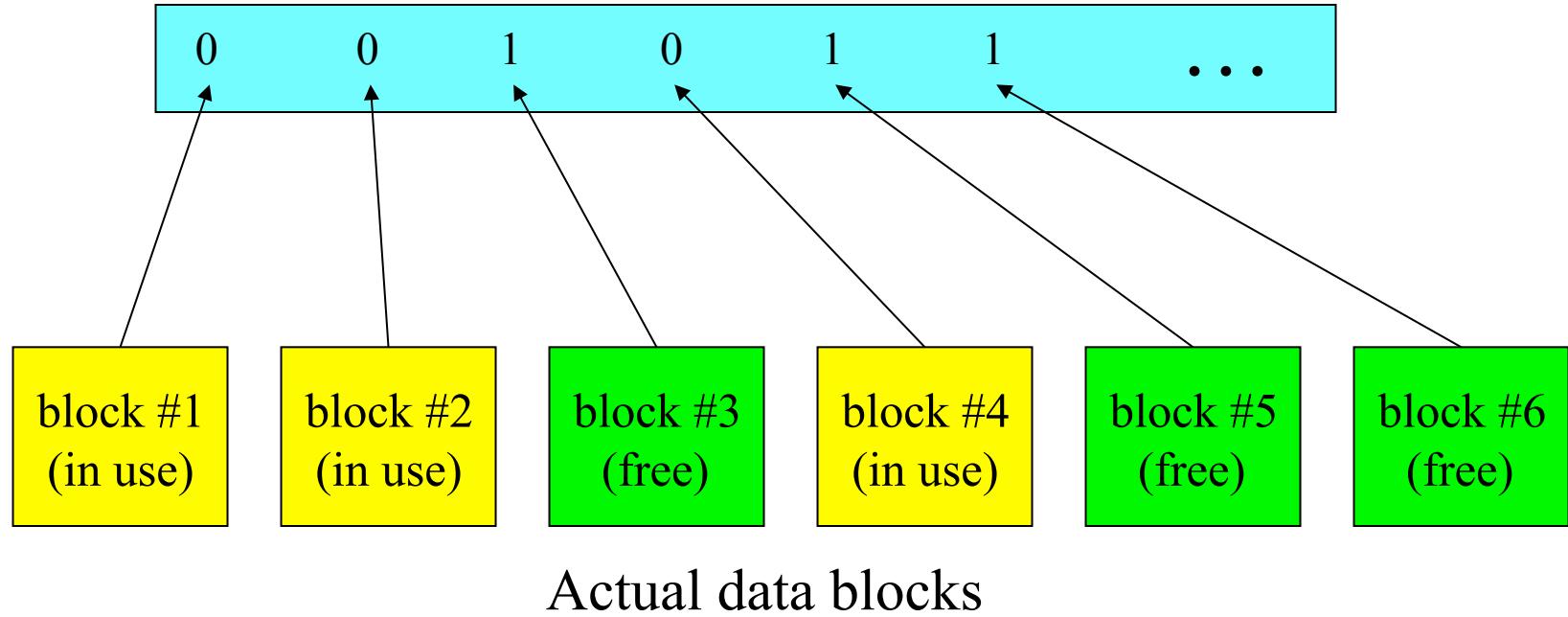
The BSD Approach

- Instead of all control information at start of disk,
- Divide file system into cylinder groups
 - Each cylinder group has its own control information
 - The *cylinder group summary*
 - Active cylinder group summaries are kept in memory
 - Each cylinder group has its own inodes and blocks
 - Free block list is a bit-map in cylinder group summary
- Enables significant reductions in head motion
 - Data blocks in file can be allocated in same cylinder
 - Inode and its data blocks in same cylinder group
 - Directories and their files in same cylinder group

BSD Cylinder Groups and Free Space



Bit Map Free Lists



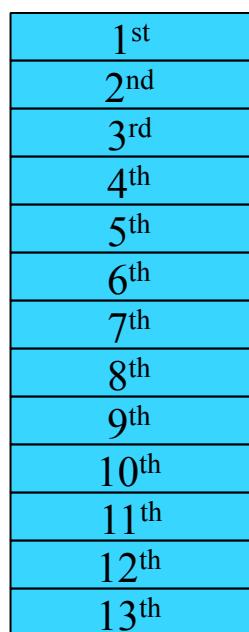
BSD Unix file systems use bit-maps to keep track of both free blocks and free I-nodes in each cylinder group

Extending a BSD/Unix File

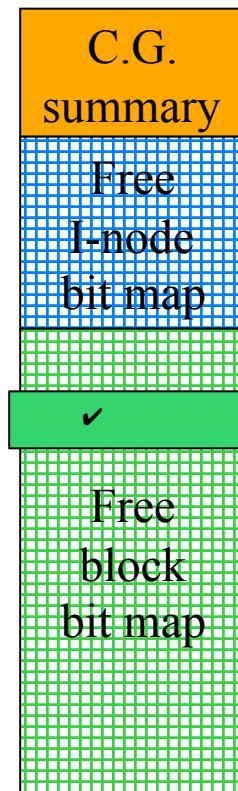
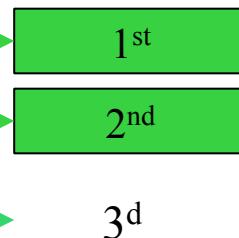
- Determine the cylinder group for the file's inode
 - Calculated from the inode's identifying number
- Find the cylinder for the previous block in the file
- Find a free block in the desired cylinder
 - Search the free-block bit-map for a free block in the right cylinder
 - Update the bit-map to show the block has been allocated
- Update the inode to point to the new block
 - Go to appropriate block pointer in inode/indirect block
 - If new indirect block is needed, allocate/assign it first
 - Update inode/indirect to point to new block

Unix File Extension

block pointers
(in I-node)



1. Determine cylinder group and get its information
2. Consult the cylinder group free block bit map to find a good block
3. Allocate the block to the file
 - 3.1 Set appropriate block pointer to it
 - 3.2 Update the free block bit map



Other Performance Improvement Strategies

- Beyond disk layout issues
 - Which are only relevant for hard drives, not flash or other solid state devices
- Transfer size
- Caching

Allocation/Transfer Size

- Per operation overheads are high
 - DMA startup, seek, rotation, interrupt service
- Larger transfer units more efficient
 - Amortize fixed per-op costs over more bytes/op
 - Multi-megabyte transfers are very good
- This requires space allocation units
 - Allocate space to files in much larger chunks
 - Large fixed size chunks -> internal fragmentation
 - Therefore we need variable partition allocation

Efficient Disk Allocation

- Allocate space in large, contiguous extents
 - Few seeks, large DMA transfers
- Variable partition disk allocation is difficult
 - More complicated to find something that fits than to always use a single allocation size
 - Many files are allocated for a very long time
- External fragmentation eventually wins
 - New files get smaller chunks, farther apart
 - File system performance degrades with age

Caching

- Caching for reads
- Caching for writes

Read Caching

- Disk I/O takes a very long time
 - Deep queues, large transfers improve efficiency
 - They do not make it significantly faster
- We must eliminate much of our disk I/O
 - Maintain an in-memory cache
 - Depend on locality, reuse of the same blocks
 - Check cache before scheduling I/O

Read-Ahead

- Request blocks from the disk before any process asked for them
- Reduces process wait time
- When does it make sense?
 - When client specifically requests sequential access
 - When client seems to be reading sequentially
- What are the risks?
 - May waste disk access time reading unwanted blocks
 - May waste buffer space on unneeded blocks

Write Caching

- Most disk writes go to a write-back cache
 - They will be flushed out to disk later
- Aggregates small writes into large writes
 - If application does less than full block writes
- Eliminates moot writes
 - If application subsequently rewrites the same data
 - If application subsequently deletes the file
- Accumulates large batches of writes
 - A deeper queue to enable better disk scheduling

Common Types of Disk Caching

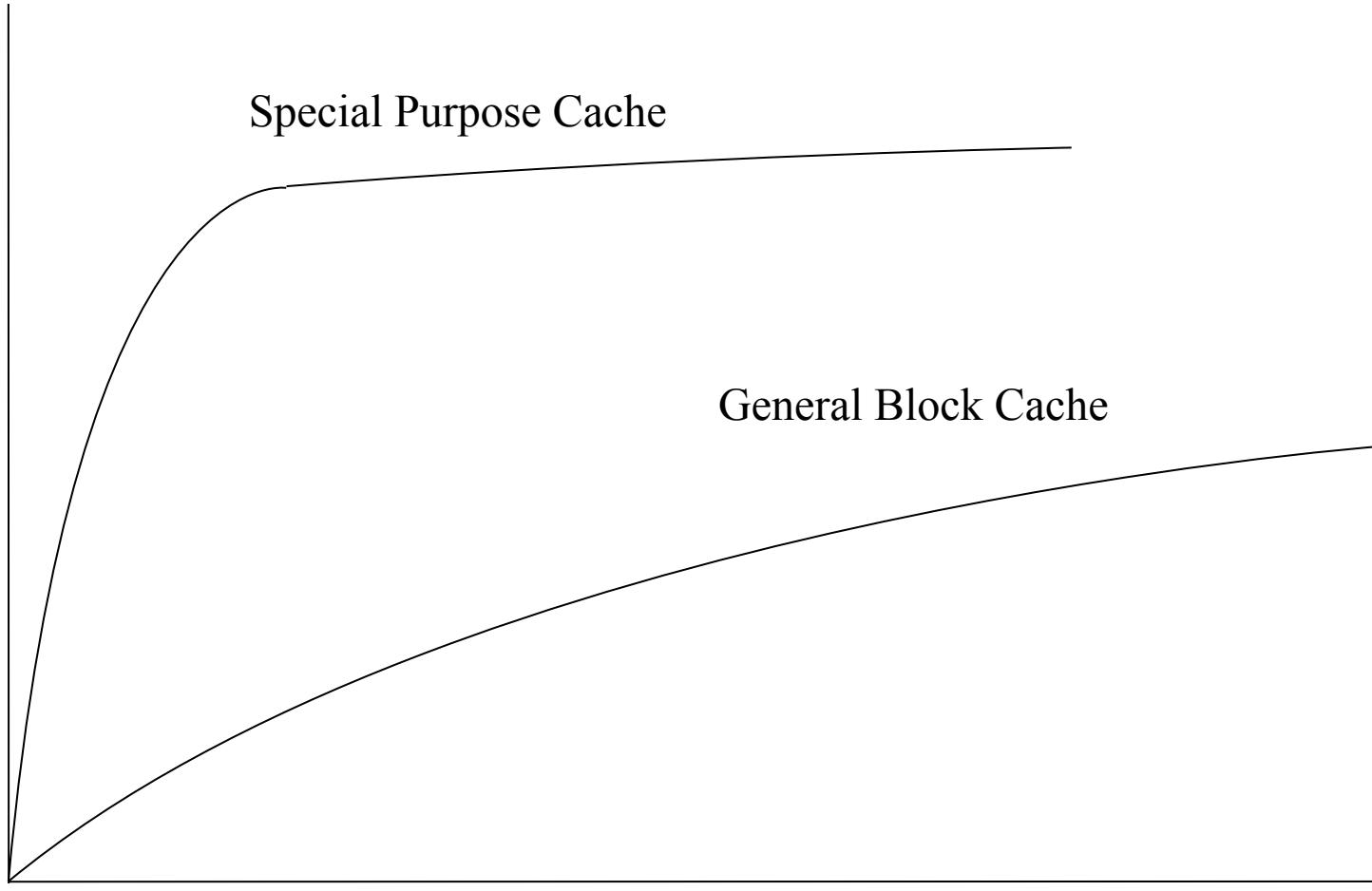
- General block caching
 - Popular files that are read frequently
 - Files that are written and then promptly re-read
 - Provides buffers for read-ahead and deferred write
- Special purpose caches
 - Directory caches speed up searches of same dirs
 - Inode caches speed up re-uses of same file
- Special purpose caches are more complex
 - But they often work much better by matching cache granularities to actual needs

Performance Gain For Different Types of Caches

Performance

Special Purpose Cache

General Block Cache



Naming in File Systems

- Each file needs some kind of handle to allow us to refer to it
- Low level names (like inode numbers) aren't usable by people or even programs
- We need a better way to name our files
 - User friendly
 - Allowing for easy organization of large numbers of files
 - Readily realizable in file systems

File Names and Binding

- File system knows files by descriptor structures
- We must provide more useful names for users
- The file system must handle name-to-file mapping
 - Associating names with new files
 - Finding the underlying representation for a given name
 - Changing names associated with existing files
 - Allowing users to organize files using names
- *Name spaces* – the total collection of all names known by some naming mechanism
 - Sometimes all names that *could* be created by the mechanism

Name Space Structure

- There are many ways to structure a name space
 - Flat name spaces
 - All names exist in a single level
 - Hierarchical name spaces
 - A graph approach
 - Can be a strict tree
 - Or a more general graph (usually directed)
- Are all files on the machine under the same name structure?
- Or are there several independent name spaces?

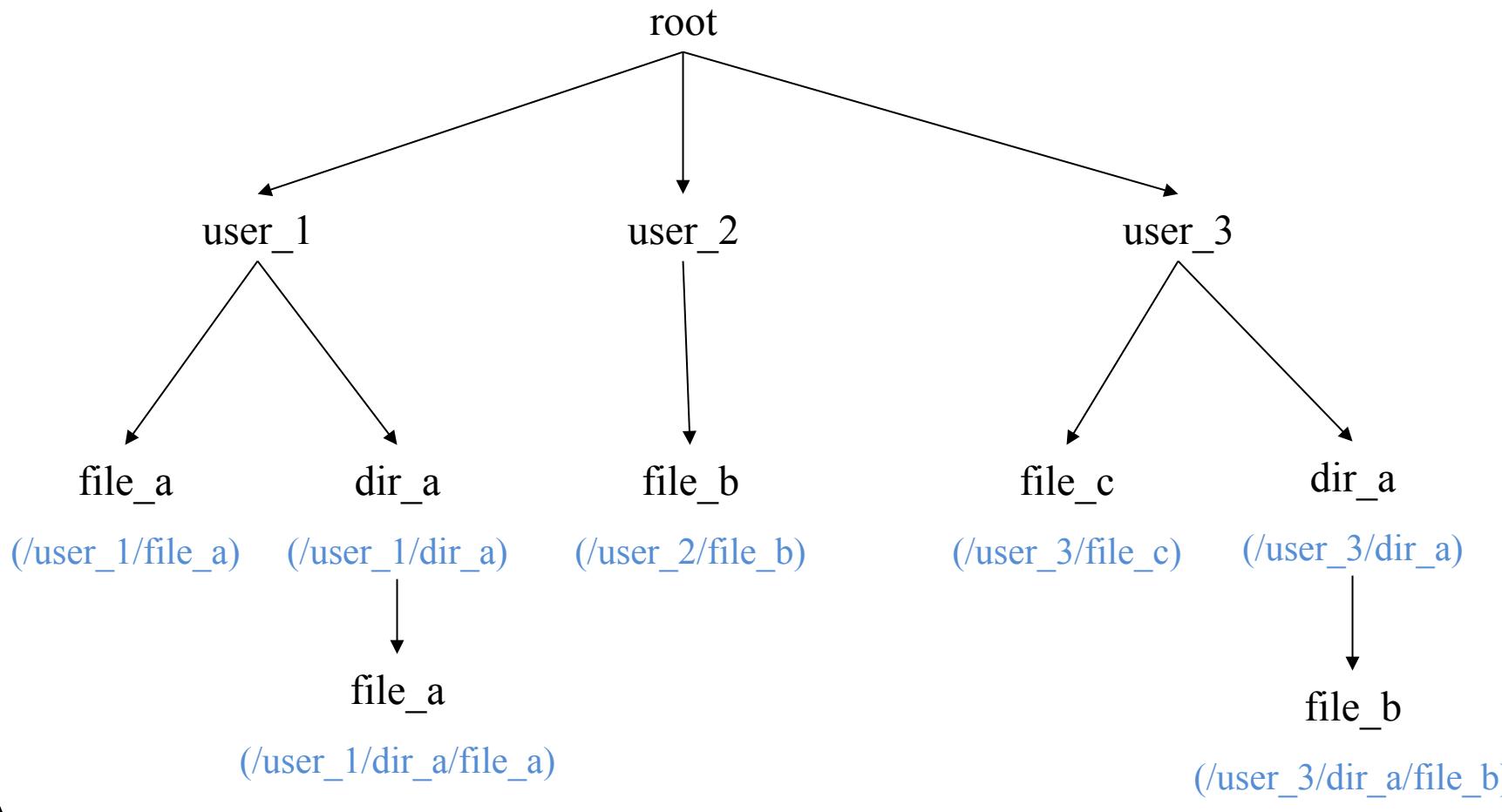
Some Issues in Name Space Structure

- How many files can have the same name?
 - One per file system ... flat name spaces
 - One per directory ... hierarchical name spaces
- How many different names can one file have?
 - A single “true name”
 - Only one “true name”, but aliases are allowed
 - Arbitrarily many
 - What’s different about “true names”?
- Do different names have different characteristics?
 - Does deleting one name make others disappear too?
 - Do all names see the same access permissions?

Hierarchical Name Spaces

- Essentially a graphical organization
- Typically organized using directories
 - A file containing references to other files
 - A non-leaf node in the graph
 - It can be used as a naming context
 - Each process has a *current directory*
 - File names are interpreted relative to that directory
- Nested directories can form a tree
 - A file name describes a path through that tree
 - The directory tree expands from a “root” node
 - A name beginning from root is called “fully qualified”
 - May actually form a directed graph
 - If files are allowed to have multiple names

A Rooted Directory Tree



Directories Are Files

- Directories are a special type of file
 - Used by OS to map file names into the associated files
- A directory contains multiple directory entries
 - Each directory entry describes one file and its name
- User applications are allowed to read directories
 - To get information about each file
 - To find out what files exist
- Usually only the OS is allowed to write them
 - Users can cause writes through special system calls
 - The file system depends on the integrity of directories

Traversing the Directory Tree

- Some entries in directories point to child directories
 - Describing a lower level in the hierarchy
- To name a file at that level, name the parent directory and the child directory, then the file
 - With some kind of delimiter separating the file name components
- Moving up the hierarchy is often useful
 - Directories usually have special entry for parent
 - Many file systems use the name “..” for that

File Names Vs. Path Names

- In some name space systems, files had “true names”
 - Only one possible name for a file,
 - Kept in a record somewhere
- E.g., in DOS, a file is described by a directory entry
 - Local name is specified in that directory entry
 - Fully qualified name is the path to that directory entry
 - E.g., start from root, to user_3, to dir_a, to file_b
- What if files had no intrinsic names of their own?
 - All names came from directory paths

Example: Unix Directories

- A file system that allows multiple file names
 - So there is no single “true” file name, unlike DOS
- File names separated by slashes
 - E.g., /user_3/dir_a/file_b
- The actual file descriptors are the inodes
 - Directory entries only point to inodes
 - Association of a name with an inode is called a *hard link*
 - Multiple directory entries can point to the same inode
- Contents of a Unix directory entry
 - Name (relative to this directory)
 - Pointer to the inode of the associated file

Unix Directories

But what's this “.” entry?

It's a directory entry that points to the directory itself!

We'll see why that's useful later

Directory /user_3, inode #114

inode #	file name
114	.
1	..
194	dir_a
307	file_c

Here's a “..” entry, pointing to the parent directory

Root directory, inode #1
inode # file name

1	.
1	..
9	user_1
31	user_2
114	user_3

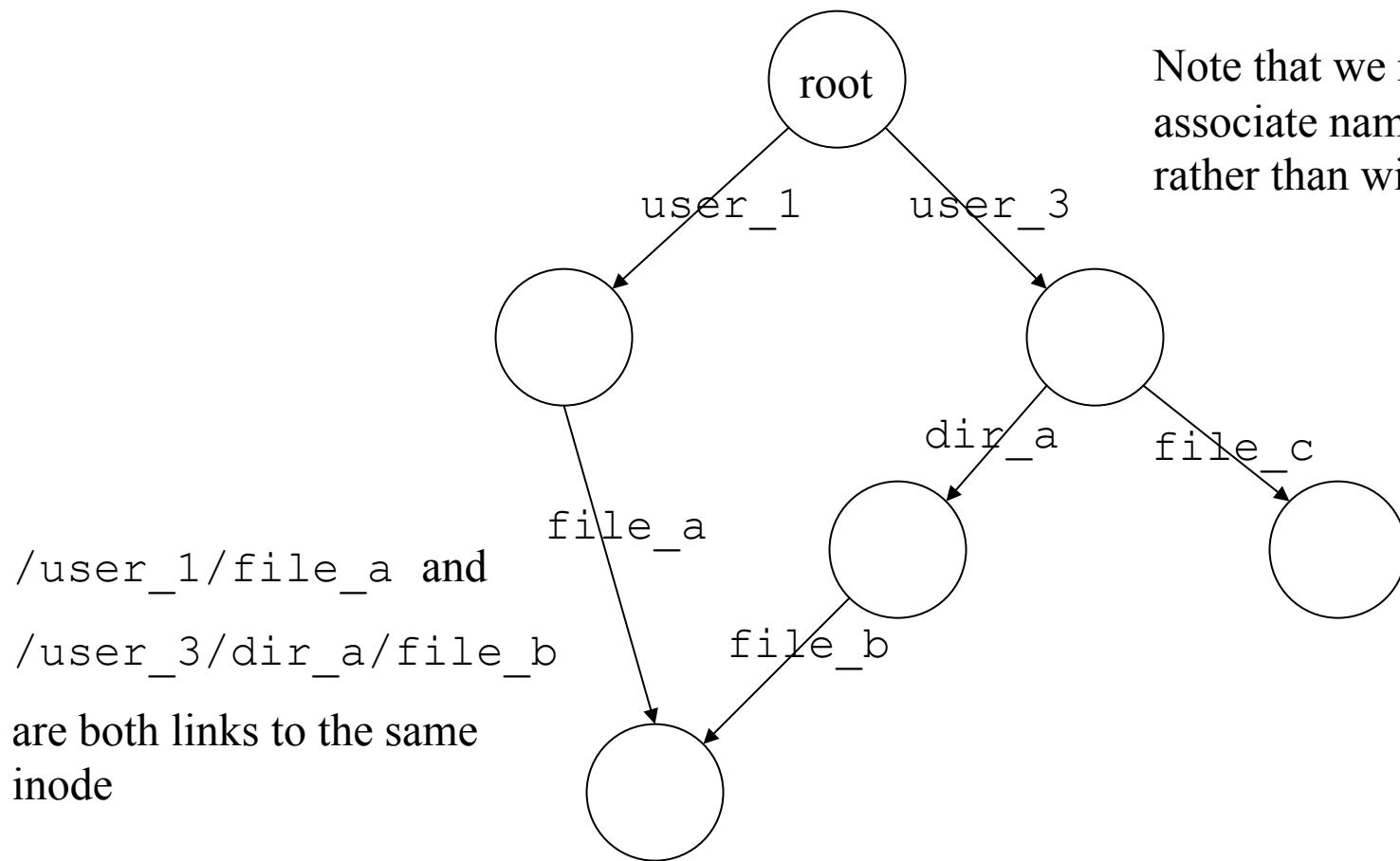
Multiple File Names In Unix

- How do links relate to files?
 - They're the names only
- All other metadata is stored in the file inode
 - File owner sets file protection (e.g., read-only)
- All links provide the same access to the file
 - Anyone with read access to file can create new link
 - But directories are protected files too
 - Not everyone has read or search access to every directory
- All links are equal
 - There is nothing special about the first (or owner's) link

Links and De-allocation

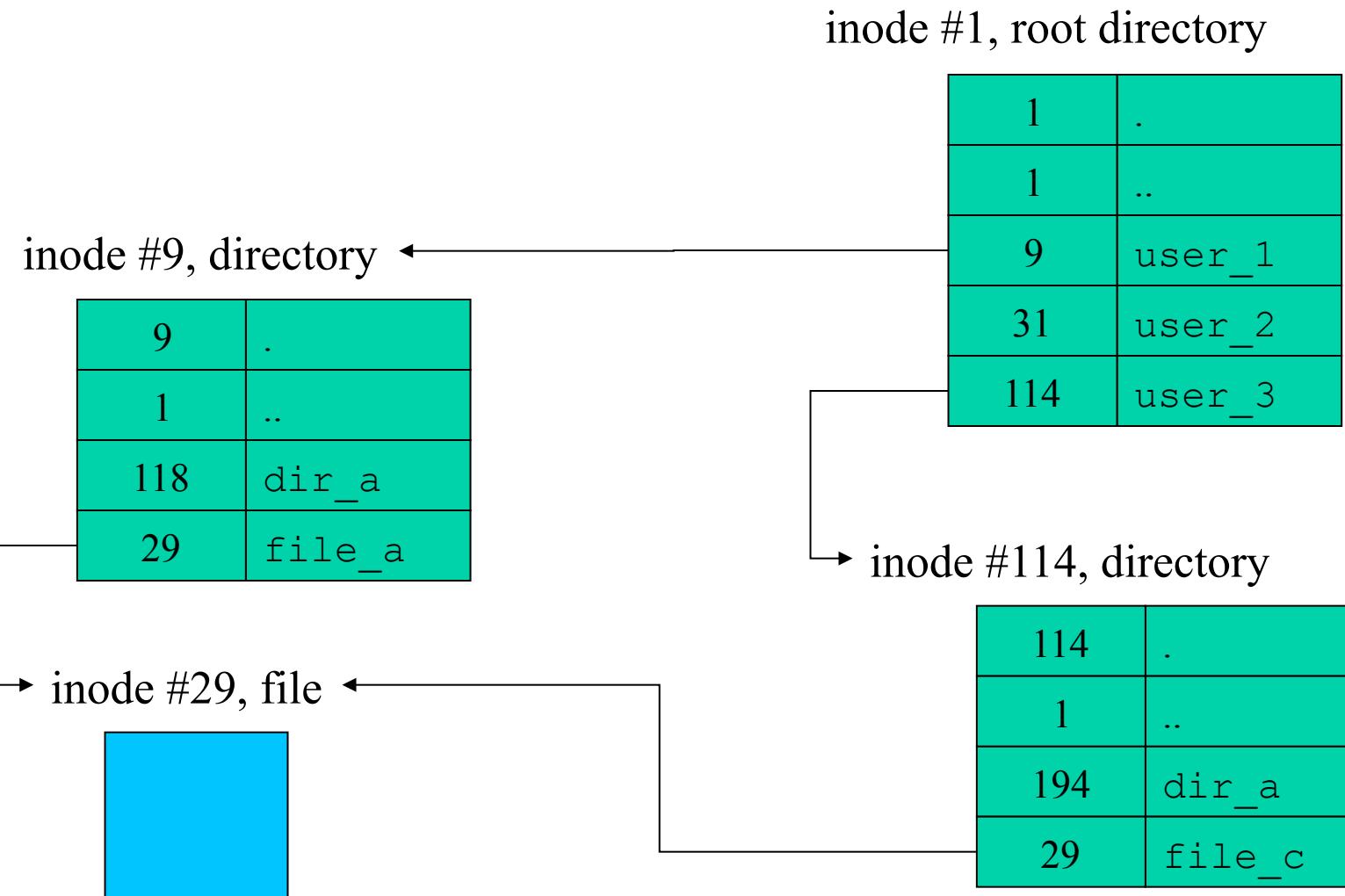
- Files exist under multiple names
- What do we do if one name is removed?
- If we also removed the file itself, what about the other names?
 - Do they now point to something non-existent?
- The Unix solution says the file exists as long as at least one name exists
- Implying we must keep and maintain a reference count of links
 - In the file inode, not in a directory

Unix Hard Link Example



Note that we now associate names with links rather than with files.

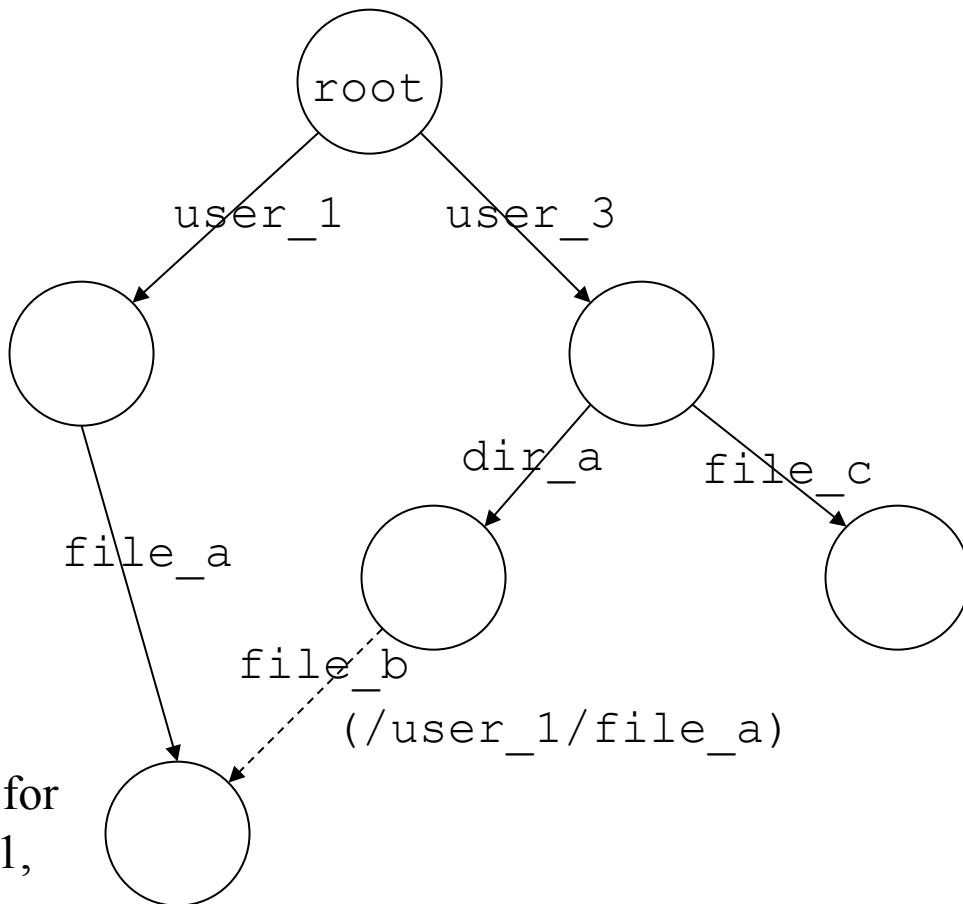
Hard Links, Directories, and Files



Symbolic Links

- A different way of giving files multiple names
- Symbolic links implemented as a special type of file
 - An indirect reference to some other file
 - Contents is a path name to another file
- OS recognizes symbolic links
 - Automatically opens associated file instead
 - If file is inaccessible or non-existent, the open fails
- Symbolic link is not a reference to the inode
 - Symbolic links will not prevent deletion
 - Do not guarantee ability to follow the specified path
 - Internet URLs are similar to symbolic links

Symbolic Link Example



The link count for
this file is still 1,
though

Symbolic Links, Files, and Directories

inode #1, root directory

inode #9, directory

9	.
1	..
118	dir_a
29	file_a

inode #29, file



Link count
still equals 1!

1	.
1	..
9	user_1
31	user_2
114	user_3

inode #114, directory

114	.
1	..
194	dir_a
46	file_c

inode #46, symlink

/user_1/file_a

File Systems Reliability

- What can go wrong in a file system?
- Data loss
 - File or data is no longer present
 - Some/all of data cannot be correctly read back
- File system corruption
 - Lost free space
 - References to non-existent files
 - Corrupted free-list multiply allocates space
 - File contents over-written by something else
 - Corrupted directories make files un-findable
 - Corrupted inodes lose file info/pointers

Storage Device Failures

- Unrecoverable read errors
 - Signal degrades beyond ECC ability to correct
 - Background *scrubbing* can greatly reduce
- Misdirected or incomplete writes
 - Detectable with independent checksums
- Complete mechanical/electronic failures
- All are correctable with redundant copies
 - Mirroring, parity, or erasure coding
 - Individual block or whole volume recovery
 - At worst, backup

File Systems – System Failures

- Caused by system crashes or OS bugs
- Queued writes that don't get completed
 - Client writes that will not be persisted
 - Client creates that will not be persisted
 - Partial multi-block file system updates
- Can also be caused by power failures
 - Solution: NVRAM disk controllers
 - Solution: Uninterruptable Power Supply (UPS)
 - Solution: super-caps and fast flush

Deferred Writes – A Worst Case Scenario

- Process allocates a new block to file A
 - We get a new block (x) from the free list
 - We write out the updated inode for file A
 - We defer free-list write-back (happens all the time)
- The system crashes, and after it reboots
 - A new process wants a new block for file B
 - We get block x from the (stale) free list
- Two different files now contain the same block
 - When file A is written, file B gets corrupted
 - When file B is written, file A gets corrupted

Robustness – Ordered Writes

- Carefully ordered writes can reduce potential damage
- Write out data before writing pointers to it
 - Unreferenced objects can be garbage collected
 - Pointers to incorrect info are more serious
- Write out deallocations before allocations
 - Disassociate resources from old files ASAP
 - Free list can be corrected by garbage collection
 - Shared data is more serious than missing data

Practicality of Ordered Writes

- Greatly reduced I/O performance
 - Eliminates head/disk motion scheduling
 - Eliminates accumulation of near-by operations
 - Eliminates consolidation of updates to same block
- May not be possible
 - Modern disk drives re-order queued requests
- Doesn't actually solve the problem
 - Does not eliminate incomplete writes
 - It chooses minor problems over major ones

Robustness – Audit and Repair

- Design file system structures for audit and repair
 - Redundant information in multiple distinct places
 - Maintain reference counts in each object
 - Children have pointers back to their parents
 - Transaction logs of all updates
 - All resources can be garbage collected
 - Discover and recover unreferenced objects
- Audit file system for correctness (prior to mount)
 - All objects are well formatted
 - All references and free-lists are correct and consistent
- Use redundant info to enable automatic repair

Practicality of Audit and Repair

- Integrity checking a file system after a crash
 - Verifying check-sums, reference counts, etc.
 - Automatically correct any inconsistencies
 - A standard practice for many years (see *fsck(8)*)
- No longer practical
 - Checking a 2TB FS at 100MB/second = 5.5 hours
- We need more efficient partial write solutions
 - File systems that are immune to them
 - File systems that enable very fast recovery

Journaling

- Create a circular buffer journaling device
 - Journal writes are always sequential
 - Journal writes can be batched
 - Journal is relatively small, may use NVRAM
- Journal all intended file system updates
 - Inode updates, block write/alloc/free
- Efficiently schedule actual file system updates
 - Write-back cache, batching, motion-scheduling
- Journal completions when real writes happen

Batched Journal Entries

- Operation is safe after journal entry persisted
 - Caller must wait for this to happen
- Small writes are still inefficient
- Accumulate batch until full or max wait time

writer:

```
if there is no current in-memory journal page
    allocate a new page
add my transaction to the current journal page
if current journal page is now full
    do the write, await completion
    wake up processes waiting for this page
else
    start timer, sleep until I/O is done
```

flusher:

```
while true
    sleep()
    if current-in-memory page is due
        close page to further updates
        do the write, await completion
        wake up processes waiting for page
```

Journal Recovery

- Journal is a circular buffer
 - It can be recycled after old ops have completed
 - Time-stamps distinguish new entries from old
- After system restart
 - Review entire (relatively small) journal
 - Note which ops are known to have completed
 - Perform all writes not known to have completed
 - Data and destination are both in the journal
 - All of these write operations are idempotent
 - Truncate journal and resume normal operation

Why Does Journaling Work?

- Journal writes much faster than data writes
 - All journal writes are sequential
 - There is no competing head motion
- In normal operation, journal is write-only
 - File system never reads/processes the journal
- Scanning the journal on restart is very fast
 - It is very small (compared to the file system)
 - It can read (sequentially) with huge (efficient) reads
 - All recovery processing is done in memory

Meta-Data Only Journaling

- Why journal meta-data?
 - It is small and random (very I/O inefficient)
 - It is integrity-critical (huge potential data loss)
- Why not journal data?
 - It is often large and sequential (I/O efficient)
 - It would consume most of journal capacity bandwidth
 - It is less order sensitive (just precede meta-data)
- Safe meta-data journaling
 - Allocate new space, write the data
 - Then journal the meta-data updates

Log Structured File Systems

- The journal is the file system
 - All inodes and data updates written to the log
 - Updates are Redirect-on-Write
 - In-memory index caches inode locations
- Becoming a dominant architecture
 - Flash file systems
 - Key/value stores
- Issues
 - Recovery time (to reconstruct index/cache)
 - Log defragmentation and garbage collection

Navigating a Logging File System

- Inodes point at data segments in the log
 - Sequential writes may be contiguous in log
 - Random updates can be spread all over the log
- Updated inodes are added to end of the log
- Index points to latest version of each inode
 - Index is periodically appended to the log
- Recovery
 - Find and recover the latest index
 - Replay all log updates since then

Redirect on Write

- Many modern file systems now do this
 - Once written, blocks and inodes are immutable
 - Add new info to the log, and update the index
- The old inodes and data remain in the log
 - If we have an old index, we can access them
 - Clones and snapshots are almost free
- Price is management and garbage collection
 - We must inventory and manage old versions
 - We must eventually recycle old log entries

Operating System Principles: Security and Privacy

CS 111

Operating Systems
Peter Reiher

Outline

- Introduction
- Authentication
- Access control
- Cryptography

Introduction

- Operating systems provide the lowest layer of software visible to users
- Operating systems are close to the hardware
 - Often have complete hardware access
- If the operating system isn't protected, the machine isn't protected
- Flaws in the OS generally compromise all security at higher levels

Why Is OS Security So Important?

- The OS controls access to application memory
- The OS controls scheduling of the processor
- The OS ensures that users receive the resources they ask for
- If the OS isn't doing these things securely, practically anything can go wrong
- So almost all other security systems must assume a secure OS at the bottom

Some Important Definitions

- Security
- Protection
- Vulnerabilities
- Exploits
- Trust
- Authentication and authorization

Security and Protection

- *Security* is a policy
 - E.g., “no unauthorized user may access this file”
- *Protection* is a mechanism
 - E.g., “the system checks user identity against access permissions”
- Protection mechanisms implement security policies

Vulnerabilities and Exploits

- A *vulnerability* is a weakness that can allow an attacker to cause problems
 - Not all vulnerabilities can cause all problems
 - Most vulnerabilities are never exploited
- An *exploit* is an actual incident of taking advantage of a vulnerability
 - Allowing attacker to do something bad on some particular machine
 - Term also refers to the code or methodology used to take advantage of a vulnerability

Trust

- An extremely important security concept
- You do certain things for those you trust
- You don't do them for those you don't
- Seems simple, but . . .
 - How do you express trust?
 - Why do you trust something?
 - How can you be sure who you're dealing with?
 - What if trust is situational?
 - What if trust changes?

Trust and the Operating System

- You pretty much have to trust your operating system
- It controls all the hardware, including the memory
- It controls how your processes are handled
- It controls all the I/O devices
- If your OS is out to get you, you're gotten
- Which implies compromising an OS is a big deal

Authentication and Authorization

- In many security situations, we need to know who wants to do something
 - We allow trusted parties to do it
 - We don't allow others to do it
- That means we need to know who's asking
 - Determining that is *authentication*
- Then we need to check if that party should be allowed to do it
 - Determining that is *authorization*
 - Authorization usually requires authentication

Authentication

- Security policies tend to allow some parties to do something, but not others
- Which implies we need to know who's doing the asking
- For OS purposes, that's a determination made by a computer
- How?

Real World Authentication

- Identification by recognition
 - I see your face and know who you are
- Identification by credentials
 - You show me your driver's license
- Identification by knowledge
 - You tell me something only you know
- Identification by location
 - You're behind the counter at the DMV
- These all have cyber analogs

Authentication With a Computer

- Not as smart as a human
 - Steps to prove identity must be well defined
- Can't do certain things as well
 - E.g., face recognition
- But lightning fast on computations and less prone to simple errors
 - Mathematical methods are acceptable
- Often must authenticate non-human entities
 - Like processes or machines

Identities in Operating Systems

- We usually rely primarily on a user ID
 - Which uniquely identifies some user
 - Processes run on his behalf, so they inherit his ID
 - E.g., a forked process has the same user associated as the parent did
- Implies a model where any process belonging to a user has all his privileges
 - Which has its drawbacks
 - But that's what we use

Bootstrapping OS Authentication

- Processes inherit their user IDs
- But somewhere along the line we have to create a process belonging to a new user
 - Typically on login to a system
- We can't just inherit that identity
- How can we tell who this newly arrived user is?

Passwords

- Authenticate the user by what he knows
 - A secret word he supplies to the system on login
- System must be able to check that the password was correct
 - Either by storing it
 - Or storing a hash of it
 - That's a much better option
- If correct, tie user ID to a new command shell or window management process

Problems With Passwords

- They have to be unguessable
 - Yet easy for people to remember
- If networks connect remote devices to computers, susceptible to password sniffers
 - Programs which read data from the network, extracting passwords when they see them
- Unless quite long, brute force attacks often work on them
- Widely regarded as an outdated technology
- But extremely widely used

Proper Use of Passwords

- Passwords should be sufficiently long
- Passwords should contain non-alphabetic characters
- Passwords should be unguessable
- Passwords should be changed often
- Passwords should never be written down
- Passwords should never be shared
- Hard to achieve all this simultaneously

Challenge/Response Systems

- Authentication by what questions you can answer correctly
 - Again, by what you know
- The system asks the user to provide some information
- If it's provided correctly, the user is authenticated
- Safest if it's a different question every time
 - Not very practical

Hardware-Based Challenge/ Response

- The challenge is sent to a hardware device belonging to the appropriate user
 - Authentication based on what you have
- Sometimes mere possession of device is enough
 - E.g., text challenges sent to a smart phone to be typed into web request
- Sometimes the device performs a secret function on the challenge
 - E.g., smart cards

Problems With Challenge/Response

- If based on what you know, usually too few unique and secret challenge/response pairs
- If based on what you have, fails if you don't have it
 - And whoever does have it might pose as you
- Some forms susceptible to network sniffing
 - Much like password sniffing
 - Smart card versions usually not susceptible

Biometric Authentication

- Authentication based on what you are
- Measure some physical attribute of the user
 - Things like fingerprints, voice patterns, retinal patterns, etc.
- Convert it into a binary representation
- Check the representation against a stored value for that attribute
- If it's a close match, authenticate the user

Problems With Biometric Authentication

- Requires very special hardware
 - With some minor exceptions
- Many physical characteristics vary too much for practical use
- Generally not helpful for authenticating programs or roles
- Requires special care when done across a network

Errors in Biometric Authentication

- False positives
 - You identified Bill Smith as Peter Reiher
 - Probably because your biometric system was too generous in making matches
 - Bill Smith can pretend to be me
- False negatives
 - You didn't identify Peter Reiher as Peter Reiher
 - Probably because your biometric system was too stingy in making matches
 - I can't log in to my own account

Biometrics and Remote Authentication

- The biometric reading is just a bit pattern
- If attacker can obtain a copy, he can send the pattern over the network
 - Without actually performing a biometric reading
- Requires high confidence in security of path between biometric reader and checking device
 - Usually OK when both are on the same machine
 - Problematic when the Internet is between them

Access Control in Operating Systems

- The OS can control which processes access which resources
- Giving it the chance to enforce security policies
- The mechanisms used to enforce policies on who can access what are called access control
- Fundamental to OS security

Goals for Access Control

- Complete mediation
- Least privilege
- Useful in a networked environment
- Scalability
- Cost and usability

Access Control Lists

- ACLs
- For each protected object, maintain a single list
 - Managed by the OS, to prevent improper alteration
- Each list entry specifies who can access the object
 - And the allowable modes of access
- When something requests access to a object, check the access control list

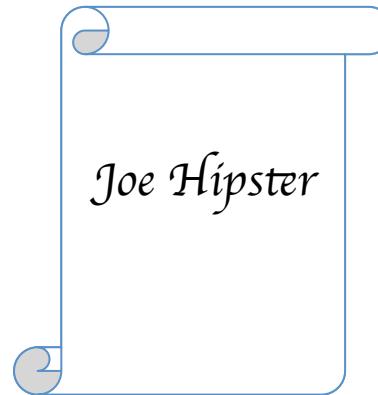
An Analogy



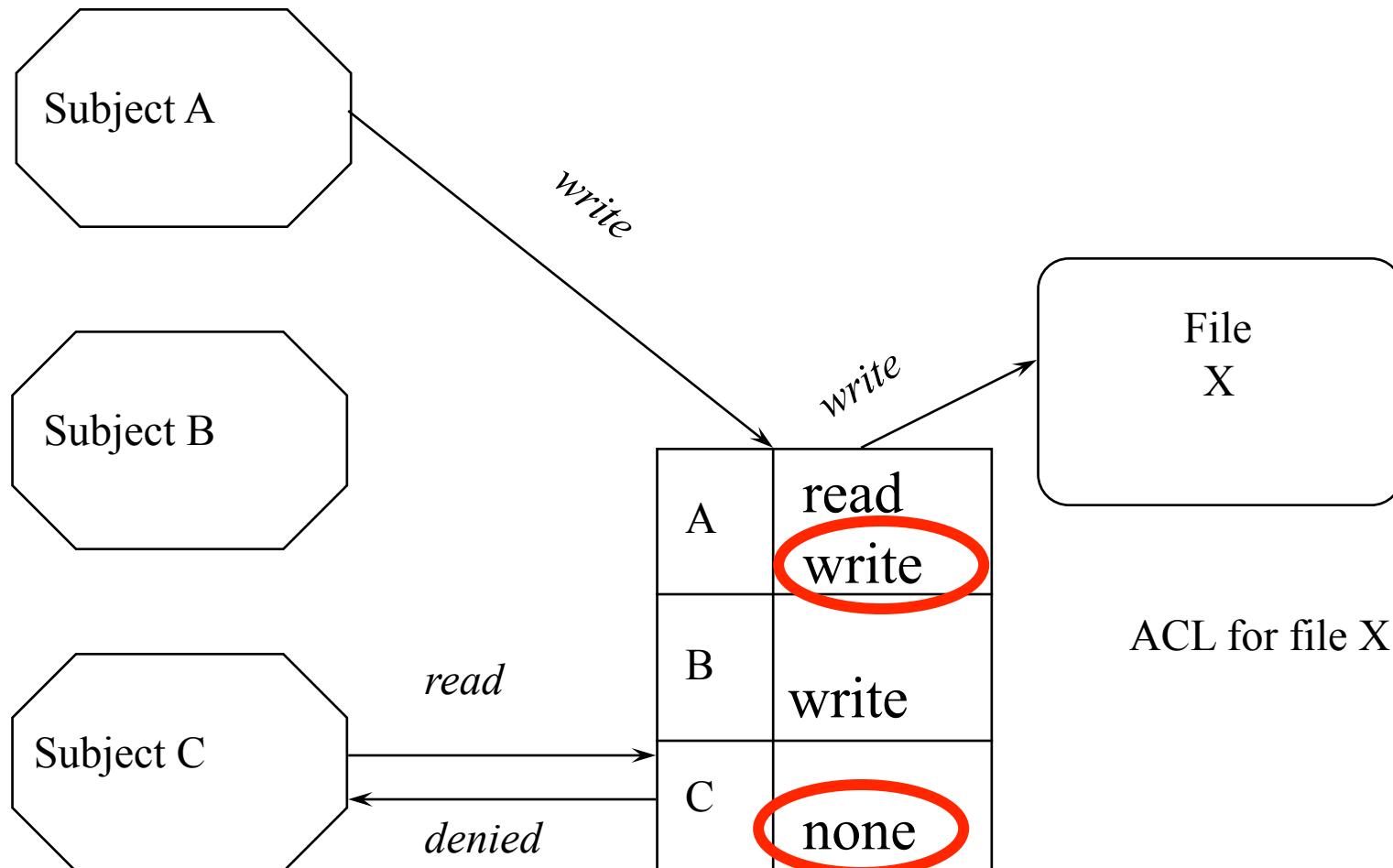
You're
Not On
the List!



This is an
access
control list



An ACL Protecting a File



An Example Use of ACLs: the Unix File System

- An ACL-based method for protecting files
 - Developed in the 1970s
- Still in very wide use today
 - With relatively few modifications
- Per-file ACLs (files are the objects)
- Three subjects on list for each file
 - Owner, group, other
- And three modes
 - Read, write, execute
 - Sometimes these have special meanings

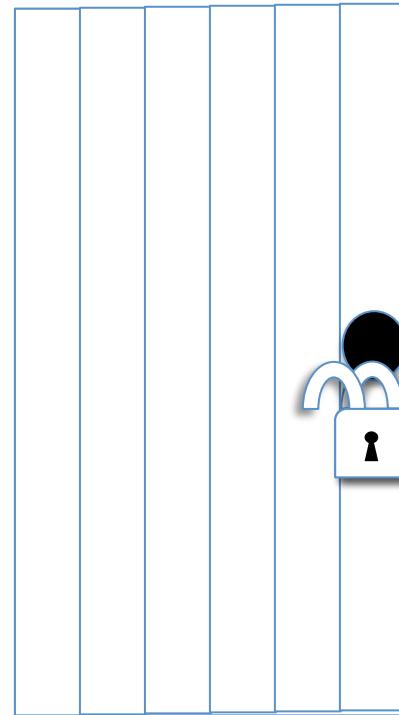
Pros and Cons of ACLs

- + Easy to figure out who can access a resource
- + Easy to revoke or change access permissions
- Hard to figure out what a subject can access
- Changing access rights requires getting to the object

Capabilities

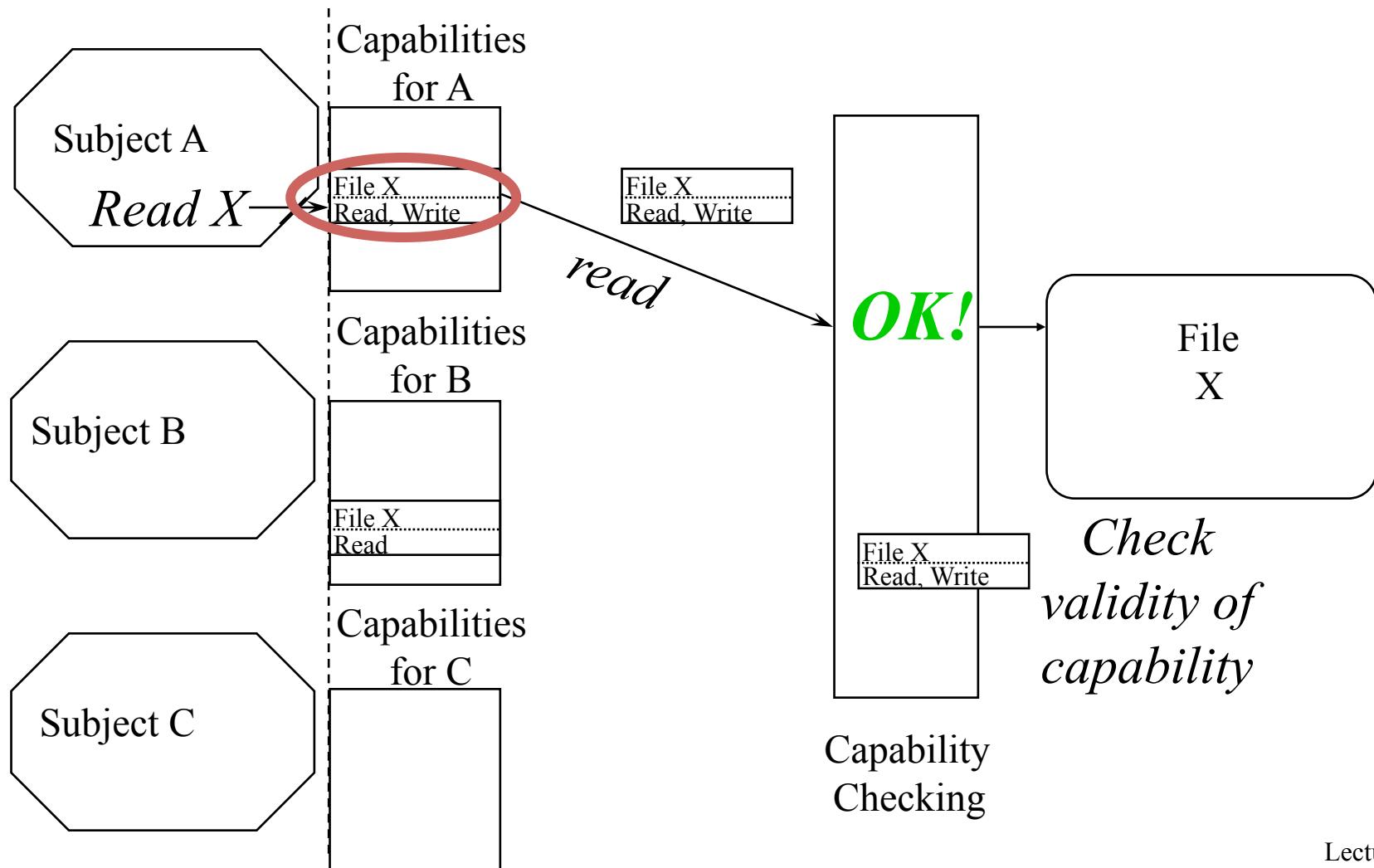
- Each entity keeps a set of data items that specify his allowable accesses
- Essentially, a set of tickets
- To access an object, present the proper capability
- Possession of the capability for an object implies that access is allowed

An Analogy



The key is a capability

Capabilities Protecting a File



Properties of Capabilities

- Capabilities are essentially a data structure
 - Ultimately, just a collection of bits
- Merely possessing the capability grants access
 - So they must not be forgeable
- How do we ensure unforgeability for a collection of bits?
- One solution:
 - Don't let the user/process have them
 - Store them in the operating system

Pros and Cons of Capabilities

- + Easy to determine what objects a subject can access
- + Potentially faster than ACLs (in some circumstances)
- + Easy model for transfer of privileges
- Hard to determine who can access an object
- Requires extra mechanism to allow revocation
- In network environment, need cryptographic methods to prevent forgery

OS Use of Access Control

- Operating systems often use both ACLs and capabilities
 - Sometimes for the same resource
- E.g., Unix/Linux uses ACLs for file opens
- That creates a file descriptor with a particular set of access rights
 - E.g., read-only
- The descriptor is essentially a capability

Enforcing Access in an OS

- Protected resources must be inaccessible
 - Hardware protection must be used to ensure this
 - So only the OS can make them accessible to a process
- To get access, issue request to OS
 - OS consults access control policy data
- Access may be granted directly
 - Resource manager maps resource into process
- Access may be granted indirectly
 - Resource manager returns a “capability” to process

Cryptography

- Much of computer security is about keeping secrets
- One method of doing so is to make it hard for others to read the secrets
- While (usually) making it simple for authorized parties to read them
- That's what cryptography is all about
 - Transforming bit patterns in controlled ways to obtain security advantages

Cryptography Terminology

- Typically described in terms of sending a message
 - Though it's used for many other purposes
- The sender is S
- The receiver is R
- *Encryption* is the process of making message unreadable/unalterable by anyone but R
- *Decryption* is the process of making the encrypted message readable by R
- A system performing these transformations is a *cryptosystem*
 - Rules for transformation sometimes called a *cipher*

Plaintext and Ciphertext

- *Plaintext* is the original form of the message (often referred to as P)
- *Ciphertext* is the encrypted form of the message (often referred to as C)

Transfer \$100
to my savings
account

Sqzmredq
#099 sn Ix
rzuhmfr
zbbntms

Cryptographic Keys

- Most cryptographic algorithms use a *key* to perform encryption and decryption
 - Referred to as K
- The key is a secret
- Without the key, decryption is hard
- With the key, decryption is easy
- Reduces the secrecy problem from your (long) message to the (short) key
 - But there's still a secret

More Terminology

- The encryption algorithm is referred to as $E()$
- $C = E(K, P)$
- The decryption algorithm is referred to as $D()$
- The decryption algorithm also has a key
- The combination of the two algorithms are often called a *cryptosystem*

Symmetric Cryptosystems

- $C = E(K, P)$
- $P = D(K, C)$
- $P = D(K, E(K, P))$
- $E()$ and $D()$ are not necessarily the same operations

Advantages of Symmetric Cryptosystems

- + Encryption and authentication performed in a single operation
- + Well-known (and trusted) ones perform much faster than asymmetric key systems
- + No centralized authority required
 - Though key servers help a lot

Disadvantages of Symmetric Cryptosystems

- Encryption and authentication performed in a single operation
 - Makes signature more difficult
- Non-repudiation hard without servers
- Key distribution can be a problem
- Scaling
 - Especially for Internet use

Some Popular Symmetric Ciphers

- The Data Encryption Standard (DES)
 - The old US encryption standard
 - Still fairly widely used, due to legacy
 - Weak by modern standards
- The Advanced Encryption Standard (AES)
 - The current US encryption standard
 - Probably the most widely used cipher
- Blowfish
- There are many, many others

Symmetric Ciphers and Brute Force Attacks

- If your symmetric cipher has no flaws, how can attackers crack it?
- *Brute force* – try every possible key until one works
- The cost of brute force attacks depends on key length
 - For N possible keys, attack must try $N/2$ keys, on average, before finding the right one
- DES uses 56 bit keys
 - Too short for modern brute force attacks
- AES uses 128 or 256 bit keys
 - Long enough

Asymmetric Cryptosystems

- Often called *public key cryptography*
 - Or PK, for short
- Encryption and decryption use different keys
 - $C = E(K_E, P)$
 - $P = D(K_D, C)$
 - $P = D(K_D, E(K_E, P))$
- Often works the other way, too
 - $C' = E(K_D, P)$
 - $P = D(K_E, C')$
 - $P = D(K_D, E(K_E, P))$

Using Public Key Cryptography

- Keys are created in pairs
- One key is kept secret by the owner
- The other is made public to the world
 - Hence the name
- If you want to send an encrypted message to someone, encrypt with his public key
 - Only he has private key to decrypt

Authentication With Public Keys

- If I want to “sign” a message, encrypt it with my private key
- Only I know private key, so no one else could create that message
- Everyone knows my public key, so everyone can check my claim directly
- Much better than with symmetric crypto
 - The receiver could not have created the message
 - Only the sender could have

Issues With PK Key Distribution

- Security of public key cryptography depends on using the right public key
- If I am fooled into using wrong one, that key's owner reads my message
- Need high assurance that a given key belongs to a particular person
 - Either a *key distribution infrastructure*
 - Or use of *certificates*
- Both are problematic, at high scale and in the real world

The Nature of PK Algorithms

- Usually based on some problem in mathematics
 - Like factoring extremely large numbers
- Security less dependent on brute force
- More on the complexity of the underlying problem
- Also implies choosing key pairs is complex and expensive

Example Public Key Ciphers

- RSA
 - The most popular public key algorithm
 - Used on pretty much everyone's computer, nowadays
- Elliptic curve cryptography
 - An alternative to RSA
 - Tends to have better performance
 - Not as widely used or studied

Security of PK Systems

- Based on solving the underlying problem
 - E.g., for RSA, factoring large numbers
- In 2009, a 768 bit RSA key was successfully factored
- Research on integer factorization suggests keys up to 2048 bits may be insecure
 - In 2013, Google went from 1024 to 2048 bit keys
- Size will keep increasing
- The longer the key, the more expensive the encryption and decryption

Combined Use of Symmetric and Asymmetric Cryptography

- Very common to use both in a single session
- Asymmetric cryptography essentially used to “bootstrap” symmetric crypto
- Use RSA (or another PK algorithm) to authenticate and establish a *session key*
- Use DES or AES with session key for the rest of the transmission

For Example,

Alice wants to share K_S only
with Bob



Alice

K_{EA}

K_{DA}

K_{DB}

K_S

$$C = E(K_S, K_{DB})$$

$$M = E(C, K_{EA})$$

Bob wants to be sure it's
Alice's key



Bob

K_{EB}

K_{DB}

K_{DA}

M

$$C = D(M, K_{DA})$$

$$K_S = D(C, K_{EB})$$

Operating System Principles: Distributed Systems

CS 111

Operating Systems
Peter Reiher

Outline

- Introduction
- Distributed system paradigms
- Remote procedure calls
- Distributed synchronization and consensus
- Distributed system security

Goals of Distributed Systems

- Scalability and performance
 - Apps require more resources than one computer has
 - Grow system capacity /bandwidth to meet demand
- Improved reliability and availability
 - 24x7 service despite disk/computer/software failures
- Ease of use, with reduced operating expenses
 - Centralized management of all services and systems
 - Buy (better) services rather than computer equipment
- Enable new collaboration and business models
 - Collaborations that span system (or national) boundaries
 - A global free market for a wide range of new services

Transparency

- Ideally, a distributed system would be just like a single machine system
- But better
 - More resources
 - More reliable
 - Faster
- *Transparent* distributed systems look as much like single machine systems as possible

Deutsch's “Seven Fallacies of Network Computing”

1. The network is reliable
2. There is no latency (instant response time)
3. The available bandwidth is infinite
4. The network is secure
5. The topology of the network does not change
6. There is one administrator for the whole network
7. The cost of transporting additional data is zero

Bottom Line: true transparency is not achievable

Heterogeneity in Distributed Systems

- Distributed systems aren't uniform
- Heterogeneous clients
 - Different instruction set architectures
 - Different operating systems and versions
- Heterogeneous servers
 - Different implementations
 - Offered by competing service providers
- Heterogeneous networks
 - Public and private
 - Managed by different orgs in different countries
- Another problem for achieving transparency
 - And sometimes correctness

Fundamental Building Blocks Change

- The old model:
 - Programs run in processes
 - Programs use APIs to access system resources
 - API services implemented by OS and libraries
- The new model:
 - Clients and servers run on nodes
 - Clients use APIs to access services
 - API services are exchanged via protocols
- Local is a (very important) special case

Changing Paradigms

- Network connectivity becomes “a given”
 - New applications assume/exploit connectivity
 - New distributed programming paradigms emerge
 - New functionality depends on network services
- Applications demand new kinds of services:
 - Location independent operations
 - Rendezvous between cooperating processes
 - WAN scale communication, synchronization

Distributed System Paradigms

- Parallel processing
 - Relying on special hardware
- Single system images
 - Make all the nodes look like one big computer
 - Somewhere between hard and impossible
- Loosely coupled systems
 - Work with difficulties as best as you can
 - Typical modern approach to distributed systems
- Cloud computing
 - A recent variant

Loosely Coupled Systems

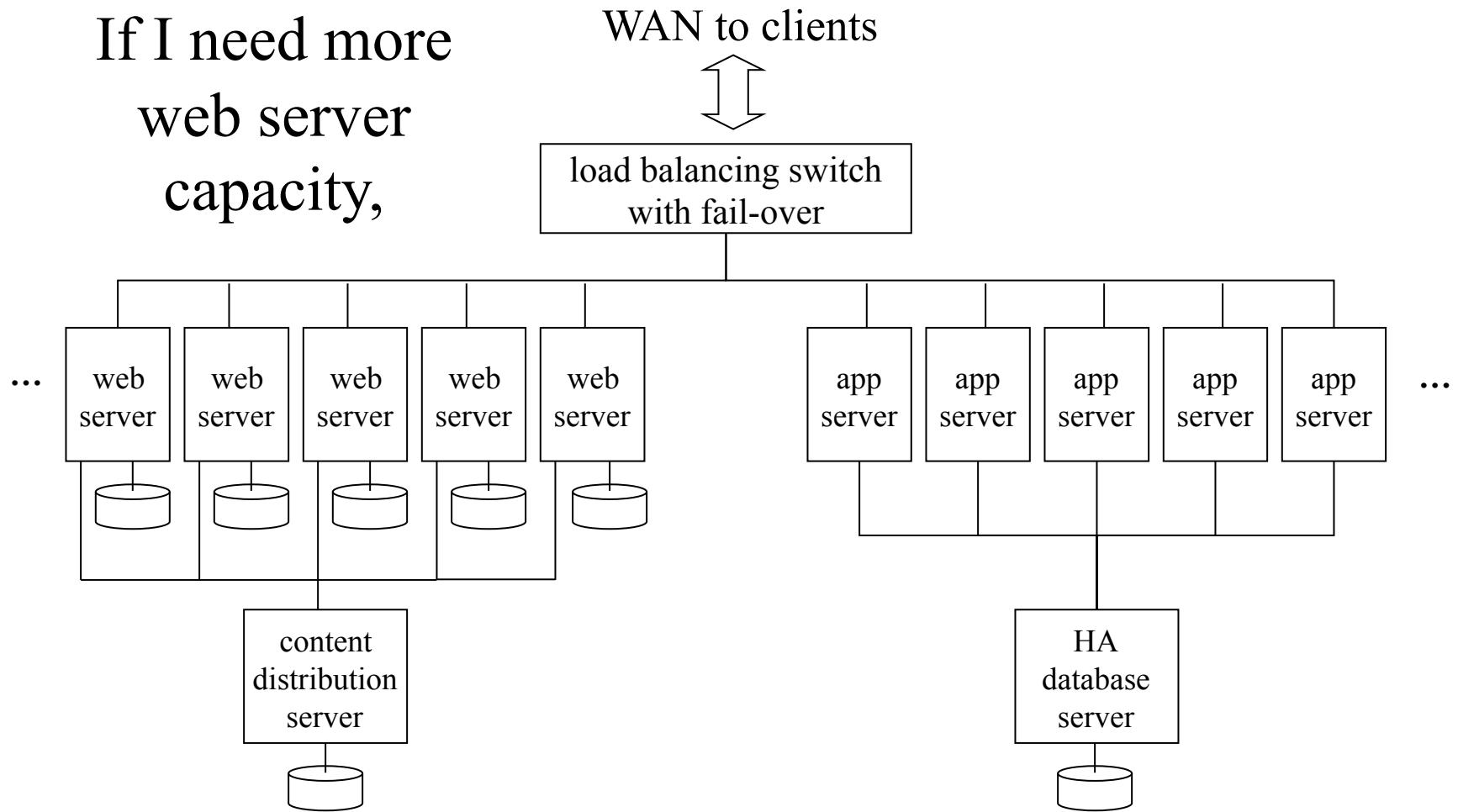
- Characterization:
 - A parallel group of independent computers
 - Serving similar but independent requests
 - Minimal coordination and cooperation required
- Motivation:
 - Scalability and price performance
 - Availability – if protocol permits stateless servers
 - Ease of management, reconfigurable capacity
- Examples:
 - Web servers, app servers, cloud computing

Horizontal Scalability

- Each node largely independent
- So you can add capacity just by adding a node “on the side”
- Scalability can be limited by network, instead of hardware or algorithms
 - Or, perhaps, by a load balancer
- Reliability is high
 - Failure of one of N nodes just reduces capacity

Horizontal Scalability Architecture

If I need more
web server
capacity,



Elements of Loosely Coupled Architecture

- Farm of independent servers
 - Servers run same software, serve different requests
 - May share a common back-end database
- Front-end switch
 - Distributes incoming requests among available servers
 - Can do both load balancing and fail-over
- Service protocol
 - Stateless servers and idempotent operations
 - Successive requests may be sent to different servers

Horizontally Scaled Performance

- Individual servers are very inexpensive
 - Blade servers may be only \$100-\$200 each
- Scalability is excellent
 - 100 servers deliver approximately 100x performance
- Service availability is excellent
 - Front-end automatically bypasses failed servers
 - Stateless servers and client retries fail-over easily
- The challenge is managing thousands of servers
 - Automated installation, global configuration services
 - Self monitoring, self-healing systems
 - Scaling limited by management, not HW or algorithms

Cloud Computing

- The most recent twist on distributed computing
- Set up a large number of machines all identically configured
- Connect them to a high speed LAN
 - And to the Internet
- Accept arbitrary jobs from remote users
- Run each job on one or more nodes
- Entire facility probably running mix of single machine and distributed jobs, simultaneously

Distributed Computing and Cloud Computing

- In one sense, these are orthogonal
- Each job submitted might or might not be distributed
- Many of the hard problems of the distributed jobs are the user's problem, not the system's
 - E.g., proper synchronization and locking
- But the cloud facility must make communications easy

What Runs in a Cloud?

- In principle, anything
- But general distributed computing is hard
- So much of the work is run using special tools
- These tools support particular kinds of parallel/distributed processing
- Either embarrassingly parallel jobs
- Or those using a method like map-reduce
- Things where the user need not be a distributed systems expert

Embarrassingly Parallel Jobs

- Problems where it's really, really easy to parallelize them
- Probably because the data sets are easily divisible
- And exactly the same things are done on each piece
- So you just parcel them out among the nodes and let each go independently
- Everyone finishes at more or less same time

MapReduce

- Perhaps the most common cloud computing software tool/technique
- A method of dividing large problems into compartmentalized pieces
- Each of which can be performed on a separate node
- With an eventual combined set of results

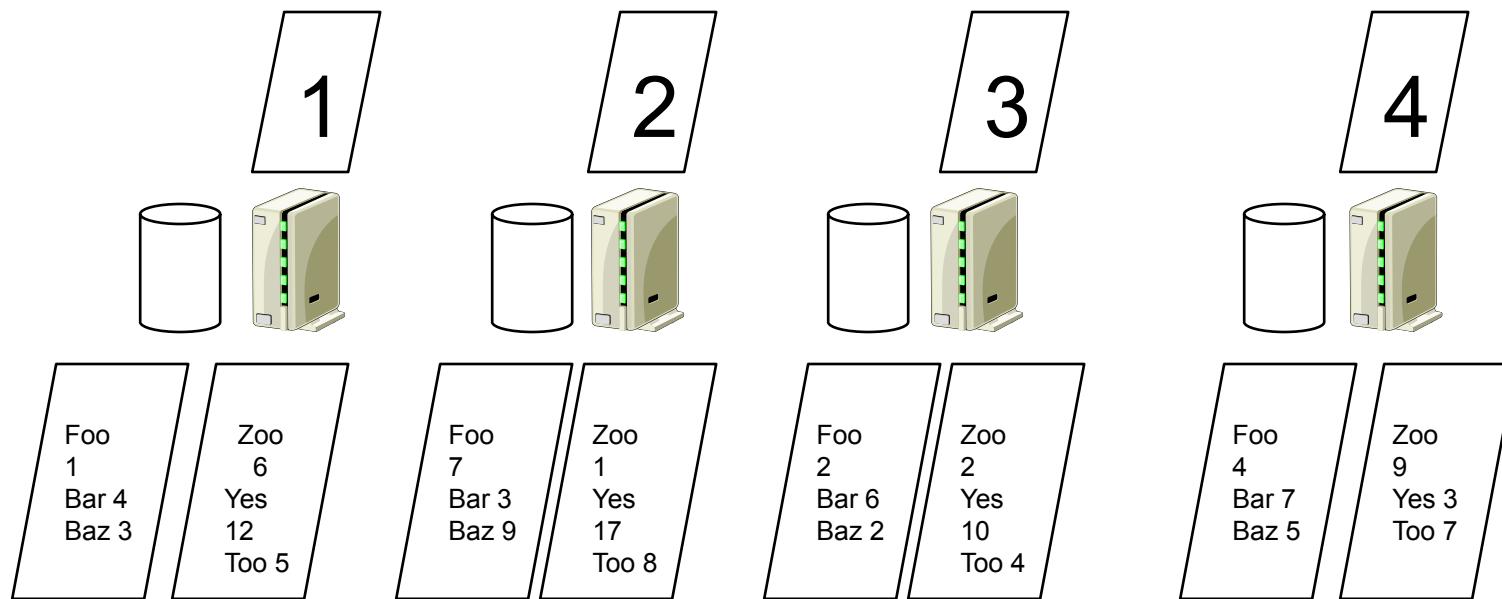
The Idea Behind MapReduce

- There is a single function you want to perform on a lot of data
 - Such as searching it for a string
- Divide the data into disjoint pieces
- Perform the function on each piece on a separate node (*map*)
- Combine the results to obtain output (*reduce*)

An Example

- We have 64 megabytes of text data
- Count how many times each word occurs in the text
- Divide it into 4 chunks of 16 Mbytes
- Assign each chunk to one processor
- Perform the map function of “count words” on each

The Example Continued

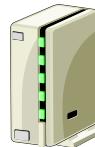
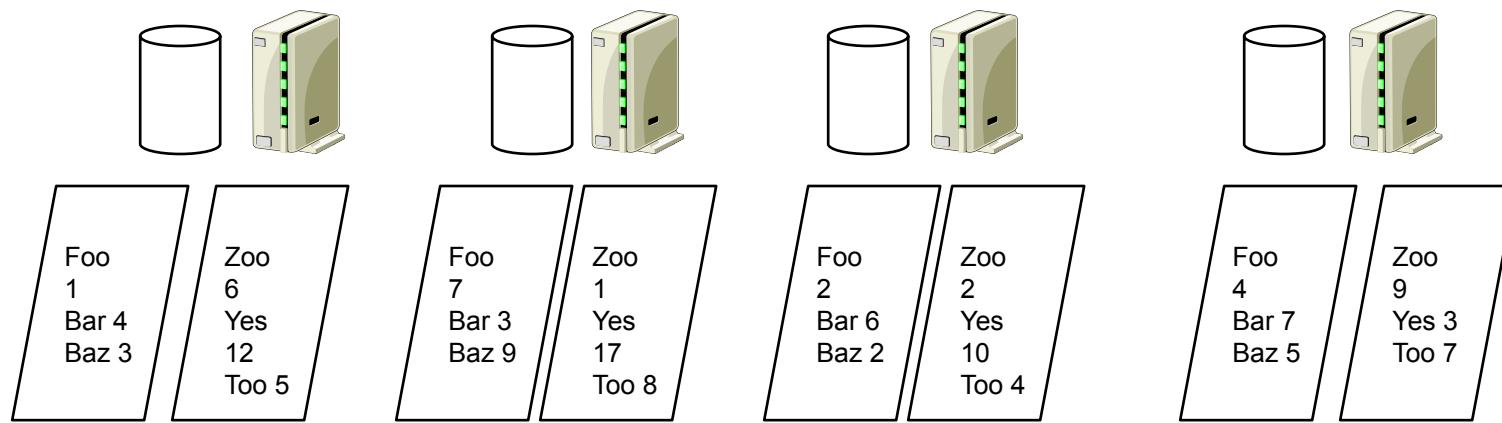


That's the map stage

On To Reduce

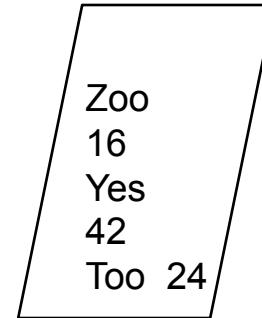
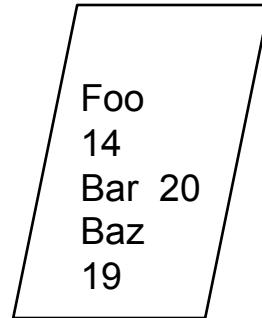
- We might have two more nodes assigned to doing the reduce operation
- They will each receive a share of data from a map node
- The reduce node performs a reduce operation to “combine” the shares
- Outputting its own result

Continuing the Example



The Reduce Nodes Do Their Job

Write out the results to files
And MapReduce is done!



But I Wanted A Combined List

- No problem
- Run another (slightly different) MapReduce on the outputs
- Have one reduce node that combines everything

Synchronization in MapReduce

- Each map node produces an output file for each reduce node
- It is produced atomically
- The reduce node can't work on this data until the whole file is written
- Forcing a synchronization point between the map and reduce phases

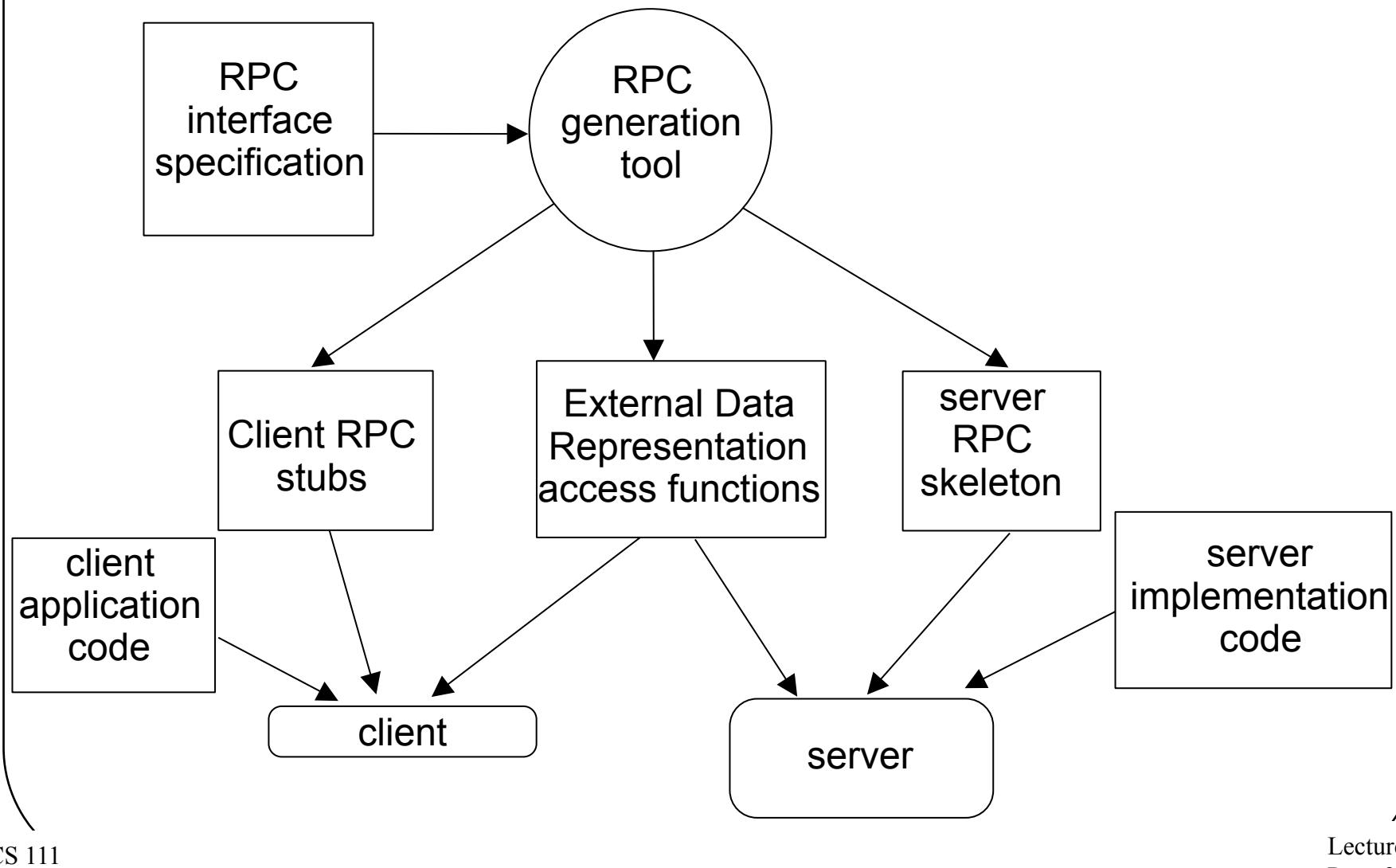
Remote Procedure Calls

- RPC, for short
- One way of building a distributed system
- Procedure calls are a fundamental paradigm
 - Primary unit of computation in most languages
 - Unit of information hiding in most methodologies
 - Primary level of interface specification
- A natural boundary between client and server
 - Turn procedure calls into message send/receives
- A few limitations
 - No implicit parameters/returns (e.g. global variables)
 - No call-by-reference parameters
 - Much slower than procedure calls (TANSTAAFL)

Remote Procedure Call Concepts

- Interface Specification
 - Methods, parameter types, return types
- eXternal Data Representation
 - Machine independent data-type representations
 - May have optimizations for like client/server
- Client stub
 - Client-side proxy for a method in the API
- Server stub (or skeleton)
 - Server-side recipient for API invocations

RPC Tool Chain



Key Features of RPC

- Client application links against local procedures
 - Calls local procedures, gets results
- All RPC implementation inside those procedures
- Client application does not know about RPC
 - Does not know about formats of messages
 - Does not worry about sends, timeouts, resents
 - Does not know about external data representation
- All of this is generated automatically by RPC tools
- The key to the tools is the interface specification

RPC Is Not a Complete Solution

- Requires client/server binding model
 - Expects to be given a live connection
- Threading model implementation
 - A single thread service requests one-at-a-time
 - Numerous one-per-request worker threads
- Limited failure handling
 - Client must arrange for timeout and recovery
- Higher level abstractions improve RPC
 - e.g. Microsoft DCOM, Java RMI, DRb, Pyro

Distributed Synchronization and Consensus

- Why is it hard to synchronize distributed systems?
- What tools do we use to synchronize them?
- How can a group of cooperating nodes agree on something?

What's Hard About Distributed Synchronization?

- Spatial separation
 - Different processes run on different systems
 - No shared memory for (atomic instruction) locks
 - They are controlled by different operating systems
- Temporal separation
 - Can't "totally order" spatially separated events
 - Before/simultaneous/after lose their meaning
- Independent modes of failure
 - One partner can die, while others continue

Leases – More Robust Locks

- Obtained from resource manager
 - Gives client exclusive right to update the file
 - Lease “cookie” must be passed to server on update
 - Lease can be released at end of critical section
- Only valid for a limited period of time
 - After which the lease cookie expires
 - Updates with stale cookies are not permitted
 - After which new leases can be granted
- Handles a wide range of failures
 - Process, client node, server node, network

Lock Breaking and Recovery

- Revoking an expired lease is fairly easy
 - Lease cookie includes a “good until” time
 - Based on server’s clock
 - Any operation involving a “stale cookie” fails
- This makes it safe to issue a new lease
 - Old lease-holder can no longer access object
 - Was object left in a “reasonable” state?
- Object must be restored to last “good” state
 - Roll back to state prior to the aborted lease
 - Implement all-or-none transactions

Distributed Consensus

- Achieving simultaneous, unanimous agreement
 - Even in the presence of node & network failures
 - Required: agreement, termination, validity, integrity
 - Desired: bounded time
 - Provably impossible in fully general case
 - But can be done in useful special cases, or if some requirements are relaxed
- Consensus algorithms tend to be complex
 - And may take a long time to converge
- They tend to be used sparingly
 - E.g., use consensus to elect a leader
 - Who makes all subsequent decisions by fiat

Typical Consensus Algorithm

1. Each interested member broadcasts his nomination.
2. All parties evaluate the received proposals according to a fixed and well known rule.
3. After allowing a reasonable time for proposals, each voter acknowledges the best proposal it has seen.
4. If a proposal has a majority of the votes, the proposing member broadcasts a claim that the question has been resolved.
5. Each party that agrees with the winner's claim acknowledges the announced resolution.
6. Election is over when a quorum acknowledges the result.

Security for Distributed Systems

- Security is hard in single machines
- It's even harder in distributed systems
- Why?

Ensuring Single Machine Security

- All key resources are kept inside of the OS
 - Protected by hardware (mode, memory management)
 - Processes cannot access them directly
- All users are authenticated to the OS
 - By a trusted agent that is (essentially) part of the OS
- All access control decisions are made by the OS
 - The only way to access resources is through the OS
 - We trust the OS to ensure privacy and proper sharing

Distributed Security Is Harder

- Your OS cannot guarantee privacy and integrity
 - Network transactions happen outside of the OS
- Authentication is harder
 - All possible agents may not be in local password file
- The wire connecting the user to the system is insecure
 - Eavesdropping, replays, man-in-the-middle attacks
- Even with honest partners, hard to coordinate distributed security
- The Internet is an open network for all
 - Many sites on the Internet try to serve all comers
 - Core Internet makes no judgments on what's acceptable
 - Even supposedly private systems may be on Internet

Goals of Network Security

- Secure conversations
 - Privacy: only you and your partner know what is said
 - Integrity: nobody can tamper with your messages
- Positive identification of both parties
 - Authentication of the identity of message sender
 - Assurance that a message is not a replay or forgery
 - Non-repudiation: he cannot claim “I didn't say that”
- Availability
 - The network and other nodes must be reachable when they need to be

Elements of Network Security

- Cryptography
 - Symmetric cryptography for protecting bulk transport of data
 - Public key cryptography primarily for authentication
 - Cryptographic hashes to detect message alterations
- Digital signatures and public key certificates
 - Powerful tools to authenticate a message's sender
- Filtering technologies
 - Firewalls and the like
 - To keep bad stuff from reaching our machines

Symmetric Encryption

- Simple fast algorithms
 - Encryption and decryption use the same key
 - Requires sender and receiver to both know the key
 - If you know who shares the key, you also get authentication
- Symmetric encryption provides privacy
 - In order to decrypt the data, you must know the key
- Symmetric encryption provides integrity
 - In order to generate false messages, you must know the key
- Symmetric encryption relies on key secrecy
 - Challenging to achieve in many circumstances
 - Large step between theoretical key secrecy and actual key secrecy in real systems

Tamper Detection: Cryptographic Hashes

- Check-sums often used to detect data corruption
 - Add up all bytes in a block, send sum along with data
 - Recipient adds up all the received bytes
 - If check-sums agree, the data is probably OK
 - Check-sum (parity, CRC, ECC) algorithms are weak
- Cryptographic hashes are very strong check-sums
 - Unique – two messages vanishingly unlikely to produce same hash
 - Particularly hard to find two messages with the same hash
 - One way – cannot infer original input from output
 - Well distributed – any change to input changes output

Using Cryptographic Hashes

- Start with a message you want to protect
- Compute a cryptographic hash for that message
 - E.g., using the Secure Hash Algorithm 3 (SHA-3)
- Transmit the hash securely
- Recipient does same computation on received text
 - If both hash results agree, the message is intact
 - If not, the message has been corrupted/
compromised

Secure Hash Transport

- Why must hash be transmitted securely?
 - Cryptographic hashes aren't keyed, so anyone can produce them (including a bad guy)
- How to transmit hash securely?
 - Typically encrypt it with symmetric cryptography
 - Unless secrecy required, cheaper than encrypting entire message
 - If you have a secure channel, could transmit it that way
 - But if you have secure channel, why not use it for everything?

Public Key Cryptography

- Uses two keys instead of one
- A secret key known only to the owner encrypts
- The public key known to everyone (potentially) decrypts
- Or you can reverse the keys and operations
 - With different effects
- The two keys are related by mathematical properties
 - But must be hard to derive from each other

Practical Use of PK

- Public key cryptography algorithms are computationally expensive
 - 10x to 100x as expensive as symmetric ones
- We use PK only when we can't use symmetric cryptography
- When is that?
 - Typically to communicate to someone we don't share a symmetric key with
 - We can share a new symmetric key using PK (*session key*)
 - Not very expensive, since the symmetric key is small

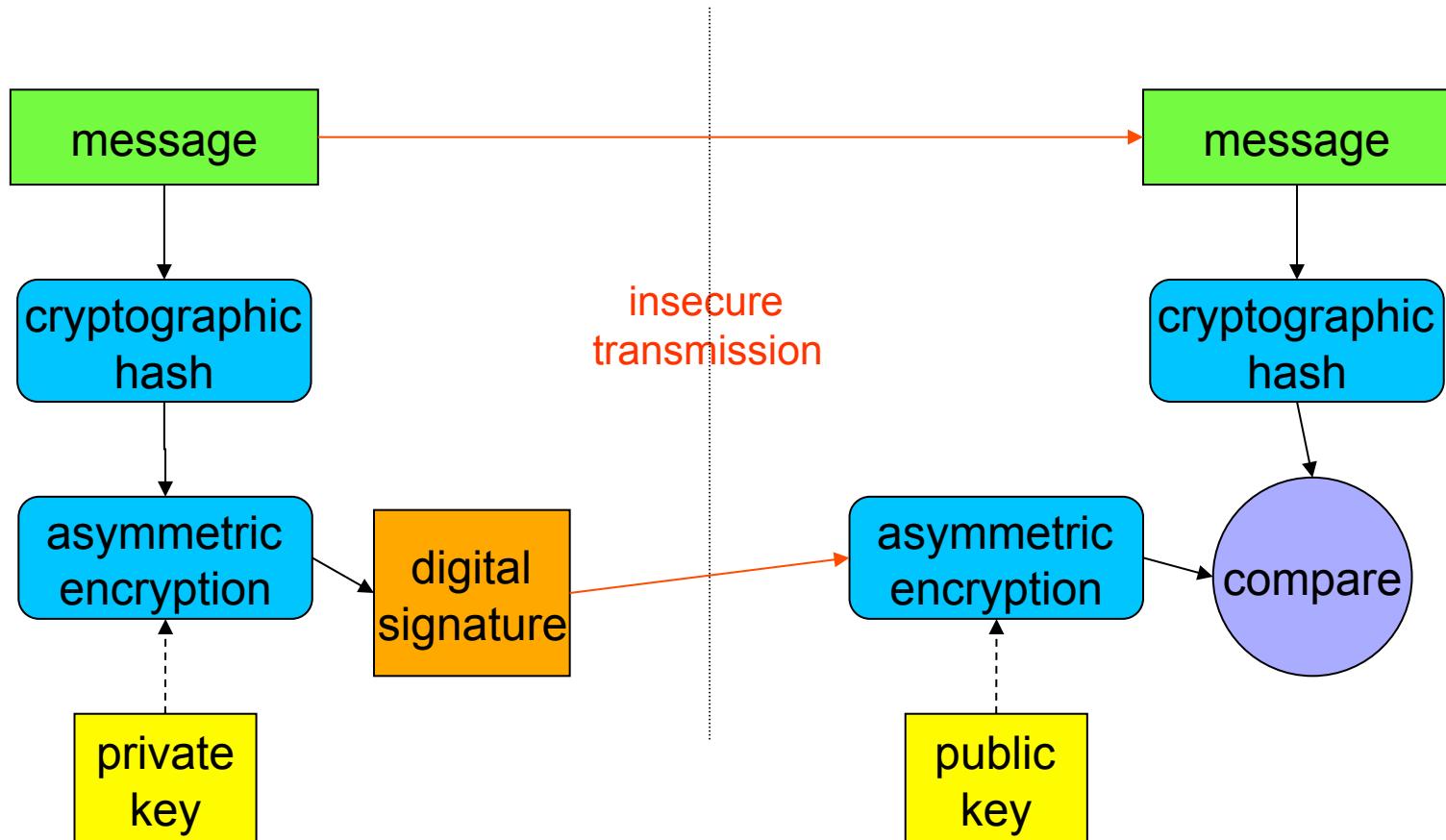
A Principle of Key Use

- Both symmetric and PK cryptography rely on a secret key for their properties
- The more you use one key, the less secure
 - The key stays around in various places longer
 - There are more opportunities for an attacker to get it
 - There is more incentive for attacker to get it
 - Brute force attacks may eventually succeed
- Therefore:
 - Use a given key as little as possible
 - Change them often
 - Within the limits of practicality and required performance

Putting It Together: Secure Socket Layer (SSL)

- A general solution for securing network communication
- Built on top of existing socket IPC
- Establishes secure link between two parties
 - Privacy – nobody can snoop on conversation
 - Integrity – nobody can generate fake messages
- Certificate-based authentication of server
 - Typically, but not necessarily
 - Client knows what server he is talking to
- Optional certificate-based authentication of client
 - If server requires authentication and non-repudiation
- PK used to distribute a symmetric session key
 - New key for each new socket
- Rest of data transport switches to symmetric crypto
 - Giving safety of public key and efficiency of symmetric

Digital Signatures



Digital Signatures

- Encrypting a message with private key signs it
 - Only you could have encrypted it, it must be from you
 - It has not been tampered with since you wrote it
- Encrypting everything with your private key is a bad idea
 - Asymmetric encryption is extremely slow
- If you only care about integrity, you don't need to encrypt it all
 - Compute a cryptographic hash of your message
 - Encrypt the cryptographic hash with your private key
 - Faster than encrypting whole message

Signed Load Modules

- How do we know we can trust a program?
 - Is it really the new update to Windows, or actually evil code that will screw me?
 - Digital signatures can answer this question
- Designate a certification authority
 - Perhaps the OS manufacturer (Microsoft, Apple, ...)
- They verify the reliability of the software
 - By code review, by testing, etc.
 - They sign a certified module with their private key
- We can verify signature with their public key
 - Proves the module was certified by them
 - Proves the module has not been tampered with

An Important Public Key Issue

- If I have a public key
 - I can authenticate received messages
 - I know they were sent by the owner of the private key
- But how can I be sure who that person is?
 - How do I know that this is really my bank's public key?
 - Could some swindler have sent me his key instead?
- I can get Microsoft's public key when I first buy their OS
 - So I can verify their load modules and updates
 - But how to handle the more general case?
- I would like a certificate of authenticity
 - Guaranteeing who the real owner of a public key is

What Is a PK Certificate?

- Essentially a data structure
- Containing an identity and a matching public key
 - And perhaps other information
- Also containing a digital signature of those items
- Signature usually signed by someone I trust
 - And whose public key I already have

Using Public Key Certificates

- If I know public key of the authority who signed it
 - I can validate the signature is correct
 - I can tell the certificate has not been tampered with
- If I trust the authority who signed the certificate
 - I can trust they authenticated the certificate owner
 - E.g., we trust drivers licenses and passports
- But first I must know and trust signing authority
 - Which really means I know and trust their public key

A Chicken and Egg Problem

- I can learn the public key of a new partner using his certificate
- But to use his certificate, I need the public key of whoever signed it
- So how do I get that public key?
- Ultimately, *out of band*
- Which means through some other means
- Commonly by having the key in a trusted program, like a web browser
- Or hand delivered (as in project 4)

Conclusion

- Distributed systems offer us much greater power than one machine can provide
- They do so at costs of complexity and security risk
- We handle the complexity by using distributed systems in a few carefully defined ways
- We handle the security risk by proper use of cryptography and other tools

Operating System Principles: Accessing Remote Data

CS 111

Operating Systems
Peter Reiher

Outline

- Data on other machines
- Remote file access architectures
- Challenges in remote data access
 - Security
 - Reliability and availability
 - Performance
 - Scalability

Remote Data: Goals and Challenges

- Sometimes the data we want isn't on our machine
 - A file
 - A database
 - A web page
- We'd like to be able to access it, anyway
- How do we provide access to remote data?

Basic Goals

- Transparency
 - Indistinguishable from local files for all uses
 - All clients see all files from anywhere
- Performance
 - Per-client: at least as fast as local disk
 - Scalability: unaffected by the number of clients
- Cost
 - Capital: less than local (per client) disk storage
 - Operational: zero, it requires no administration
- Capacity: unlimited, it is never full
- Availability: 100%, no failures or service down-time

Key Characteristics of Remote Data Access Solutions

- APIs and transparency
 - How do users and processes access remote data?
 - How closely does remote data mimic local data?
- Performance and robustness
 - Is remote data as fast and reliable as local data?
- Architecture
 - How is solution integrated into clients and servers?
- Protocol and work partitioning
 - How do client and server cooperate?

Remote File Systems

- Provide files to local user that are stored on remote machine
- Using the same or similar model as file access
- Not the only case for remote data access
 - Remote storage devices
 - Accessed by low level device operations over network
 - Remote databases
 - Accessed by database queries on remote nodes

Remote Data Access and Networking

- ALL forms of remote data access rely on networking
- Which is provided by the operating system as previously discussed
- Remote data access must take networking realities into account
 - Unreliability
 - Performance
 - Security

Remote File Access Architectures

- Client/server
- Remote file transfer
- Remote disk access
- Remote file access
- Cloud model

Client/Server Models

- Peer-to-peer
 - Most systems have resources (e.g. disks, printers)
 - They cooperate/share with one-another
- Thin client
 - Few local resources (e.g. CPU, NIC, display)
 - Most resources on work-group or domain servers
- Cloud Services
 - Clients access services rather than resources
 - Clients do not see individual servers

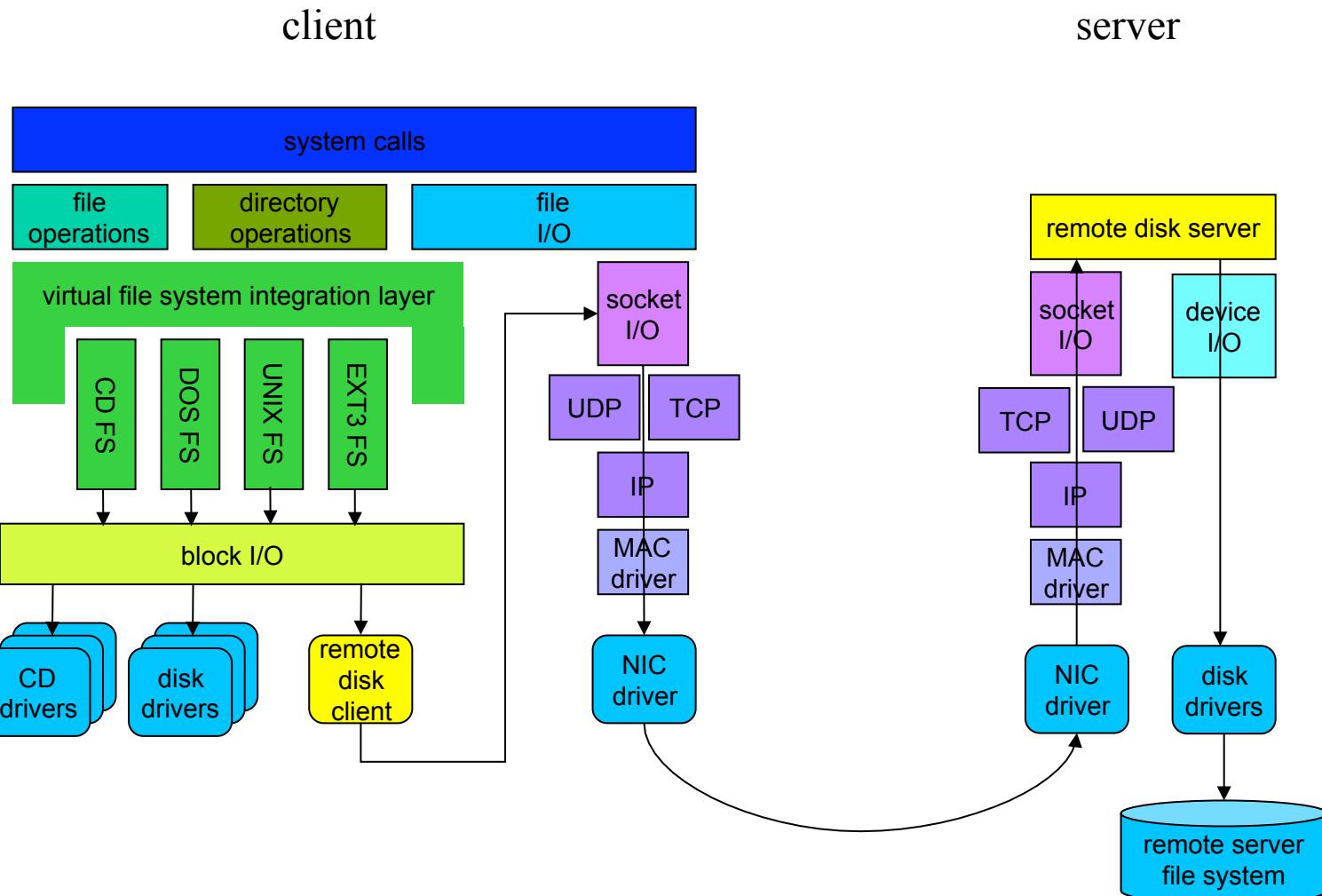
Remote File Transfer

- Explicit commands to copy remote files
 - OS specific: *scp(1)*, *rsync(1)*, S3 tools
 - IETF protocols: FTP, SFTP
- Implicit remote data transfers
 - Browsers (transfer files with HTTP)
 - Email clients (move files with IMAP/POP/SMTP)
- Advantages: efficient, requires no OS support
- Disadvantages: latency, lack of transparency

Remote Disk Access

- Goal: complete transparency
 - Normal file system calls work on remote files
 - All programs “just work” with remote files
- Typical architectures
 - Storage Area Network (SCSI over Fibre Channel)
 - Very fast, very expensive, moderately scalable
 - iSCSI (SCSI over ethernet)
 - Client driver turns reads/writes into network requests
 - Server daemon receives/serves requests
 - Moderate performance, inexpensive, highly scalable

Remote Disk Access Architecture



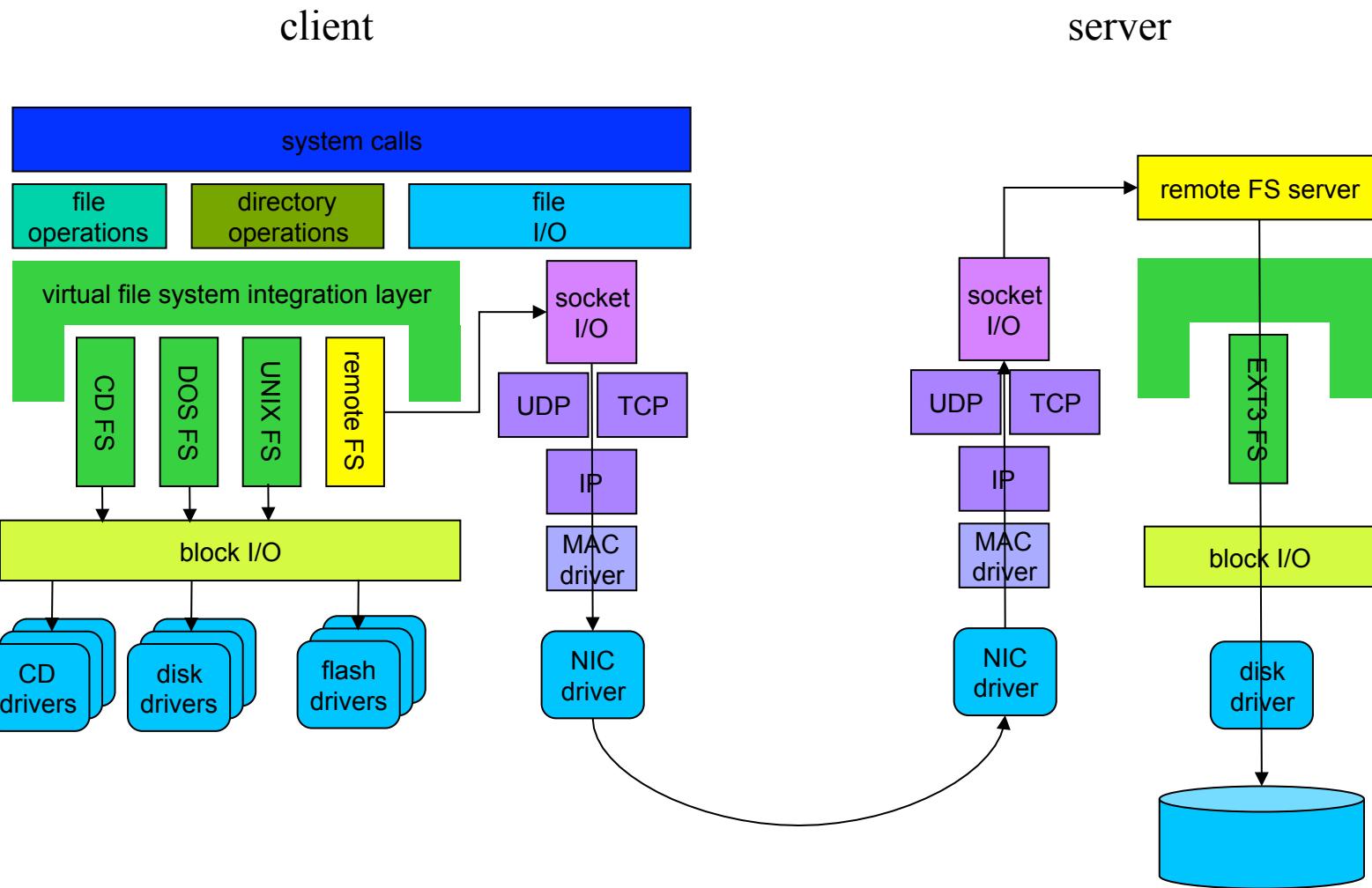
Rating Remote Disk Access

- Advantages:
 - Provides excellent transparency
 - Decouples client hardware from storage capacity
 - Performance/reliability/availability per back-end
- Disadvantages
 - Inefficient fixed partition space allocation
 - Can't support file sharing by multiple client systems
 - Message losses can cause file system errors
- This is THE model for Virtual Machines

Remote File Access

- Goal: complete transparency
 - Normal file system calls work on remote files
 - Support file sharing by multiple clients
 - Performance, availability, reliability, scalability
- Typical architecture
 - Exploits plug-in file system architecture
 - Client-side file system is a local proxy
 - Translates file operations into network requests
 - Server-side daemon receives/process requests
 - Translates them into real file system operations

Remote File Access Architecture



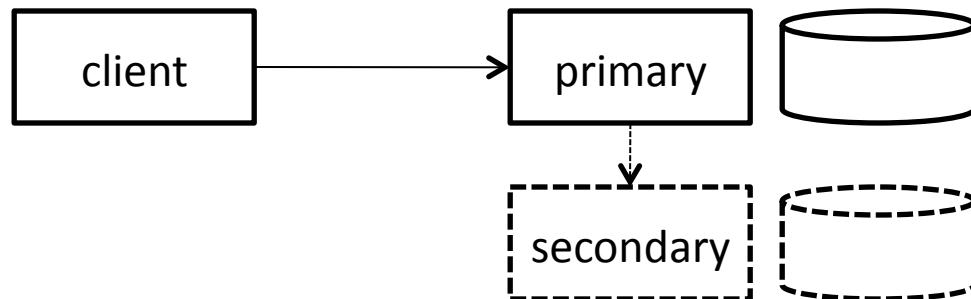
Rating Remote File Access

- Advantages
 - Very good application level transparency
 - Very good functional encapsulation
 - Able to support multi-client file sharing
 - Potential for good performance and robustness
- Disadvantages
 - At least part of implementation must be in the OS
 - Client and server sides tend to be fairly complex
- This is THE model for client/server storage

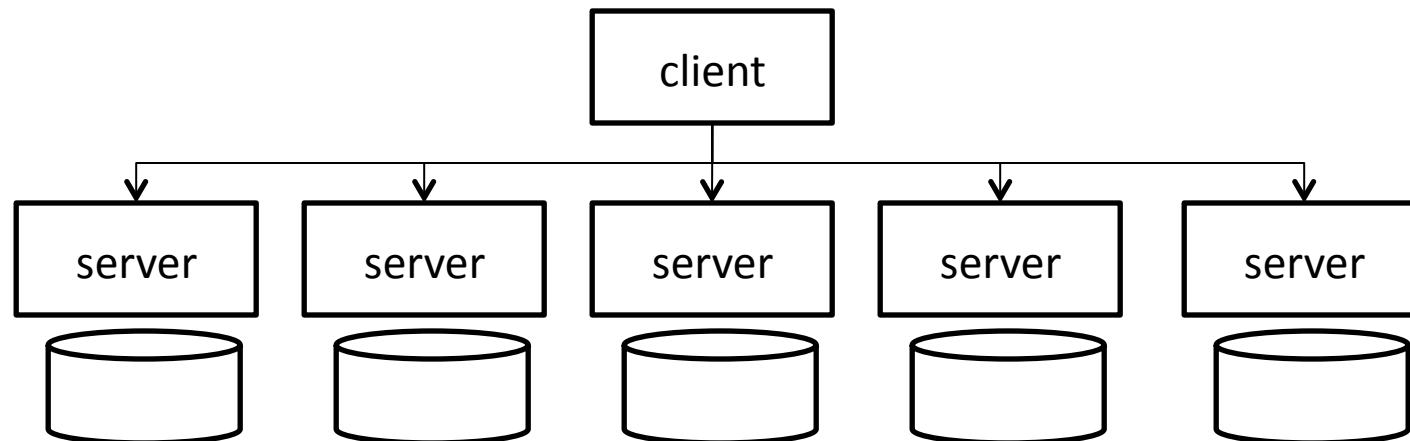
Cloud Model

- A logical extension of client/server model
 - All services accessed via standard protocols
- Opaque encapsulation of servers/resources
 - Resources are abstract/logical, thin-provisioned
 - One highly available IP address for all services
 - Mirroring/migration happen under the covers
- Protocols likely to be WAN-scale optimized
- Advantages:
 - Simple, scalable, highly available, low cost
 - A very compelling business model

Remote Disk/File Access



Distributed File System



Remote vs. Distributed FS

- Remote file access (e.g., NFS, CIFS)
 - Client talks to (per FS) primary server
 - Secondary server may take over if primary fails
 - Advantages: simplicity
- Distributed file system (e.g., Ceph, RAMCloud)
 - Data is spread across numerous servers
 - Client may talk directly to many/all of them
 - Advantages: performance, scalability
 - Disadvantages: complexity++

Security For Remote File Systems

- Major issues:
 - Privacy and integrity for data on the network
 - Solution: encrypt all data sent over network
 - Authentication of remote users
 - Solution: various approaches
 - Trustworthiness of remote sites
 - Solution: various approaches

Authentication Approaches

- Anonymous access
- Peer-to-peer approaches
- Server authentication approaches
- Domain authentication approaches

Anonymous Access

- All files are available to all users
 - No authentication required
 - May be limited to read-only access
 - Examples: anonymous FTP, HTTP
- Advantages
 - Simple implementation
- Disadvantages
 - Can't provide information privacy
 - Usually unacceptable for write access
 - Which is often managed by other means

Peer-to-Peer Security

- All participating nodes are trusted peers
- Client-side authentication/authorization
 - All users are known to all systems
 - All systems are trusted to enforce access control
 - Example: basic NFS
- Advantages:
 - Simple implementation
- Disadvantages:
 - You can't always trust all remote machines
 - Doesn't work in heterogeneous OS environment
 - Universal user registry is not scalable

Server Authenticated Approaches

- Client agent authenticates to each server
 - Authentication used for entire session
 - Authorization based on credentials produced by server
 - Example: CIFS
- Advantages
 - Simple implementation
- Disadvantages
 - May not work in heterogeneous OS environment
 - Universal user registry is not scalable
 - No automatic fail-over if server dies

Domain Authentication Approaches

- Independent authentication of client & server
 - Each authenticates with independent authentication service
 - Each knows/trusts only the authentication service
- Authentication service may issue signed “tickets”
 - Assuring each of the others’ identity and rights
 - May be revocable or timed lease
- May establish secure two-way session
 - Privacy – nobody else can snoop on conversation
 - Integrity – nobody can generate fake messages
- Kerberos is one example

Distributed Authorization

1. Authentication service returns credentials
 - Which server checks against Access Control List
 - Advantage: auth service doesn't know about ACLs
 2. Authentication service returns capabilities
 - Which server can verify (by signature)
 - Advantage: servers do not know about clients
- Both approaches are commonly used
 - Credentials: if subsequent authorization required
 - Capabilities: if access can be granted all-at-once

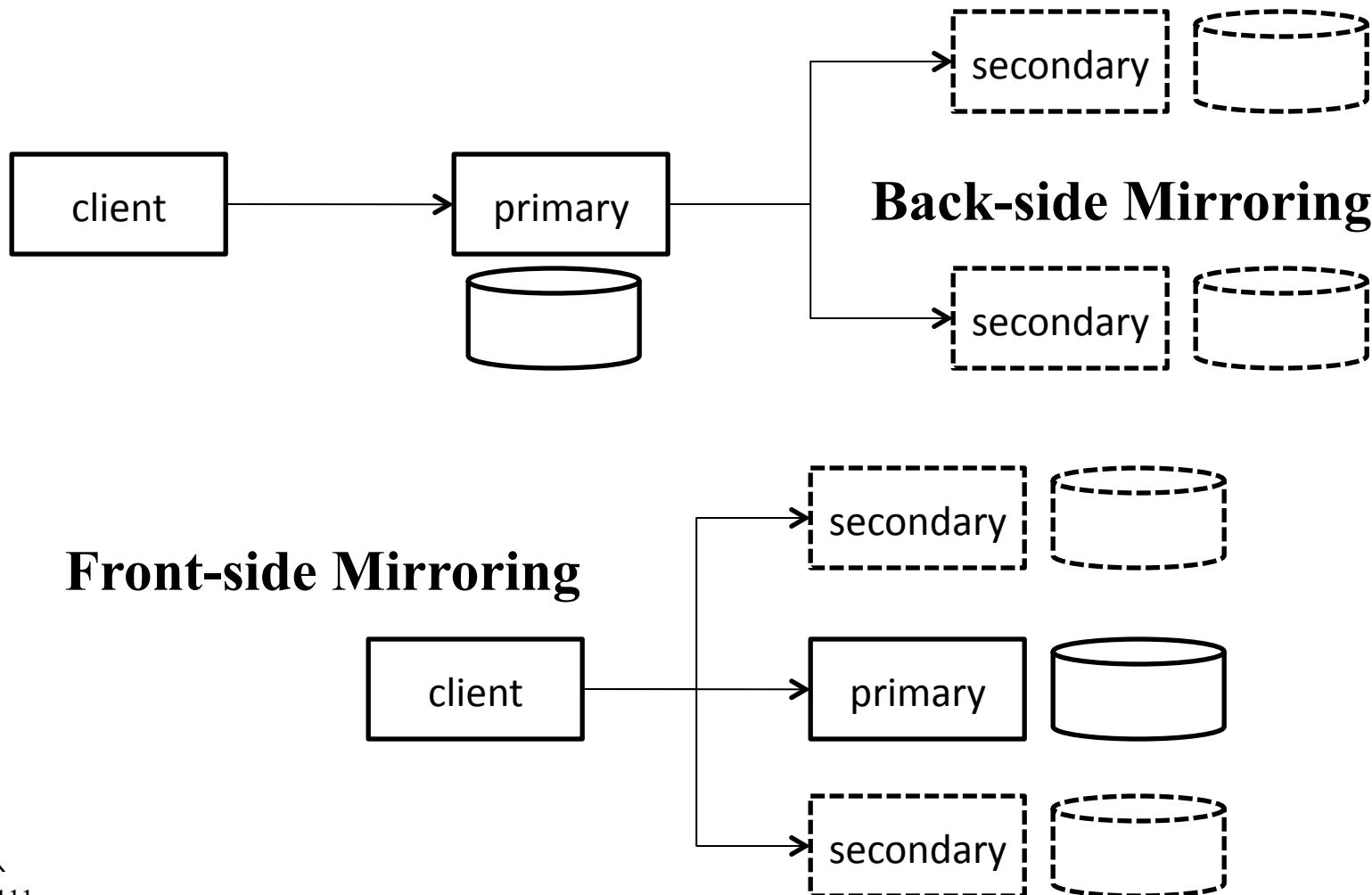
Reliability and Availability

- Reliability is high degree of assurance that service works properly
 - Challenging in distributed systems, because of partial failures
 - Data is not lost despite failures
- Availability is high degree of assurance that service is available whenever needed
 - Failures of some system elements don't prevent data access
 - Certain kinds of distributed systems can greatly improve availability
- Both, here, in the context of accessing remote files

Achieving Reliability

- Must reduce probability of data loss
- Typically by some form of redundancy
 - Disk/server failures don't result in data loss
 - RAID (mirroring, parity, erasure coding)
 - copies on multiple servers
- Also important to automatically recover after failure
 - Remote copies of data become available again
 - Any redundancy loss due to failure must be made up

Reliability: Data Mirroring



Mirroring, Parity, and Erasure Coding

- Similar to trade-offs we made in RAID
 - Extra copies of some data prevent data loss
 - In this case on another machine
 - But the extra copies mean more network I/O
- Mirroring – multiple copies
 - Fast, but requires a great deal of space
- Parity – able to recover from one/two errors
 - Lower space overhead
 - Requires full strip write buffering
- Erasure coding – recover with N/M copies
 - Very space efficient
 - Very slow/expensive reads and writes

Availability and Fail-Over

- Fail-over means transferring work/requests from failed server to some other server
- Data must be mirrored to secondary server
- Failure of primary server must be detected
- Client must be failovered to secondary
- Session state must be reestablished
 - Client authentication/credentials
 - Session parameters (e.g. working directory, offset)
- In-progress operations must be retransmitted
 - Client must expect timeouts, retransmit requests
 - Client responsible for writes until server ACKs

Availability: Failure Detect/Rebind

- If a server fails, need to detect it and rebind to a different server
- Client driven recovery
 - Client detects server failure (connection error)
 - Client reconnects to (successor) server
 - Client reestablishes session
- Transparent failure recovery
 - System detects server failure (health monitoring)
 - Successor assumes primary's IP address (or other redirection)
 - State reestablishment
 - Successor recovers last primary state check-point
 - Stateless protocol

Availability: Stateless Protocols

- Stateful protocols (e.g., TCP)
 - Operations occur within a context
 - Server must save state
 - Each operation depends on previous operations
 - Replacement server must obtain session state to operate properly
- Stateless protocols (e.g., HTTP)
 - Client supplies necessary context with each request
 - Each operation is self-contained and unambiguous
 - Successor server needs no memory of past events
- Stateless protocols make fail-over easy

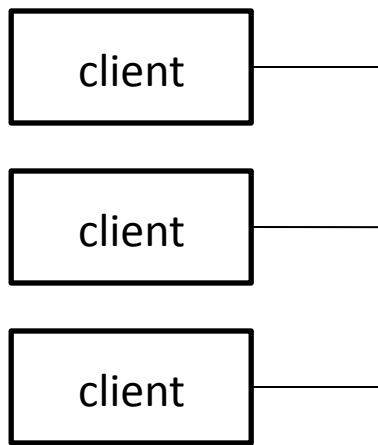
Availability: Idempotent Operations

- Idempotent operations can be repeated many times with same effect
 - Read block 100 of file X
 - Write block 100 of file X with contents Y
 - Delete file X version 3
 - Non-idempotent operations
 - Read next block of current file
 - Append contents Y to end of file X
- If client gets no response, resend request
 - If server gets multiple requests, no harm done
 - Works for server failure, lost request, lost response
 - But no ACK does not mean operation did not happen

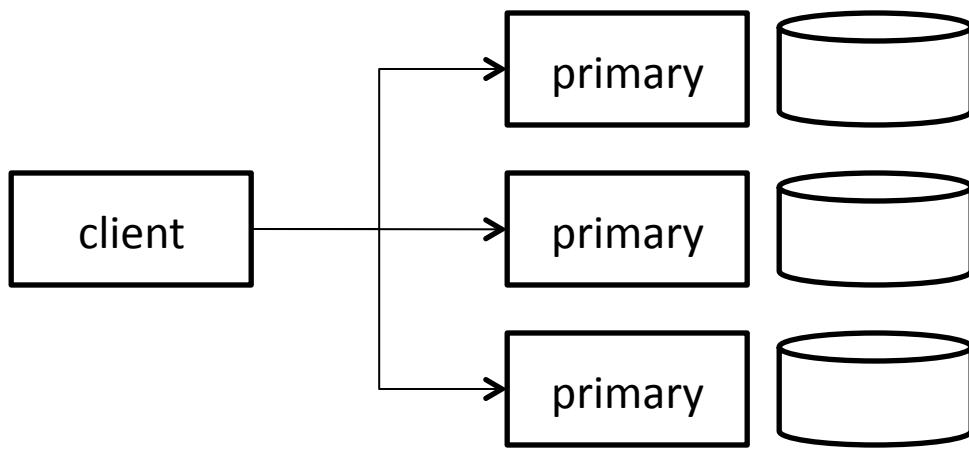
Remote File System Performance

- Disk bandwidth and performance
- Performance for reads
- Performance for writes
- Overheads particular to remote file systems
- Performance and availability

Disk Bandwidth Implications



a single server has limited throughput



striping files across
multiple servers
provides scalable
throughput

Network Impacts on Performance

- Bandwidth limitations
 - Implications for client
 - Implications for server
- Delay implications
 - Particularly important if acknowledgements required
- Packet loss implications
 - If loss rate high, will require acknowledgements

Cost of Reads

- Most file system operations are reads, so read performance is critical
- Common way to improve read performance is through caching
- Can use read-ahead, but costs of being wrong are higher than for local disk

Caching For Reads

- Client-side caching
 - Cache data permanently stored at the server at the client
 - Eliminates waits for remote read requests
 - Reduces network traffic
 - Reduces per-client load on server
- Server-side caching
 - Typically performed similarly to single machine caching
 - Reduces disk delays, but not network problems

Whole File Vs. Block Caching

- Many distributed file systems use whole file caching
 - E.g., AFS
- Higher network latency justifies whole file pulls
- Stored in local (cache-only) file system
- Satisfy early reads before entire file arrives
- Block caching is also common (NFS)
 - Typically integrated into shared block cache

Cost of Writes

- Writes at clients need to get to server(s) that store the data
 - And what about other clients caching that data?
- Not caching the writes is very expensive
 - Since they need to traverse the network
 - And probably be acknowledged
- Caching approaches improve performance at potential cost of consistency

Caching Writes For Distributed File Systems

- Write-back cache
 - Create the illusion of fast writes
 - Combine small writes into larger writes
 - Fewer, larger network and disk writes
 - Enable local read-after-write consistency
- Whole-file updates
 - No writes sent to server until *close(2)* or *fsync(2)*
 - Reduce many successive updates to final result
 - Possible file will be deleted before it is written
 - Enable atomic updates, close-to-open consistency
 - But may lead to more potential problems of inconsistency

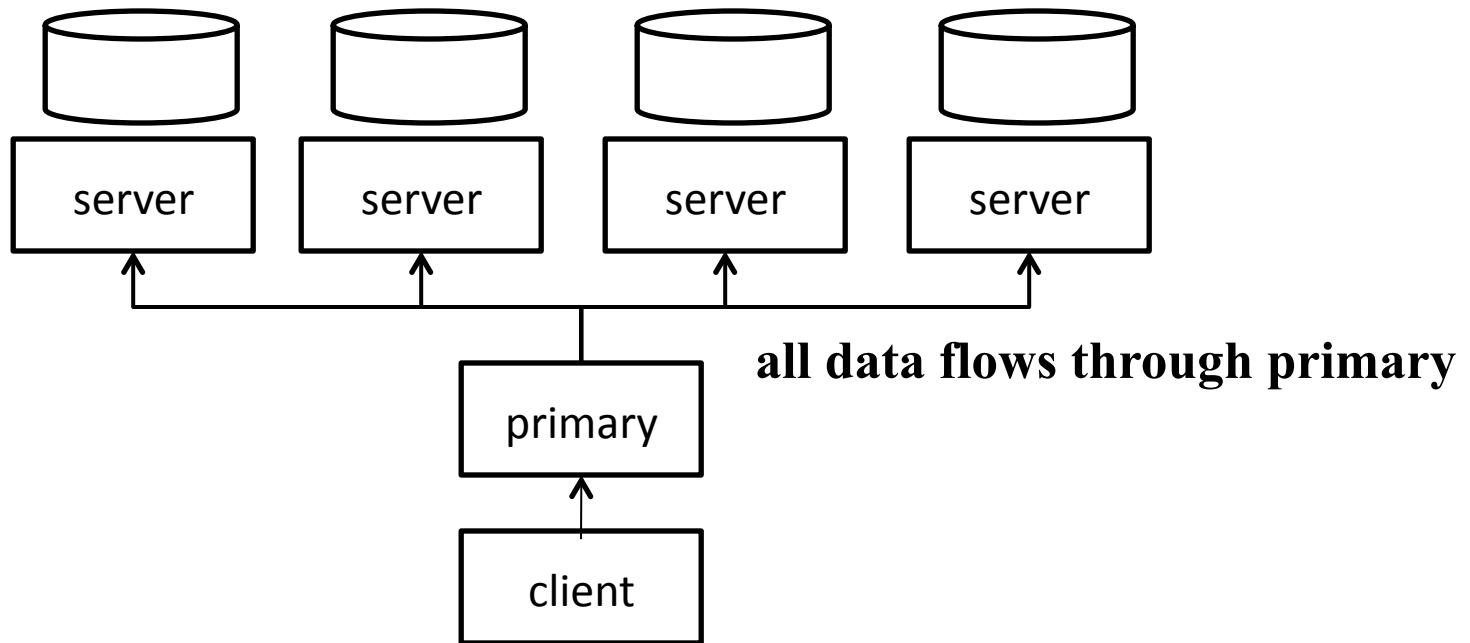
Cost of Consistency

- Caching is essential in distributed systems
 - For both performance and scalability
- Caching is easy in a single-writer system
 - Force all writes to go through the cache
- Multi-writer distributed caching is hard
 - Time To Live is a cute idea that doesn't work
 - Constant validity checks defeat the purpose
 - One-writer-at-a-time is too restrictive for most FS
 - Change notifications are a reasonable alternative

Cost of Mirroring

- Multi-host vs. multi-disk mirroring
 - Protects against host and disk failures
 - Creates much additional network traffic
- Mirroring by primary
 - Primary becomes throughput bottleneck
 - Move replication traffic to back-side network
- Mirroring by client
 - Data flows directly from client to storage servers
 - Replication traffic goes through client NIC
 - Parity/erasure code computation on client CPU

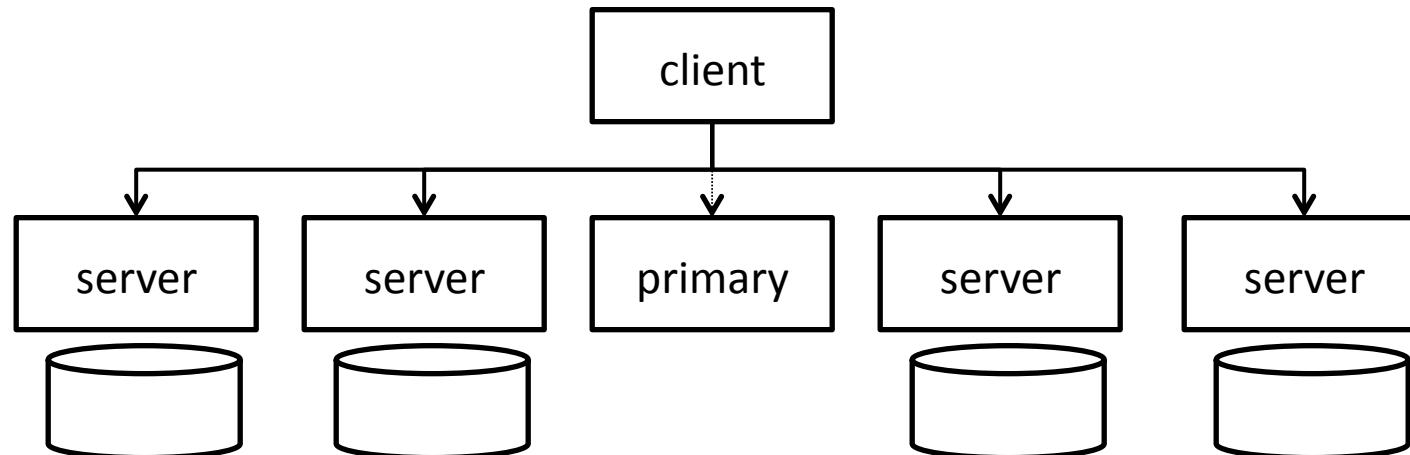
Mirroring Through Primary



Mirroring Through Direct Data Flow

Primary directs client to storage nodes

Data flows direct to storage nodes



Benefits of Direct Data Path

- Architecture
 - Primary tells clients where which data resides
 - Client communicates directly with storage servers
- Throughput
 - Data can be striped across multiple storage servers
- Latency
 - No intermediate relay through primary server
- Scalability
 - Fewer messages on network
 - Much less data flowing through primary servers

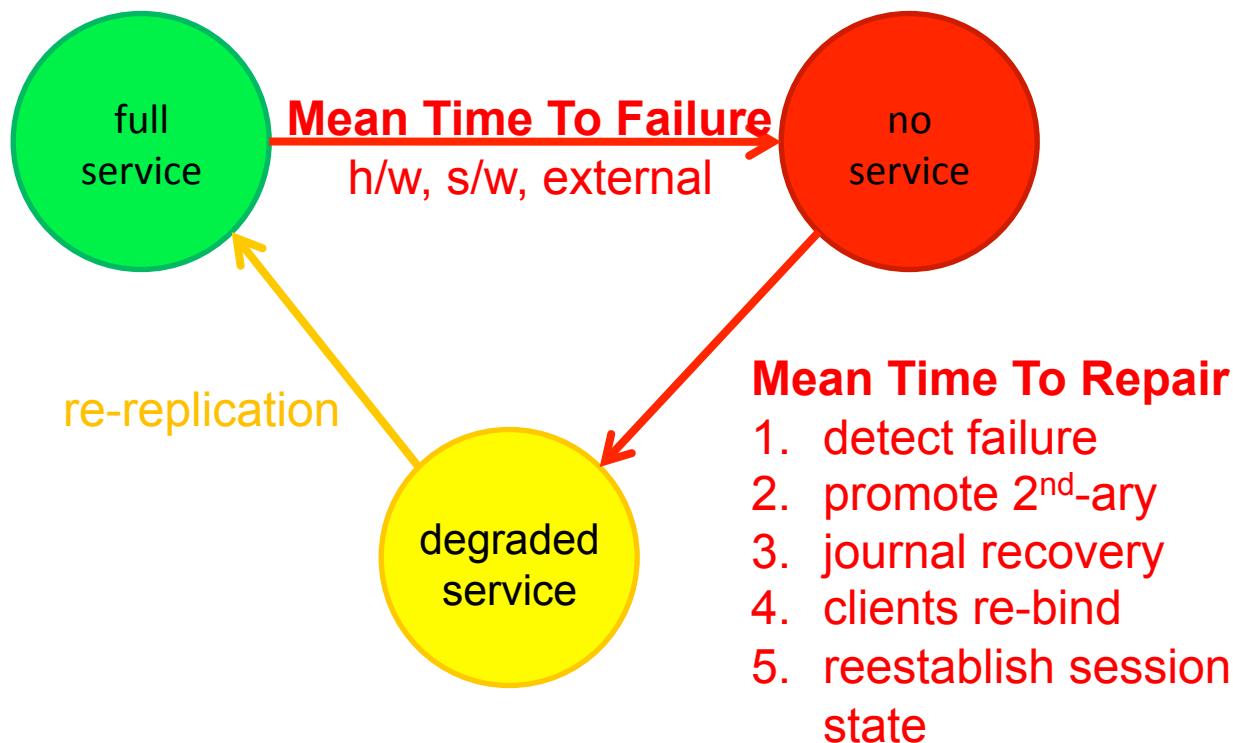
Reliability and Availability

Performance

- Distributed systems must expect some failures
- Distributed file systems are expected to offer good service despite those failures
- How do we characterize that performance characteristic?
- How do we improve it?

Recovery Time

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$



Improving Availability

- Reduce MTTF
 - Use more reliable components
 - Get rid of bugs
- Or reduce MTTR
 - Use architectures that provide service quickly once recovery starts
 - There are several places where you can improve MTTR

Availability Performance

- Improving MTTR
 - Detect failures more quickly
 - Promote secondary to primary role quickly
 - Recover recent/in-progress operations quickly
 - Inform and rebind clients quickly
 - Re-establish session state (if any) quickly
- Degraded service may persist longer
 - Restoring lost redundancy may take a while
 - Heavily loading servers, disks, and network

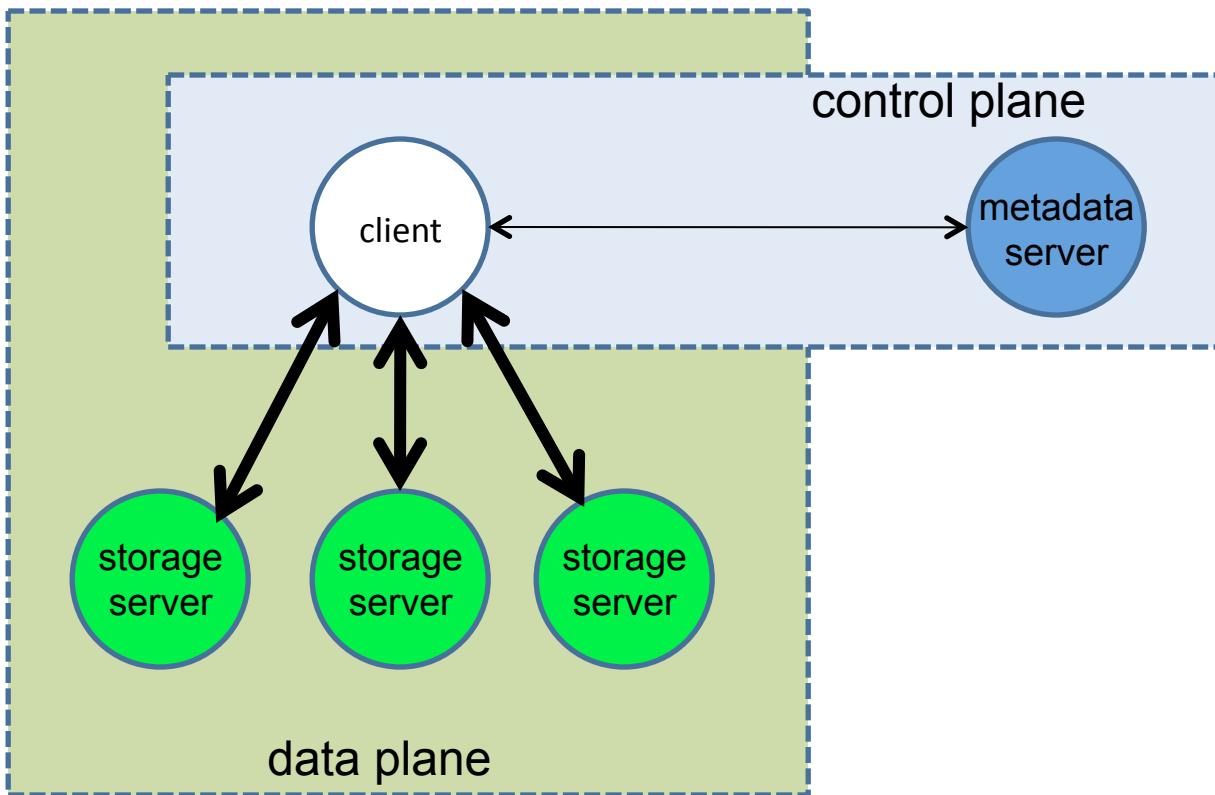
Scalability and Performance: Network Traffic

- Network messages are expensive
 - NIC and network capacity to carry them
 - Server CPU cycles to process them
 - Client delays awaiting responses
- Minimize messages/client/second
 - Cache results to eliminate requests entirely
 - Enable complex operations with single request
 - Buffer up large writes in write-back cache
 - Pre-fetch large reads into local cache

Scalability Performance: Bottlenecks

- Avoid single control points
 - Partition responsibility over many nodes
- Separated data- and control-planes
 - Control nodes choreograph the flow of data
 - Where data should be stored or obtained from
 - Ensuring coherency and correct serialization
 - Data flows directly from producer to consumer
 - Data paths are optimized for throughput/efficiency
- Dynamic re-partitioning of responsibilities
 - In response to failures and/or load changes

Control and Data Planes



Scalability Performance: Cluster Protocols

- Consensus protocols do not scale well
 - They only work fast for small numbers of nodes
- Minimize number of consensus operations
 - Elect a single master who makes decisions
 - Partitioned and delegated responsibility
- Avoid large-consensus/transaction groups
 - Partition work among numerous small groups
- Avoid high communications fan-in/fan-out
 - Hierarchical information gathering/distribution

Hierarchical Communication Structure

