

Introduction
CS 111
Winter 2023
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
- 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: No in person office hours this quarter
 - Office hours: TTh 10-11 AM, via Zoom
 - Zoom ID for office hours to be announced later
 - Often available at other times

TAs

- Section 1A: Victor Zhang
 - Email: victorzhangyf@ucla.edu
- Section 1B: Rustem Can Aygun
 - Email: canaygun10@gmail.com
- Section 1C: Yadi Cao
 - Email: yadicao95@ucla.edu
- Section 1D: Salekh Parkhati
 - Email: salekh@cs.ucla.edu
- Section 1E: Shruthi Srinarasi
 - Email: shruthi223@g.ucla.edu
- Office hours to be announced later

Instructor/TA Division of Responsibilities

- Instructor handles all lectures, readings, and tests
 - Ask me about issues related to these
- TAs handles 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

- We'll primarily use a web site set up for this class
 - <https://bruinlearn.ucla.edu/courses/153928>
 - Schedules for reading, lectures, exams, projects
 - Project materials
 - And uploads of completed projects
 - Copies of lecture slides
 - Also Zoom IDs for lectures and taped lectures
 - 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
- 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/>
- Supplemented with web-based materials

Course Grading

- Basis for grading:
 - Class evaluation 1%
 - 1 midterm exam 20%
 - Final exam 26%
 - Lab 0 9%
 - Other labs 11% 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
 - Scores on BruinLearn not authoritative

Midterm Examination

- When: 6th week (Tuesday, February 14)
 - Replacing that day's class
 - You can take it online during any two hour period that day
- Scope: All material up to the exam date
 - Approximately 60% lecture, 40% text
 - No questions on purely project materials
- Format:
 - On line, multiple choice, open book/notes
- Goals:
 - Test understanding of key concepts
 - Test ability to apply principles to practical problems

Final Exam

- When: Friday, March 24
 - You can take it online during any 3 hour period that day
- Scope: Entire course
- Format:
 - On line, multiple choice, open book/notes
- Goals:
 - Determining if you have mastered the full range of material presented in the class

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

Late Assignments & Make-ups

- Labs
 - Due dates set by TAs
 - *NOTE: They may change from the dates listed on the syllabus*
 - 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 your 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

Academic Honesty - Tests

- It shouldn't be necessary to say this, but . . .
- The rules for the tests will be stated before the test
- You must take the test yourself
- You must follow general UCLA academic honesty principles

Introduction to the Course

- Purpose of course and relationships to other courses
- Why study operating systems?
- What is an operating system?

What Will CS 111 Do?

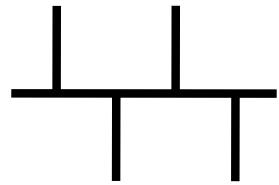
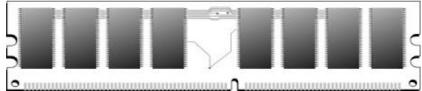
- Build on concepts from other courses
 - Data structures, programming languages, assembly language programming, computer architectures, ...
- Prepare you for advanced courses
 - Data bases, data mining, and distributed computing
 - Security, fault-tolerance, high availability
 - Network protocols, computer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

Why Study Operating Systems?

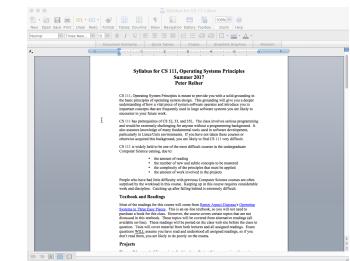
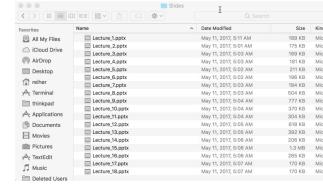
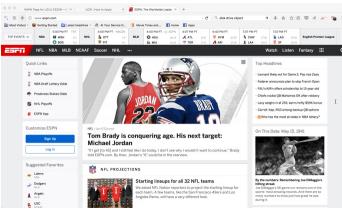
- Why do we have them, in the first place?
- Why are they important?
- What do they do for us?

Starting From the Bottom

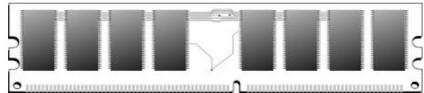
Here's
what
you've
got



Here's
what
you
want



What Can You Do With What You've Got?



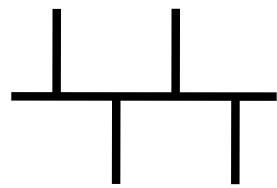
Read or write some binary words



MOV
ADD
JMP
SQRTPD



Report X and Y movements



READ
REQUEST SENSE



Write to groups of pixels



Read or write a block of data

And You Want This?



You're Going to Need Some Help

- And that's what the operating system is about
- Helping you perform complex operations
 - That interact
 - Using various hardware
 - And probably various bits of software
- While hiding the complexity
- And making sure nothing gets in the way of anything else

What Is An Operating System, Anyway?

- System software intended to provide support for higher level applications
 - Including higher level system software applications
 - But primarily for user processes
- The software that sits between the hardware and everything else
- The software that hides nasty details
 - Of hardware, software, and common tasks
- On a good day, the OS is your best computing friend

But Why Are You Studying Them?

- High probability none of you will ever write an operating system
 - Or even fix an operating system bug
- Not very many different operating systems are in use
 - So the number of developers for them is small
- So why should you care about them?

Everybody Has One

- Practically every computing device you will ever use has an operating system
 - Servers, laptops, desktop machines, tablets, smart phones, game consoles, set-top boxes
- Many things you don't think of as computers have CPUs inside
 - Usually with an operating system
 - Internet of Things devices
- So you will work with operating systems

How Do You Work With OSes?

- You configure them
- You use their features when you write programs
- You rely on *services* that they offer
 - Memory management
 - Persistent storage
 - Scheduling and synchronization
 - Interprocess communications
 - Security

Another Good Reason

- Many hard problems have been tackled in the context of operating systems
 - How to coordinate separate computations
 - How to manage shared resources
 - How to virtualize hardware and software
 - How to organize communications
 - How to protect your computing resources
- The operating system solutions are often applicable to programs and systems you write

Some OS Wisdom

- View services as objects and operations
 - Behind every object there is a data structure
- Interface vs. implementation
 - An implementation is not a specification
 - Many compliant implementations are possible
 - Inappropriate dependencies cause problems
- An interface specification is a contract
 - Specifies responsibilities of producers & consumers
 - Basis for product/release interoperability

More OS Wisdom

- Modularity and functional encapsulation
 - Complexity hiding and appropriate abstraction
- 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
 - Correctness doesn't always win . . .

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

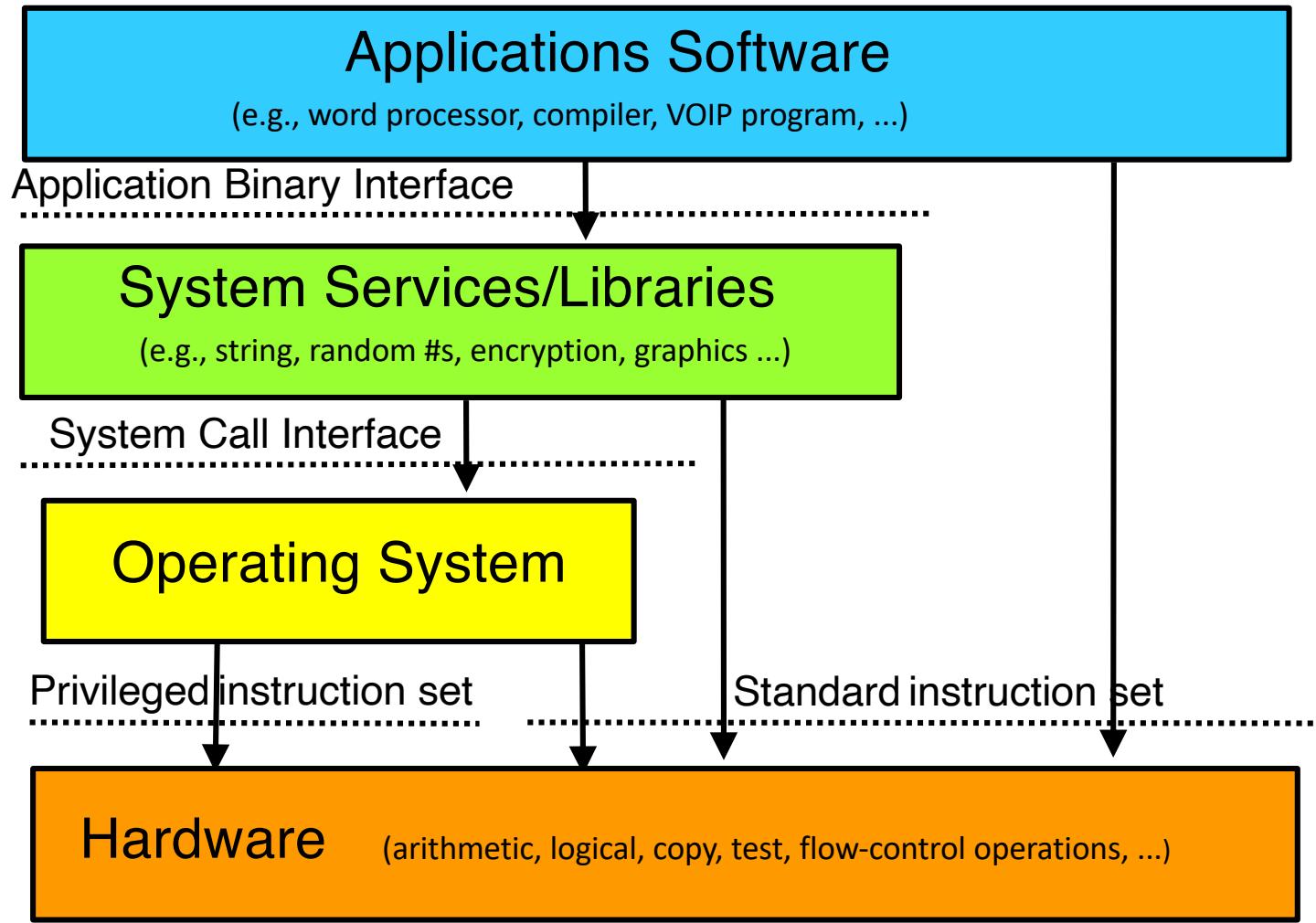
What Does an OS Do?

- 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?



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 . . .

Instruction Set Architectures (ISAs)

- The set of instructions supported by a computer
 - Which 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 (x86, ARM, 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 the 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
 - There are lots of them

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
- Which implies that the OS will abstract the ISA
- 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
- How can an OS manufacturer distribute to all these different ISAs and platforms?

Binary Distribution Model

- Binary is the derivative of source
 - The OS is written in source
 - But a source distribution must be compiled
 - A binary distribution is ready to run
- OSes usually distributed in binary
- One (or more) binary distributions per ISA
- 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

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

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

Conclusion

- Understanding operating systems is critical to understanding how computers work
- Operating systems interact directly with the hardware
- Operating systems rely on stable interfaces

Operating System Services

CS 111

Winter 2023

Operating System Principles

Peter Reiher

Outline

- Operating systems and abstractions
- Trends in operating systems
- Operating system services

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 flash drives (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

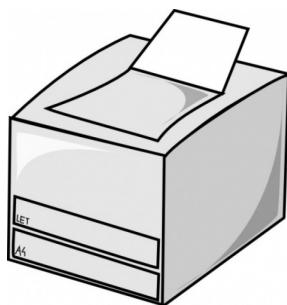
- Lots of variations in machines' HW and SW
- Make many different types appear the same
 - So applications can deal with single common class
- Usually involves a common unifying model
 - E.g., portable document format (pdf) for printers
 - Or SCSI standard for disks, CDs and tapes
- For example:
 - Printer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

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:

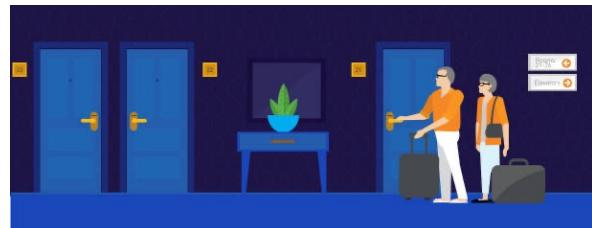
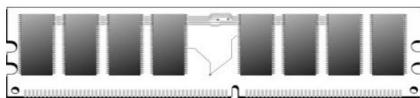


What Is A Graceful Transition?

- 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:



Do We Still Need Graceful Transitions?

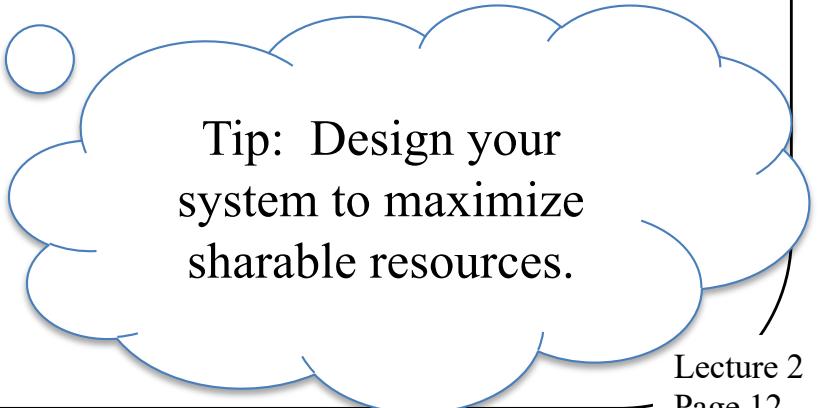
- Yes
- Most partitionable resources aren't permanently allocated
 - The piece of RAM you're using now will belong to another process later
- As long as it's “yours,” no transition required
- But sooner or later it's likely to become someone else's

Shareable Resources

- Usable by multiple concurrent clients
 - Clients don't “wait” for access to resource
 - Clients don't “own” a particular subset of the resource
- May involve (effectively) limitless resources
 - Air in a room, shared by occupants
 - Copy of the operating system, shared by processes

Do We Still Need Graceful Transitions?

- Typically not
- The shareable resource usually doesn't change state
- Or isn't “reused”
- We never have to clean up what doesn't get dirty
 - Like an execute-only copy of the OS
- Shareable resources are great!
 - When you can have them . . .



Tip: Design your system to maximize sharable resources.

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

What about
Chrome OS?

- There are a handful of widely used OSes



1985



Mac OS

1984



1991

And a few special purpose ones (e.g., real time and embedded system OSes)



FreeBSD



QNX



- OSes in the same family are used for vastly different purposes
 - Challenging for the OS designer
 - Most OSes are based on pretty old models

It's Linux-based.

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
 - Which requires other parties to do a lot of work
- You need to have some clear advantage over present acceptable alternatives

Where Are The Popular OSes Used?

- Windows
 - The most popular choice for personal computers
 - Laptops, desktops, etc.
 - Some use in servers and small devices
- MacOS
 - Exclusively in Apple products
 - But in all Apple products (Macbooks, iPhones, Apple Watches, etc.)
- Linux
 - The choice in industrial servers (e.g., cloud computing)
 - And the choice of CS nerds and embedded systems

OS Services

- The operating system offers important services to other programs
- 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 widgets, 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

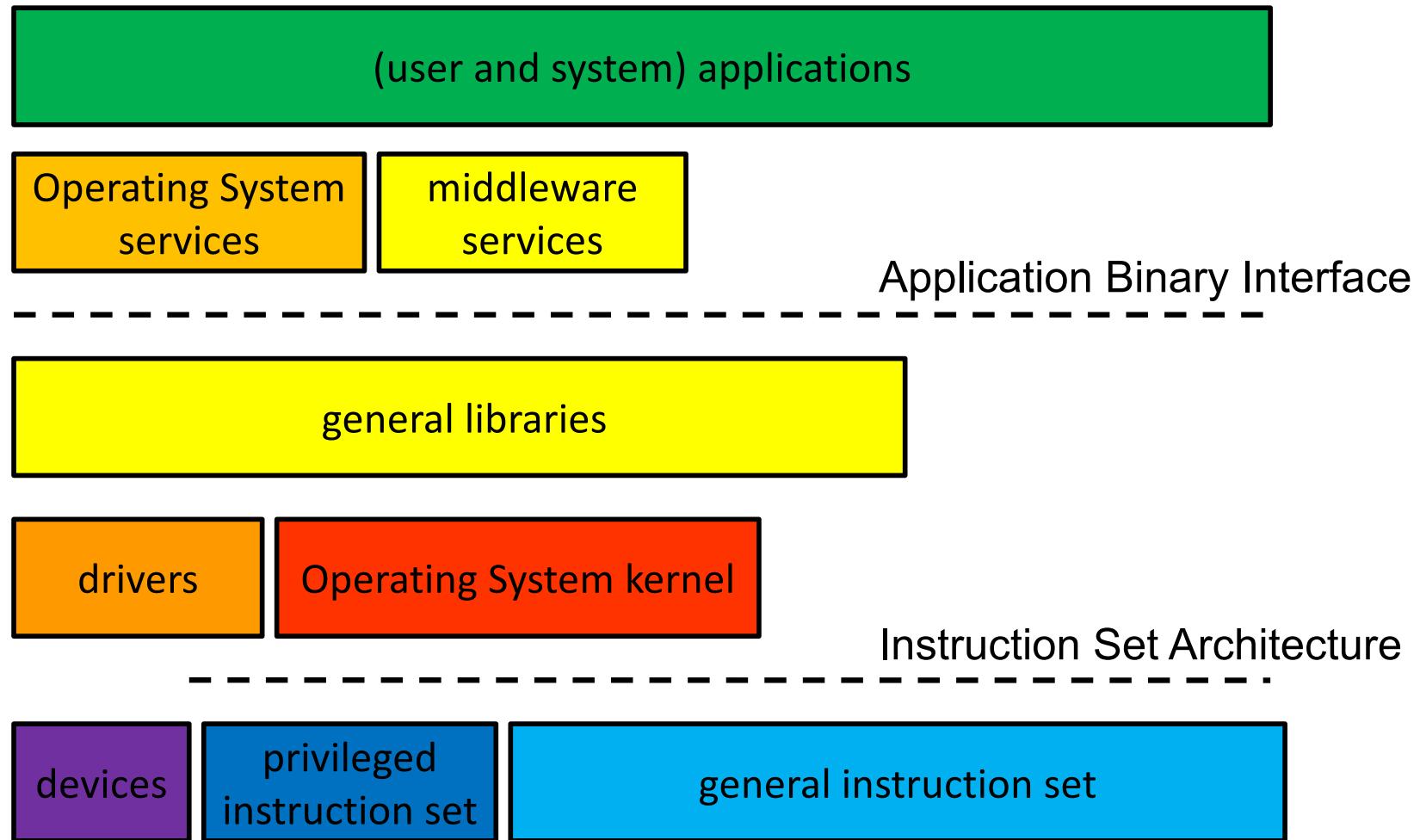
How Can the OS Deliver These Services?

- Several possible ways
 - Applications could just call subroutines
 - Applications could make system calls
 - Applications could send messages to software that performs the services
- Each option works at a different *layer* of the stack of software

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

Software Layering



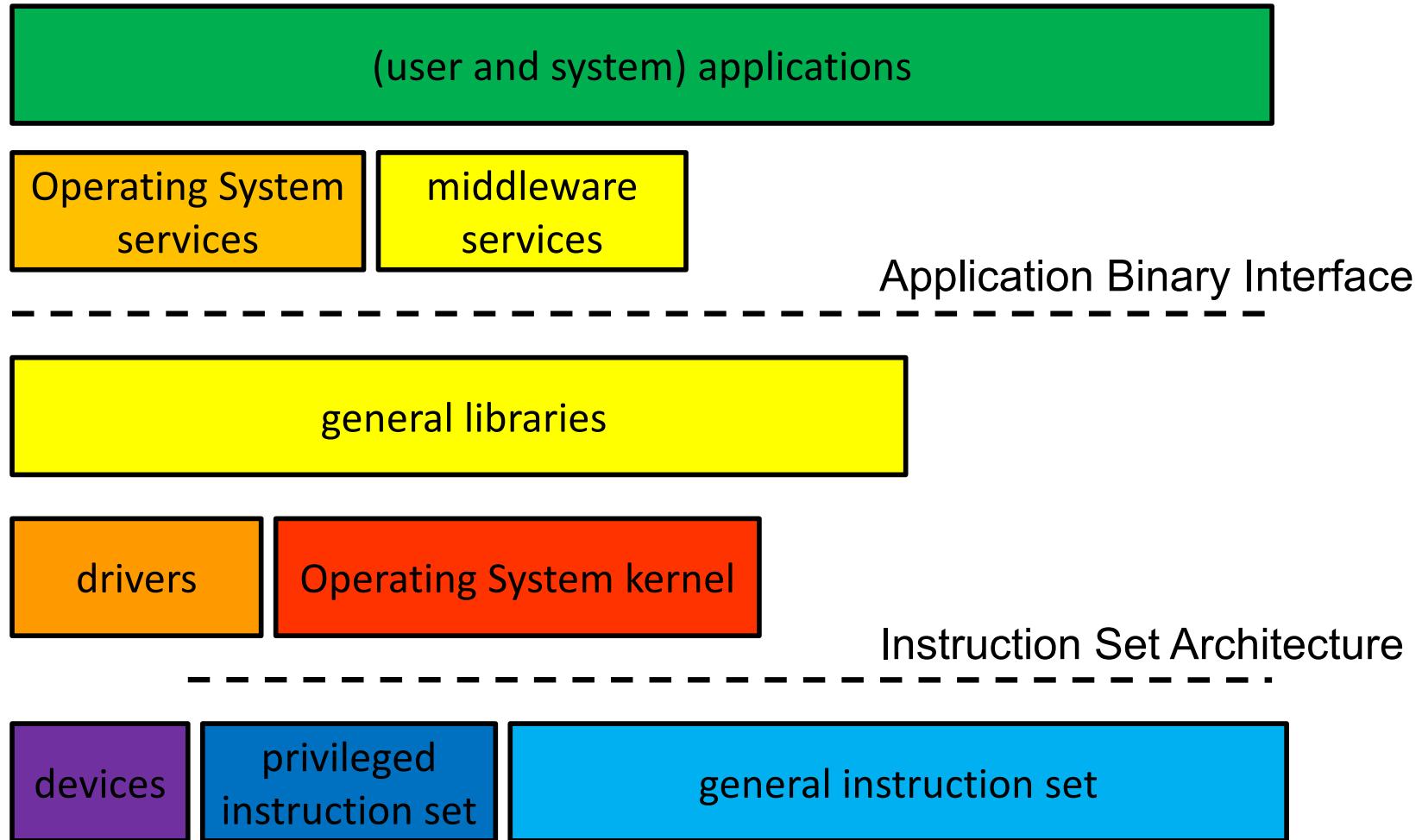
Service Delivery via Subroutines

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

Service Delivery via Libraries

- One subroutine service delivery approach
- 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

Sharing Libraries

- *Static library* modules are 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

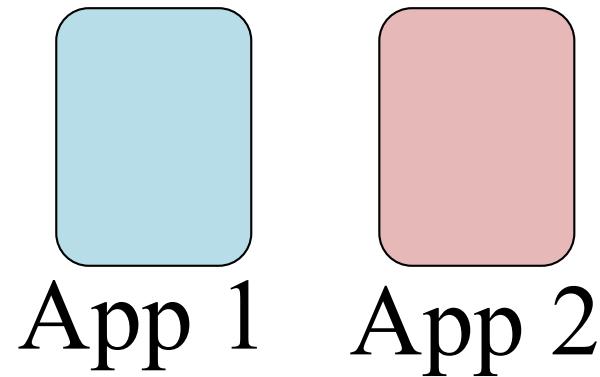
Advantages of Shared Libraries

- 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 easily (e.g., a new version with bug fixes)
 - Programs automatically get the newest version when they are restarted

Limitations of Shared Libraries

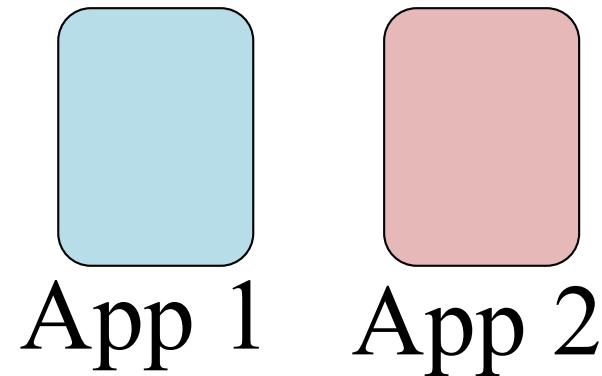
- Not all modules will work in a shared library
 - They cannot define/include global data storage
- They are added into program memory
 - Whether they are actually needed or not
- Called routines must be known at compile-time
 - Only the fetching of the code is delayed 'til run-time
 - Symbols known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

Where Is the Library?



RAM

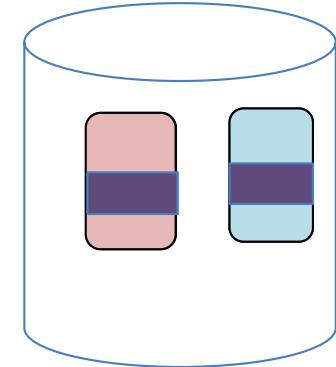
Static Libraries



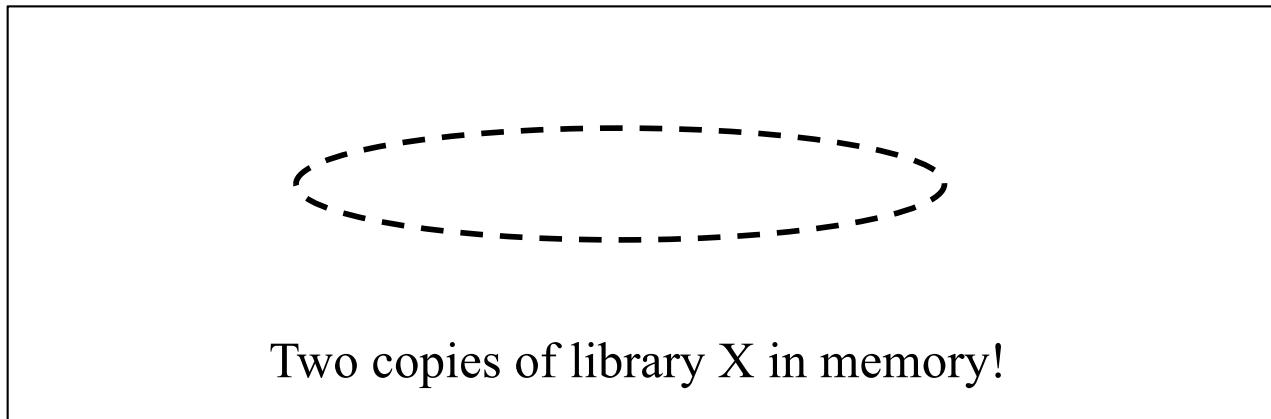
Compile App 1 Compile App 2
Run App 1 Run App 2



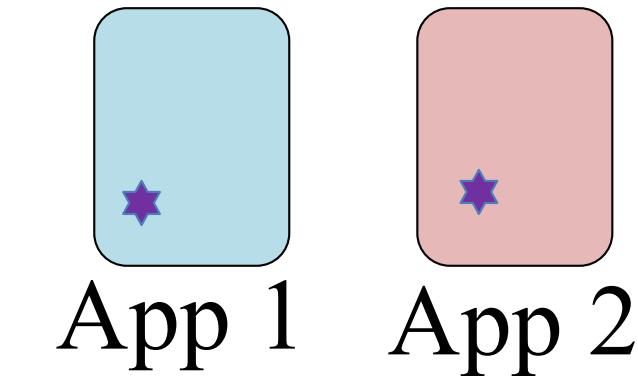
Library X



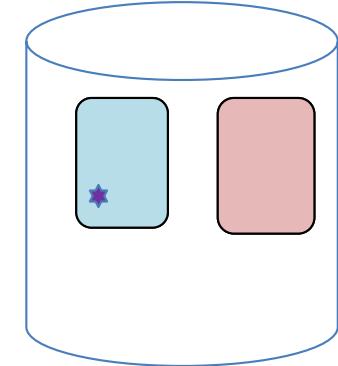
RAM



Shared Libraries



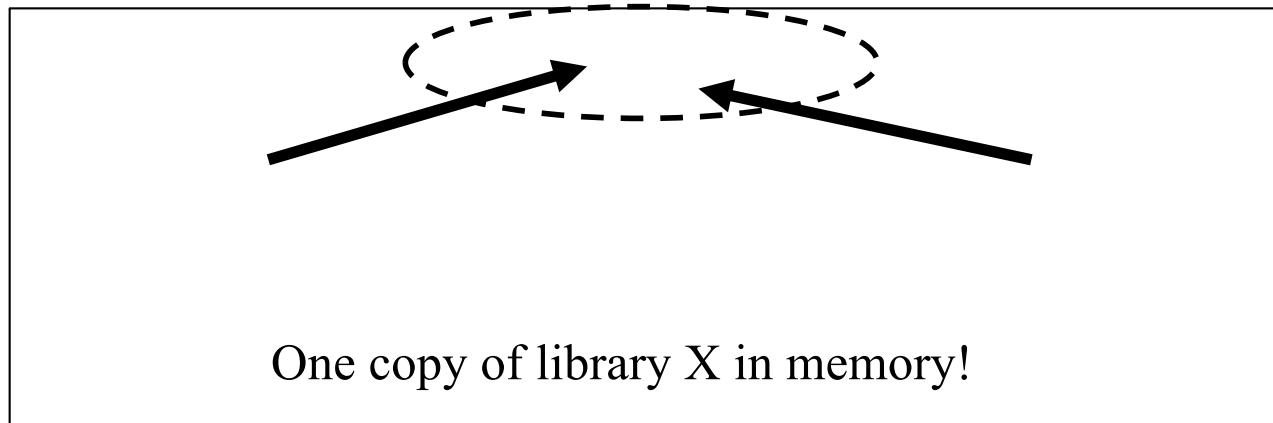
Library X



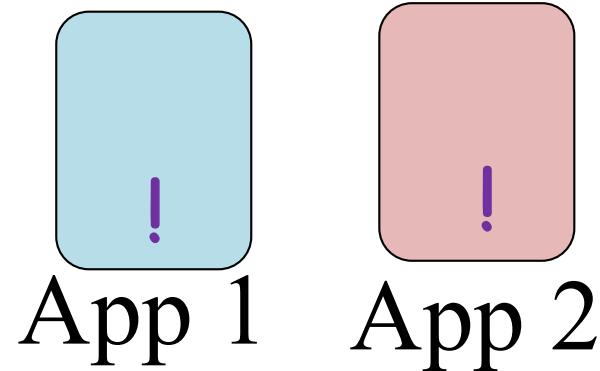
Secondary
Storage

Compile App 1 Compile App 2
Run App 1 Run App 2

RAM



Dynamic Libraries



Compile App 1

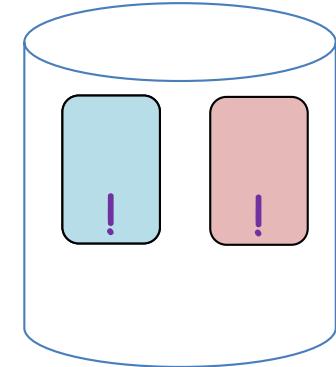
Run App 1

Compile App 2

App 1 calls library function

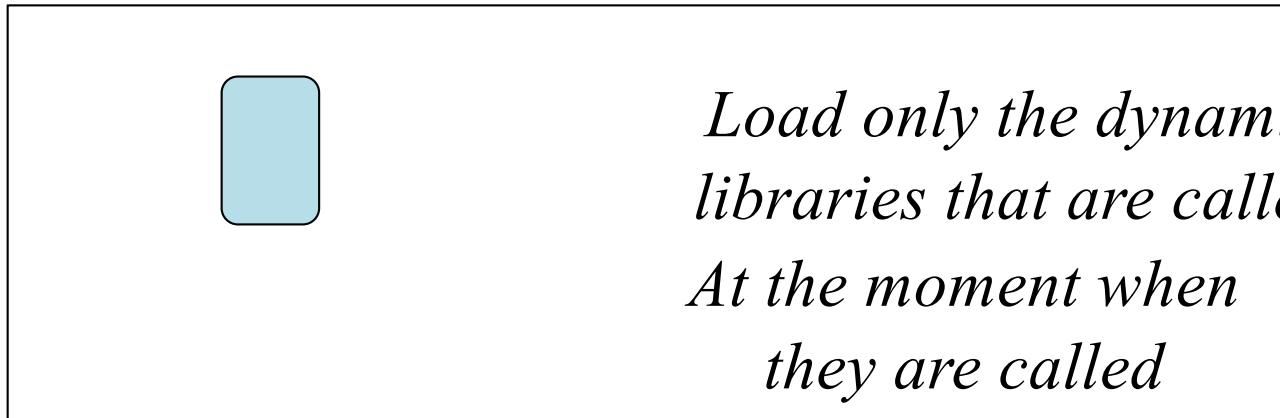


Library X



Secondary
Storage

RAM



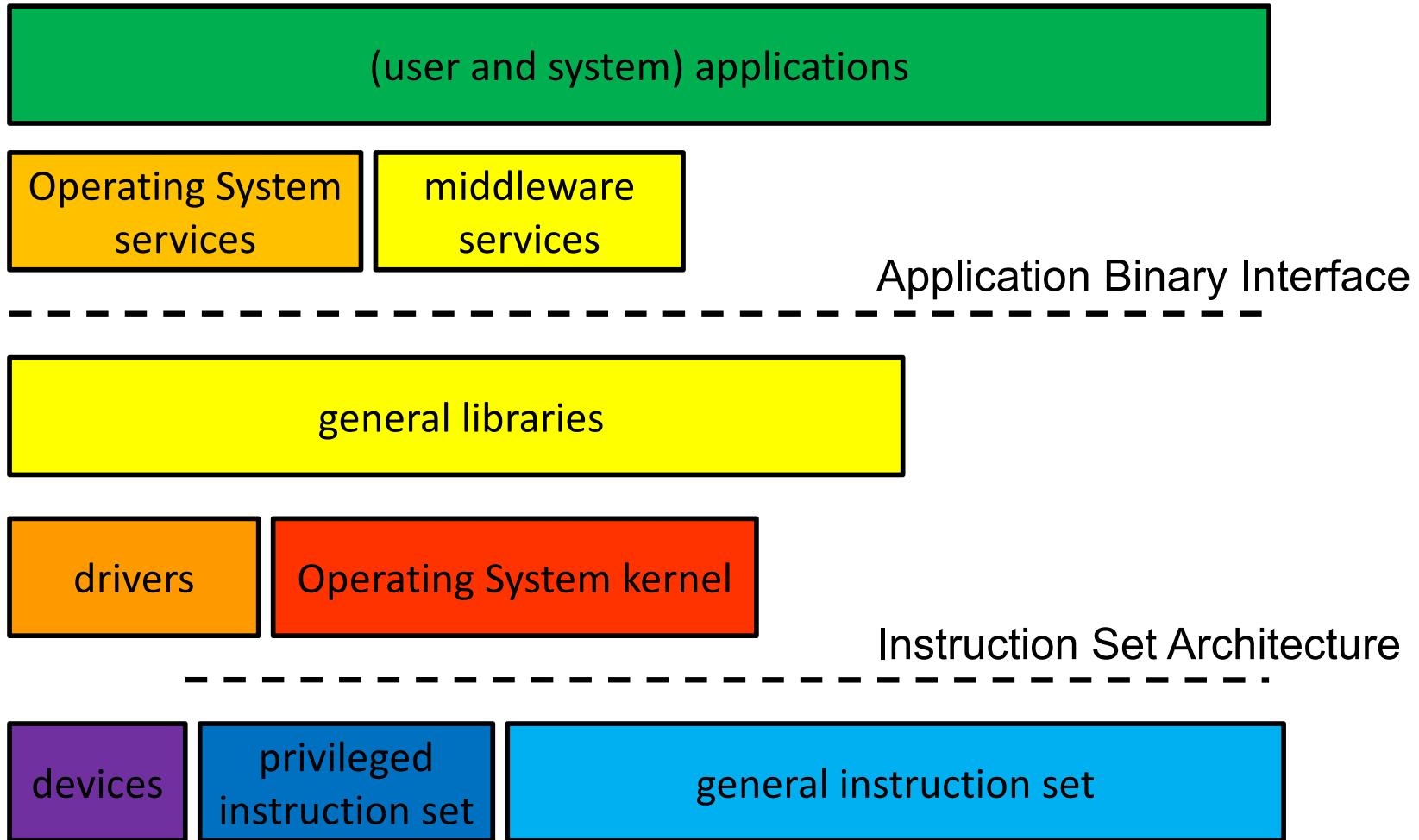
Service Delivery via System Calls

- 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
 - 100x-1000x slower than subroutine calls

Providing Services via 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

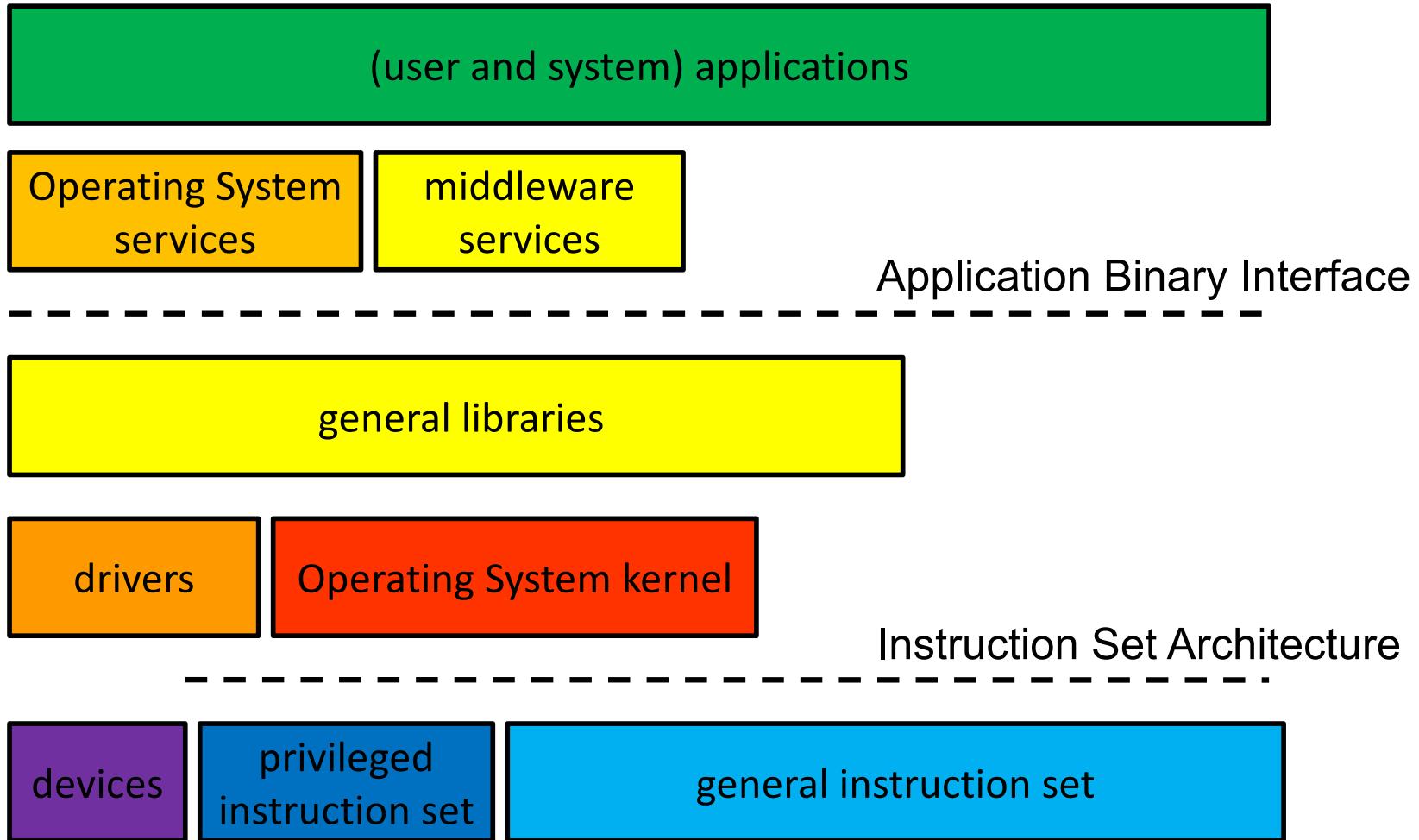
The Kernel Layer



System Services Outside the Kernel

- 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



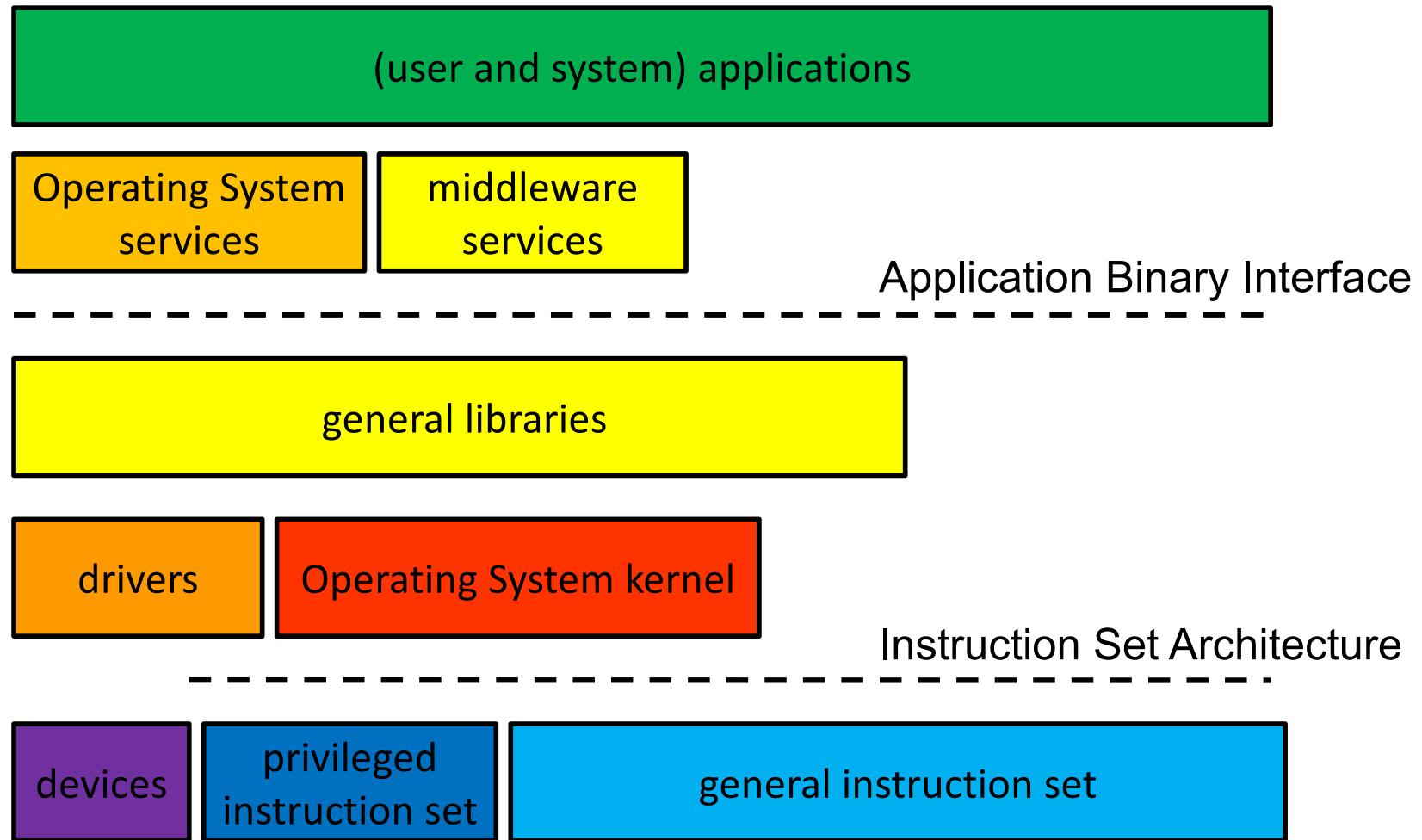
Service Delivery via Messages

- Exchange messages with a server (via syscalls)
 - Parameters in request, returns in response
- Advantages:
 - Server can be anywhere on earth (or local)
 - 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

System Services via 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



Conclusion

- Operating systems have converged on a few popular systems
- Operating systems provide services via abstractions
- Operating systems offer services at several layers in the software stack

OS Interfaces and Abstractions

CS 111

Winter 2023

Operating System Principles

Peter Reiher

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

APIs help you
write programs
for your OS

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

ABIs help you
install binaries
on your OS

Libraries and Interfaces

- Normal libraries (shared and otherwise) are accessed through an API
 - Source-level definitions of how to access the library
 - Readily portable between different machines
- Dynamically loadable libraries also called through an API
 - But the dynamic loading mechanism is ABI-specific
 - Issues of word length, stack format, linkages, etc.

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

Side Effects

- A *side effect* occurs when an action on one object has non-obvious consequences
 - Effects not specified by interfaces
 - Perhaps even to other objects
- 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

Tip: Avoid all side effects in complex systems!

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

Simplifying Abstractions

- Hardware is fast, but complex and limited
 - Using it correctly is extremely complicated
 - It may not support the desired functionality
 - It is not a solution, but merely a building block
- Abstractions . . .
 - Encapsulate implementation details
 - Error handling, performance optimization
 - Eliminate behavior that is irrelevant to the user
 - Provide 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
 - You read them and you write them
 - But there are complications

Some Complicating Factors

- Persistent vs. transient memory
- Size of memory 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 that lacks 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*
- We expect the entire read/write to occur
 - *Atomicity*
- If there are several reads/writes to the file, we expect them to occur in some order

What Is Implementing the File?

- Often a flash drive
- Flash drives have peculiar characteristics
 - Write-once (sort of) semantics
 - Re-writing requires an erase cycle
 - Which erases a whole block
 - And is slow
 - Atomicity of writing typically at word level
 - Blocks can only be erased so many times
- So the operating system needs to smooth out these oddities

What Does That Lead To?

- Different structures for the file system
 - Since you can't easily overwrite data words in place
- Garbage collection to deal with blocks largely filled with inactive data
- Maintaining a pool of empty blocks
- Wear-leveling in use of blocks
- Something to provide desired atomicity of multi-word writes

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 are almost always multiple processes
- The OS has limited physical memory
 - To hold the environment information
- There is usually only one set of registers
 - Or one per core
- The process shares the CPU or core
 - With 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

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

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 for transfer
 - Copy 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 Does That Lead To?

- Need to optimize costs of copying
- Tricky memory management
- Inclusion of complex network protocols in the OS itself
- Worries about message loss, retransmission, etc.
- New security concerns that OS might need to address

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 (PDF) 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 generic file system layers on a particular file system, which layers on abstract disk, which layers on 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 network link may hide causes of packet losses

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

Conclusion

- Stable interfaces are critical to proper performance of an operating system
 - For program development (API)
 - For user experience (ABI)
- Abstractions make operating systems easier to use for both programmers and consumers
- The most important OS abstractions involve memory, interpreters, and communications

Operating System Principles: Processes, Execution, and State

CS 111

Winter 2023

Operating System Principles
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?

- A type of interpreter
- An executing instance of a program
- A virtual private computer
- A process is an *object*
 - Characterized by its properties (*state*)
 - Characterized by its *operations*
 - Of course, not all OS objects are processes
 - But processes are a central and vital OS object type

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 them from other objects
 - Characterizing object's current condition
- Contents of state depends on object
 - Complex operations often mean complex state
 - But representable as a set of bits
 - We can save/restore the bits of the aggregate/total state
 - We can talk of a state subset (e.g., scheduling state)

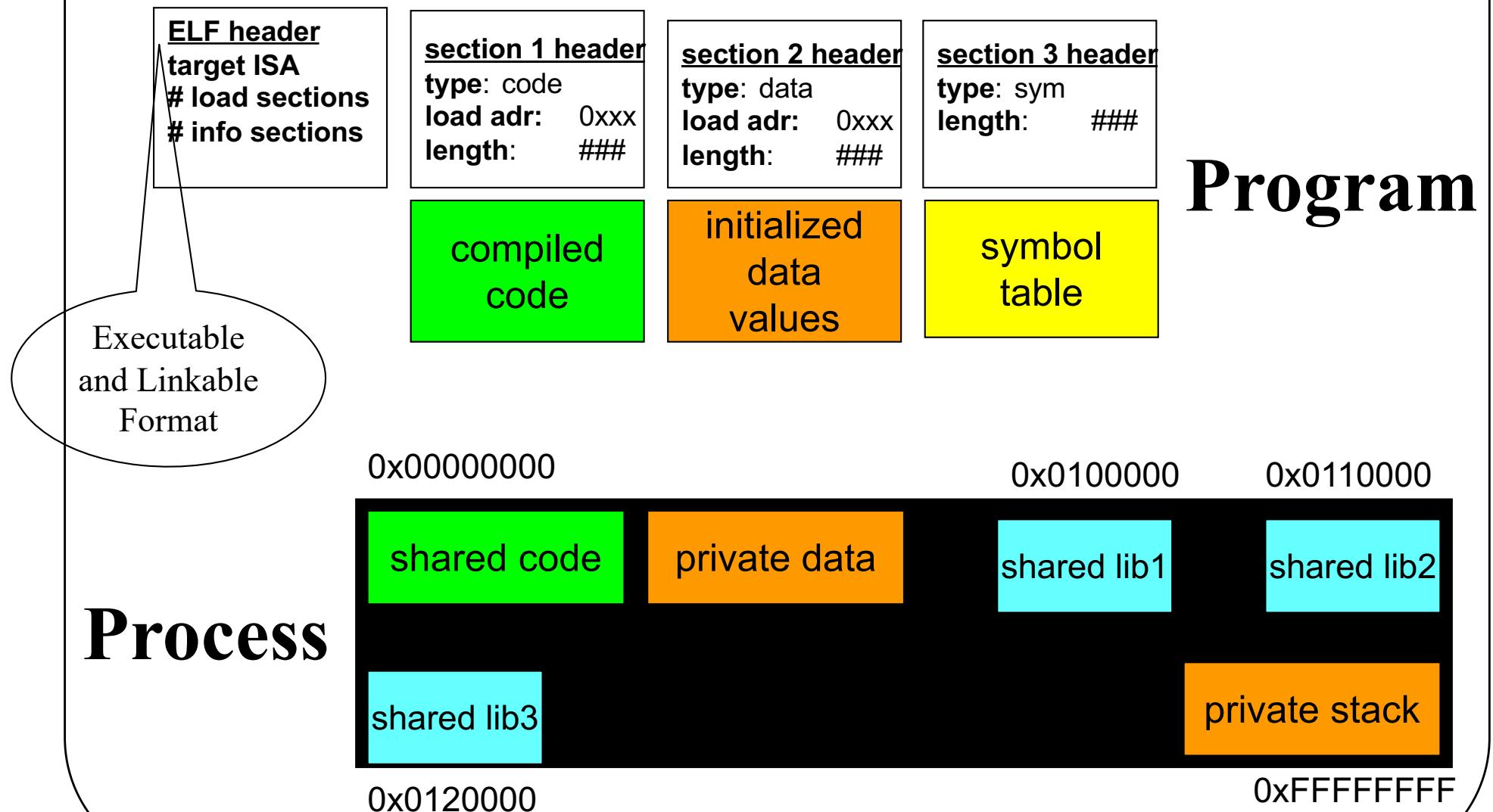
Examples Of OS Object State

- Scheduling priority of a process
- Current pointer into a file
- Completion condition of an I/O operation
- List of memory pages allocated to a process
- OS objects' state is mostly managed by the OS itself
 - Not (directly) by user code
 - It must ask the OS to access or alter state of OS objects

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
 - If an address isn't in its address space, the process can't request access to it
- Modern OSes pretend that every process' address space can include all of memory
 - But that's not true, under the covers

Program vs. Process Address Space



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
 - E.g., code is not writable but must be executable
 - And stacks are readable and writable but not executable
- Each operating system has some strategy for where to put these process memory segments

Layout of Unix Processes in Memory



0x00000000

0xFFFFFFFF

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

Address Space: Code Segments

- We start with a load module
 - The output of a linkage editor
 - All external references have been resolved
 - All modules combined into a few segments
 - Text, data, BSS, etc.
- Code must be loaded into memory
 - Instructions can only be run from RAM
 - A code segment must be created
 - Code must be read in from the load module
 - Map segment into process' address space
- Code segments are read/execute 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 and mapped into the process' address space
 - Initial contents must be copied from load module
 - BSS¹ segments must be initialized to all zeroes
- 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
- 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 CPUs 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
 - E.g., amount of local storage used by each routine
 - Grows larger as calls nest more deeply
 - After calls return, their stack frames can be recycled
- OS manages the process' stack segment
 - Stack segment created at same time as data segment
 - Some OSes allocate fixed sized stack at program load time
 - Some dynamically extend stack as program needs it
- Stack segments are read/write and process private
 - Usually not executable

Address Space: 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
- Shared libraries use less space
 - 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

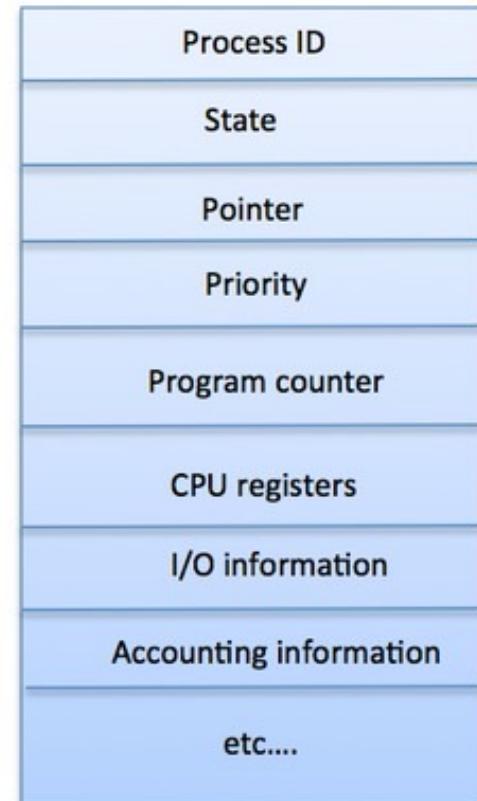
- Registers
 - General registers
 - Program counter, processor status, stack pointer, frame pointer
- Process' own OS resources
 - Open files, current working directory, locks
- But also OS-related state information
- The OS needs some data structure to keep track of all this information

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
- Managed by the OS
- 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)
 - 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 several 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
 - Process table contains one entry (e.g., a PCB) for each process in the system

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
 - This is another data structure, itself
- 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 parent process
- On assumption that the new child process is a lot like the old one
 - Designed decades ago for reasons no longer relevant
 - But the approach has advantages, like easing creation of pipelines

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 Memory

Parent `fork()`

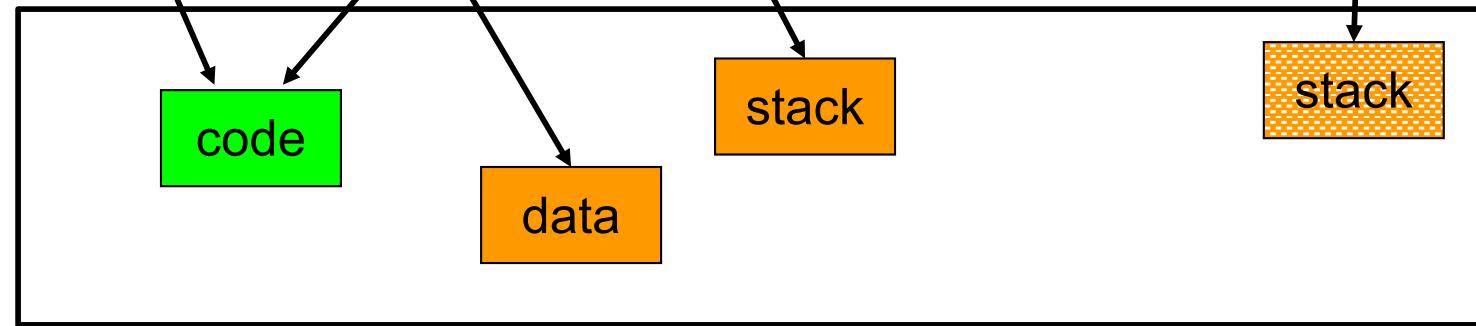


Child



?

RAM



Forking and the Data Segments

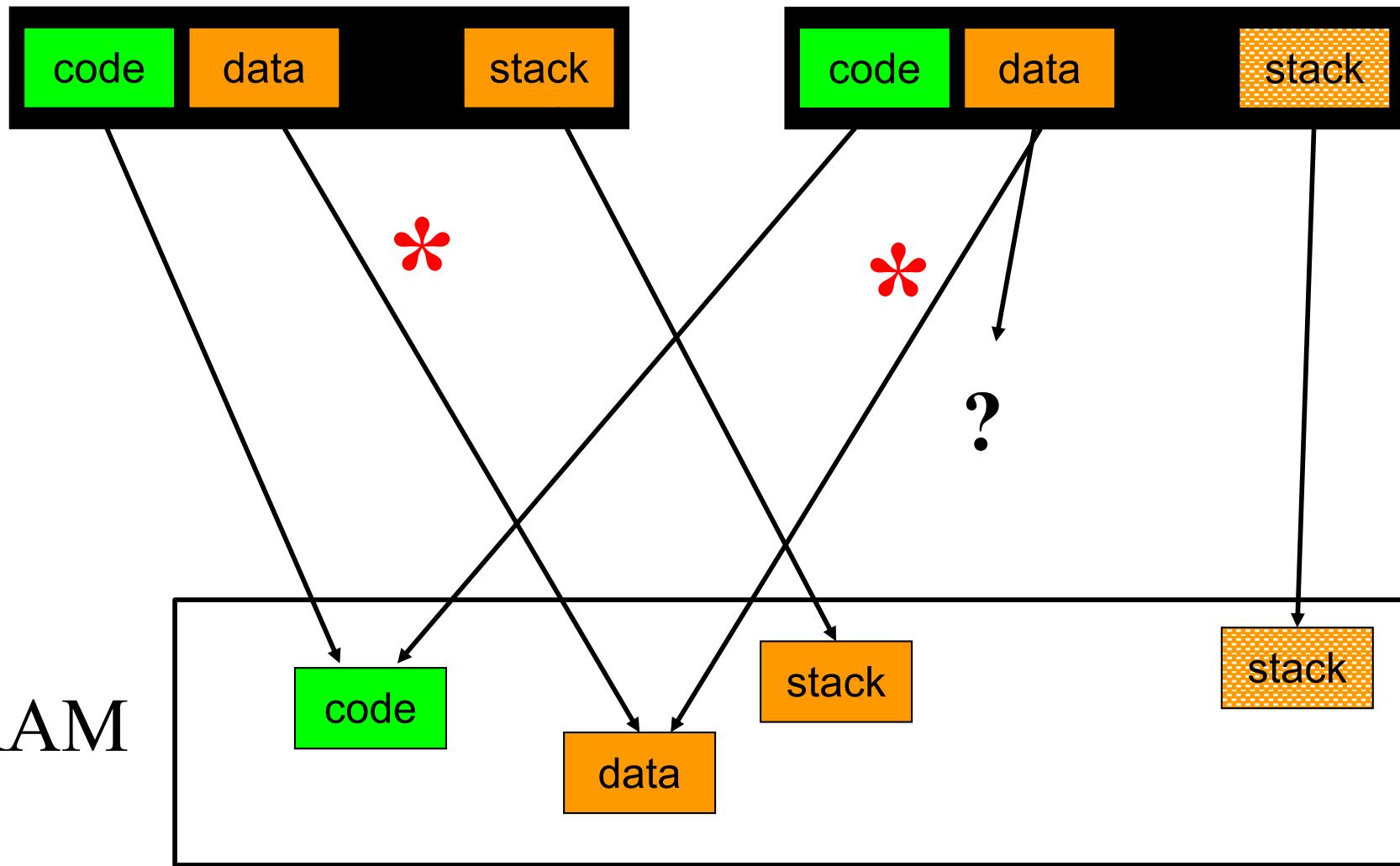
- Forked child shares the parent's code
- But not its stack
 - It has its own stack, initialized to match the parent's
 - Just as if a second process running the same program had reached the same point in its run
- Child should also have its own data segment
 - Forked processes do not share their data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

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

Forking and Copy on Write

Parent Child



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
 - Though you could call it without forking

How Does the OS Handle Exec?

- Must get rid of the child's old code
 - More precisely, don't point to it any more
- 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

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 process table
 - And reclaim its memory

Running Processes

- Processes must execute code to do their job
- Which means the OS must give them access to a processor core
- But usually more processes than cores
 - Easily 400-600 on a typical modern machine
- So processes will need to share the cores
 - So 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 core's hardware must be initialized
 - Either to an initial state or whatever state the process 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
 - Without any OS intervention
- 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 most 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 and user-space code processes
 - For OS emulators (e.g., QEMU on Linux)
 - For virtual machines
- Enter the OS as seldom as possible
 - Get back to the application as quickly as possible

The key to
good system
performance

!

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

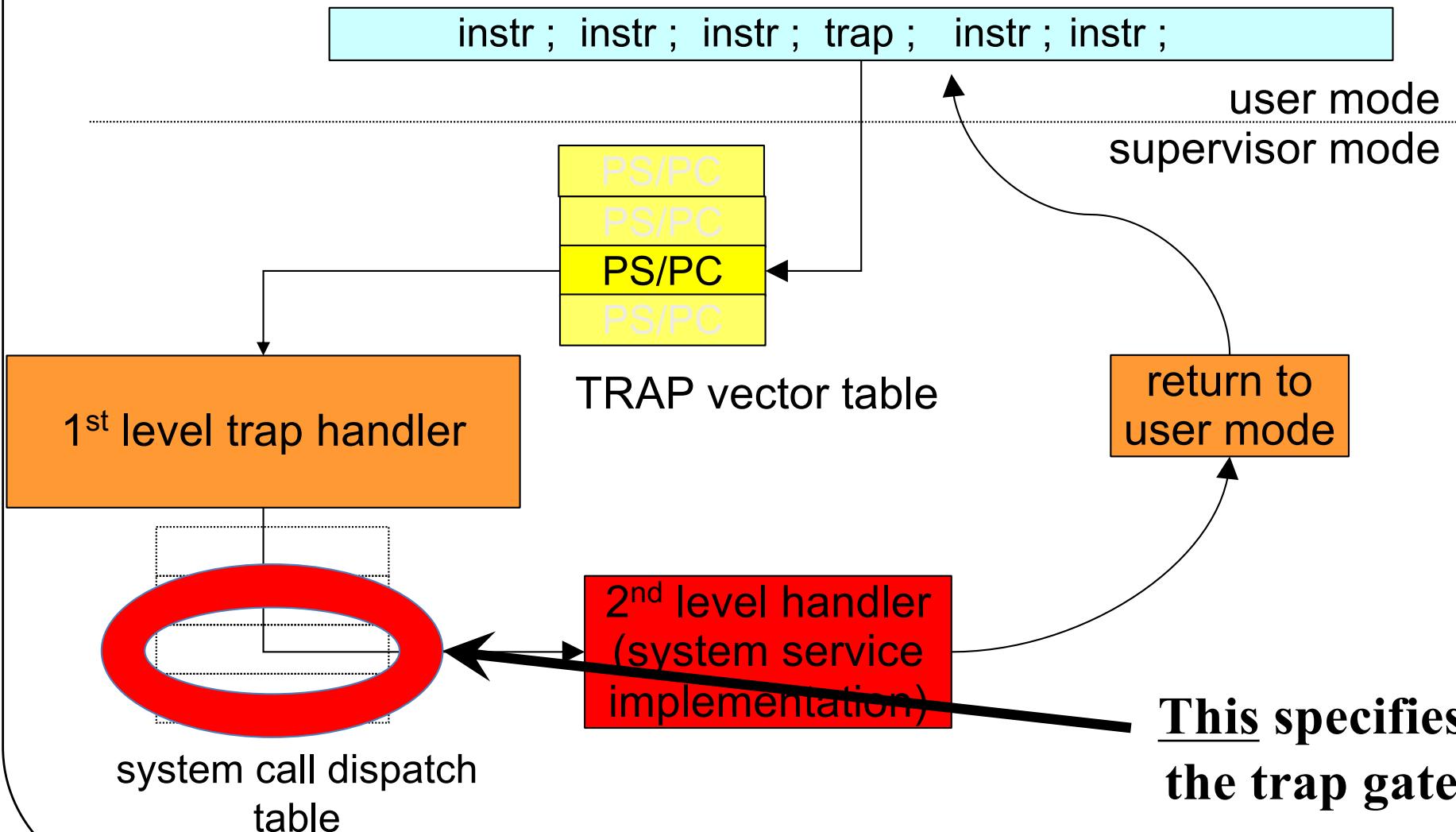
- Inherently unpredictable
- Programs can't check for them, since no way of knowing when and if they happen
- Some languages support try/catch operations
- Hardware and OS support traps
 - Which catch these exceptions and transfer control to the OS
- Operating systems also use these for *system calls*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

Using Traps for System Calls

- Made possible at processor design time, not OS design time
- Reserve one or more privileged instruction for system calls
 - Most ISAs specifically define such instructions
- Define system call linkage conventions
 - Call: r0 = system call number, r1 points to arguments
 - Return: r0 = return code, condition code indicates success/failure
- Prepare arguments for the desired system call
- Execute the designated system call instruction
 - Which causes an exception that traps to the OS
- OS recognizes & performs the requested operation
 - Entering the OS through a point called a *gate*
- Returns to instruction after the system call

System Call Trap Gates

Application Program



Trap Handling

- Partially hardware, partially software
- Hardware portion of trap handling
 - Trap cause an 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 ...
 - Could run a lot of OS code to do this

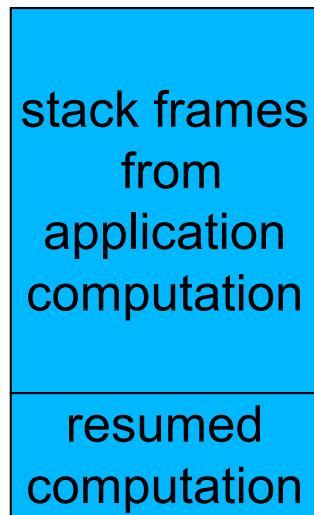
Where do you
think this table
came from?

Traps and the Stack

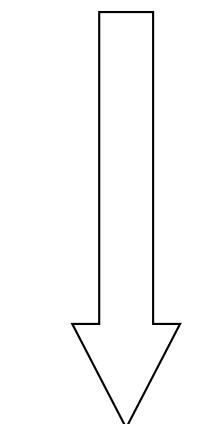
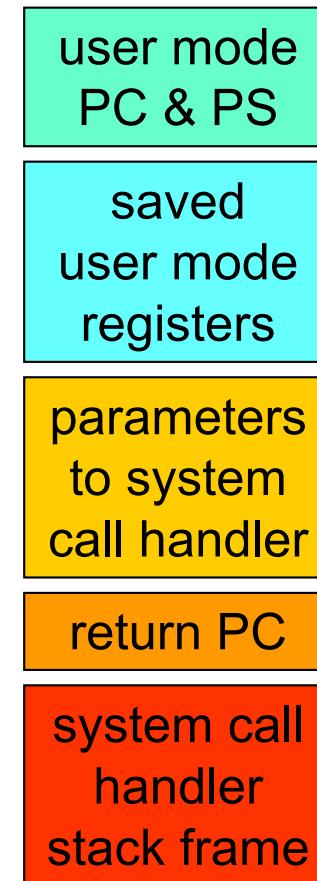
- The code to handle a trap is just code
 - Although run in privileged mode
- It requires a stack to run
 - Since it might call many routines
- How does the OS provide it with the necessary stack?
- While not losing track of what the user process was doing?
- Or leaving sensitive data in the user's stack area?

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
- Other time 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 segments
 - Open files and devices
 - Supervisor stack
 - And many other things

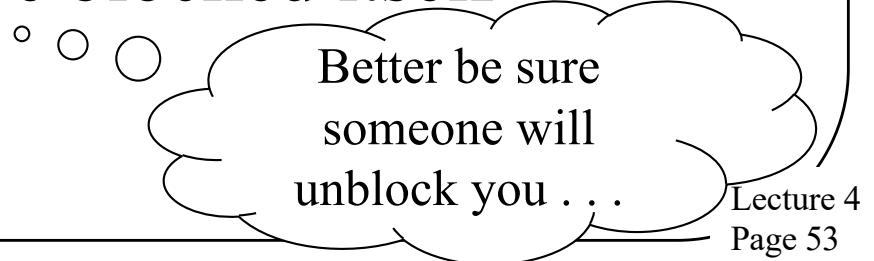
Which implies that they can screw these up if they aren't careful

Blocked Processes

- One important process state element is whether a process is ready to run
 - No point in trying to run it if it isn't ready to run
 - Processes not ready to run are *blocked*
- Why might it not be ready to run?
- 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 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 remove them
 - And a process can ask to be blocked itself
 - Through a system call



Who Handles Blocking?

- Usually happens in a resource manager
 - When process needs an unavailable resource
 - Change process' scheduling state to “blocked”
 - Call the scheduler and yield the CPU
 - When the required resource becomes available
 - Change process' scheduling state to “ready”
 - Notify scheduler that a change has occurred

Conclusion

- Processes are the fundamental OS interpreter abstraction
- They are created by the OS at application request and managed via process descriptors
- There are different methods for creating processes
- Processes use system calls to transfer control to the OS to obtain system services

Operating System Principles: Scheduling

CS 111

Winter 2023

Operating System Principles
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 flash drive?
- If multiple messages are to be sent over the network, in what order should they be sent?
- We'll primarily consider scheduling processes

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

- How should we schedule our cores?
- 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
- Service Level Agreement (SLA)
 - To share resources between multiple customers
 - Make sure all agreements are met
 - Various agreements may differ in details

Scheduling: Policy and Mechanism

- The scheduler will move jobs onto and off of a processor core (*dispatching*)
 - Requiring various mechanics to do so
 - Part of the scheduling mechanism
- 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

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 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 work to run something else, it's preemptive

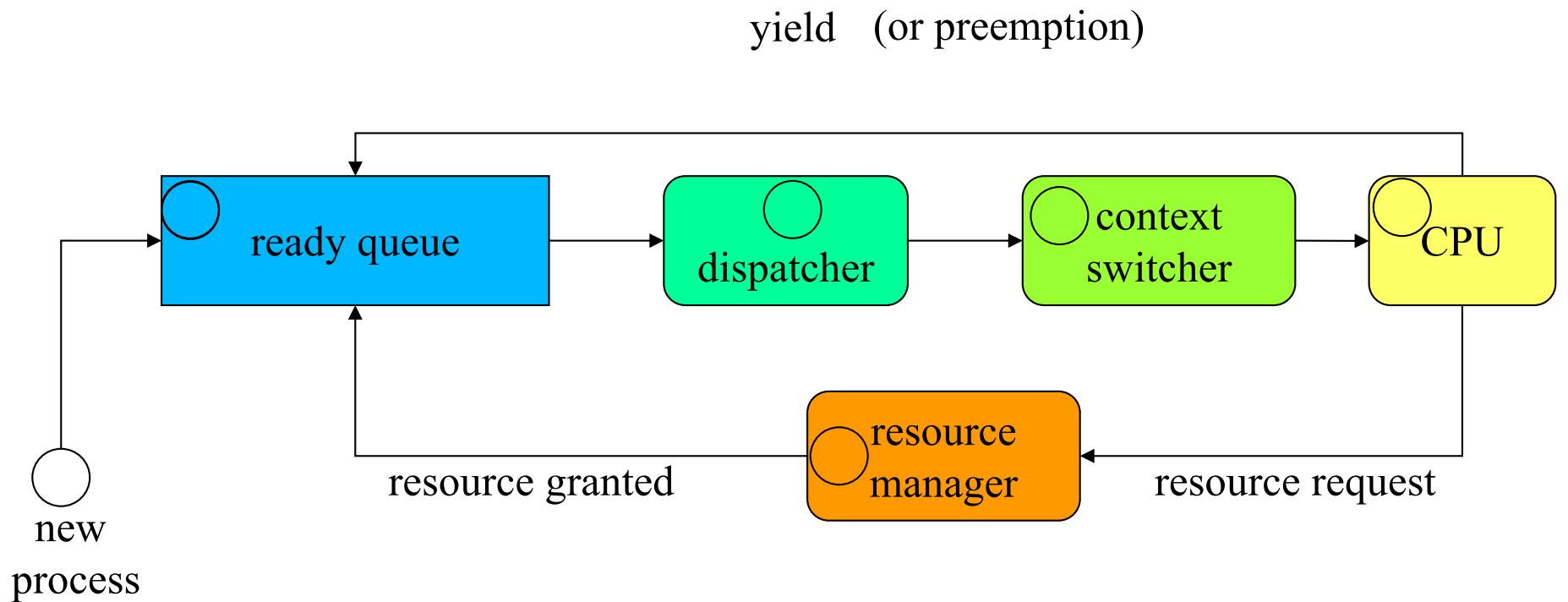
Pros and Cons of Non-Preemptive Scheduling

- + Low scheduling overhead
- + Tends to produce high throughput
- + Conceptually very simple
- Poor response time
- Bugs can cause machine to freeze up
 - If process contains infinite loop, e.g.
- Poor 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
- + Good for real-time and priority scheduling
- More complex
- Requires ability to cleanly halt process and save its state
- May not get good throughput
- Possibly higher overhead

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 a brief description
 - 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

Software can't optimize what it doesn't control.

Other Scheduling Metrics

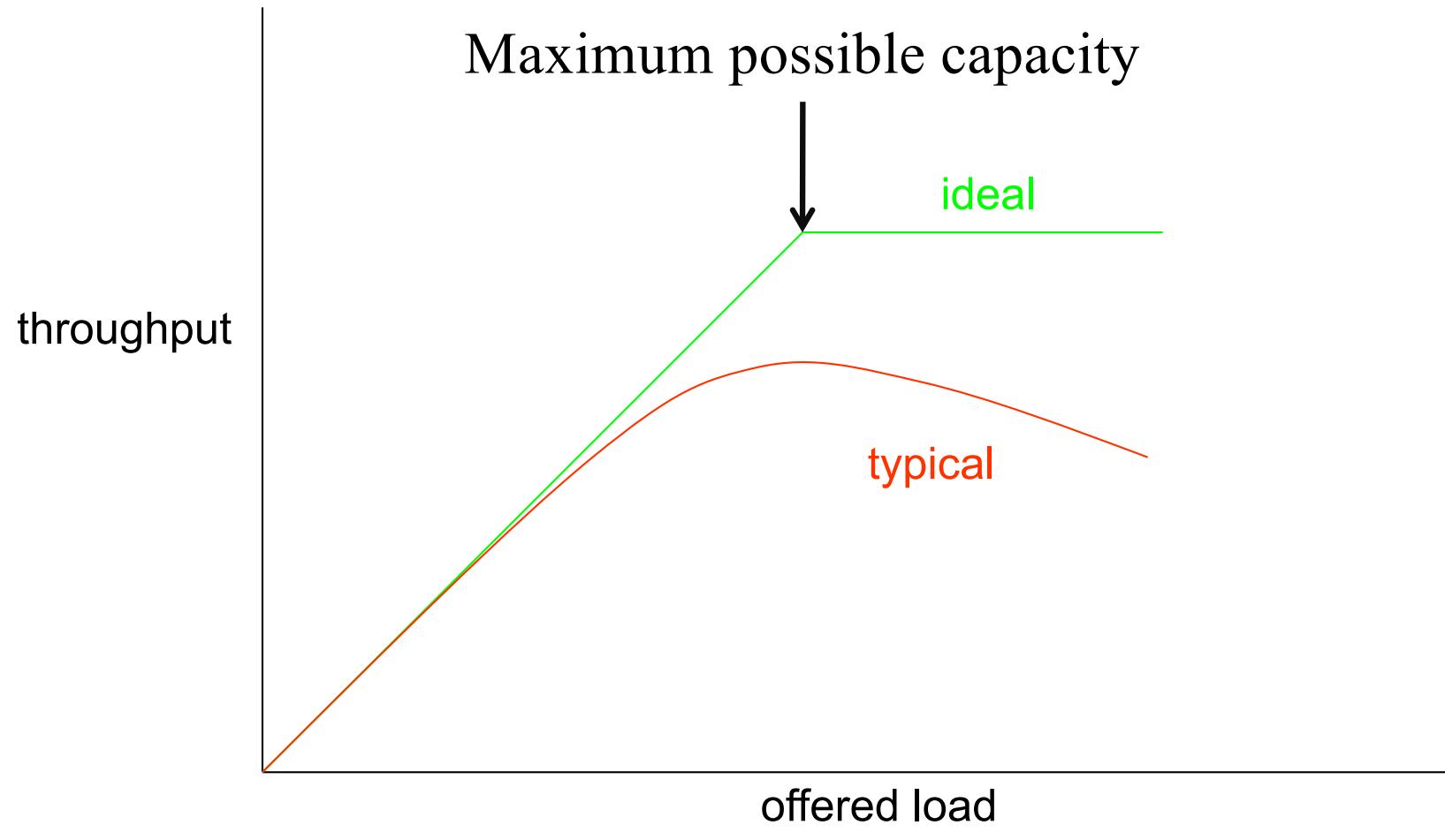
- Mean time to completion (seconds)
 - For a particular job mix (benchmark)
- Throughput (operations per second)
 - For a particular activity or job mix (benchmark)
- Mean response time (milliseconds)
 - Time spent on the ready queue
- Overall “goodness”
 - Requires a customer-specific weighting function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

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 when ready
 - This time is controlled by the scheduler
- Proposed metric:
 - Time that “ready” processes spend waiting for the CPU

So the scheduler can optimize this!

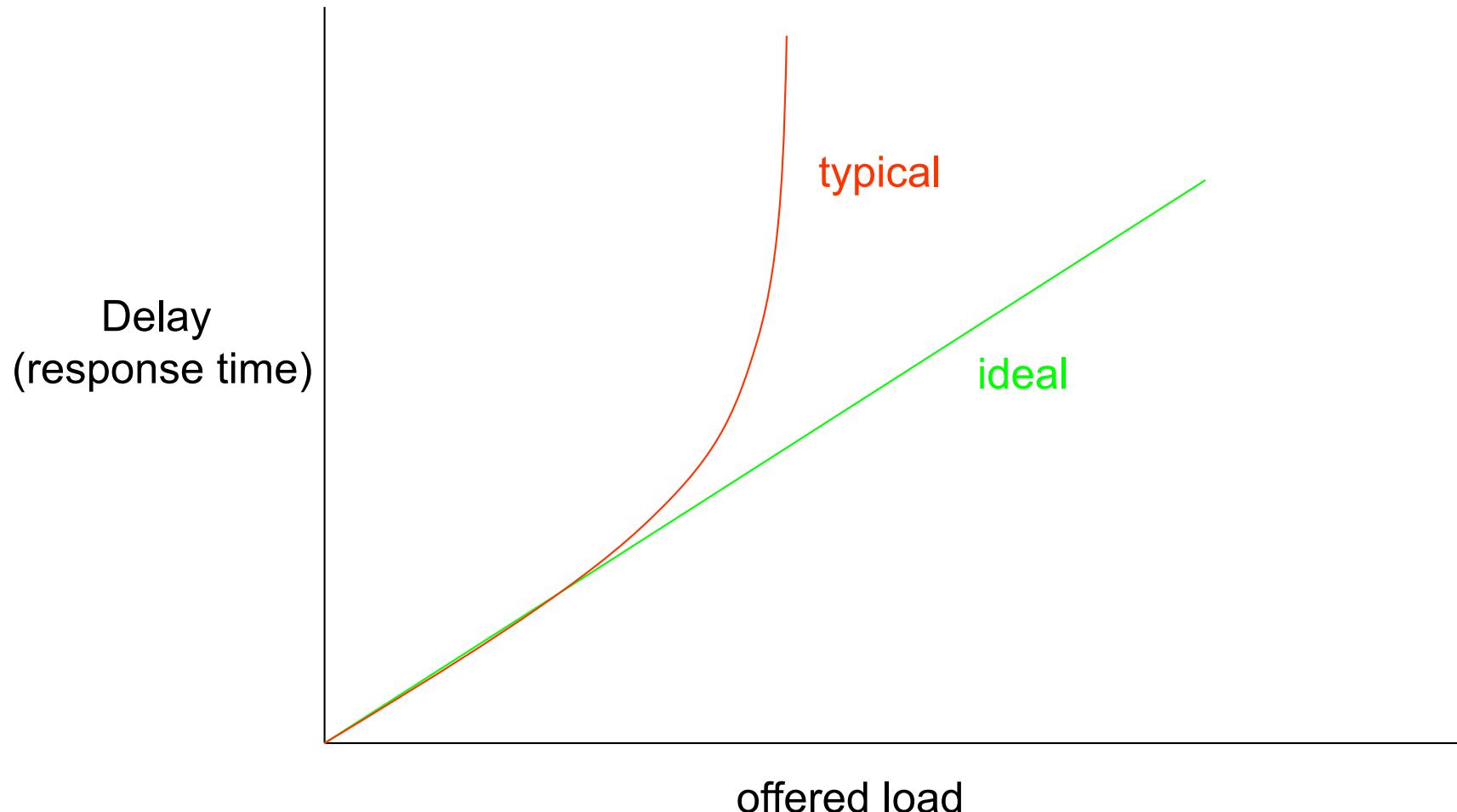
Typical Throughput vs. Load Curve



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 you're careful, overheads explode during periods of heavy load

Graceful Degradation

- When is a system “overloaded”?
 - When it is no longer able to meet its 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

- 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 in lecture
 - It's in the readings
- 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	

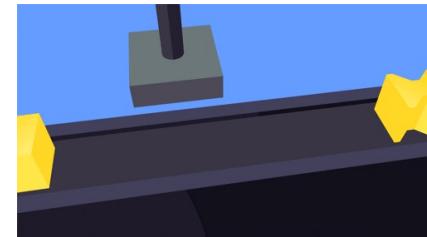
Note: Average is worse than total/5 because four other processes had to wait for the slow-poke who ran first.

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. When response time is not important (e.g., batch)
 2. Where minimizing overhead more important than any single job's completion time (e.g., expensive HW)
 3. 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 stamp the widget before the conveyer belt moves on, 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 enforces 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 miss a few
- 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
- Prune the queue to remove missed deadlines
- Goal: Minimize *total lateness*

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 more important process becomes ready
 - Perhaps as a result of an I/O completion interrupt
 - If running process's importance 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 a 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
 - They “halt themselves”
- 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*)

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						550	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:										1275	27

First process completed: 475 msec

Comparing Round Robin to FIFO

- Context switches: 27 vs. 5 for FIFO
 - Clearly more expensive
- First job completed: 475 msec vs. 350 for FIFO
 - Can take longer to complete first process
- Average waiting time: 100 msec vs. 595 for FIFO
 - For first opportunity to compute
 - Clearly more responsive

Choosing a Time Slice

- Performance of a preemptive scheduler depends heavily on how long the 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

Probably the most
important cost
nowadays

Priority Scheduling Algorithms

- 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

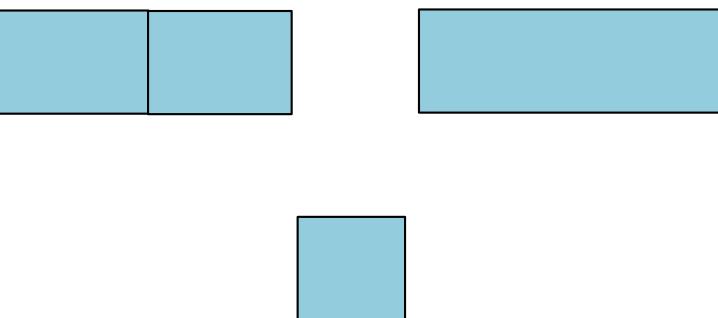
- If non-preemptive, priority scheduling is just about ordering processes
- Much like shortest job first, but ordered by priority instead
- But what if scheduling is preemptive?
- In that case, when new process is created, it might preempt running process
 - If its priority is higher

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

- A soft priority system
- 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

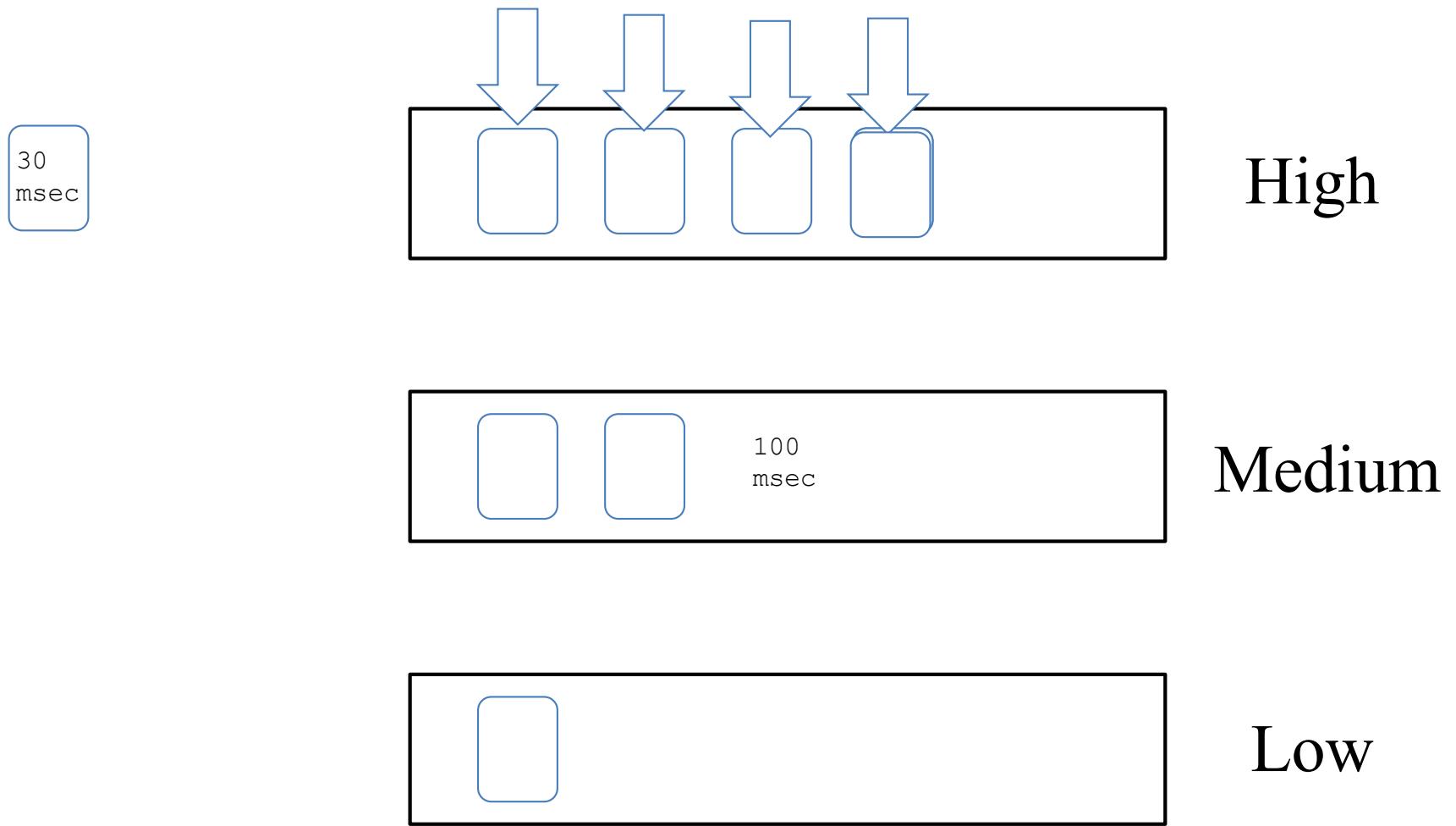
Multi-Level Feedback Queue (MLFQ) Scheduling

- One time slice length may not fit all processes
- Create multiple ready queues
 - Short quantum (foreground) tasks that finish quickly
 - Short but high priority time slices
 - To optimize response time
 - Long quantum (background) tasks that run longer
 - Longer but low priority time slices
 - To minimize overhead
- Round robin within a queue

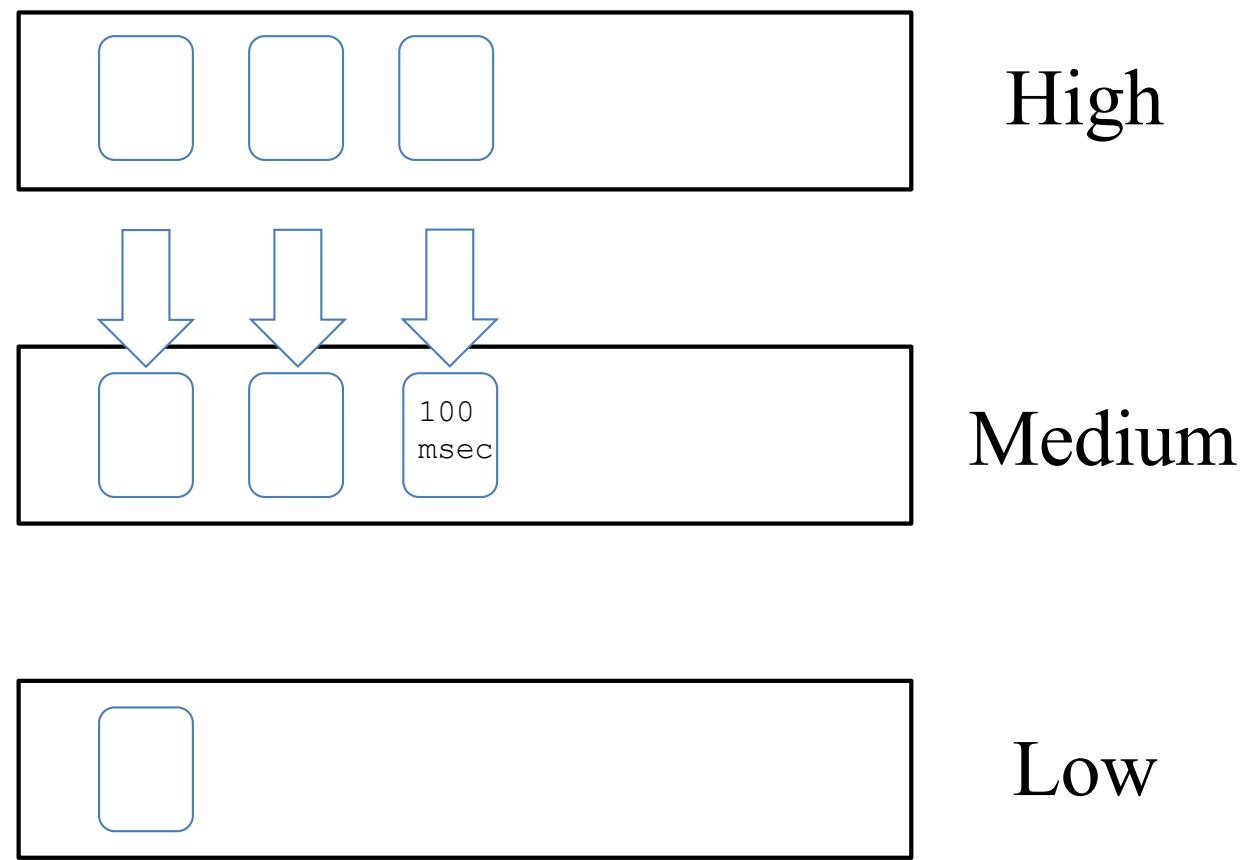
How Do I Know What Queue To Put New Process Into?

- If it's in the wrong queue, its scheduling discipline causes it problems
- Start all processes in short quantum (high priority) queue
 - Give it a standard allocation of CPU
 - Every time it runs, reduce its allocation
 - Move to longer quantum (lower priority) queue after it uses its allocation
- Periodically move all processes to a higher priority queue
 - To avoid starvation

MLFQ At Work

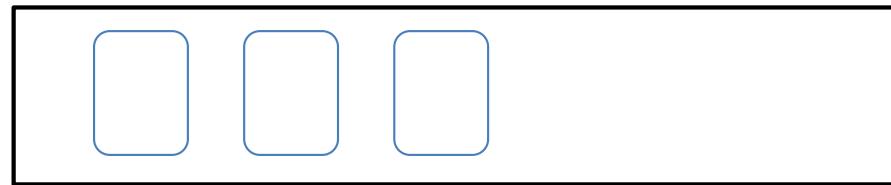


MLFQ Continuing



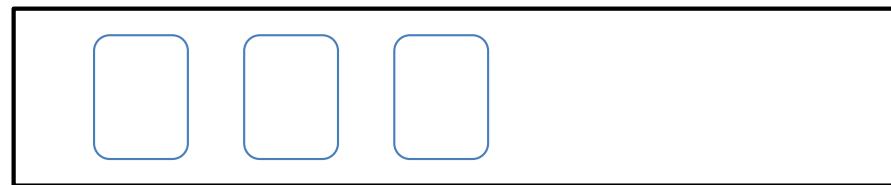
What About Fairness?

Periodically
promote
everyone



High

Resetting
time slices
accordingly



Medium



Low

What Benefits Do We Expect From MLFQ?

- Acceptable response time for interactive jobs
 - Or other jobs with regular external inputs
 - It won't be too long before they're scheduled
 - But they won't waste CPU running for a long time
- Efficient but fair CPU use for non-interactive jobs
 - They run for a long time slice without interruption
 - If they're starved, eventually they get a priority boost
- Dynamic and automatic adjustment of scheduling based on actual behavior of jobs

Operating System Principles: Memory Management

CS 111

Winter 2023

Operating System Principles
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

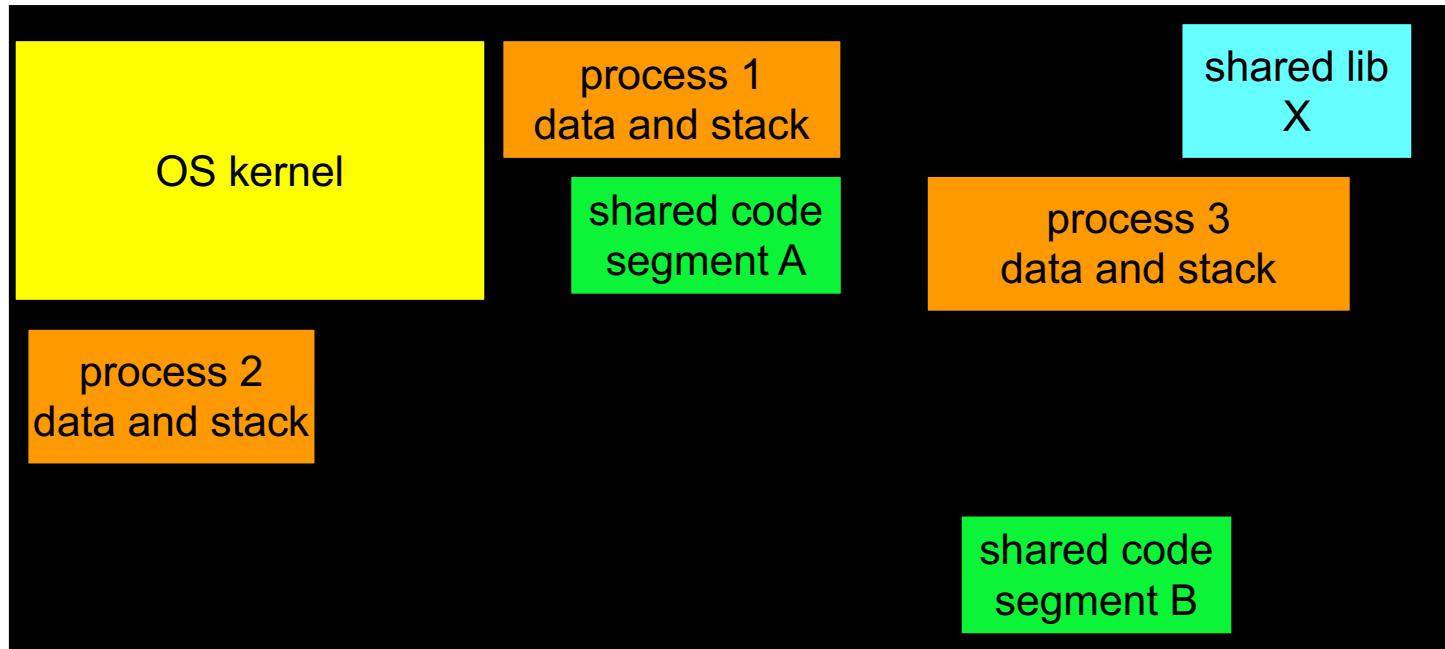
2. Efficiency

- 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 RAM cell has a particular physical address
 - Essentially a location on a memory chip
- Decades ago, that address was used by processes to name memory locations
- Now processes use virtual addresses
 - Which is not a location on a memory chip
 - And usually isn't the same as the actual physical address
- More flexibility in memory management, but requires virtual to physical translation

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

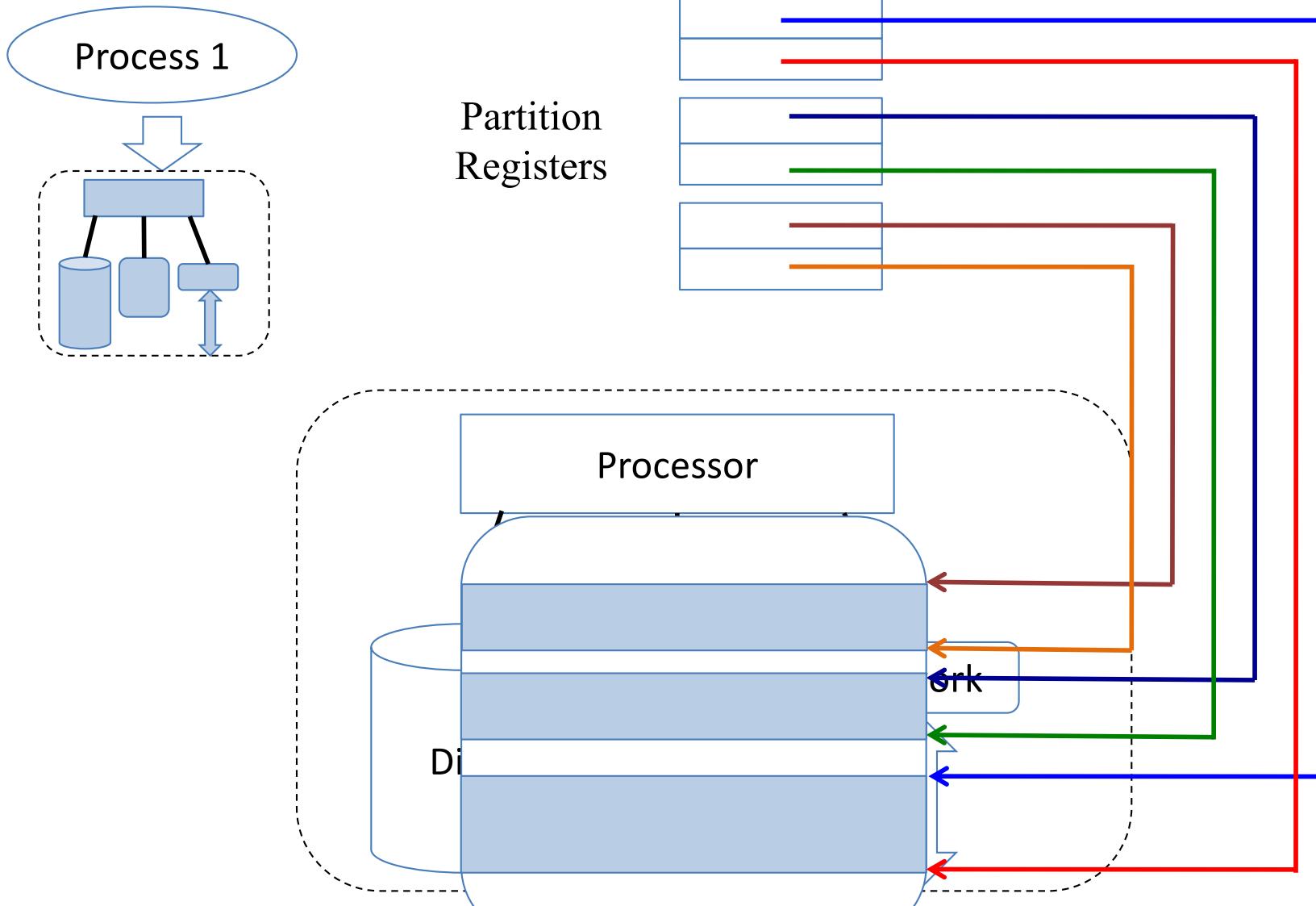
Fixed Partition Allocation

- 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 inefficiencies in memory allocation
- With too much fragmentation,
- 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

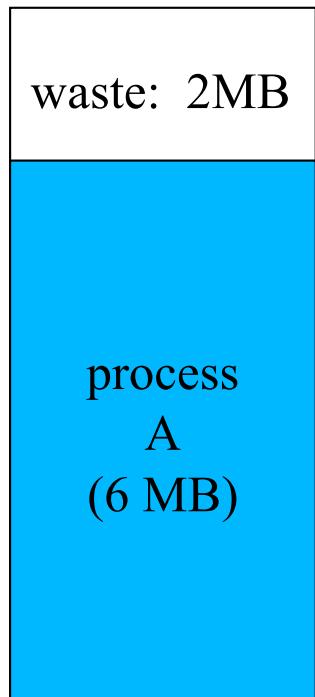
Available partition sizes:

8 Mbytes

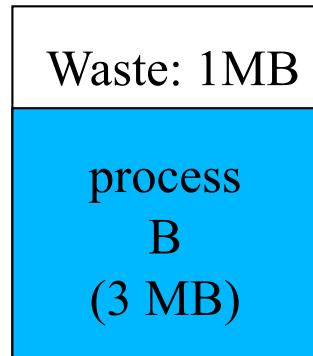
4 Mbytes

4 Mbytes

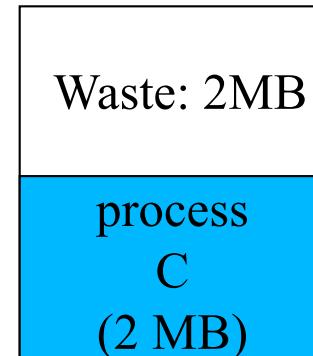
$$\begin{aligned}\text{Total waste} &= 2\text{MB} + 1\text{MB} + 2\text{MB} = \\ &5/16\text{MB} = 31\%\end{aligned}$$



Partition 1
8MB



Partition 2
4MB



Partition 3
4MB

If someone asks for a 3MB partition, you can't provide it

Even though there's 5 MB unused

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 *inside* 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 size of a fixed-sized block
 - The actual sizes that processes use
- Average waste: 50% of each block

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 almost any size requested
 - Each partition has contiguous memory addresses
 - Processes have access permissions for the partitions
 - Potentially shared between processes
- Each process could have multiple partitions
 - With different sizes and characteristics
- In basic scheme, still only physical addresses

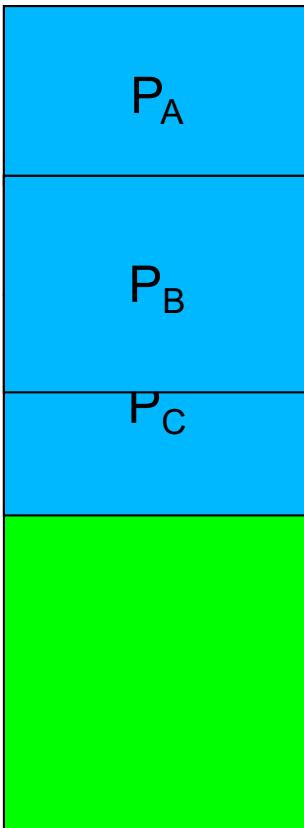
Problems With Dynamic Partitions

- 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
 - Of a different kind . . .

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”

Illustrating the Expansion 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

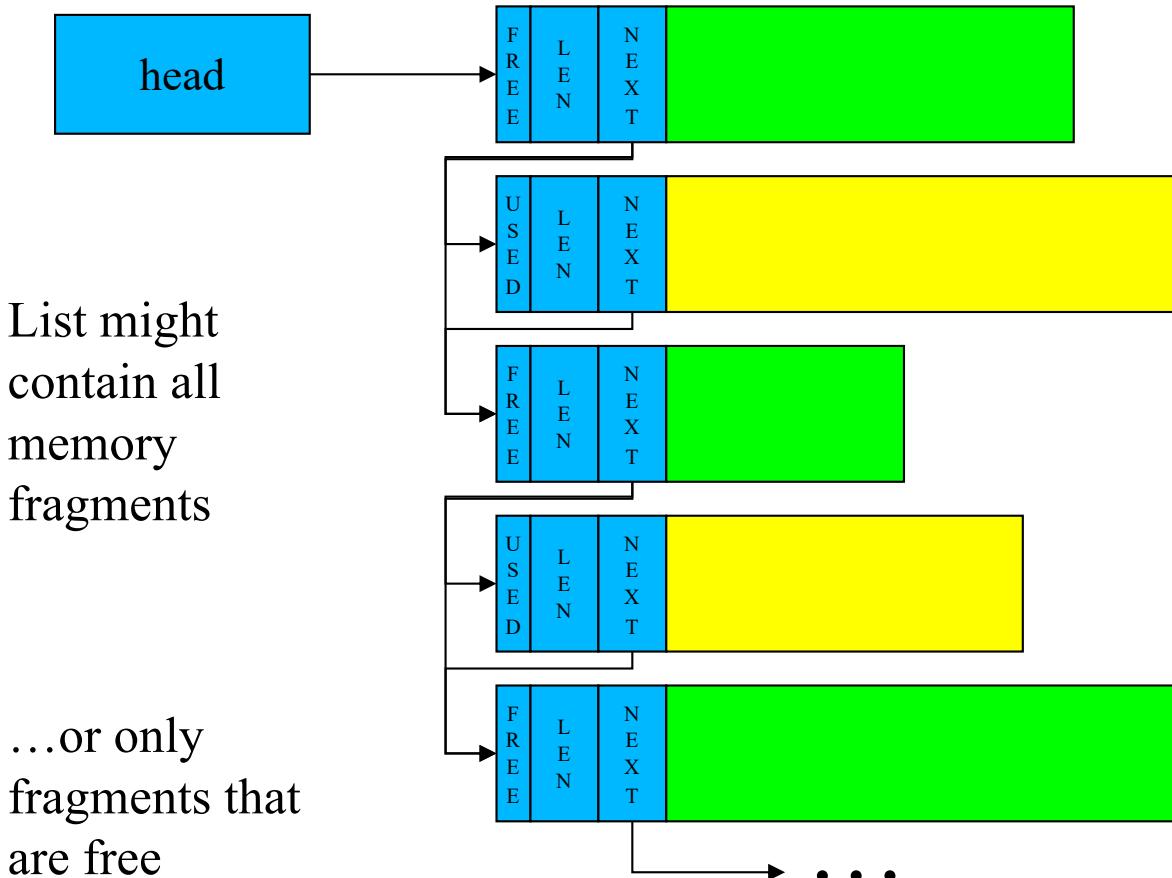
How To Keep Track of Variable Sized Partitions?

- 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 the free list
- When a process frees memory
 - Put freed memory back on the free list

Managing the Free List

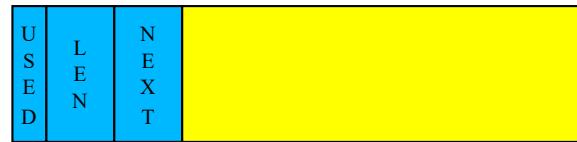
- Fixed sized blocks are easy to track
 - A bit map indicating which blocks are free
- Variable chunks require more information
 - A linked list of descriptors, one per chunk
 - Each descriptor lists the size of the chunk and whether it is free
 - Each has a pointer to the next chunk on list
 - Descriptors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

The Free List

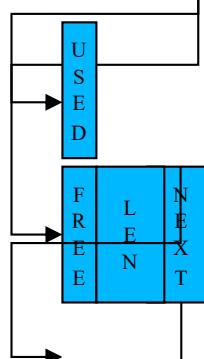


Free Chunk Carving

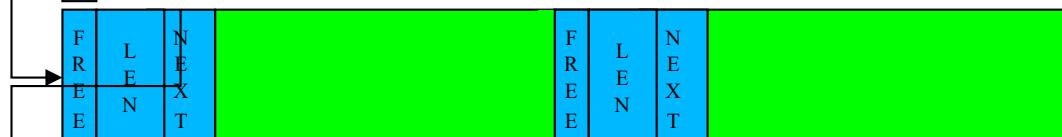
1. Find a large enough free chunk



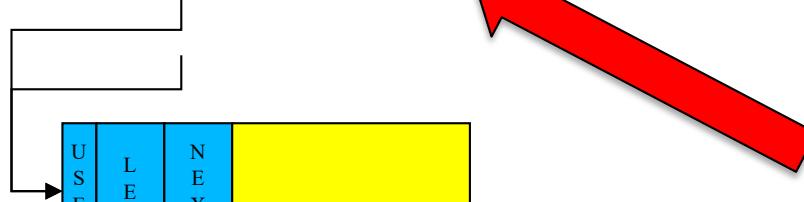
2. Reduce its len to requested size



3. Create a new header for residual chunk



4. Insert the new chunk into the list



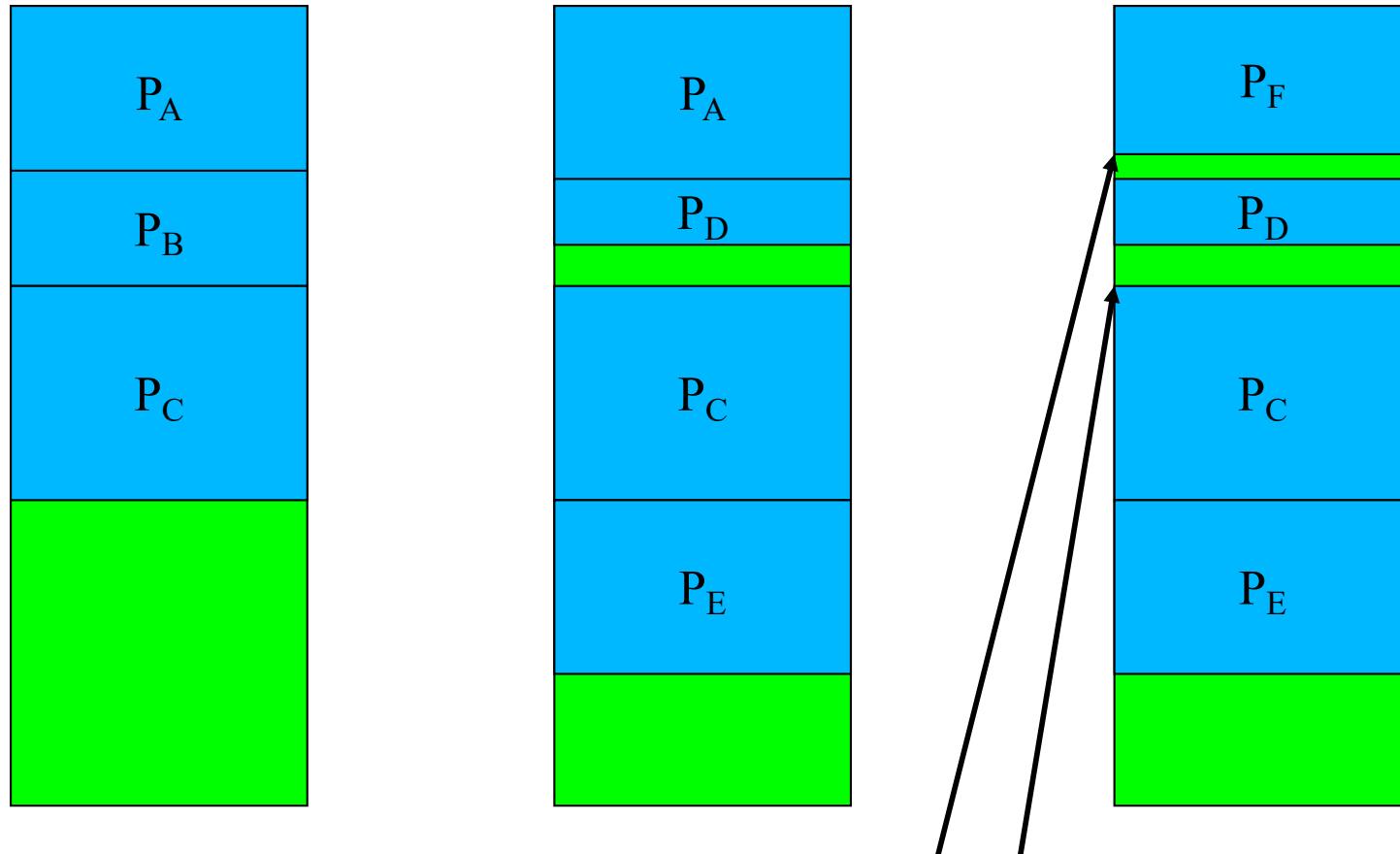
5. Mark the carved piece as in use



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

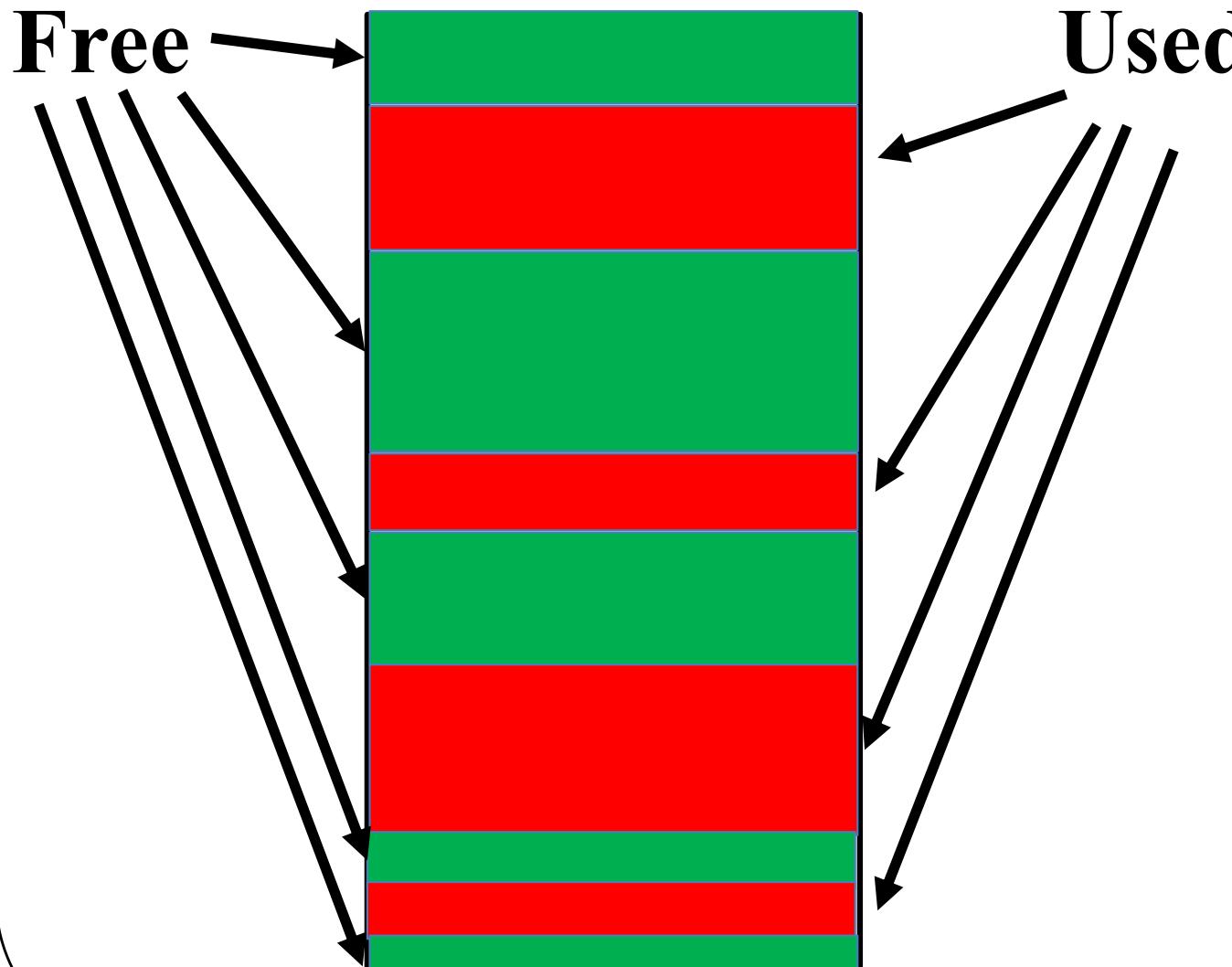
External Fragmentation: Causes and Effects

- Each allocation creates left-over free 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

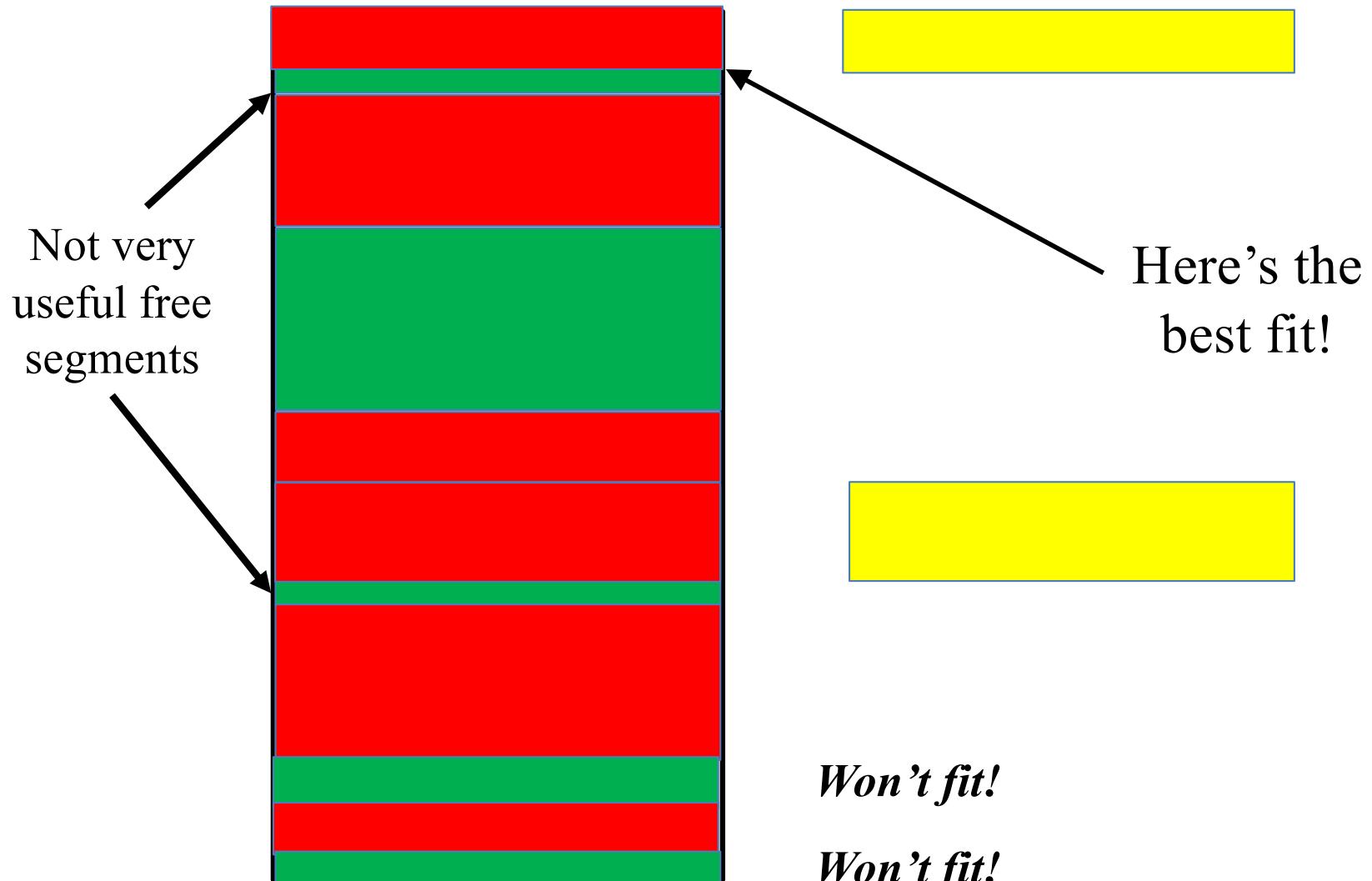
Allocating Partitions in Memory



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

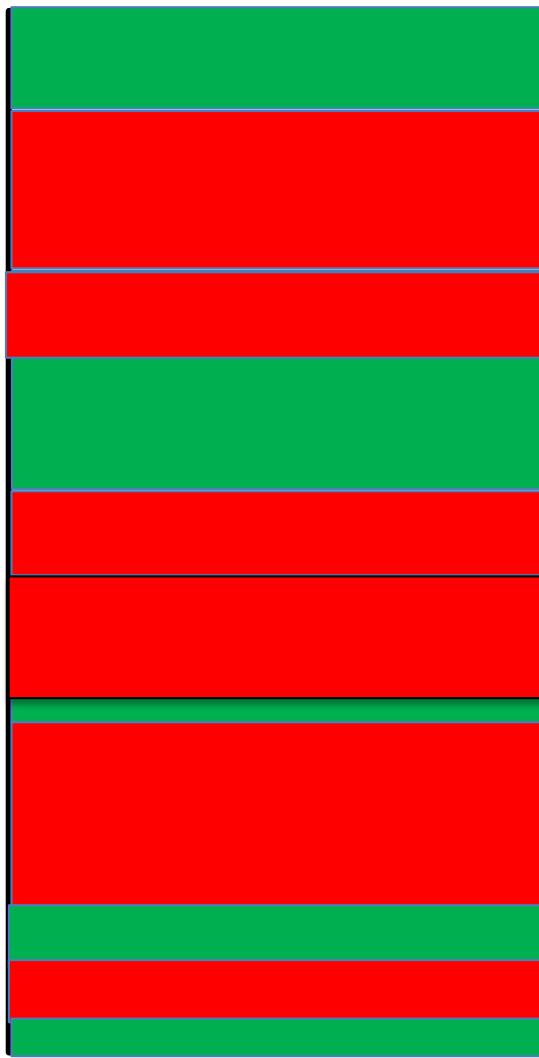
Best Fit in Action



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

Worst Fit in Action



Comparing Best and Worst Fit

Best fit



Worst fit



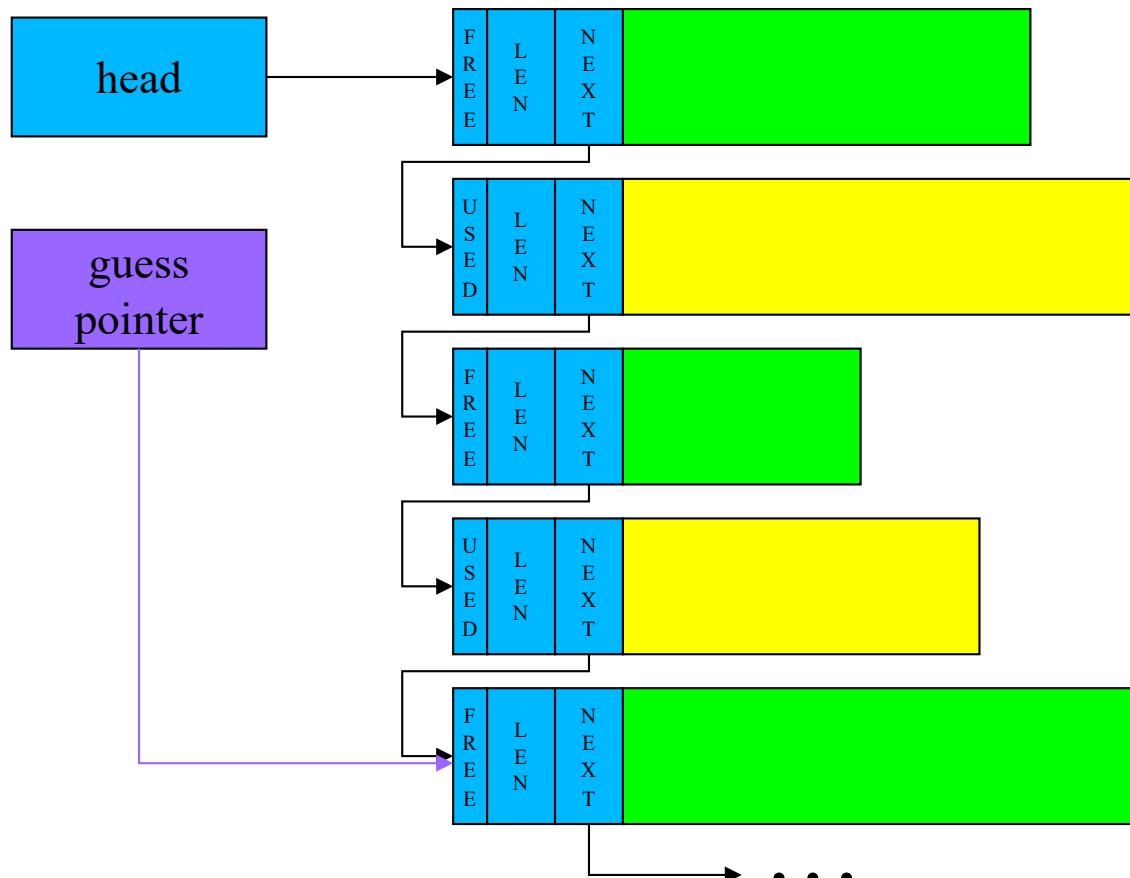
First Fit

- 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.



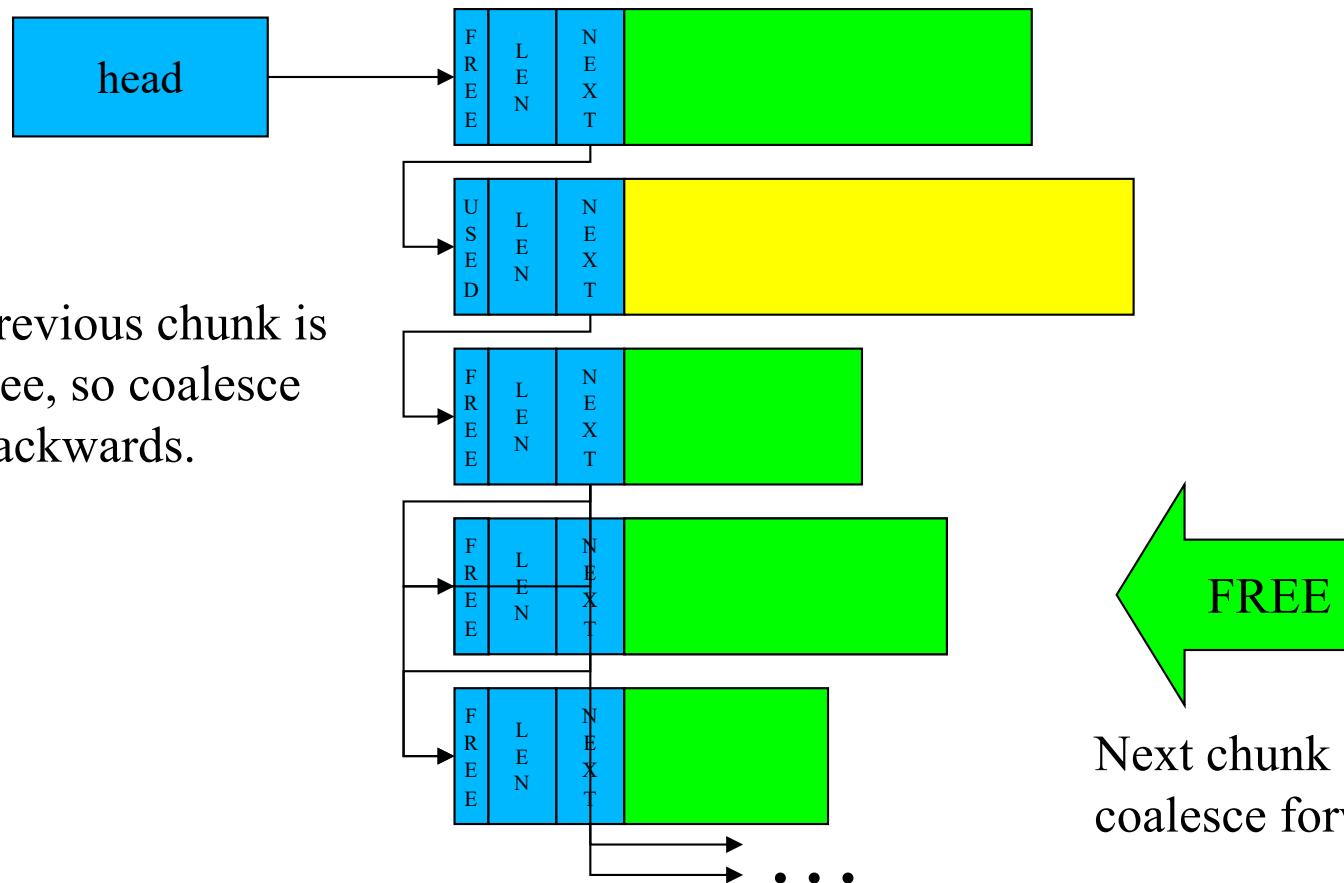
Next Fit Properties

- 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
 - 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
 - Order list by chunk address, so neighbors are close
- Counters forces of external fragmentation

Free Chunk Coalescing



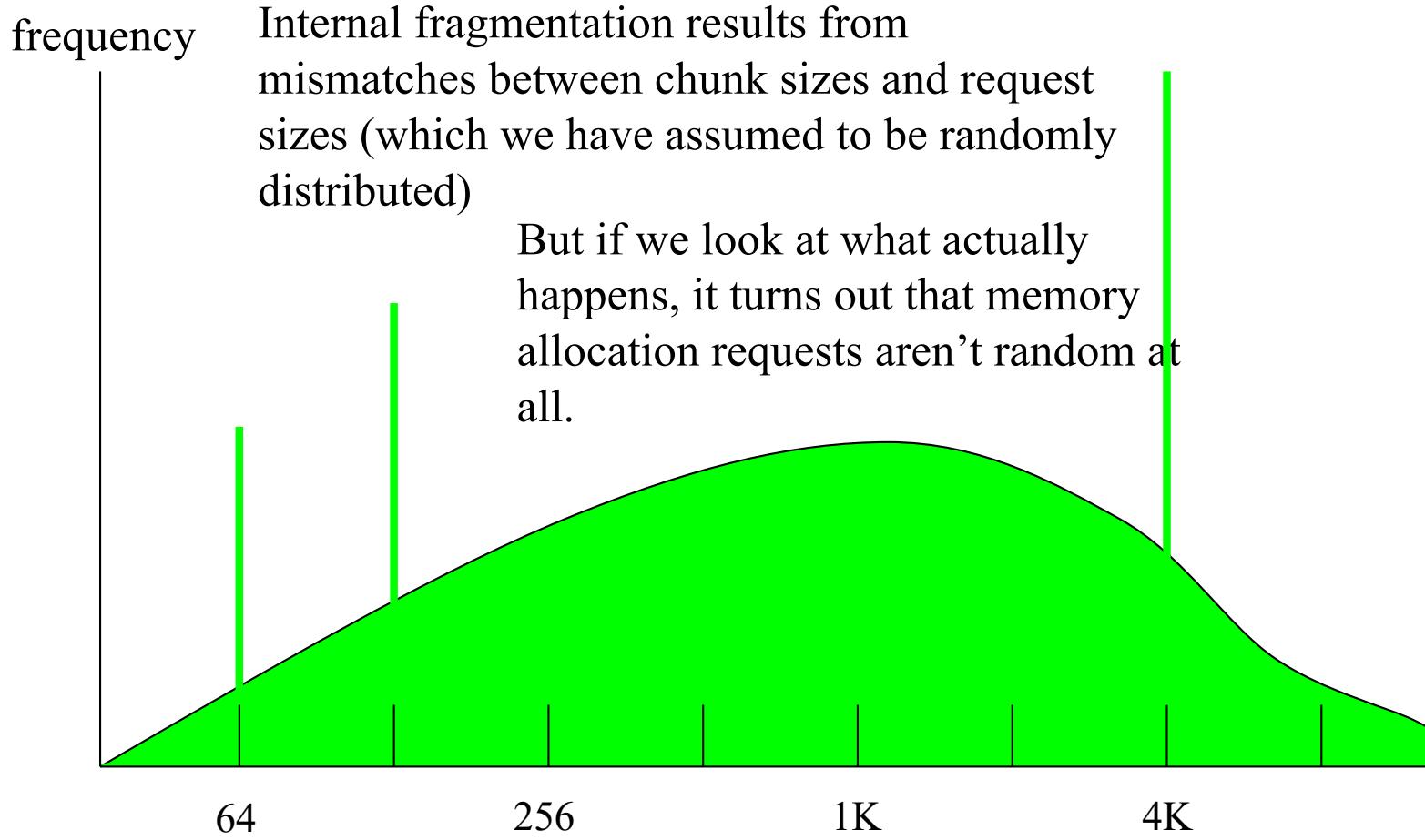
Fragmentation and Coalescing

- 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

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?

A Special Case for Fixed Allocations



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

How Big Should the Buffer Pool Be?

- Resize it automatically and dynamically
- 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 buffer
 - 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 complement (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?

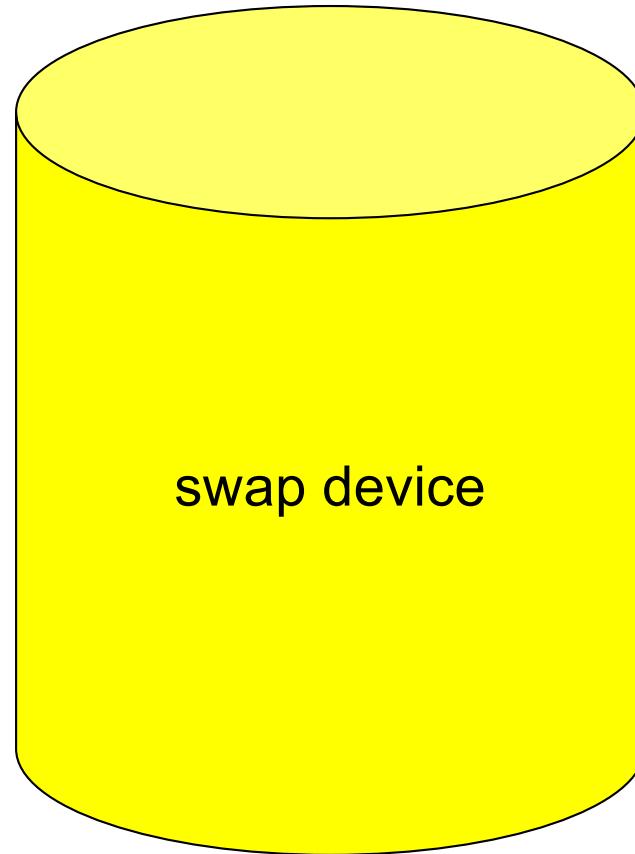
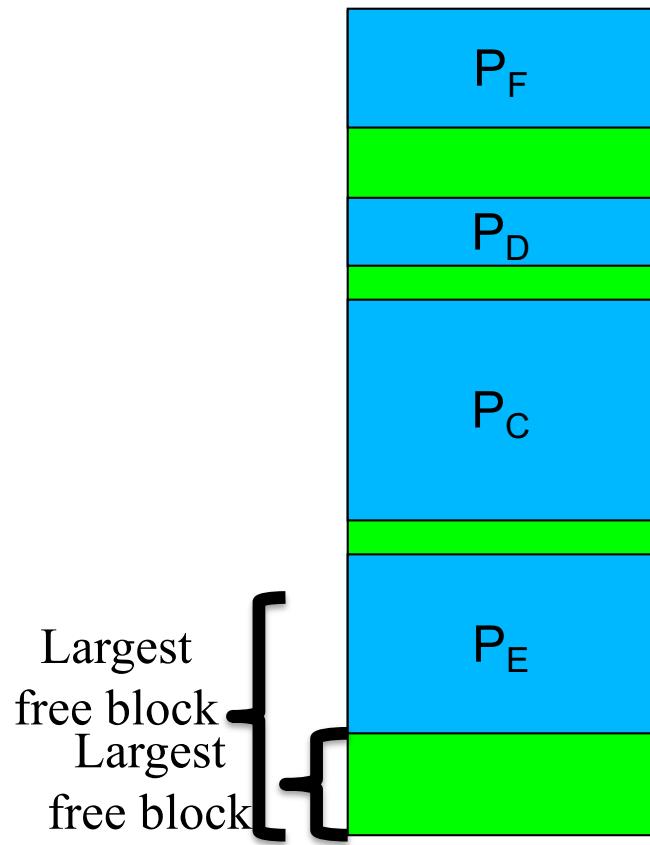
Problems With General Garbage Collection

- 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 processes needing more memory would block until some is freed, slowing the system
- 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

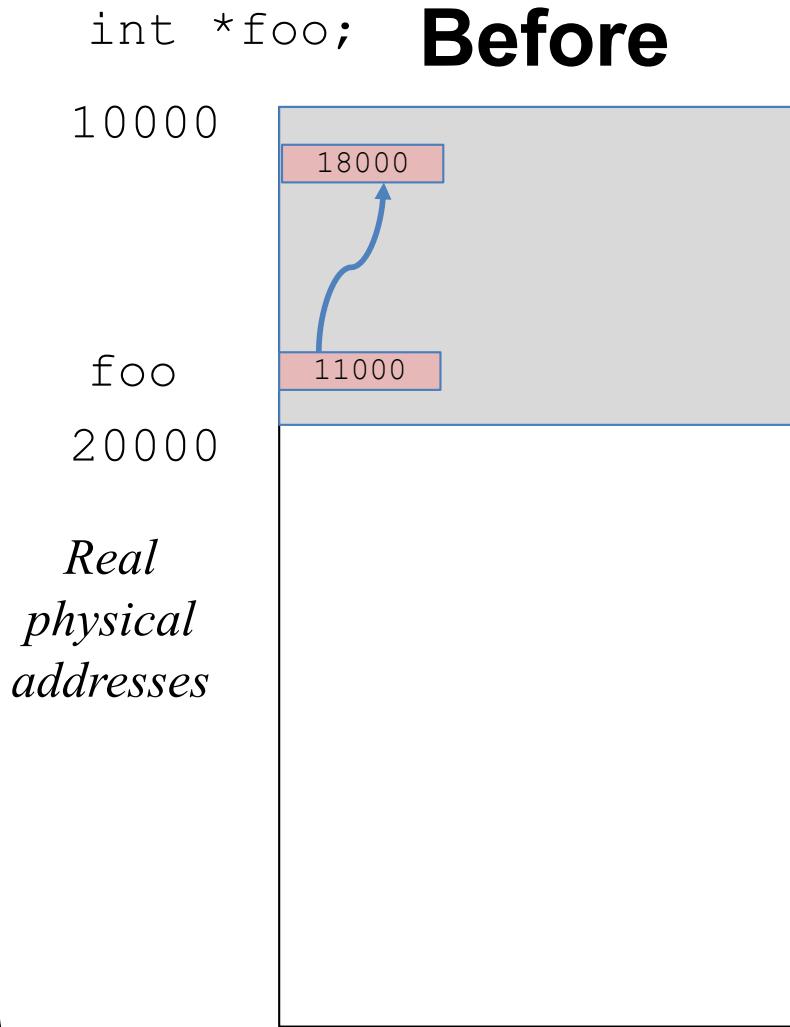


*An obvious
improvement!*

All This Requires Is Relocation . . .

- The ability to move a process' data
 - 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

Why Is Relocation Hard?



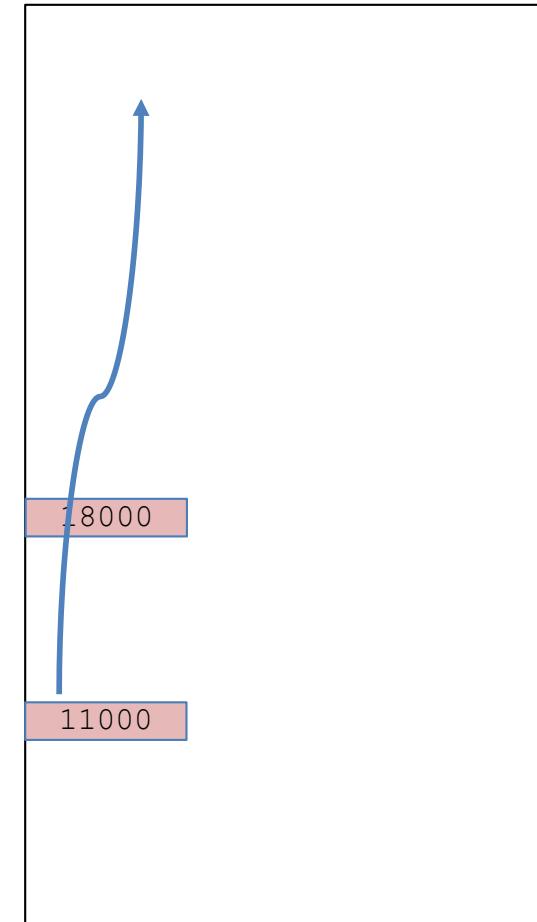
Let's move the partition!

What's going to happen the next time we access foo?

23000

foo
33000

After

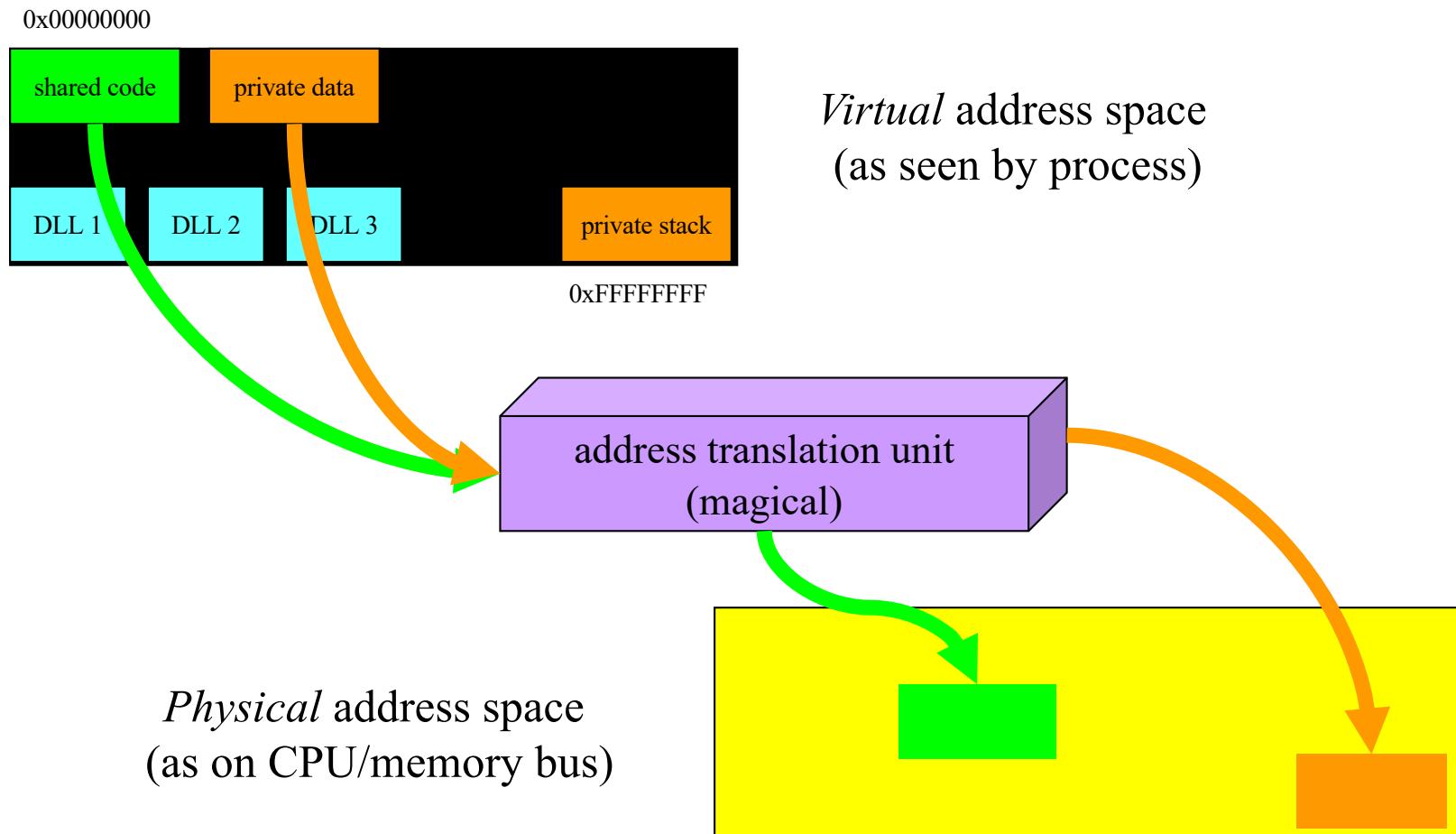


Of course, we copy the partition's contents when we move it

The Relocation Problem

- It is not generally feasible to 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?

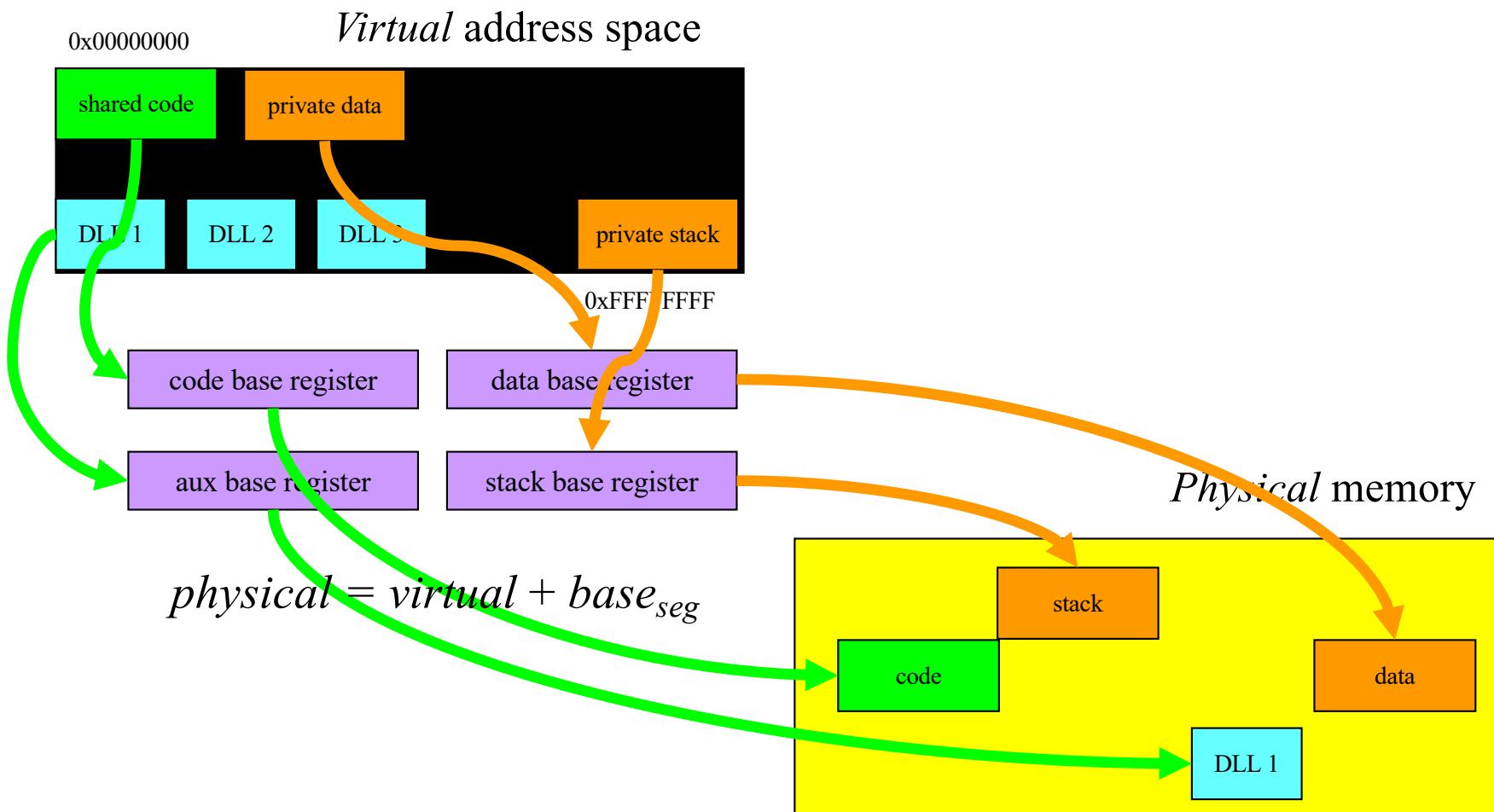
Virtual Address Spaces



Memory Segment Relocation

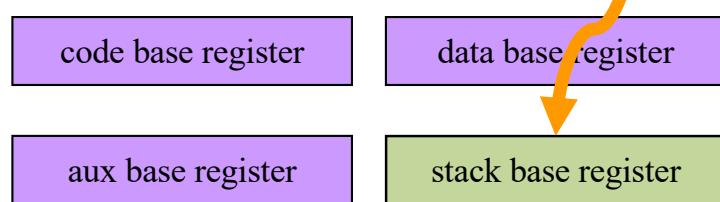
- A natural model
 - Process address space is made up of multiple segments
 - Use the segment as the unit of relocation
 - Long tradition, from the IBM system 360 to Intel x86 architecture
- Computer has special relocation registers
 - They are called *segment base registers*
 - They point to the start (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

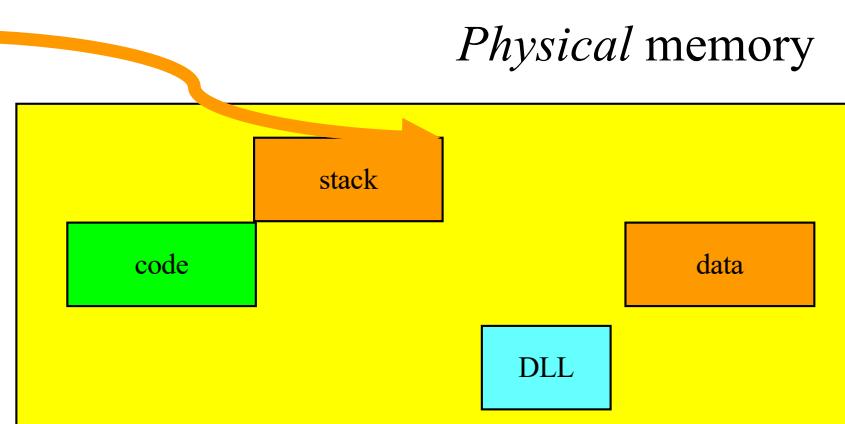
The virtual address of the stack doesn't change



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

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
 - 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

Memory Management – Swapping, Paging, and Virtual Memory

CS 111

Winter 2023

Operating System Principles

Peter Reiher

How Much of Our Problem Have We Solved?

- We can use variable sized partitions
 - Cutting down on internal fragmentation
- We can move partitions around
 - Which helps coalescing be more effective
 - But segments still need contiguous chunks of memory
 - So external fragmentation is still a problem
- We still can't support more process data needs than we have physical memory

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 (hard or flash)
- 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 or is blocked, 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

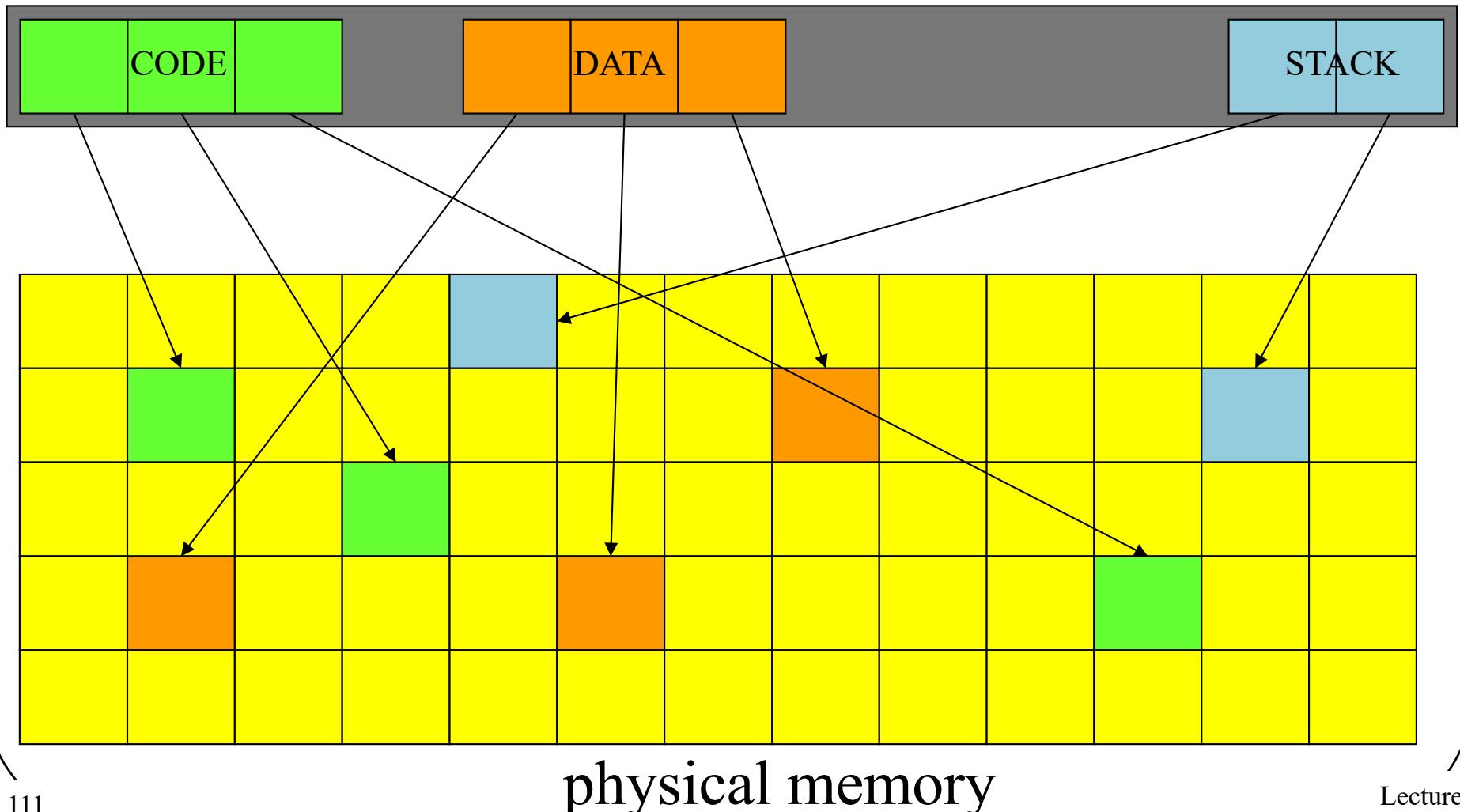
- Segment relocation solved the relocation problem for us
- It used base registers to compute a physical address from a virtual address
 - Allowing us to move data around in physical memory
 - By only updating the base register
- It did nothing about external fragmentation
 - Because segments are still 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
 - Call each virtual unit a *page*
- For each virtual address space page, store its data in one physical address page frame
 - Any page frame, not one specific to this page
- Use some magic per-page translation mechanism to convert virtual to physical pages

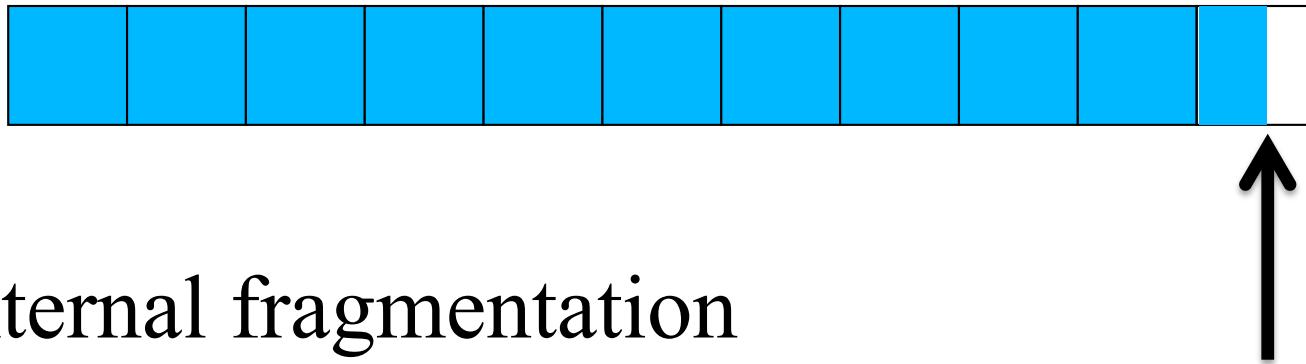
Paged Address Translation

process virtual address space



Paging and Fragmentation

- A segment is implemented as a set of virtual pages



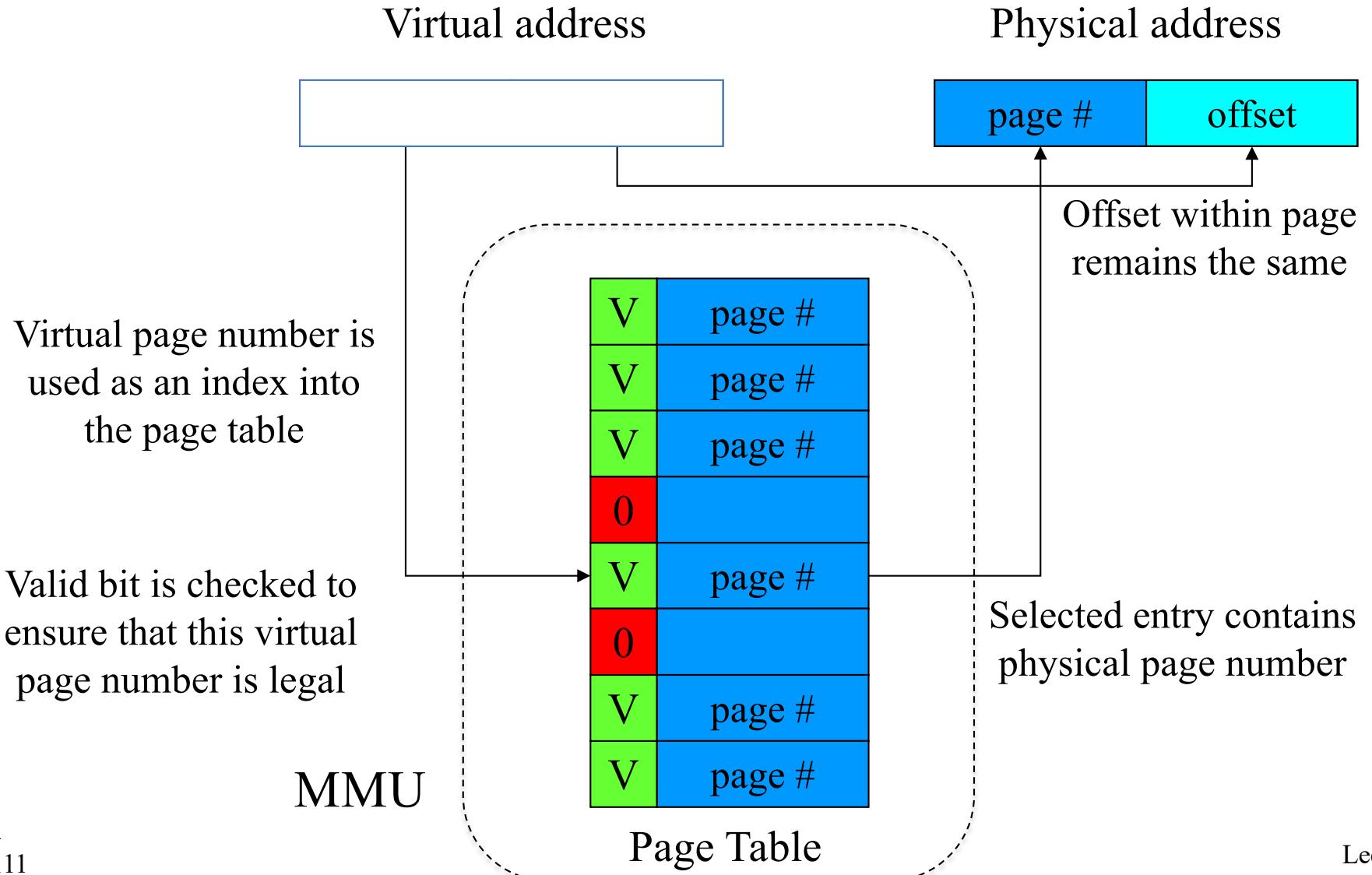
- Internal fragmentation
 - Averages only $\frac{1}{2}$ page (half of the last one)
- External fragmentation
 - Completely non-existent
 - We never carve up pages

Tremendous
reduction in
fragmentation costs!

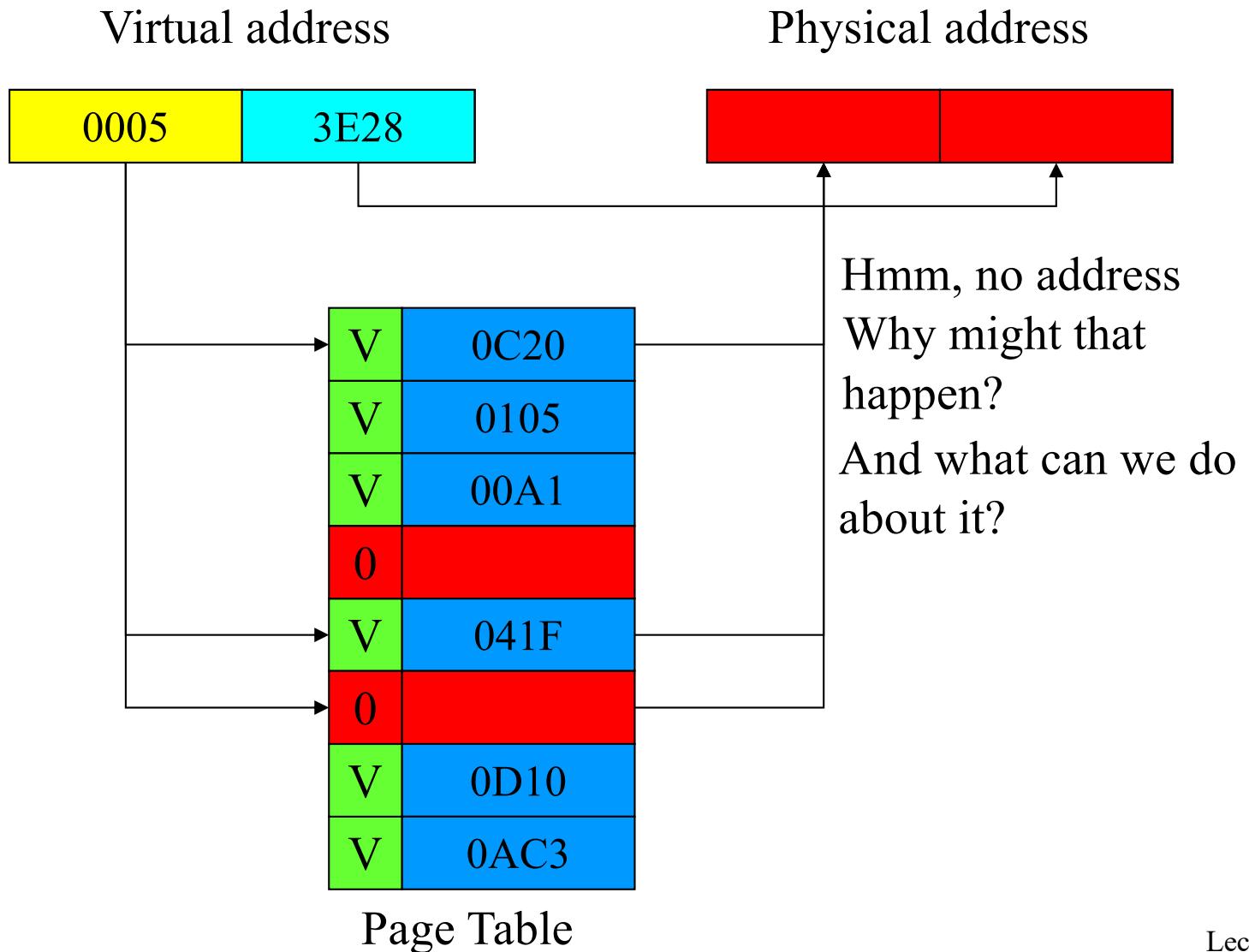
Providing the Magic Translation Mechanism

- On per page basis, we need to change a virtual address to a physical address
 - On every memory reference
- 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?

Where Do We Keep Extra Pages?

- We have more pages than RAM
- So some of them must be somewhere other than RAM
- Where else do we have the ability to store data?
- How about on our flash drive?
- In paged system, typically some pages are kept on persistent memory device
- But code can only access a page in RAM . . .

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

- If we can't keep all our pages in RAM, some are out on disk
- But we can't directly use them on disk
- So which ones do we put out on disk?
- And how do we get them back if it turns out we need to use them?
- Demand paging is the modern approach to that problem

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 the requested 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 swapping 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!

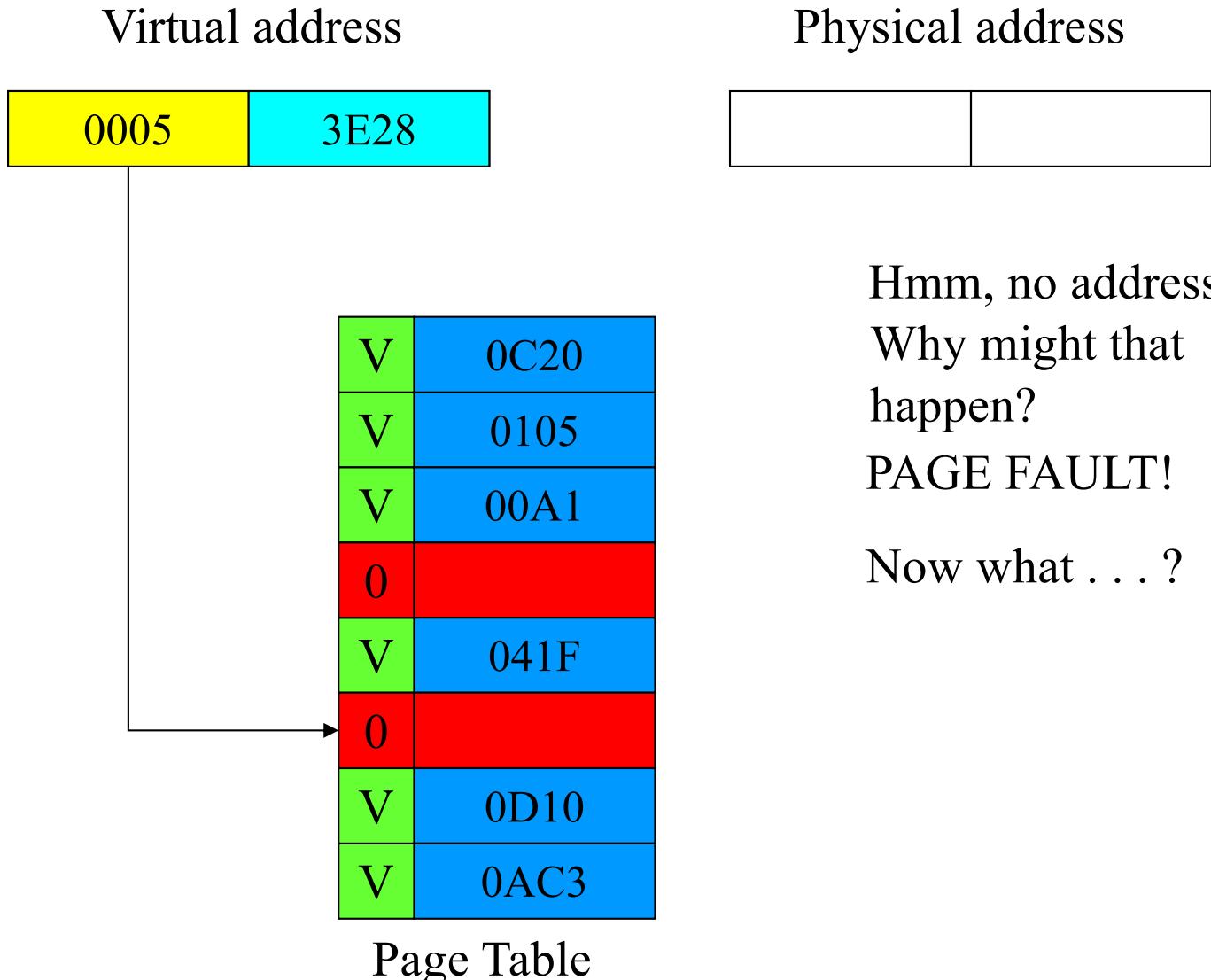
Why is Locality of Reference Usually Present?

- 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 pointers to pages of RAM
- In some cases, the pages are not in RAM, at the moment
 - They're out on disk (hard or flash)
- 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

A Page Fault Example



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 exception
 - 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

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 which 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 which 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 pages in memory
 - As many as will fit
 - Probably not all belonging to a single 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

Oracles are systems
that perfectly predict
the future.

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
 - How to actually implement LRU?

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

With 64 Gbytes of RAM and 4K pages, 16 megabytes of timestamps – sounds expensive.

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

Loads 4
Replacements 7

Maintaining Information for LRU

- Can we keep it in the MMU?
 - MMU would note the time whenever a page is referenced
 - But MMU translation must be blindingly fast
 - Getting/storing time on every fetch would be very expensive
 - At best MMU will maintain a *read* and a *written* bit per page
- Can we maintain this information in software?
 - Complicated and time consuming
 - If we maintain real timestamps, multiple overhead instructions for each real memory reference
- We need a cheap software surrogate for LRU
 - No extra instructions per memory reference
 - Mustn't cause extra page faults
 - Can't scan entire list each time on replacement, 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's reference bit is set
 - If so, clear 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

Remember guess
pointers from
Next Fit
algorithm?

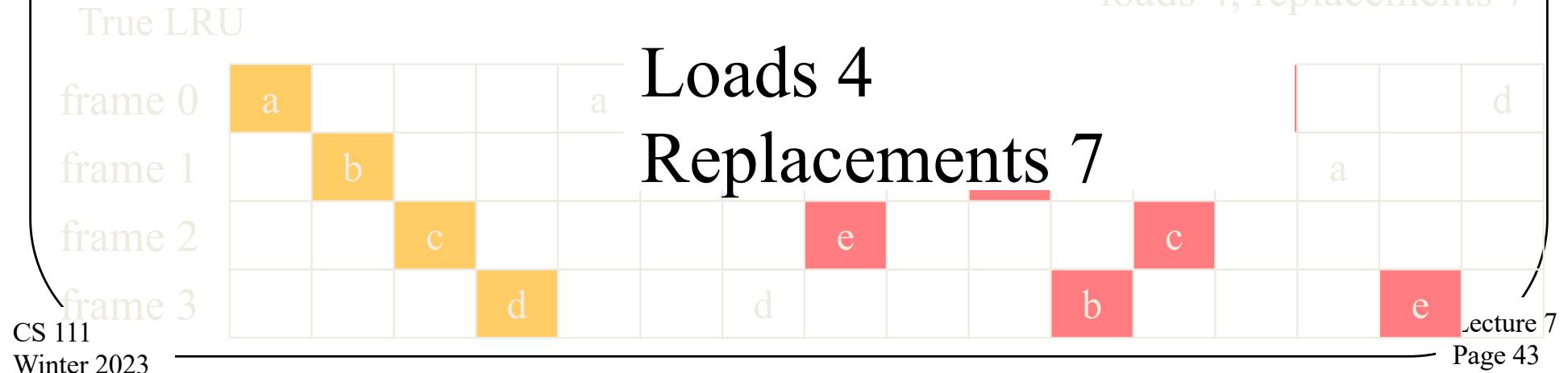
Clock Algorithm Page Replacement

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
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
 - When switched out process runs again, we don't want a bunch of page faults
- 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's scheduled, lots of page faults

Per-Process Page Frame Pools

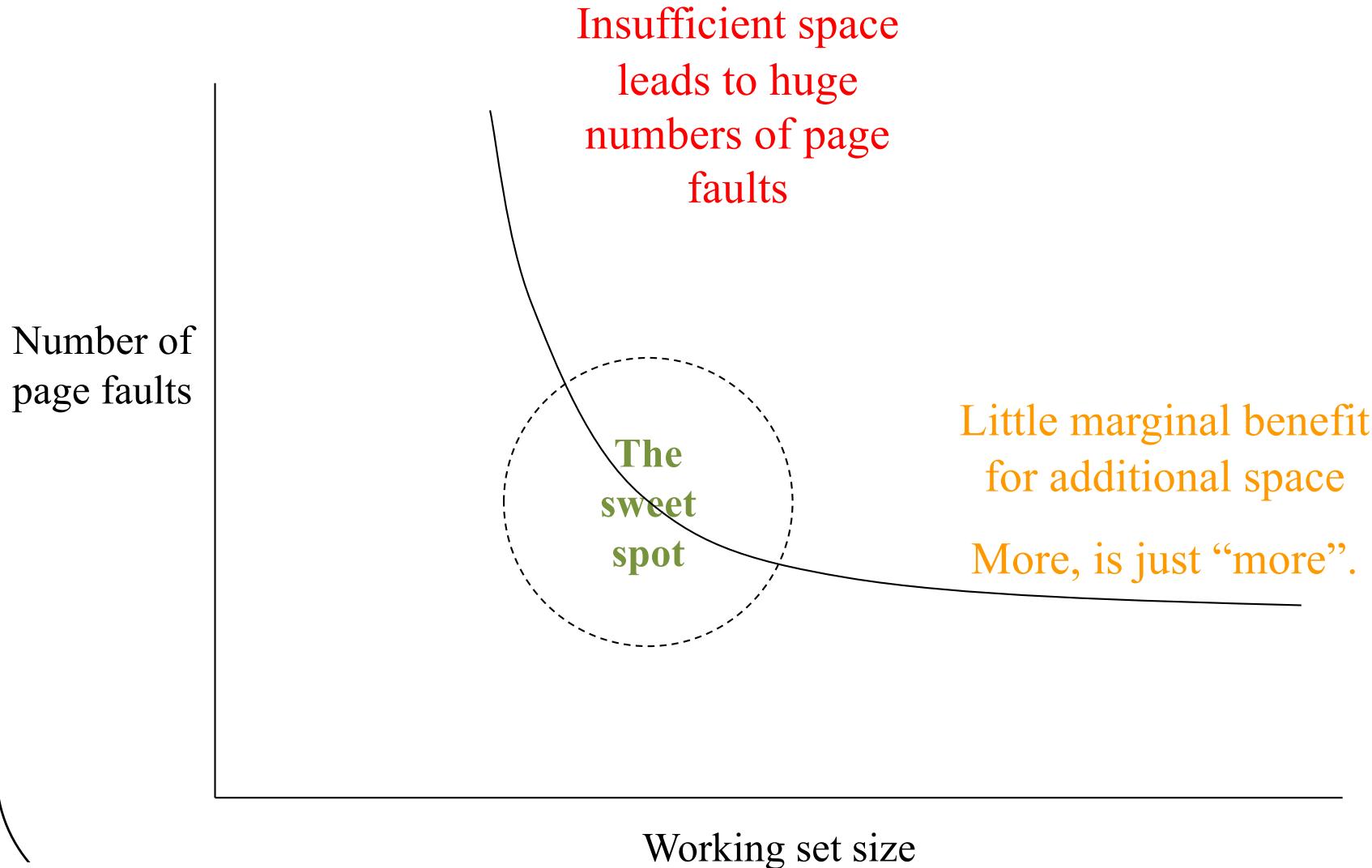
- Set aside some number of page frames for each running process
 - Use an LRU approximation separately for each
- How many page frames per process?
- Fixed number of pages per process is bad
 - Different processes exhibit different locality
 - Which pages are needed changes over time
 - Number of pages 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 we 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 to adjust working sets
 - E.g., Working Set-Clock
 - Track last use time for each page, for owning process
 - Find page (approximately) 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 page frames
 - 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 usually cannot add more memory
- 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

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 an identical 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 paged 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 manager 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

Conclusion

- Paging allows us to use RAM more efficiently
 - More processes runnable
- Virtual memory provides a good abstraction for programmers
 - But typically requires demand paging to work
- Key technology for VM is page replacement
- Working set approaches allow us to minimize page faults

Operating System Principles: Threads, IPC, and Synchronization

CS 111

Winter 2023

Operating System Principles
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 them 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
 - If threads require protection from each other, it's your problem

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

User Level Threads Vs. Kernel Threads

- Kernel threads:
 - An abstraction provided by the kernel
 - Still share one address space
 - But scheduled by the kernel
 - So multiple threads can use multiple cores at once
- User level threads:
 - Kernel knows nothing about them
 - Provided and managed via user-level library
 - Scheduled by library, not by kernel

By now you should
be able to deduce the
advantages and
disadvantages of each

Communications Between Processes

- 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
- These mechanisms are referred to as “inter-process communications”
 - 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

OS IPC Mechanics

- For local processes
- Data is in memory space of sender
- Data needs to get to memory space of receiver
- Two choices:
 1. The OS copies the data
 2. The OS uses VM techniques to switch ownership of memory to the receiver

Which To Choose?

- Copying the data
 - Conceptually simple
 - Less likely to lead to user/programmer confusion
 - Since each process has its own copy of the bits
 - Potentially high overhead
- Using VM
 - Much cheaper than copying the bits
 - Requires changing page tables
 - Only one of the two processes sees the data at a time

IPC: Synchronous and Asynchronous

- Synchronous IPC
 - Writes block until message is 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

Known by application, not by IPC mechanism

The IPC mechanism knows about these.

IPC and Flow Control

- Flow control: making sure a fast sender doesn't overwhelm a slow receiver
- Queued IPC consumes system resources
 - Buffered in the OS until the receiver asks for it
- 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 communication
 - Receiving side: stifle sender, flush old data
 - Handled by network protocols or OS mechanism
- Mechanisms for feedback to sender

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 receiver’s 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
- Shared memory
- There are others we won't discuss in detail
 - Mailboxes
 - Named pipes
 - Simple messages
 - IPC signals

Pipelines

- Data flows through a series of programs
 - ls | grep | sort | mail
 - Macro processor | compiler | 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 datagrams
 - 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*

Shared Memory

- OS arranges for processes to share read/write memory segments
 - Mapped into multiple processes' 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 themselves achieve whatever synchronization/consistency 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 code into simpler pieces
- Improved reliability
 - The failure of one component does not stop others
- A better fit to emerging paradigms
 - Client server computing, web based services
 - Our universe is cooperating parallel processes

Kill parallelism
and performance
goes back to the
1970s

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 makes process behavior less predictable
 - 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 too complicated
 - 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
- *Synchronization* refers to that control
- 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 interpreters
 - 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

Critical Section Example 1: Updating a File

Process 1

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

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

Process 2

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

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

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

Critical Section Example 2: Re-entrant Signals

First signal

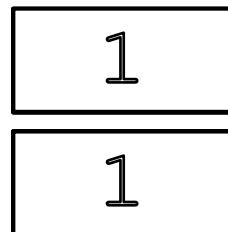
```
load r1,numsigs // = 0  
add r1,=1 // = 1  
store r1,numsigs // =1
```

```
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

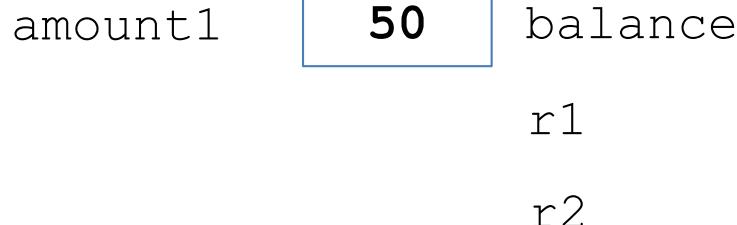
Thread 1

```
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
```

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

The \$25 debit was lost!!!

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 %eax, counter

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

If one of them is running it, the other definitely isn't!

One Solution: Interrupt Disables

- Temporarily block some or all interrupts
 - No interrupts -> nobody preempts my code in the middle
 - 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
 - Won't solve all sync problems on multi-core machines
 - Since they can have parallelism without interrupts

Downsides of Disabling Interrupts

- Not an option in user mode
 - Requires use of privileged instructions
 - Can be used in OS kernel code, though
- 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

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 these in more detail in the next class

Conclusion

- Processes are too expensive for some purposes
- Threads provide a cheaper alternative
- Threads can communicate through memory
- Processes need IPC
- Both processes and threads allow parallelism
 - Which is vital for performance
 - But raises correctness issues

Operating System Principles: Mutual Exclusion and Asynchronous Completion

CS 111

Winter 2023

Operating System Principles
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
 - If one thread is running the critical section, the other definitely isn't

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
 - Or multiple related data structures
- 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 or will be undone
 - An uncompleted update has no effect
- Correctness generally requires both



Vice versa is OK.

Options for Protecting Critical Sections

- Turn off interrupts
 - We covered that in the last lecture
 - Prevents concurrency, not usually possible
- Avoid shared data whenever possible
- Protect critical sections using hardware mutual exclusion
 - In particular, atomic CPU instructions
- Software locking

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
- Can we do entire critical section in one instruction?
 - With careful design, some data structures can be implemented this way

Usually not feasible

- Doesn't help with waiting synchronization

Locking

- Protect critical sections with a data 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

counter = counter + 1;

thread #2

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

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

Atomic Instructions – Test and Set

A C description of a machine language instruction **REAL Instructions are silicon, not C!!!**

```
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! */  
}
```

If rc was false,
nobody else ran
TS. We got the
lock!

If rc was true,
someone else already
ran TS. They got the
lock!

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                /* someone else changed *p */  
        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. */  
}
```

Using Atomic Instructions to Implement a Lock

- Assuming silicon implementation of test and set

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

Lock Enforcement

- Locking resources only works if either:
 - It's not possible to use a locked resource without the lock
 - Everyone who might use the resource carefully follows the rules
- Otherwise, a thread might use the resource when it doesn't hold the lock
- We'll return to practical options for enforcement later

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 yet again . . .

Spin Waiting



- Spin waiting for a lock is called *spin locking*
- 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
 - The cycles burned could be used by the locking party to finish its work
 - Bug may lead to infinite spin-waits

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
- Can we use spin locks for this synchronization?

Spinning Sometimes Makes Sense

1. When awaited operation proceeds in parallel
 - A hardware device accepts a command
 - Another core releases a briefly held spin lock
2. When awaited operation is guaranteed to happen 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

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
 - Potential unfairness

Fairness and Mutual Exclusion

- What if multiple processes/threads/machines need mutually exclusive access to a resource?
- Locking can provide that
- But can we make guarantees about fairness?
- Such as:
 - Anyone who wants the resource gets it sooner or later (no starvation)
 - Perhaps ensuring FIFO treatment
 - Or enforcing some other scheduling discipline

How Can We Wait?

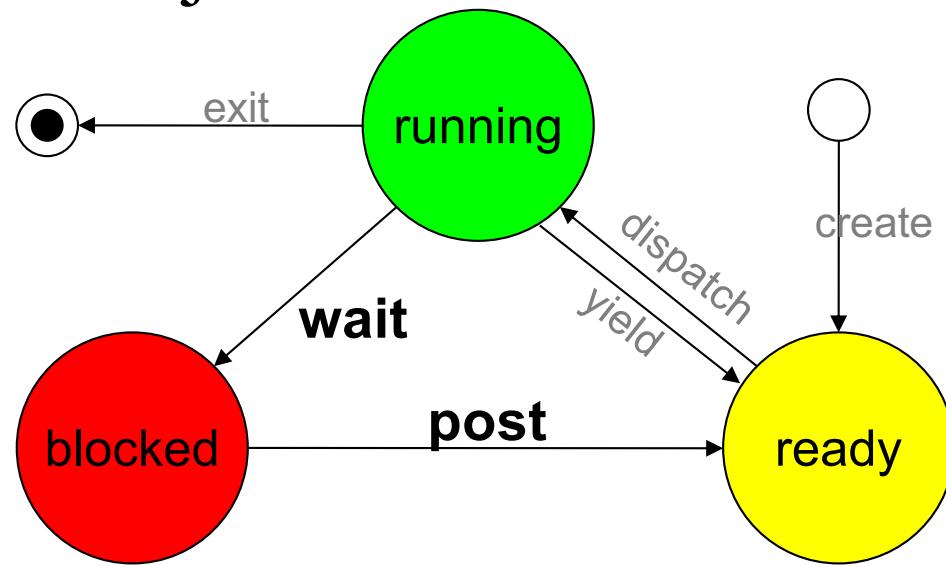
- Spin locking/busy waiting
- Yield and spin ...
- Either spin option may still require mutual exclusion
 - And any time spent spinning is wasted
- And fairness may be an issue
- *Completion events*

Completion Events

- If you can't get the lock, block
- Ask the OS to wake you when the lock is available
- Similarly for anything else you need to wait for
 - Such as I/O completion
 - Or another process to finish its work
- Implemented with *condition variables*

Condition Variables

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



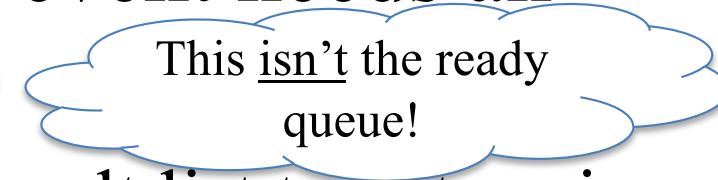
Condition Variables and the OS

- Generally the OS provides condition variables
 - Or library code that implements threads does
- Block a process or thread when a 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

Handling Multiple Waits

- Threads will wait on several different things
- Pointless to wake up everyone on every event
 - Each should wake up only when his event happens
- So OS (or thread package) should allow easy selection of “the right one”
 - When some particular event occurs
- But several threads could be waiting for the same thing . . .

Waiting Lists

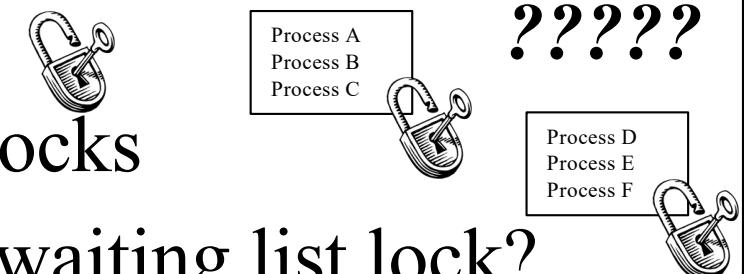
- Suggests each completion event needs an associated waiting list 
 - When posting an event, consult list to determine who's waiting for that event
 - Then what?
 - Wake up everyone on that event's waiting list?
 - One-at-a-time in FIFO order?
 - One-at-a-time in priority order (possible starvation)?
 - 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

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?
 - Where does it end?



A Real Problem For Waiting Lists

- 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 has locked a resource and thread A needs to get that lock
- So thread A will call `sleep()` to wait for the lock to be free
- Meanwhile, thread B finishes using the resource
 - So thread B will call `wakeup()` to release the lock
- No other threads are waiting for the resource

The Race At Work

Thread A Thread B

```
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

```
The event hasn't happened yet!  
void wakeup( eventp *e ) {  
    struct proce *p; Now it happens!  
  
    e->posted = TRUE;  
    p = get_from_queue( &e-> queue );  
    if (p) {  
        } /* if !p, nobody's waiting */  
    }
```

But there's no one to
wake him up

The effect?

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 and block° ° °
 - 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
 - We just need a lock°°°
- Think about why these actions are part of the critical section.
- SEE IF YOU CAN FIGURE THAT OUT FOR YOURSELF!
- But how will we handle contention for that lock?

Conclusion

- Two classes of synchronization problems:
 1. Mutual exclusion
 - Only allow one of several activities to happen at once
 2. Asynchronous completion
 - Properly synchronize cooperating events
- Locks are one way to assure mutual exclusion
- Spinning and completion events are ways to handle asynchronous completions

Operating System Principles: Semaphores and Other Synchronization Primitives

CS 111

Winter 2023

Operating System Principles
Peter Reiher

Outline

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

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

Computational Semaphores

- 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 queue non-empty, wake one of the waiting 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 wait will succeed
 - Subsequent waits will block
- Use V/post operation to release the lock
 - Increment semaphore count to indicate one less waiting request
 - 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 ...
 - The number 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;  
    wait( &a->semaphore );      /* get exclusive access to the account */  
  
    if ( a->balance >= amount ) { /* check for adequate funds */  
        amount -= balance;  
        ret = amount;  
    } else {  
        ret = -1;  
    }  
  
    post( &a->semaphore );      /* release access to the account */  
    return( ret );  
}
```

Limitations of Semaphores

- Semaphores are a very basic 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 (e.g., files) 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*
 - Lock passes with the relevant fd
 - Distinct opens are not affected
- Locking with flock() 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 and flocks() 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)
 - Process specific
 - Closing any fd for the file releases for all of a process' fds for that file
- Locking may be enforced
 - Depending on the underlying file system

Locking Problems

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

Performance of Locking

- Locking often 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?

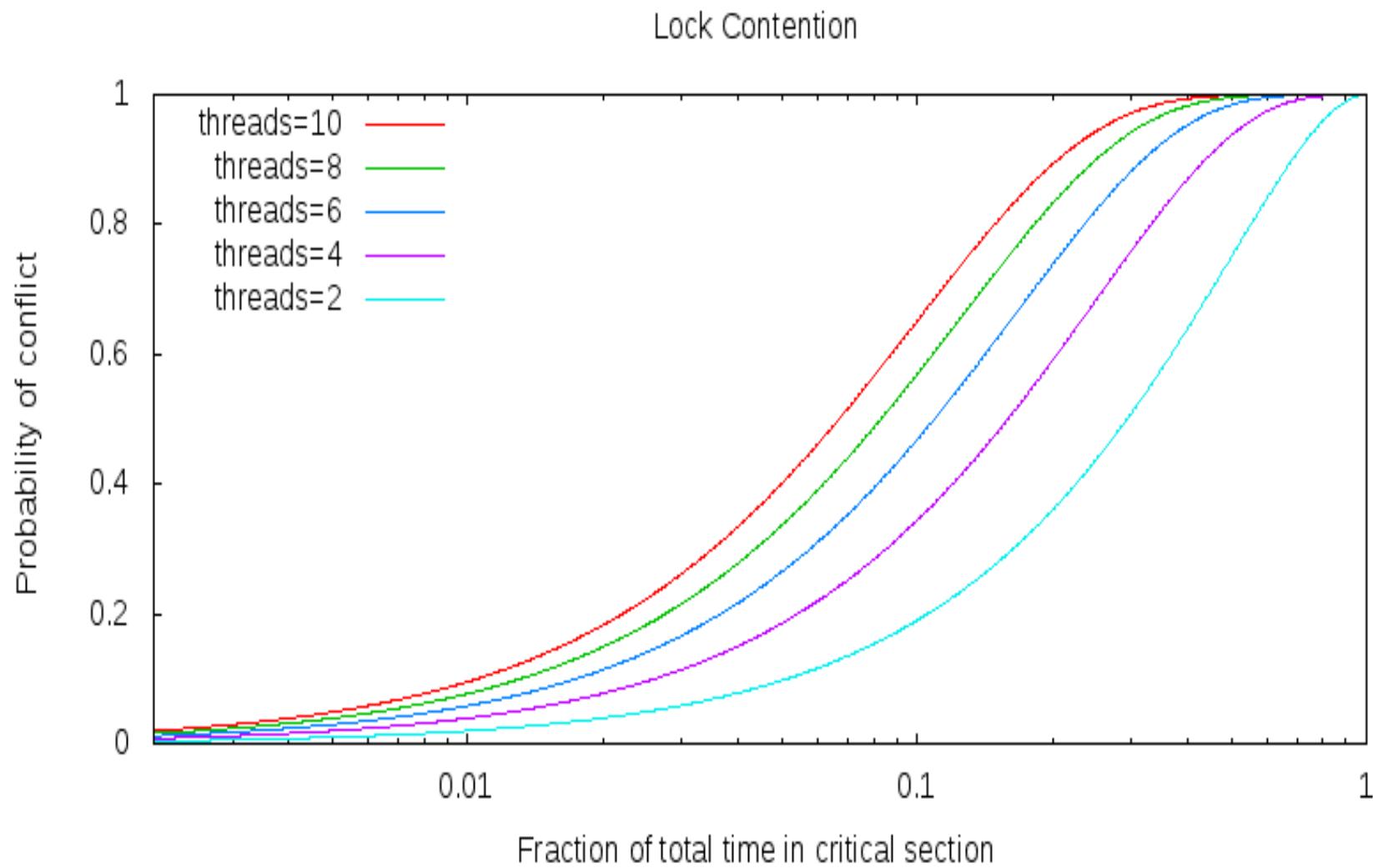
- Then you block
- Blocking is much more expensive than getting a lock
 - E.g., 1000x
 - Micro-seconds to yield and context switch
 - Milliseconds if swapped-out or a queue forms
- Performance depends on conflict probability

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

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

$$P_{\text{conflict}} = 1 - \underbrace{(1 - (T_{\text{critical}} / T_{\text{total}}))}_{\text{Nobody else in critical section at the same time}}^{\text{threads}}$$

Nobody else in critical section at the same time

- Unless a FIFO queue forms

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

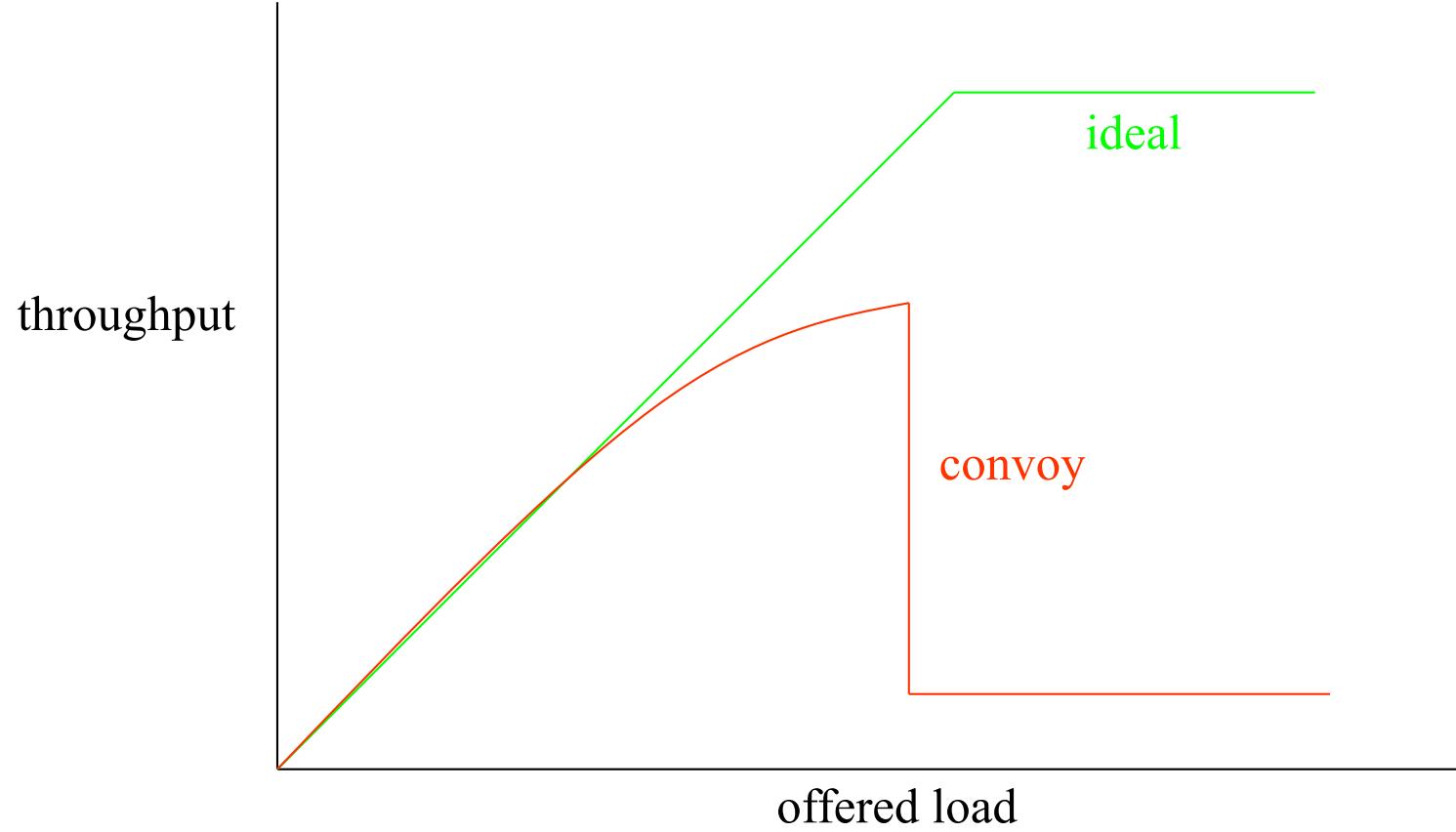
Newcomers have to get into line

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

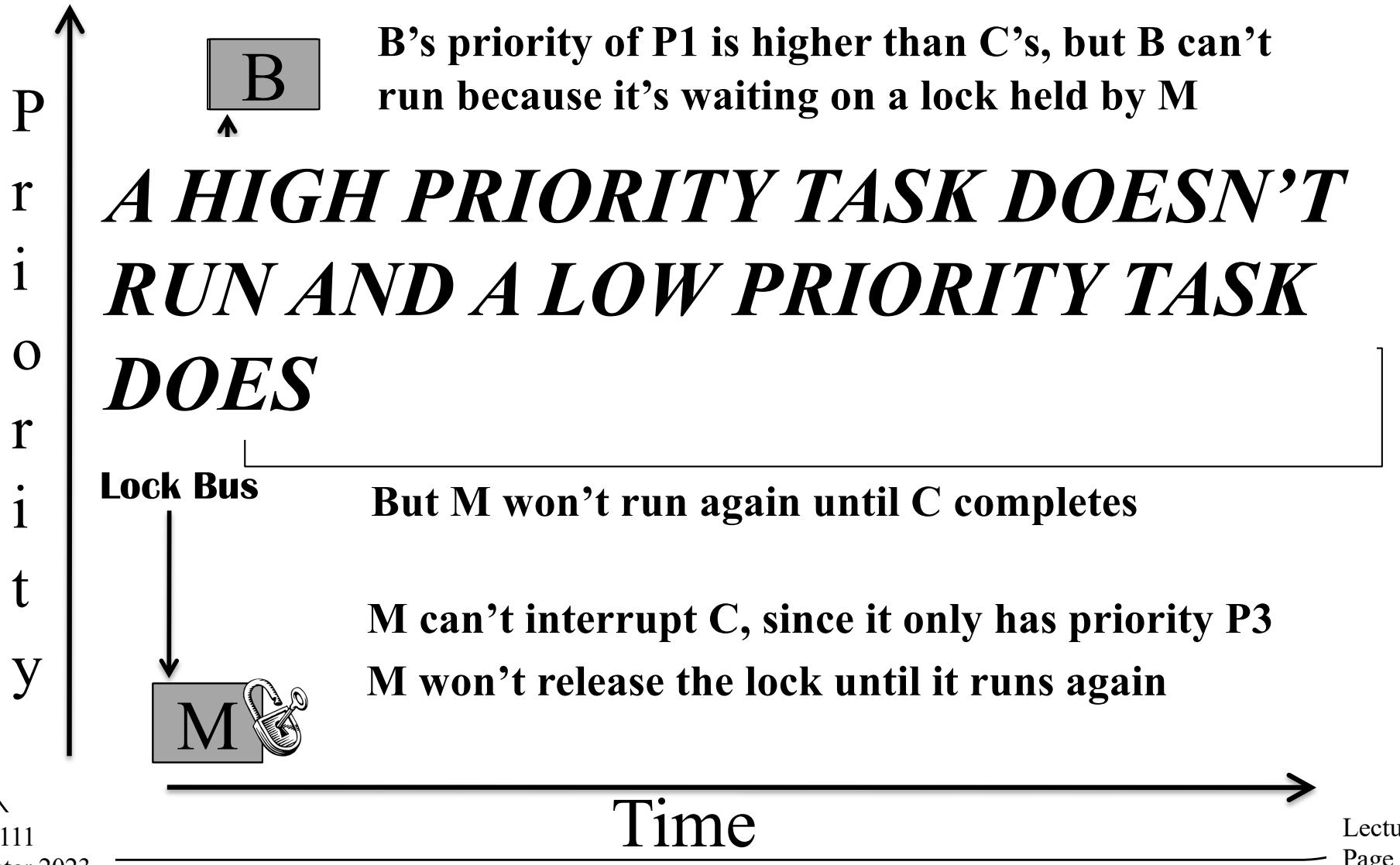
A Tale of Three Tasks

- 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



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

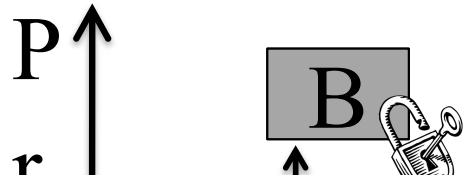
Handling Priority Inversion Problems

- 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

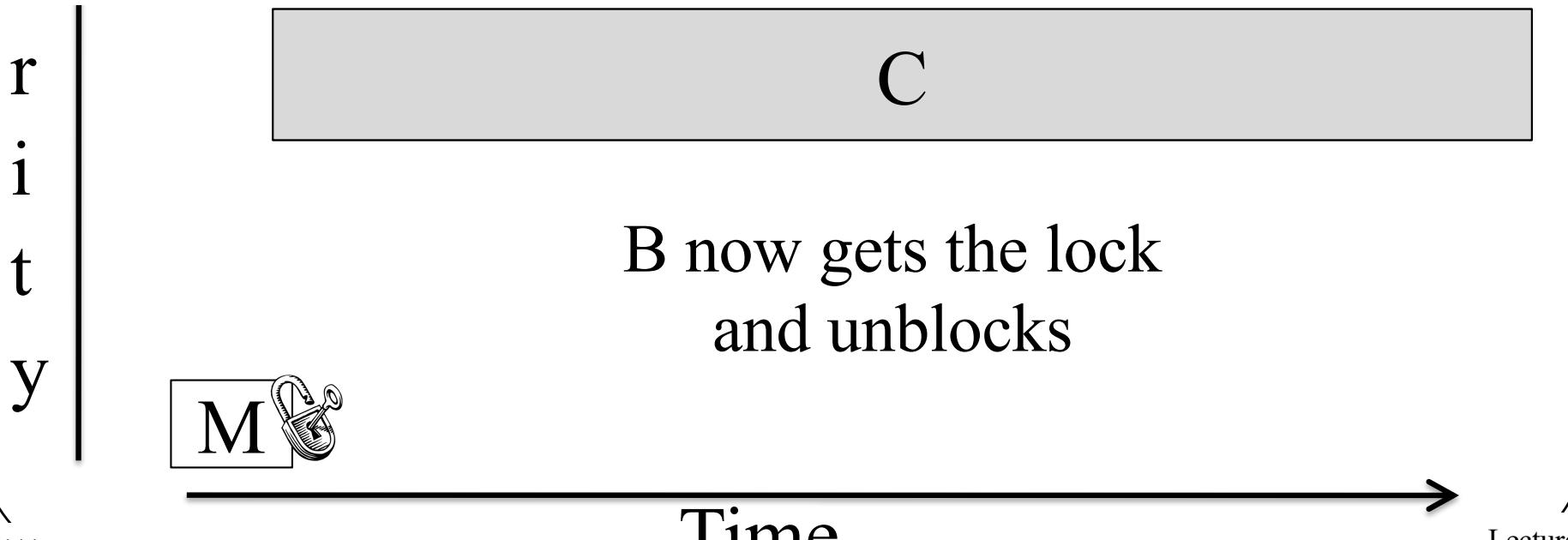
- Temporarily increase the priority of the meteorological task
 - While the high priority bus management task was blocked by it
 - So the communications task wouldn't preempt it
 - When lock is released, drop meteorological task's priority back to normal
- *Priority inheritance*: a general solution to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



Solving Locking Problems

- Reducing overhead
- Reducing contention

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

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

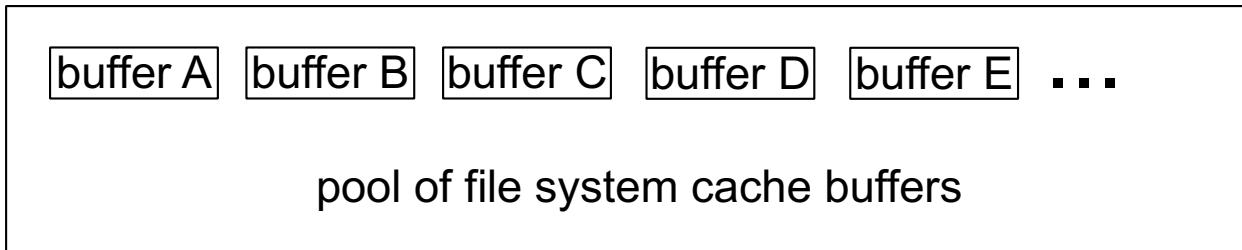
- Read/write locks
- Reads and writes are not equally common
 - File reads and writes: reads/writes > 50
 - Directory search/create: reads/writes > 1000
- Only writers require exclusive access
- Read/write locks
 - Allow many readers to share a resource
 - Only enforce exclusivity when a writer is active
 - 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 buffer AA to the cache
 - Thread 1 looks for buffer B while thread 2 is deleting it
- The pool lock could become a bottle-neck, so
 - Minimize its use
 - Reader/writer locking
 - Sub-pools ...

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 go wrong
- If we are not careful locking can lead to ou
- Deadlock



Operating System Principles: Deadlock – Problems and Solutions

CS 111

Winter 2023

Operating System Principles
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 at a table
Five plates of pasta
Five forks

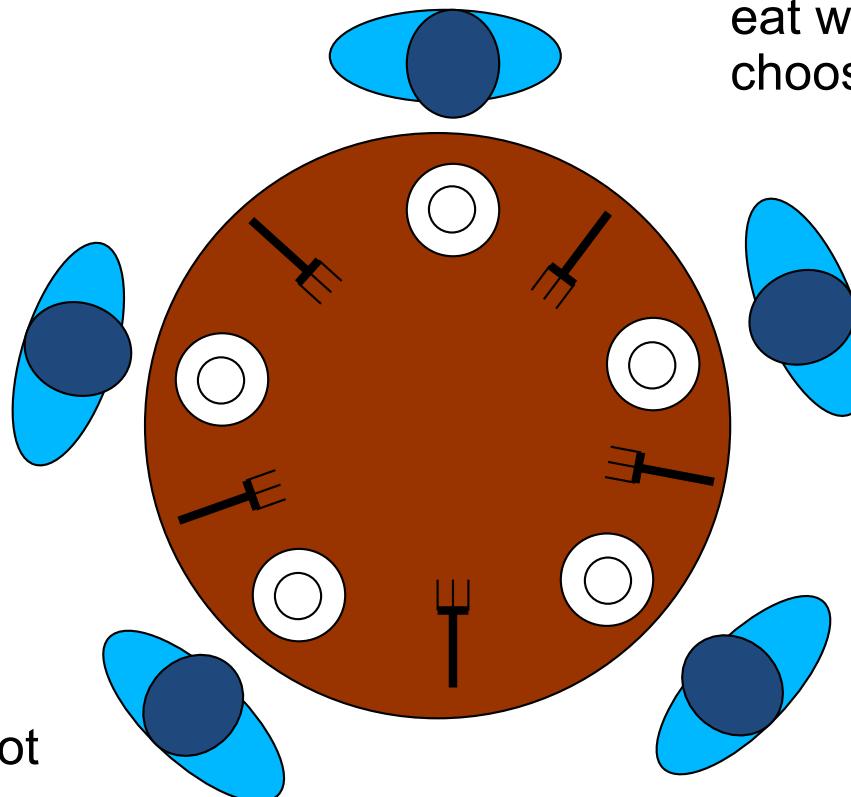
A philosopher needs two forks to eat pasta, but must pick them up one at a time

Philosophers will not negotiate with one another

Philosophers try to eat whenever they choose to

Ensure that philosophers will not deadlock while trying to eat

The problem demands an absolute solution



Dining Philosophers and Deadlock

- This problem is the classic 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 eliminates 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

One Possible Dining Philosophers Deadlock

All five philosophers try to eat at the same time

Each grabs one fork

Result?

- No philosopher has two forks
- So no philosopher can eat

- But no philosopher will release a fork before he eats



Deadlock!

- So no philosopher will ever eat again

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 are not the only synchronization problem . . .

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 anything 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!

Thread 1

Thread 2

Thread 1
acquires a
lock for
Critical
Section A

Thread 1
requests a
lock for
Critical
Section B

Deadlock!

Critical
Section
A



Critical
Section
B



Thread 2
acquires a
lock for
Critical
Section B

Thread 2
requests a
lock for
Critical
Section A

Deadlock Avoidance

- Use methods that guarantee that no deadlock can occur, by their nature
- Advance reservations
 - The problems of under/over-booking
 - Dealing with rejection

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 until 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

How does the OS handle overbooking memory?

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
 - One hopes before the new computation has begun
 - Failures will not happen at request time
 - System behavior is 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 to not 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

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 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

Note: deadlock solutions solve deadlocks – they don't necessarily solve all your other problems!

They may even create new ones

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

You solved
your deadlock,
but you broke
your resource.

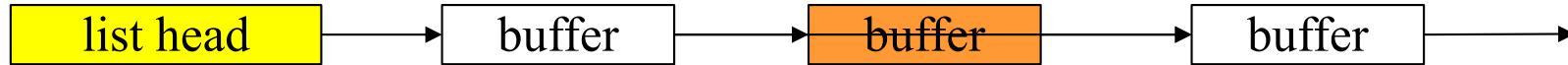
When Can The OS “Seize” a Resource?

- 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 stop honoring requests
- When can’t the OS 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:

unlock buffer

write lock list head

search for desired buffer

lock desired buffer

remove from list

unlock list head

Because we can't lock the list head while we hold the buffer lock

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

- To detect all deadlocks, need to identify all resources that can be locked
 - Not always clear in an OS
 - Especially if some locks are application level
- Must maintain wait-for graph or equivalent structure
- When lock requested, structure is updated and checked for deadlock
 - Better to just to reject the lock request?
 - And not let the requester block?

Deadlocks Outside the OS

- Some applications use locking internally
 - Not as an OS feature
 - But built into their own code
- Database systems are a main example
 - They often allow locking of records
 - And often enforce that locking
- The OS knows nothing of those locks
 - And thus offers no help in handling those deadlocks
- Deadlock detection may make sense here
 - Since the database knows of all relevant locks

Deadlocks here typically handled by rolling back one of the deadlocked transactions.

Not All Synchronization Bugs Are Deadlocks

- There are lots of reasons systems hang and make no progress
 - Sometimes it really is a deadlock
 - Sometimes it's something else
 - Livelock
 - Flaws in lock implementation
 - Simple bugs in how code operates
 - If there are no locks, it's not a deadlock
 - Even if there are locks, it might not be
- Of course, just finding out your problem isn't a deadlock doesn't necessarily help solve it*
- An approach that handles the whole range of synchronization problems would be helpful*

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 That 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
 - E.g., the sleep/wakeup race we talked about earlier
- Priority inversion hangs
 - Like the Mars Pathfinder case
- None of these is a true deadlock
 - Wouldn't be found by a deadlock detection algorithm
 - But 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 object has a semaphore
 - Automatically acquired on any method invocation
 - Automatically released on method return
- 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 {  
    // object 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 object on any method
 - 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

Java Synchronized Methods

- Each object has an associated mutex
 - Only acquired for specified methods
 - Not all object methods need be synchronized
 - 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;  
    // object is not locked when this method is invoked  
    public int balance() {  
        return(balance);  
    }  
    // object is locked when this method is invoked  
    public synchronized int debit(int amount) {  
        balance -= amount;  
        return( balance)  
    }  
}
```

Conclusion

- Parallelism is necessary in modern computers to achieve high speeds
- Parallelism brings with it many chances for serious errors
 - Generally non-deterministic errors
 - Deadlock is just one of them
- Those working with parallel code need to understand synchronization
 - Its problems and the solutions to those problems
 - And the costs associated with the solutions

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

CS 111

Winter 2023

Operating System Principles
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

Devices and Performance

- Most devices are very slow
 - Compared to CPU, bus, RAM
 - Sometimes orders of magnitude slower
- Leads to challenges in managing devices
 - Primarily performance challenges
 - System must operate at CPU speeds, not device speeds
 - But often correct application behavior requires device interactions
 - System code must handle the mismatch

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 flash 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 message, 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
 - A 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
 - Flash drive, display, printer, 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
 - Plug in the ones a given machine needs
- Making it easy to change or augment later

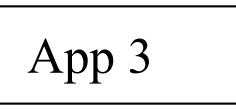
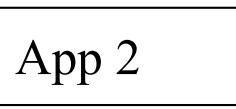
Layering Device Drivers

- 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

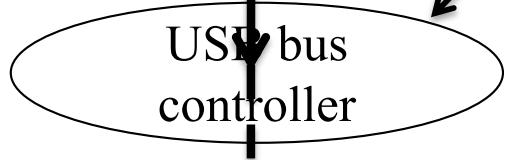
User space

System Call

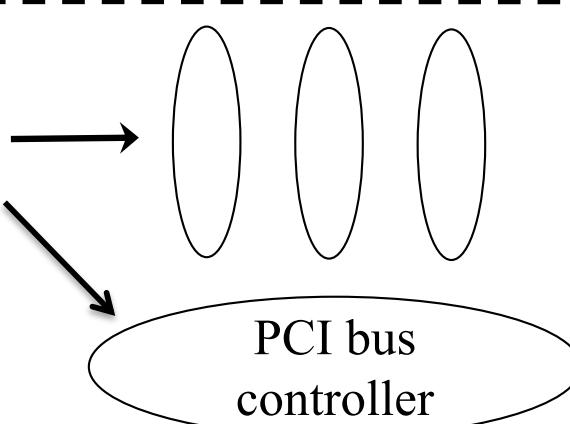


Kernel space

Device Call

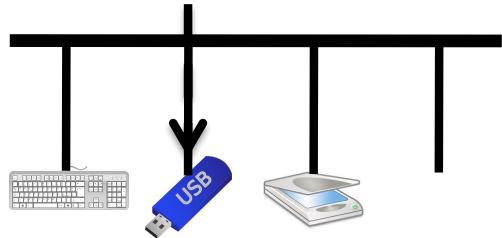


Device Drivers

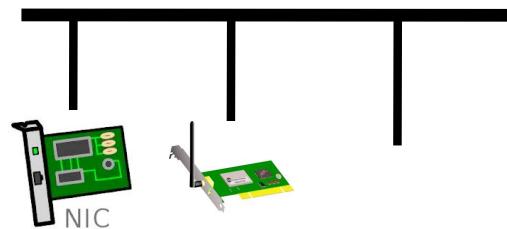


Hardware

USB bus



PCI bus



Device Drivers Vs. Core OS Code

- Device driver code can be 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
- Devices work at different speed than the CPU
 - Typically slower
 - Often much 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

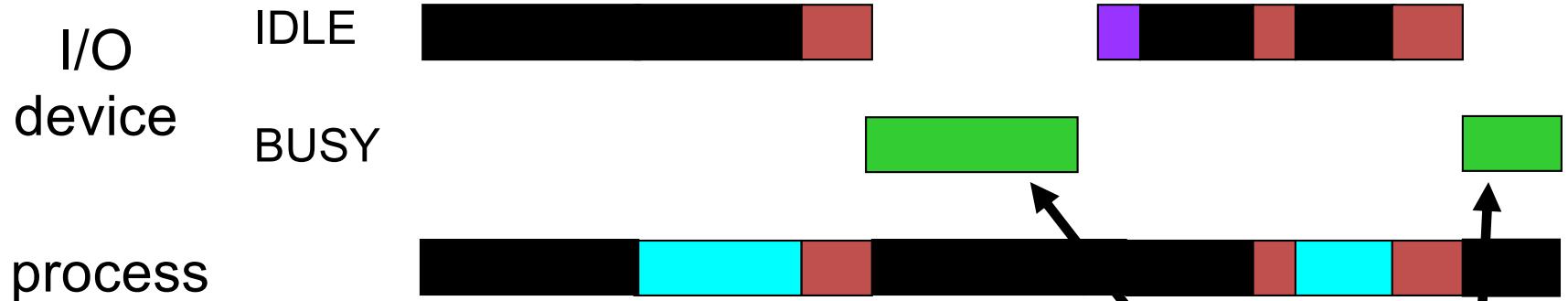
Device Performance

- 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
 - These devices are much slower than CPU
- 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
 - But CPU must not be held up waiting for devices
 - 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
- The only times the device is doing work!

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 it does, life will be slow, anyway
 - Since RAM is much slower than the CPU
- So one way to parallelize activities is to let a device use the bus instead of the CPU

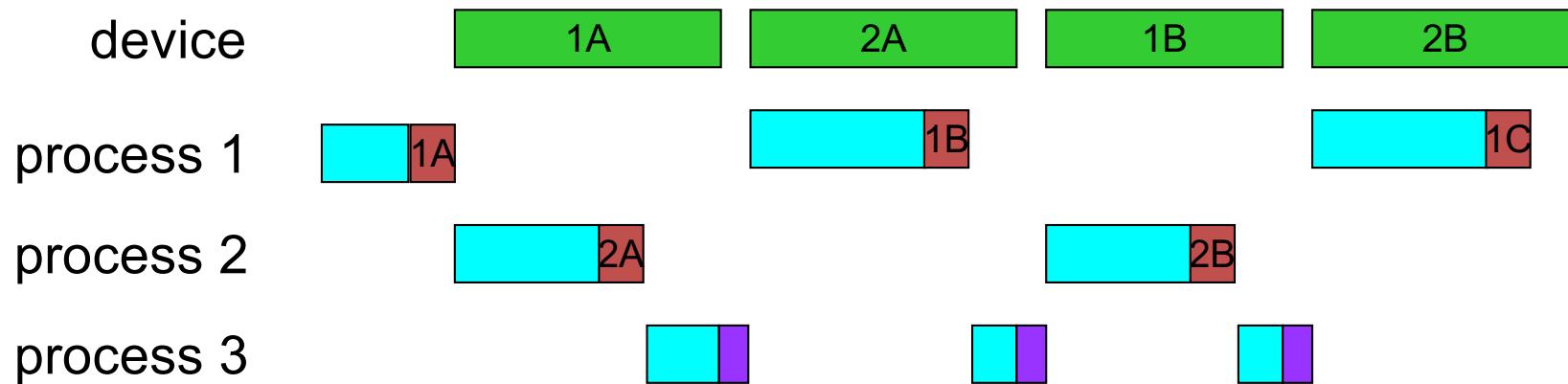
Direct Memory Access (DMA)

- 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

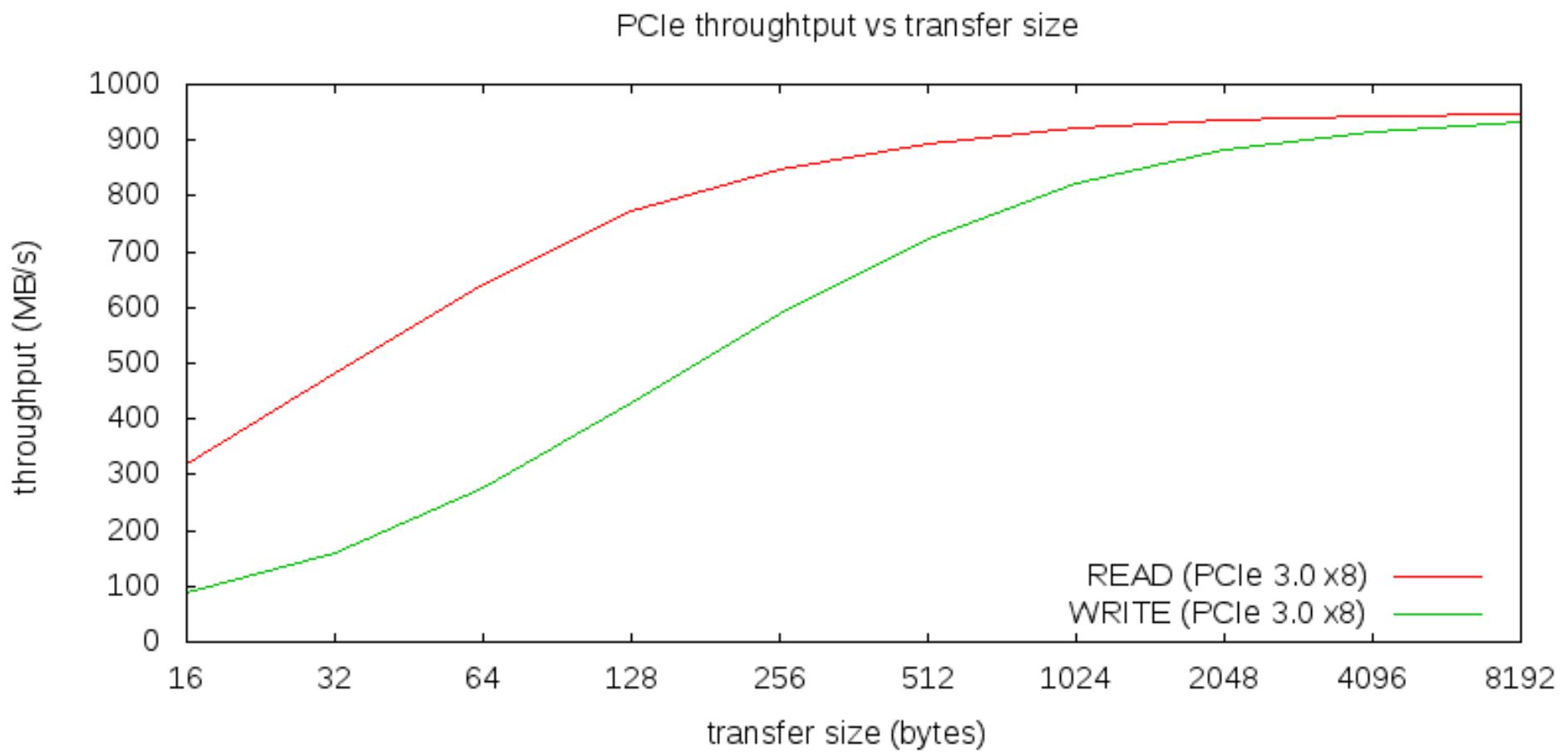
- Allow multiple requests to be pending at a time
 - Queue them, just like processes in the ready queue
 - Requesters block to await eventual completions
- Use DMA to perform the actual data transfers
 - Data transferred, with no delay, at device speed
 - Minimal overhead imposed on CPU
- When the currently 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

Multi-Tasking & Interrupt Driven I/O



1. P_1 runs, requests a read, and blocks
2. P_2 runs, requests a read, and blocks
3. P_3 runs until interrupted
4. Awaken P_1 and start next read operation
5. P_1 runs, requests a read, and blocks
6. P_3 runs until interrupted
7. Awaken P_2 and start next read operation
8. P_2 runs, requests a read, and blocks
9. P_3 runs until interrupted
10. Awaken P_1 and start next read operation
11. P_1 runs, requests a read, and blocks

Bigger Transfers are Better



Why Are Bigger Transfers Better?

- All transfers have per-operation overhead
 - DMA-related, device-related, OS-related
 - 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

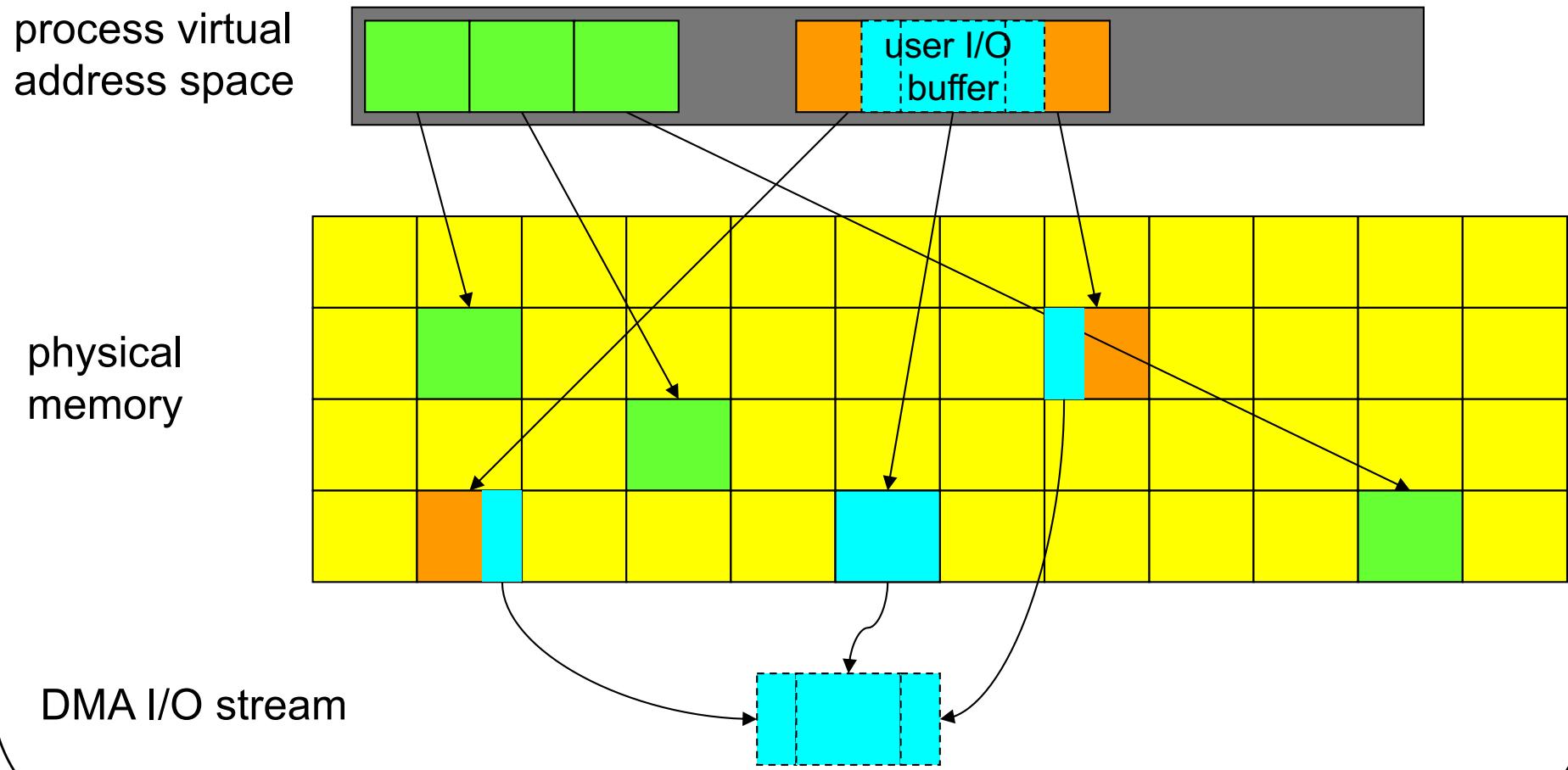
Deep Request Queues

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

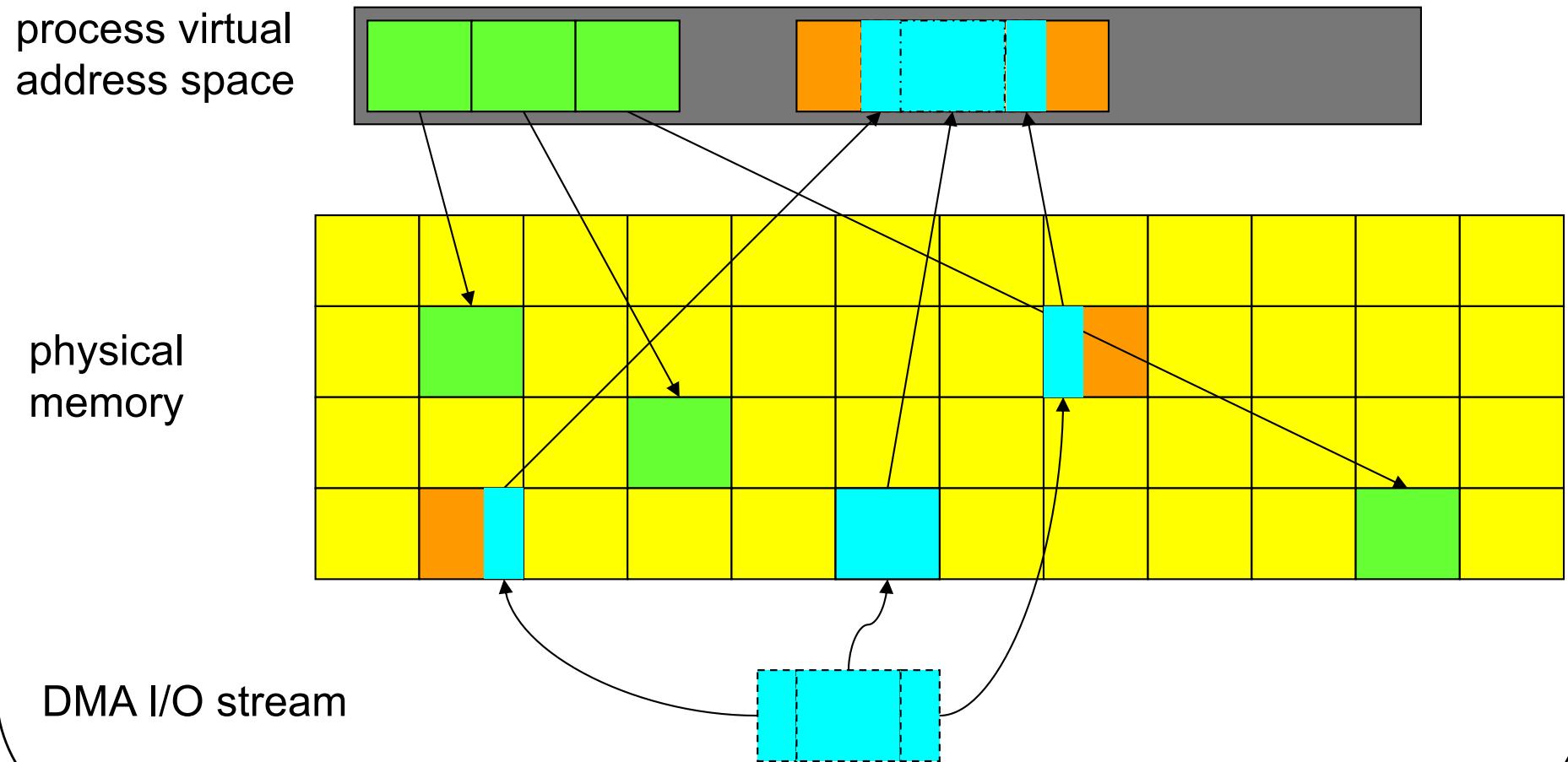
Scatter/Gather I/O

- Many device controllers support DMA transfers
 - Entire transfer must be contiguous in physical memory
- User buffers are in paged virtual memory
 - So buffers may be spread all over physical memory
 - *Scatter*: read from device to multiple page frames
 - *Gather*: writing from multiple page frames to device
- Three basic approaches apply:
 1. Copy all user data into physically contiguous 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 always 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 flash drives, all network cards, all graphics cards, etc.
- Can we simplify the OS by leveraging these commonalities?
- By defining simplifying abstractions?

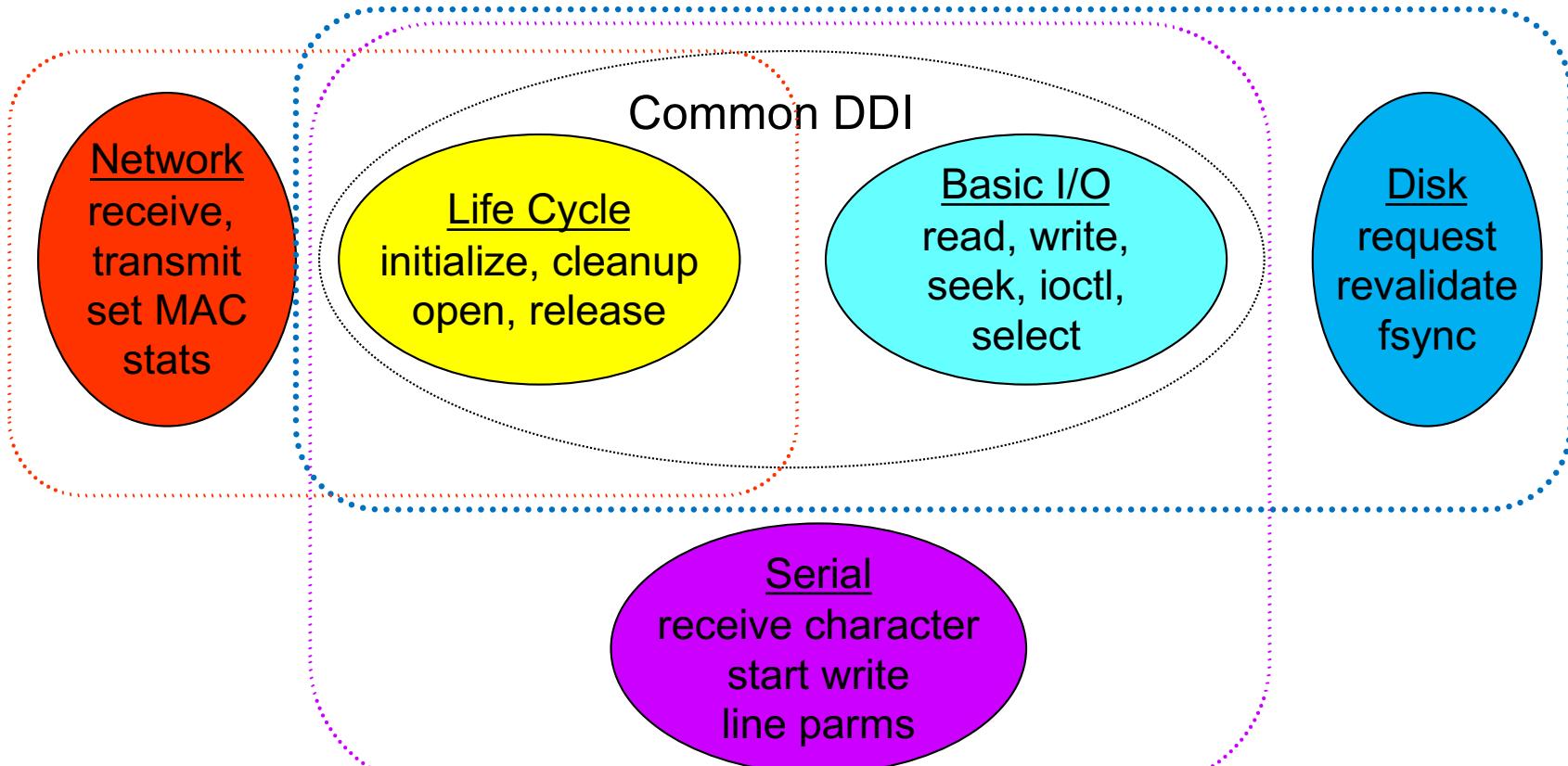
Providing the Abstractions

- The OS defines idealized device classes
 - Flash, 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

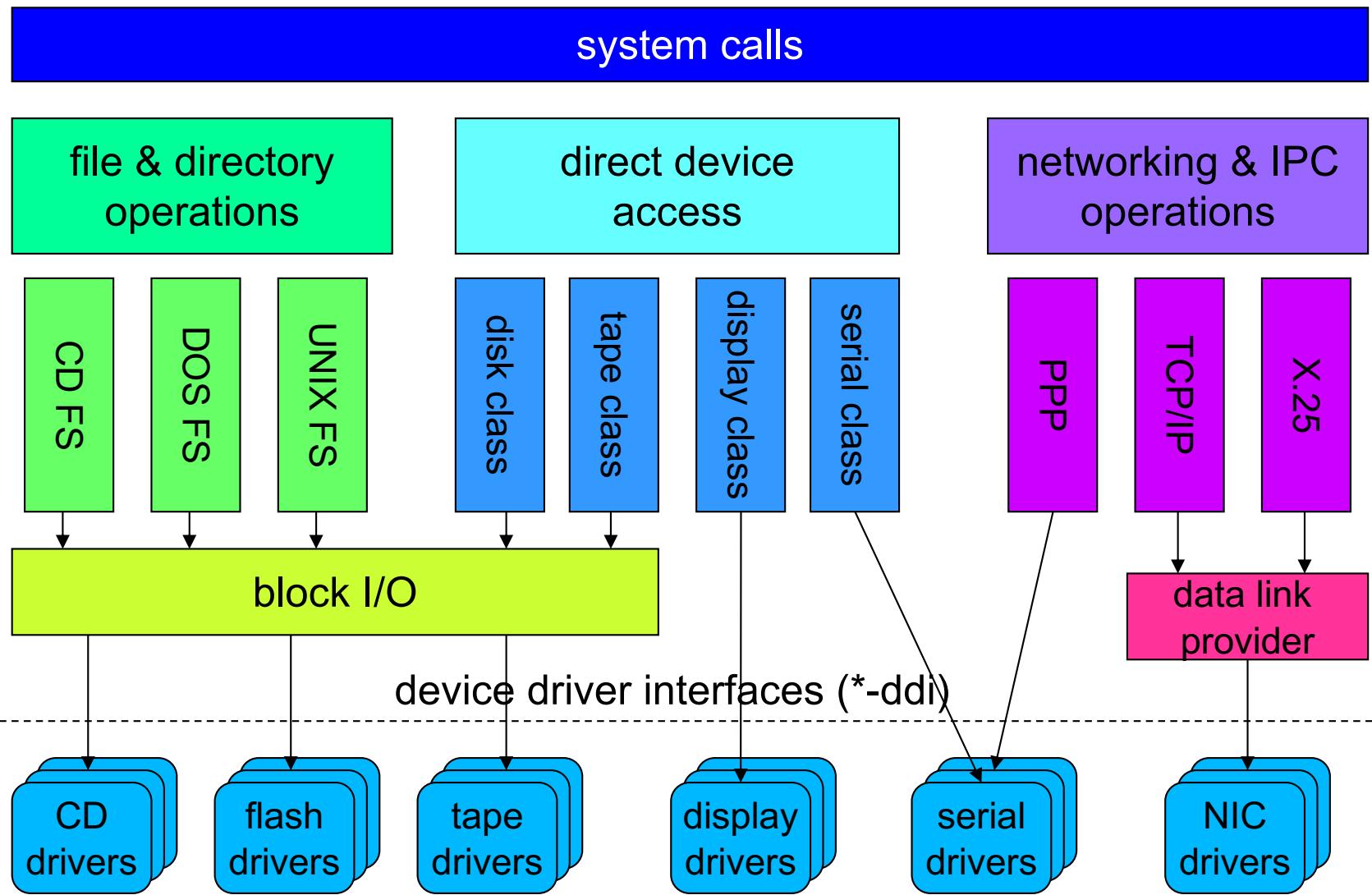
Device Driver Interface (DDI)

- 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
 - Flash drivers are meant to be called by block I/O
 - Network drivers are meant to be called by protocols

DDIs and sub-DDIs



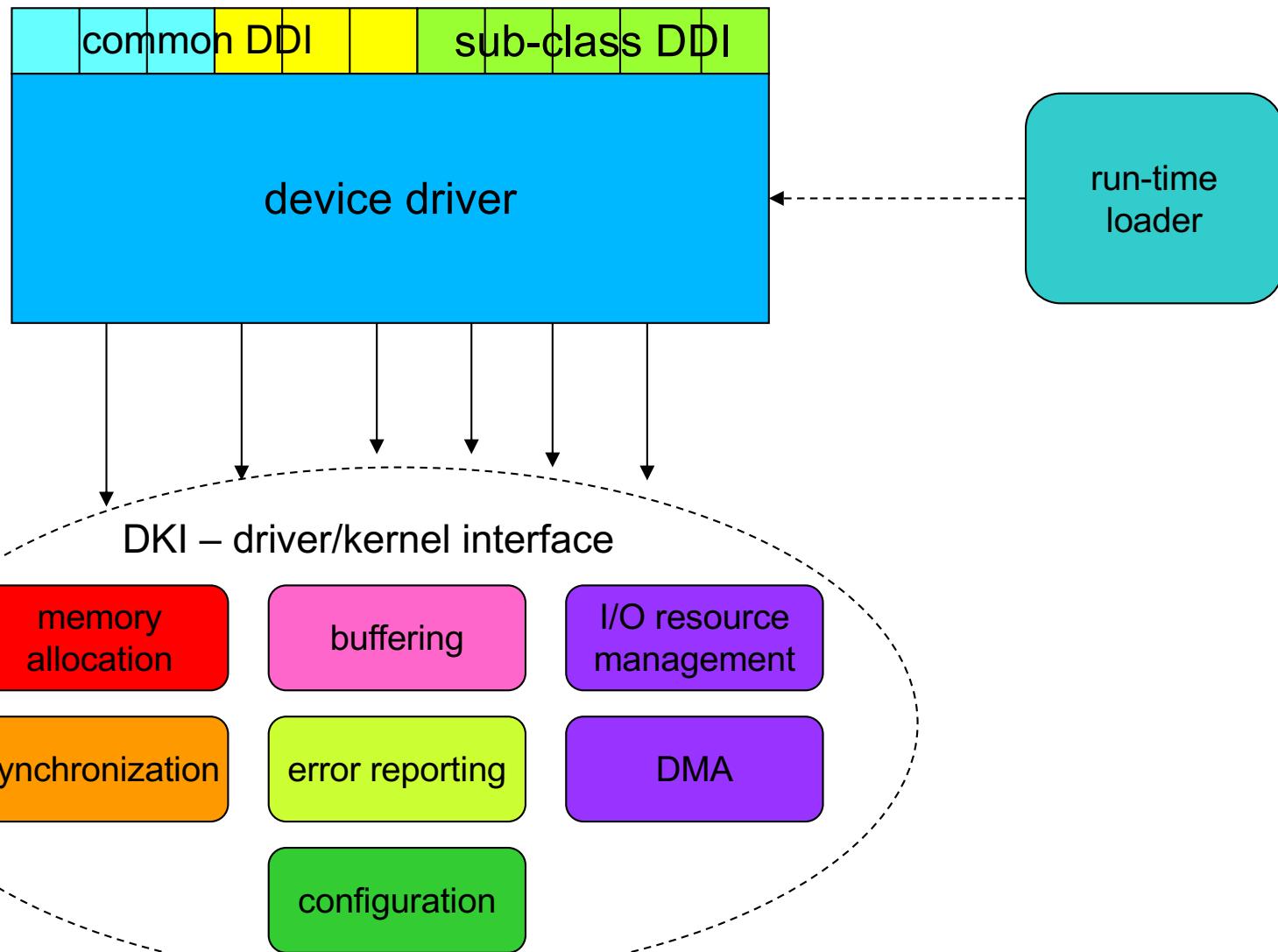
Standard Driver Classes & Clients



Drivers – Simplifying Abstractions

- Encapsulate knowledge of how to use a device
 - Map standard operations to device-specific operations
 - 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

Conclusion

- Proper handling of devices is a critical part of the OS' job
- Each device is handled by a specialized piece of software called a device drivers
- The OS uses a layered approach to get from application requests to device commands
- Poor utilization of devices can cause serious performance problems

Operating System Principles: File Systems

CS 111

Winter 2023

Operating System Principles
Peter Reiher

Outline

- File systems:
 - Why do we need them?
 - Why are they challenging?
- Basic elements of file system design
- Example file systems
 - DOS FAT
 - Unix System V file system

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

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
 - Flash drives, for example, but there are others
- 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

Flash Drives

- Solid state persistent storage devices
 - I.e., no moving parts
- Reads and writes are fairly fast
 - Reads up to 100 MB/sec
 - Writes up to 40 MB/sec
- But a given block can only be written once
 - Writing again requires erasing
 - Much slower and erases large sectors of the drive

How will these factors effect
file system design?

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

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 for flash drives of different types
- Or if we use hard disk instead of flash
- Or if we use a RAID instead of one disk
- Or if even we don't use persistent memory at all
 - E.g., RAM file systems

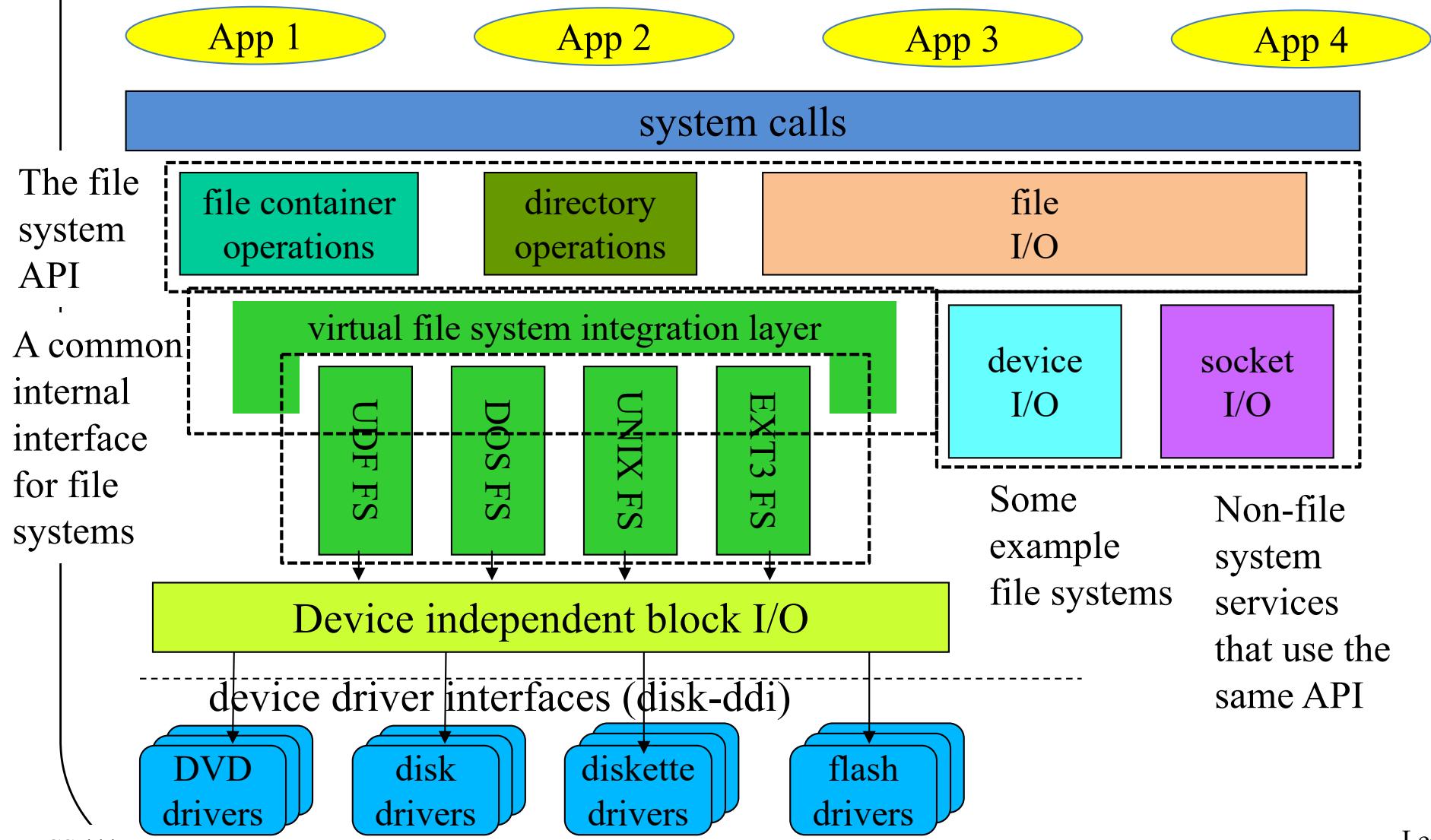
Desirable File System Properties

- 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

Basics of File System Design

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

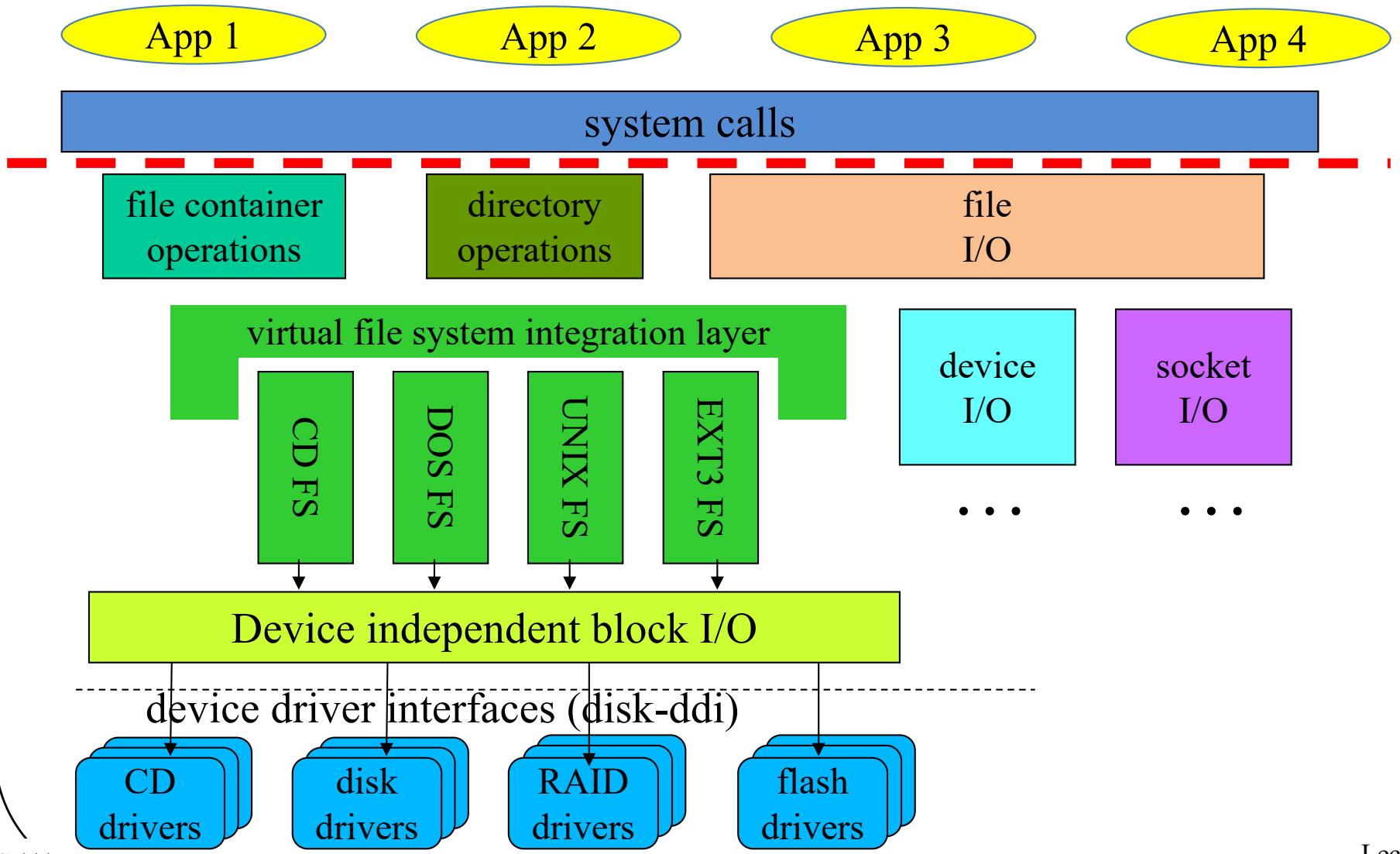
File Systems and the OS



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

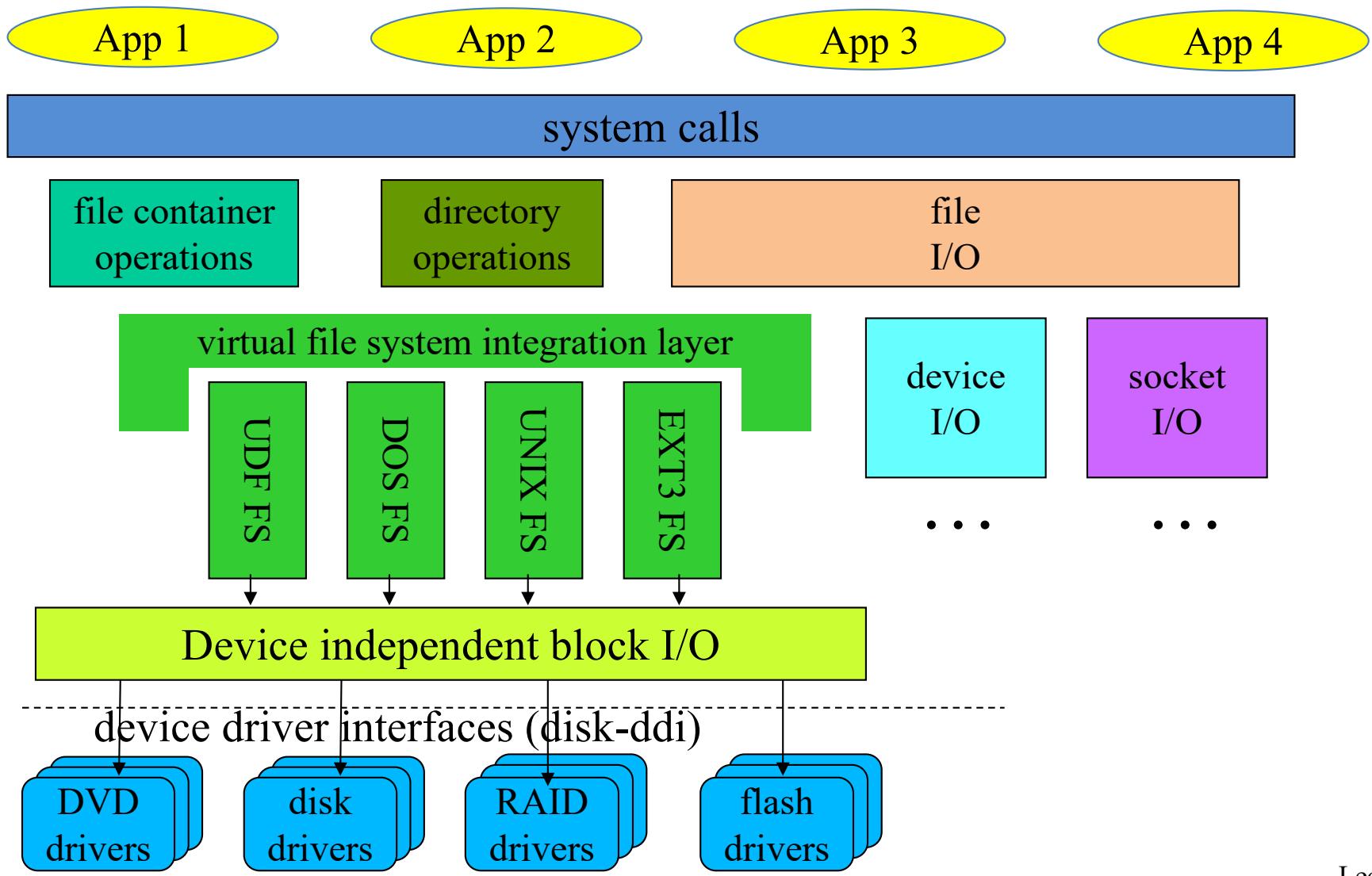
Directory Operations

- 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

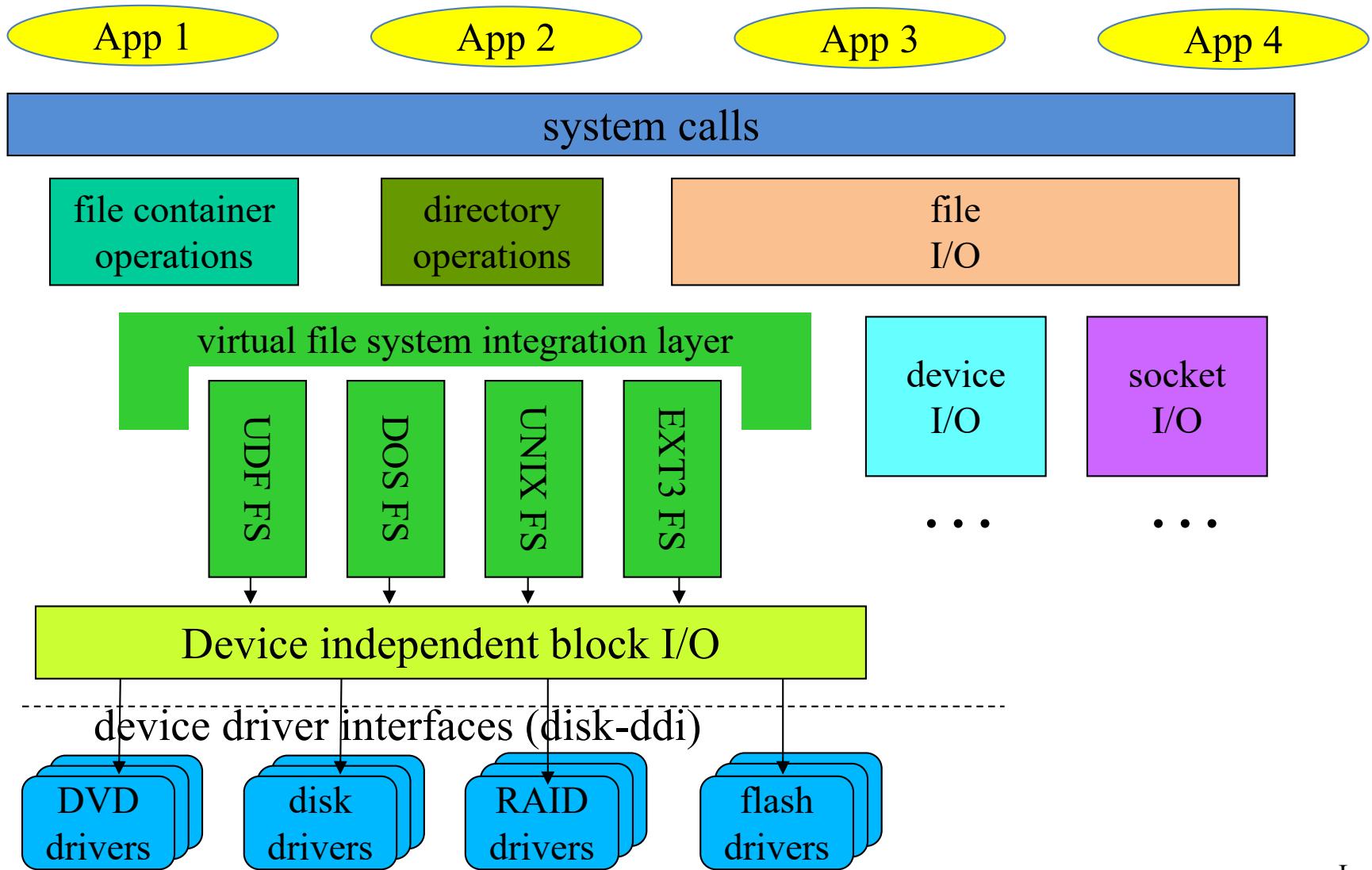
The Virtual File System Layer



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 for file system implementations
 - DOS FAT, Unix, EXT3, ISO 9660, NFS, 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



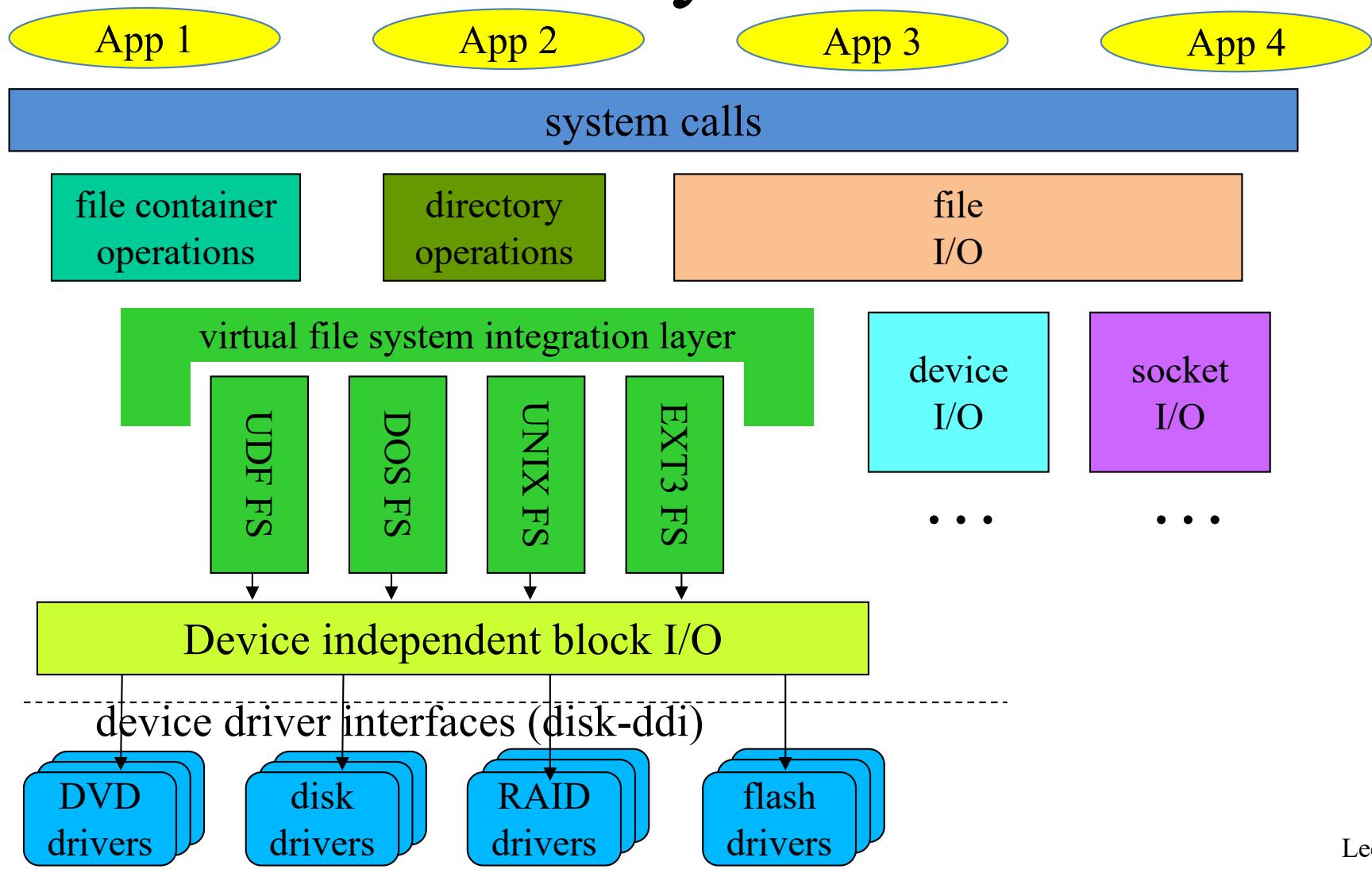
The File Systems Layer

- 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 HW 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 drives
- Allows unified LRU buffer cache for drive 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 parts of 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

Why A Single Block I/O Cache?

- Why not one per process (or user)?
- Or one per device?
- A single cache is more efficient when multiple users access the same file
- A single cache provides better hit ratio than several independent caches
 - Whether per process, user, or device
 - Generally true for caching, not just here

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 Devices

- Essentially a question of how you managed the space on your device
- Space management on a device 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 may have performance effects
 - Poor management leads to poor performance
- Must manage the space assigned to each file
 - On-device, master data structure for each file

On-Device File Control Structures

- On-device 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 be 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

The In-Memory Representation

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

File System Structure

- How do I organize a device 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 block-oriented devices
- Such 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 file 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 device is usually reserved for the *boot block*
 - Code allowing the machine to boot an OS
 - Not just for DOS, for all OSes
- Not usually under the control of a file system
 - It typically ignores the boot block entirely
- Not all devices are bootable
 - But the 0th block is usually reserved, “just in case”
- So file systems start work at block 1

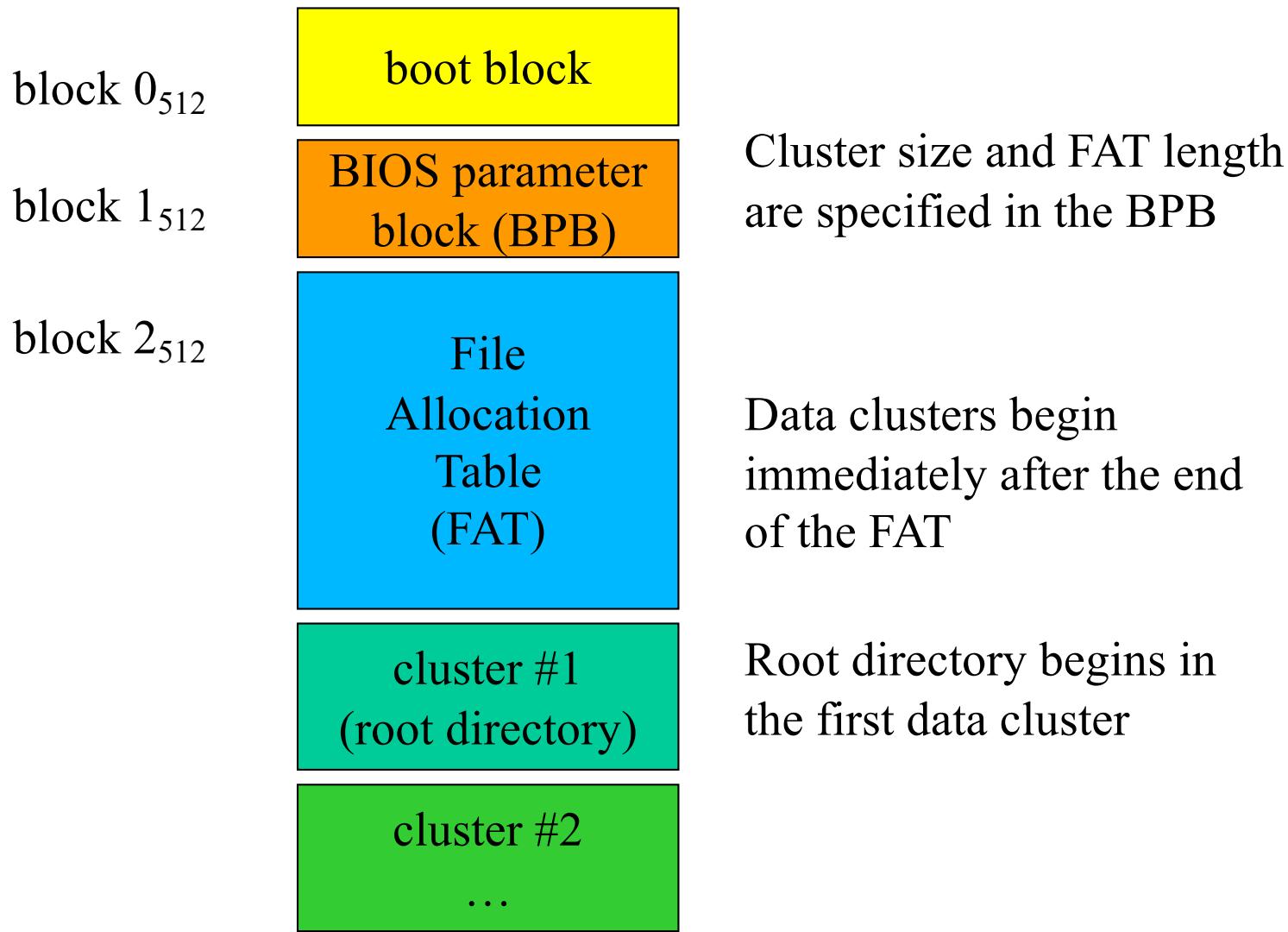
Managing Allocated Space

- A core activity for a file system, with various choices
- What if we give each file the 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 (“pages”) 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
- Or pointers can be in auxiliary “chunk linkage” table
 - Faster searches, especially if table kept in memory

The DOS File System



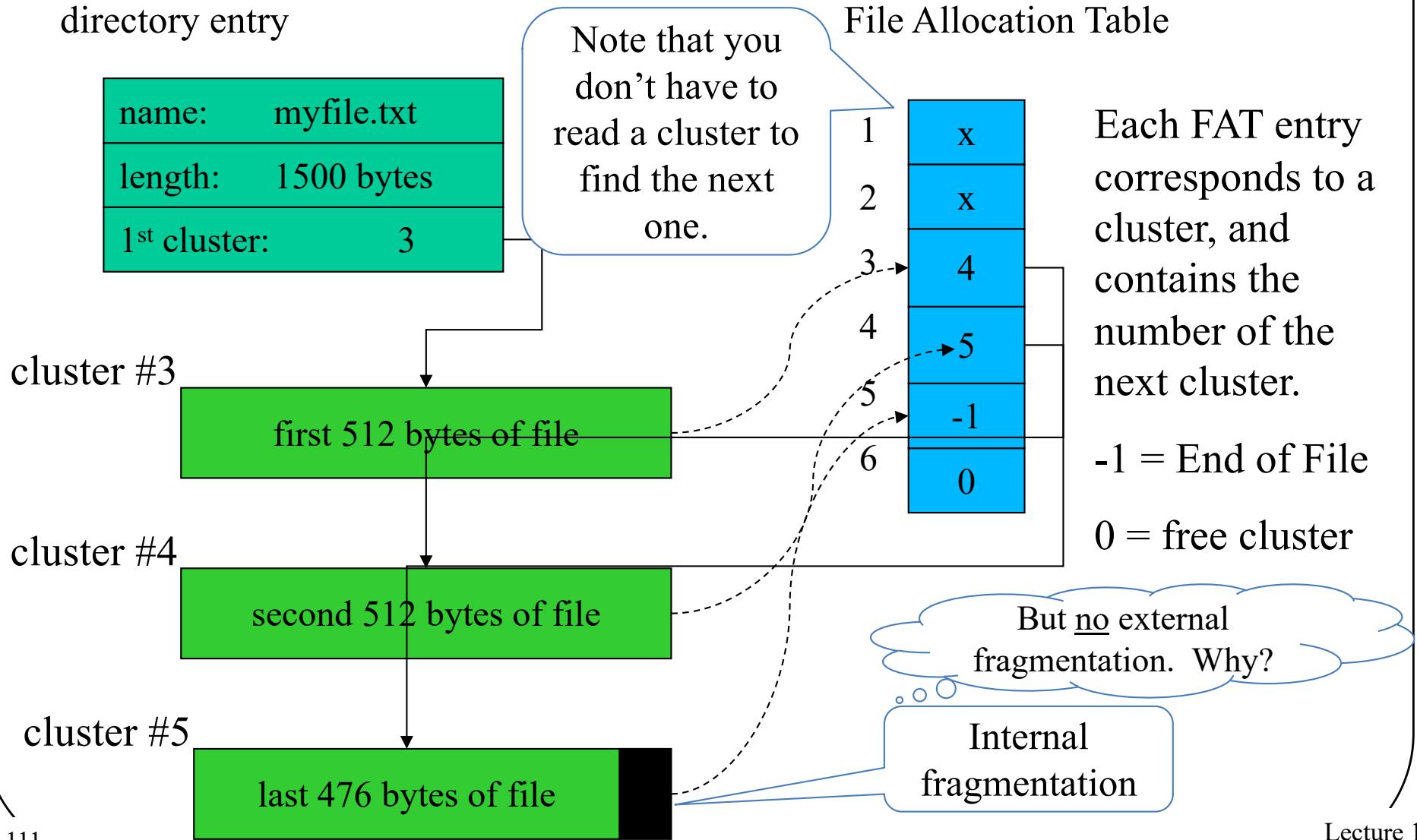
DOS File System Overview

Their name
for chunks.

- 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

An example
of a chunk
linkage
table.

DOS FAT Clusters



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 in-memory search can still be long
- No support for “sparse” files
 - If 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

How Big a File Can the DOS File System Handle?

- There's one entry in the FAT table per cluster
 - Only clusters with entries in the FAT table exist
- The FAT table has some maximum size
 - Kept in memory on a machine with little RAM
 - Originally 4096 entries
- Each cluster has some size
 - Originally 512 bytes
- Original max file size $\sim 2^{12} \times 2^9 = 2^{21} = 4\text{Mbytes}$

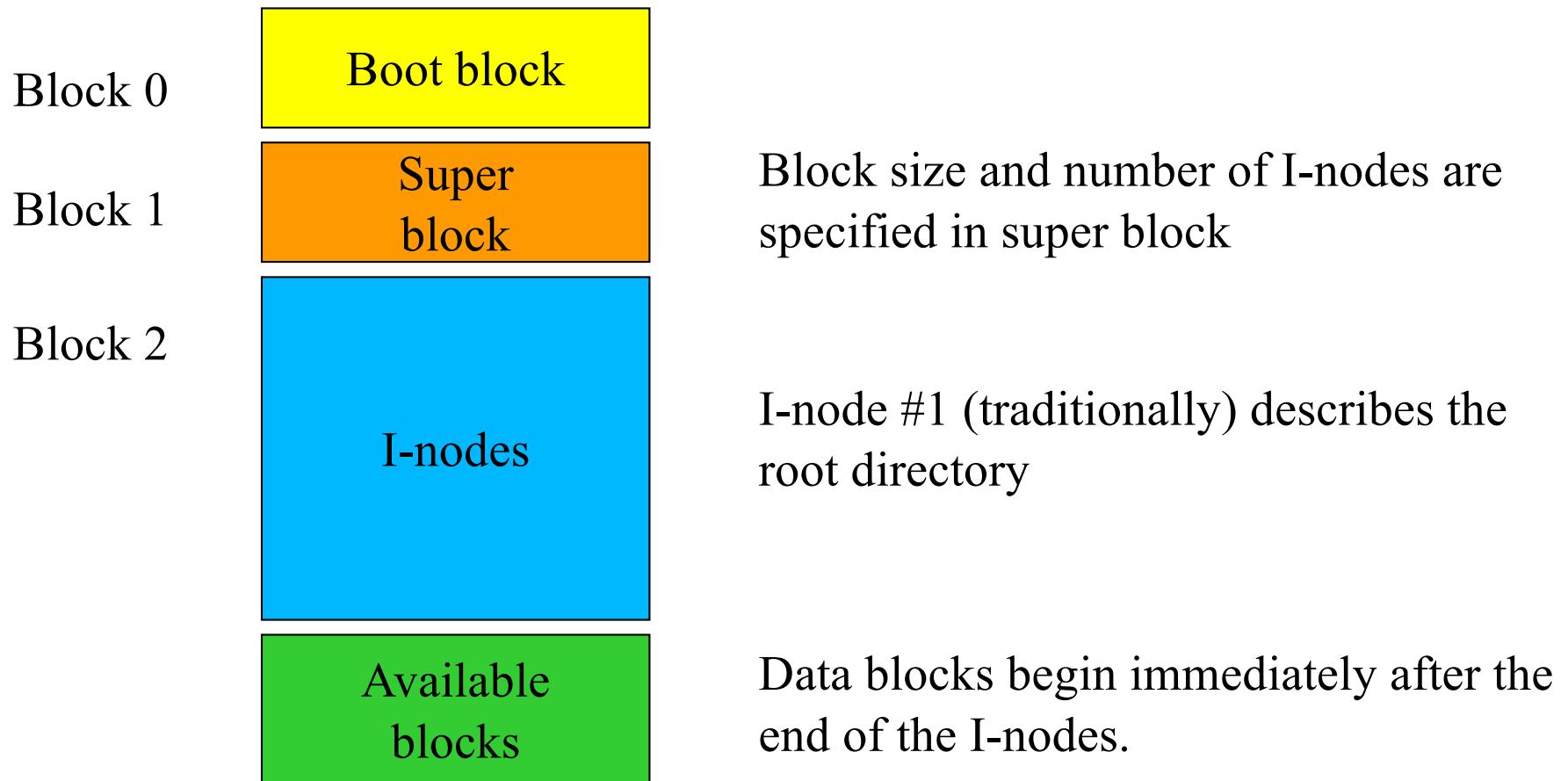
File Index Blocks

- A different way to keep track of where a file's data blocks are on the device
- 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 (of bigger than block size)
 - But this still gives us a fixed number of extents

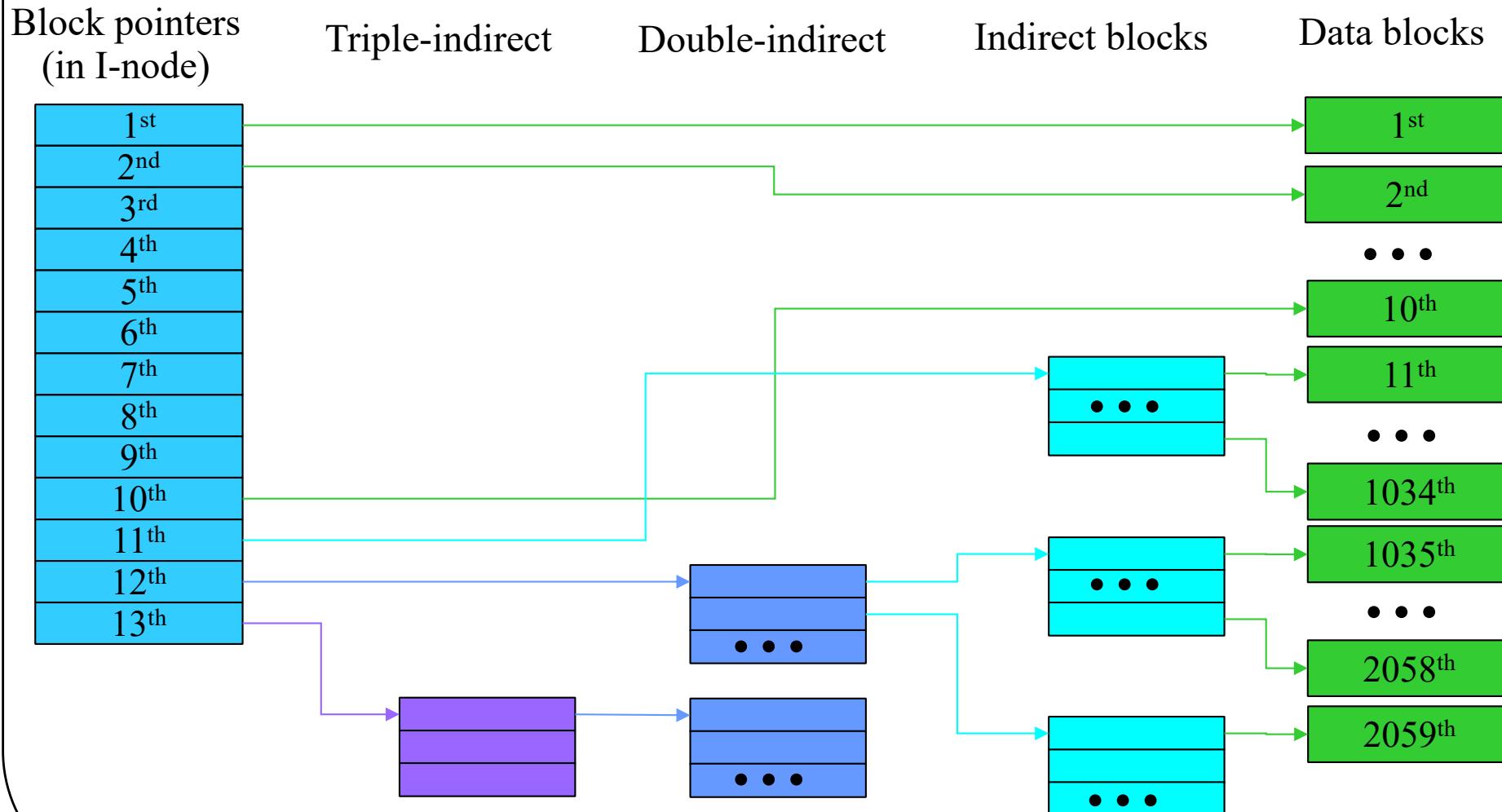
Hierarchically Structured File Index Blocks

- 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



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 blocks
 - 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

Winter 2023

Operating System Principles
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

Remember Free Lists?

- We talked about them in the context of memory allocation
 - Primarily for variable sized partitions
- These aren't variable sized partitions
 - The free elements are fixed size blocks
- But there are other issues
 - For hard disks, locality matters
 - For flash, there are issues of erasure and load leveling
- These issues may affect free list organization

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

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 which block we choose
 - Can't write fully-written flash blocks
 - May want to do wear-levelling and keep data contiguous
 - Other issues for hard disk drives
- Free-list organization must address both concerns
 - Speed of allocation and deallocation
 - Ability to allocate preferred device space

Performance Improvement Strategies

- Transfer size
- Caching

Allocation/Transfer Size

- Per operation overheads are high
 - DMA startup, interrupts, device-specific costs
- Larger transfer units are more efficient
 - Amortize fixed per-op costs over more bytes/op
 - Multi-megabyte transfers are very good
- What unit do we use to allocate storage space?
 - Small chunks reduce efficiency
 - Large fixed size chunks -> internal fragmentation
 - Variable sized chunks -> external fragmentation
 - Tradeoff between fragmentation and efficiency

Flash Drive Issues

- Flash is becoming the dominant technology
 - Sales overtook HDD in 2021
- Special flash characteristics:
 - Faster than hard disks, slower than RAM
 - Any location equally fast to access
 - But write-once/read-many access
 - Until you erase
 - You can only erase very large chunks of memory
- Think about this as we discuss other file system issues

Caching

- Caching for reads
- Caching for writes

Read Caching

- Persistent storage I/O takes a long time
 - Deep queues, large transfers improve efficiency
 - They do not make it significantly faster
- We must eliminate much of our persistent storage 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 device 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 device access time reading unwanted blocks
 - May waste buffer space on unneeded blocks

Write Caching

- Most device writes go to a write-back cache
 - They will be flushed out to the device 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

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 *That's what we mean by binding*
 - 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 means 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
 - Graph-based name spaces
 - Can be a strict hierarchical 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?

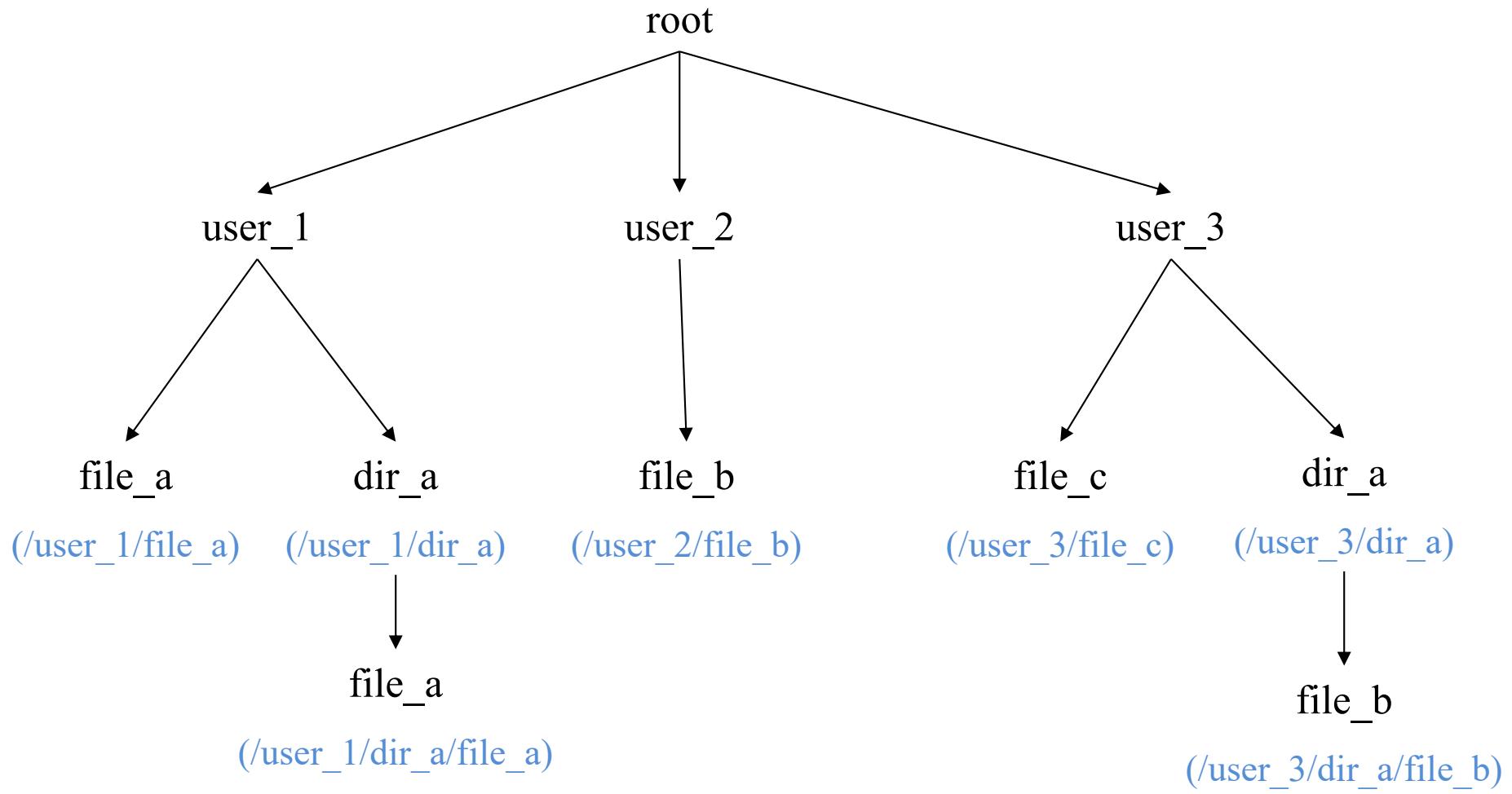
Some Issues in Name Space Structure

- 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!

Useful if we use defaults to attach prefixes to file names.

Root directory, inode #1
inode # file name

1	.
1	..
9	user_1
31	user_2
114	user_3

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

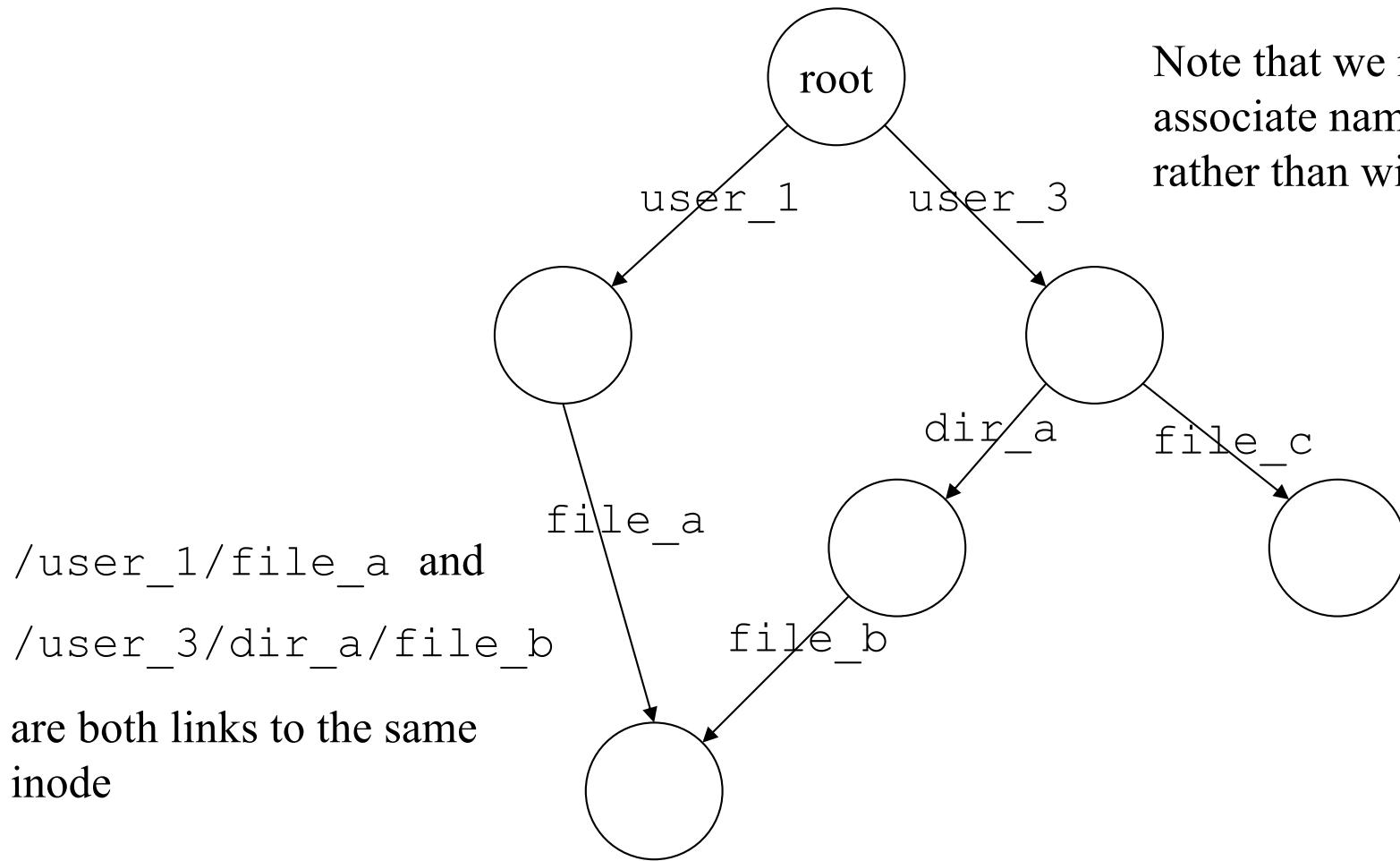
Multiple File Names In Unix

- 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

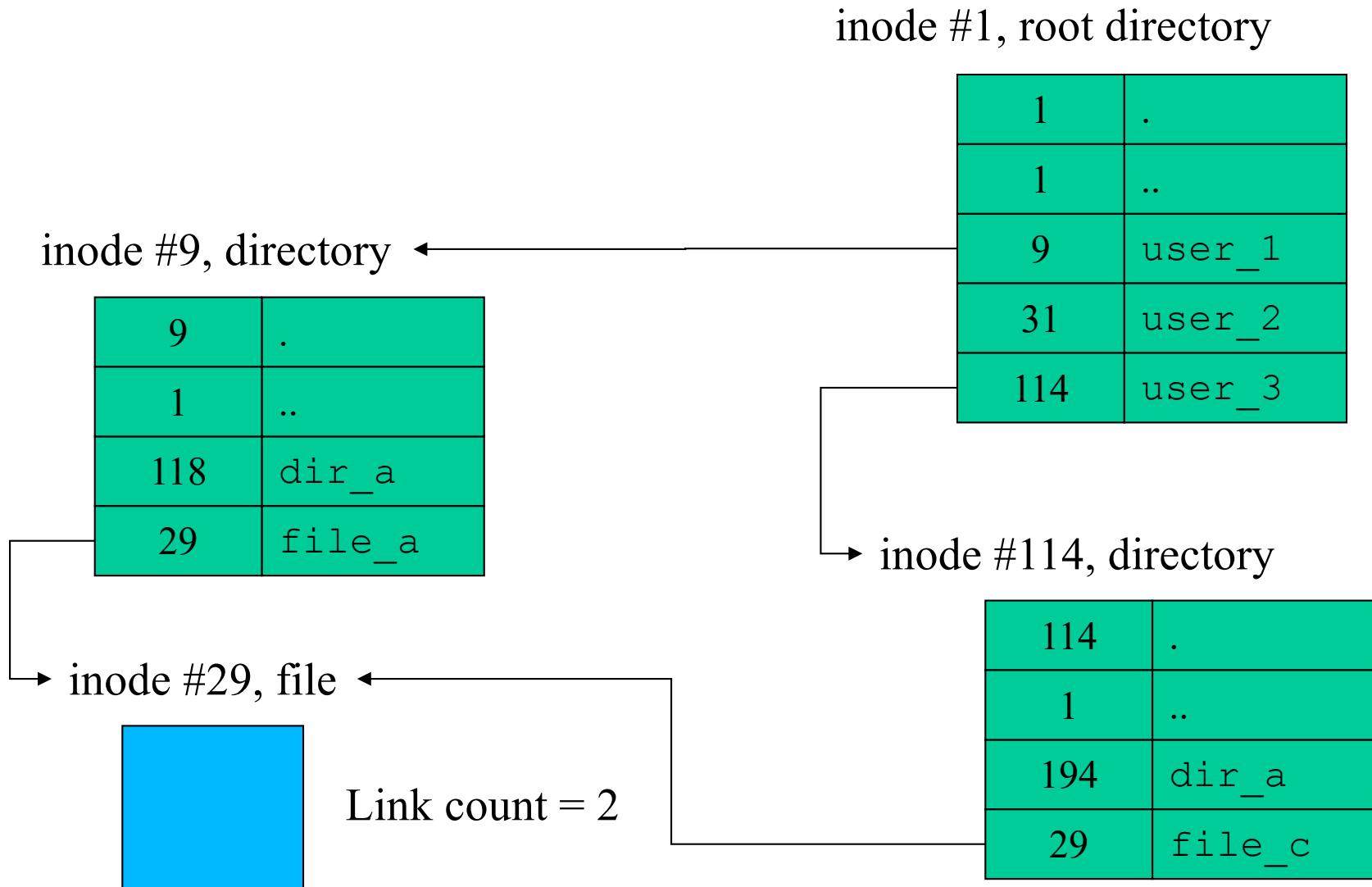
- Files exist under multiple names
- What do we do if one name is removed?
- If we also removed the file itself, what about the other names?
 - Do they now point to something non-existent?
- The Unix solution says the file exists as long as at least one name exists
- Implying we must keep and maintain 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



File Systems Reliability

- What can go wrong in a file system?
- System crashes can result in incorrect file system state
- 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

The Core Reliability Problem

- File system writes typically involve multiple operations
 - Not just writing a data block to disk/flash
 - But also writing one or more metadata blocks
 - The inode, the free list, maybe directory blocks
- All must be committed to disk for the write to succeed
- But each block write is a separate hardware operation

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

So we
need to
write a
free list
block.

So we
need to
write an
inode.

Application Expectations When Writing

- Applications make system calls to perform writes
- When system call returns, the application (and user) expect the write to be “safe”
 - Meaning it will persist even if system crashes
- We can block the writing application until really safe
- But that might block application for quite a while . . .
- Crashes are rare
 - So persistence failure caused by them are also rare
 - Must we accept big performance penalties for occasional safety?

Buffered Writes

- Don't wait for the write to actually be persisted
- Keep track of it in RAM
- Tell the application it's OK
- At some later point, actually write to persistent memory
- Up-sides:
 - Less application blocking
 - Deeper and optimizable write queues
- Down-side:
 - What if there's a crash between lying and fixing the lie?

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
 - Improperly shared data is more serious than missing data

Practicality of Ordered Writes

- Greatly reduced I/O performance
 - Eliminates accumulation of near-by operations
 - Eliminates consolidation of updates to same block
- May not be possible
 - Modern devices may 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
- Audit file system for correctness and use redundant info to enable automatic repair
- Used to be standard practice
- No longer practical
 - Checking a 2TB FS at 100MB/second = 5.5 hours
- We need more efficient partial write solutions

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

A Journaling Example

Write to /a/foo

Let's say that's two pages of data



Replacing two existing pages

Put a *start* record in the journal

Plus metadata about where to put the two blocks

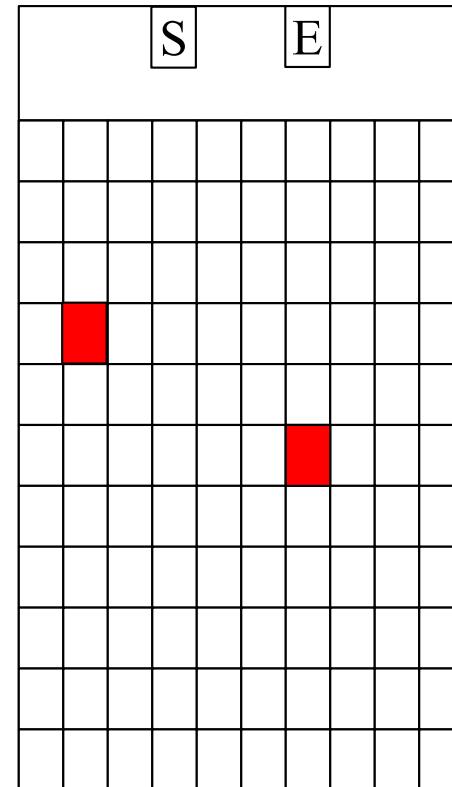
Then write the two pages to the journal

Put an *end* record in the journal

Tell the writing process that it's done

When the OS has spare time, copy the data pages to their true locations

Then get rid of the log entry



The storage device

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 small 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 are much faster than data writes
 - All journal writes are sequential
- 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
- Journal pages may contain information for multiple files
 - Performed by different processes and users

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 for the data, write it there
 - Then journal the meta-data updates

A Metadata Journaling Example

Write to /a/foo

Let's say that's two pages of data



This is more natural
for flash. Why?

Replacing two existing pages

Put a *start* record in the journal

Plus *new* metadata about where to put the two blocks

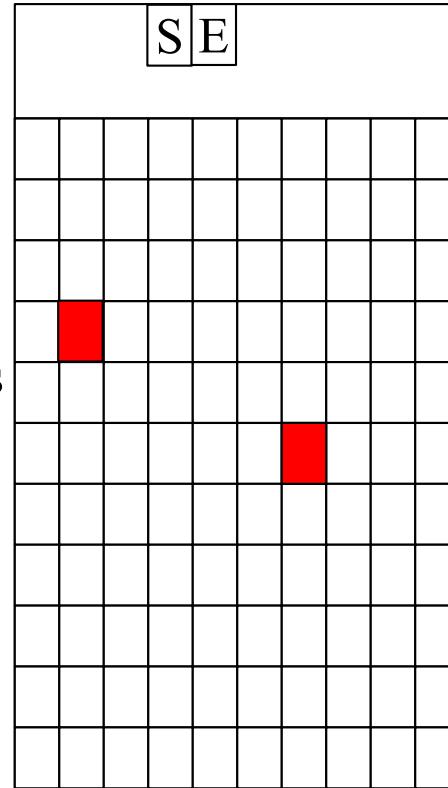
Then write the two pages to the new locations

Put an *end* record in the journal

Tell the writing process that it's done

When the OS has spare time, update the file's
metadata to show the new page locations

Then free the old data pages and get rid of the log entry



The storage device

Log Structured File Systems

- The journal is the file system
 - All inodes and data updates written to the log
 - Updates are Redirect-on-Write
 - An in-memory index caches inode locations
- Becoming a dominant architect
 - Flash file systems
 - Key/value stores
- Issues
 - Recovery time (to reconstruct index/cache)
 - Log defragmentation and garbage collection

Don't overwrite the old data.

Write it elsewhere and change the metadata (inode) pointer to it.

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

Note: This will work very nicely for flash devices

- 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

A Log Structured File System Example

The current head of the log

Write to /a/foo

Let's say that's two pages of data



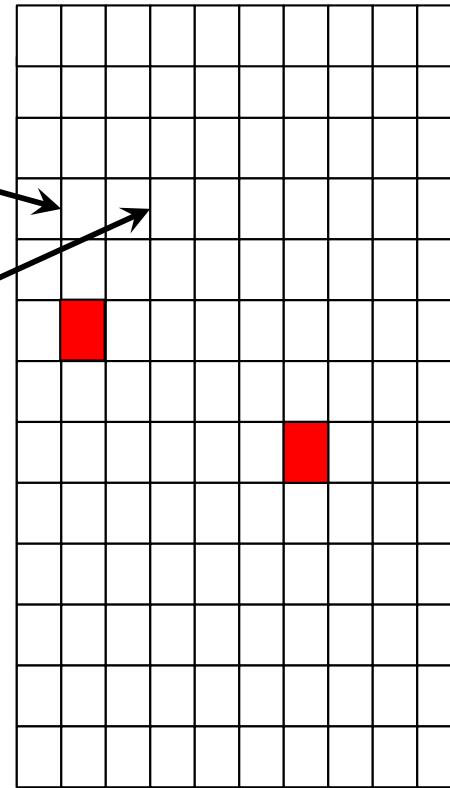
Replacing two existing pages

Write the new pages to the head of the log

Move the head of the log pointer

*But how can we find the
new pages of foo?*

*Foo's inode still points to
the old version of the pages*



The storage device

Continuing the Example

Foo's inode would still point to the old versions

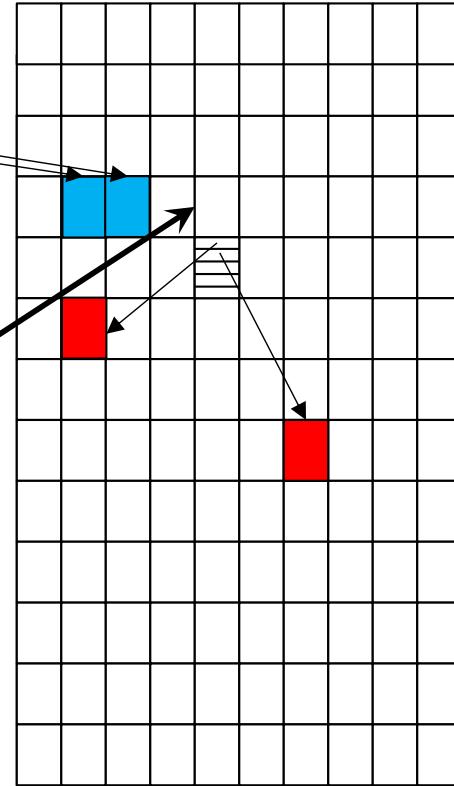
So create a new inode for foo

Pointing to the new pages

Where do we put the new inode?

In the log!

And we move the head of the log pointer



The storage device

But . . .

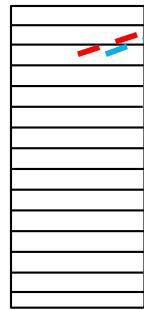
How do we find the inode for this file?

The */a* directory's entry for foo points to the old inode

Traditional Linux file systems keep all inodes
in one part of the disk

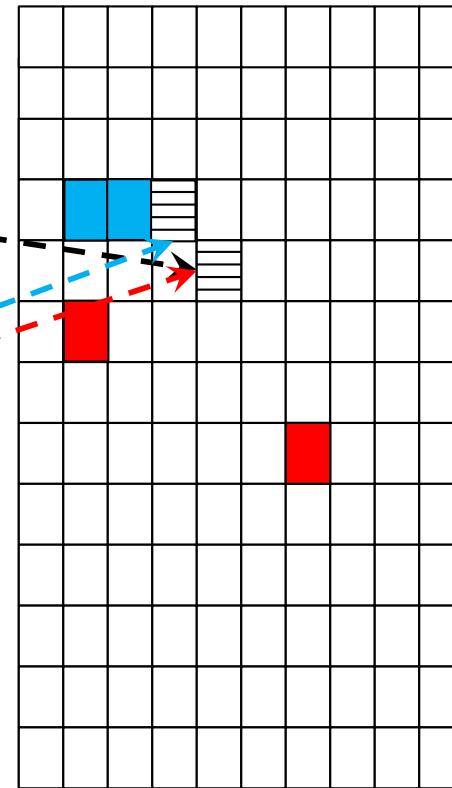
LFS scatters them all over the log

So use an inode map to keep track of them



The old location

The new location



The storage device

One Thing Leads to Another

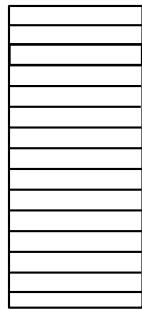
The inode map better be persistent

So we need to store it on disk

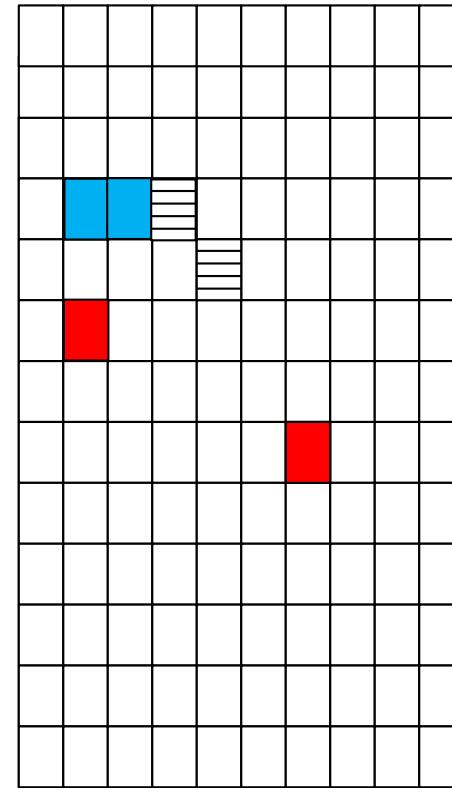
Where?

How about in the log?

Maybe just a relevant piece of it, though



But how do we find the inode
map's pieces?



The storage device

Read the book to find out

Conclusion

- We must have some solution to how to manage space on a persistent device
- We must have some scheme for users to name their files
 - And for the file system to match names to locations
- Performance and reliability are critical for file systems
- How the file system works under the covers matters a lot for those properties

Operating System Principles: Security and Privacy

CS 111

Winter 2023

Operating System Principles
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 (mostly)

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
 - Often the response can be found by attackers
- 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 picky 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

Multi-factor Authentication

- Rely on two separate authentication methods
 - E.g., a password and a text message to your cell phone
- If well done, each method compensates for some of the other's drawbacks
 - If poorly done, not so much
- The current preferred approach in authentication

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

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 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
- 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

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 a request (system call) 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 lx
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

- Hard to separate encryption from authentication
 - Complicates some signature uses
- 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 somewhat used, for legacy reasons
 - 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
 - Or I authenticate incorrectly
- 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

There are actually potential security problems with this method.

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: Virtual Machines

CS 111

Winter 2023

Operating System Principles
Peter Reiher

Outline

- What is a virtual machine?
- Why do we want them?
- How do virtual machines work?
- Issues in virtual machines

What Is a Virtual Machine?

- Remember, in CS, “virtual” means ”not real”
 - But it looks like it’s real
- So a virtual machine isn’t really a machine
 - But it looks like a machine
- What do we mean by that?
- A *virtual machine* is a software illusion meant to appear to be a real machine
- Virtual machines abbreviated as VMs

What's That Really Mean?

- We have an actual computer
- We do something in software to make it look like we have multiple computers
 - Or that it's a different kind of computer
 - Making use of the actual computer to do so
- The virtual machine must appear to apps and users to be a real machine

Graphically, . . .



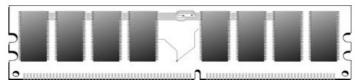
We implement a virtual server on the real hardware



With a set of virtual components



We have a real server computer



With a real CPU

And real RAM

And real peripherals



How?

- Use the real hardware to implement the virtual hardware
 - Instructions for the virtual CPU run on the real CPU
 - Real RAM stores the data for virtual RAM
 - A real disk stores data for the virtual disk
 - Etc.
- But to what purpose?

Why Do We Want Virtual Machines?

- For several reasons
 - Fault isolation
 - Better security
 - To use a different operating system
 - To provide better controlled sharing of the hardware
- Let's consider each reason separately

Fault Isolation

- Operating systems must never crash
 - Since they take everything down with them
- But crashing a virtual machine's operating system need not take down the entire machine
 - Just the virtual machine
- So our correctness requirements can be relaxed
- Similar advantages for faults that could damage devices
 - They damage the virtual device, not the physical

Better Security

- The OS is supposed to provide security for processes
- But the OS also provides shared resources
 - Such as the file system and IPC channels
- A virtual machine need not see the real shared resource
- So processes in other virtual machines are harder to reach and possibly damage

Using a Different Operating System

- Let's say you're running Windows
- But you want to run a Linux executable
- Windows has one system call interface, Linux has a different one
 - So system calls from your Linux executable won't work on Windows
- But if you have a virtual machine running Linux on top of the real machine running Windows . . .
 - Now your application can run fine
 - Assuming you get the virtualization right . . .

Sharing a Machine's Resources

- In principle, an OS can control how to share resources among processes
- But actually guaranteeing a particular allocation of resources is hard
- It's easier to guarantee an entire virtual machine gets a set allocation of resources
 - So the processes running in it will not steal resources from the other virtual machines
- A very big deal for cloud computing

How Do We Run Virtual Machines?

- Easiest if the virtual and real machine use the same ISA (Instruction Set Architecture)
 - Tricky and probably slow, otherwise
 - So the same ISA is the common case
- Basically, rely on limited direct execution
 - Run as much VM activity directly on the CPU as possible
 - When necessary, trap from the VM
- But trap to what?

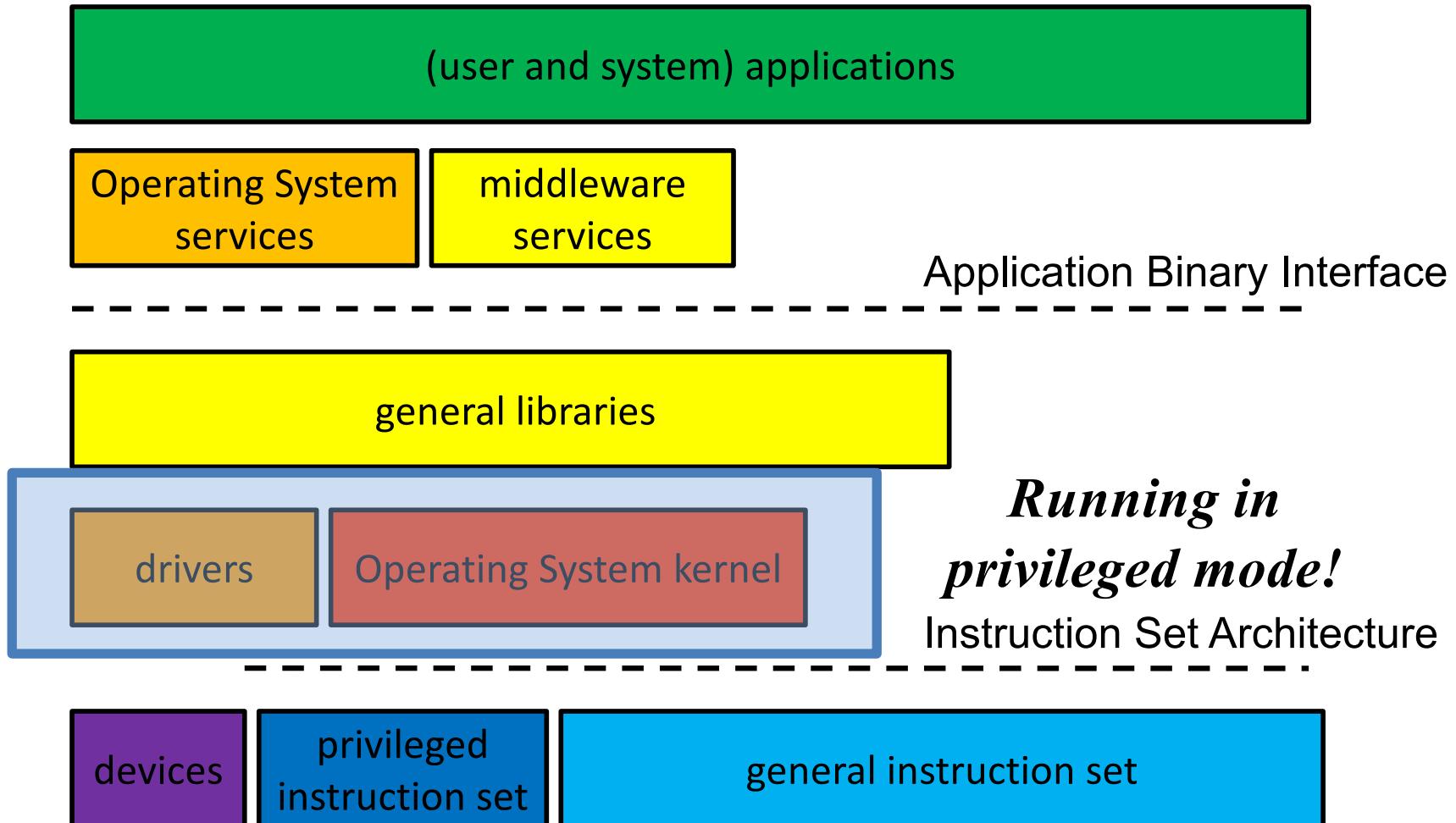
The Hypervisor

- Also known as the Virtual Machine Monitor (VMM)
- A controller that handles all virtual machines running on a real machine
- When necessary, trap from the virtual machine to the VMM
- It performs whatever magic is necessary
- And then returns to limited direct execution
- Much like a process' system call to an OS

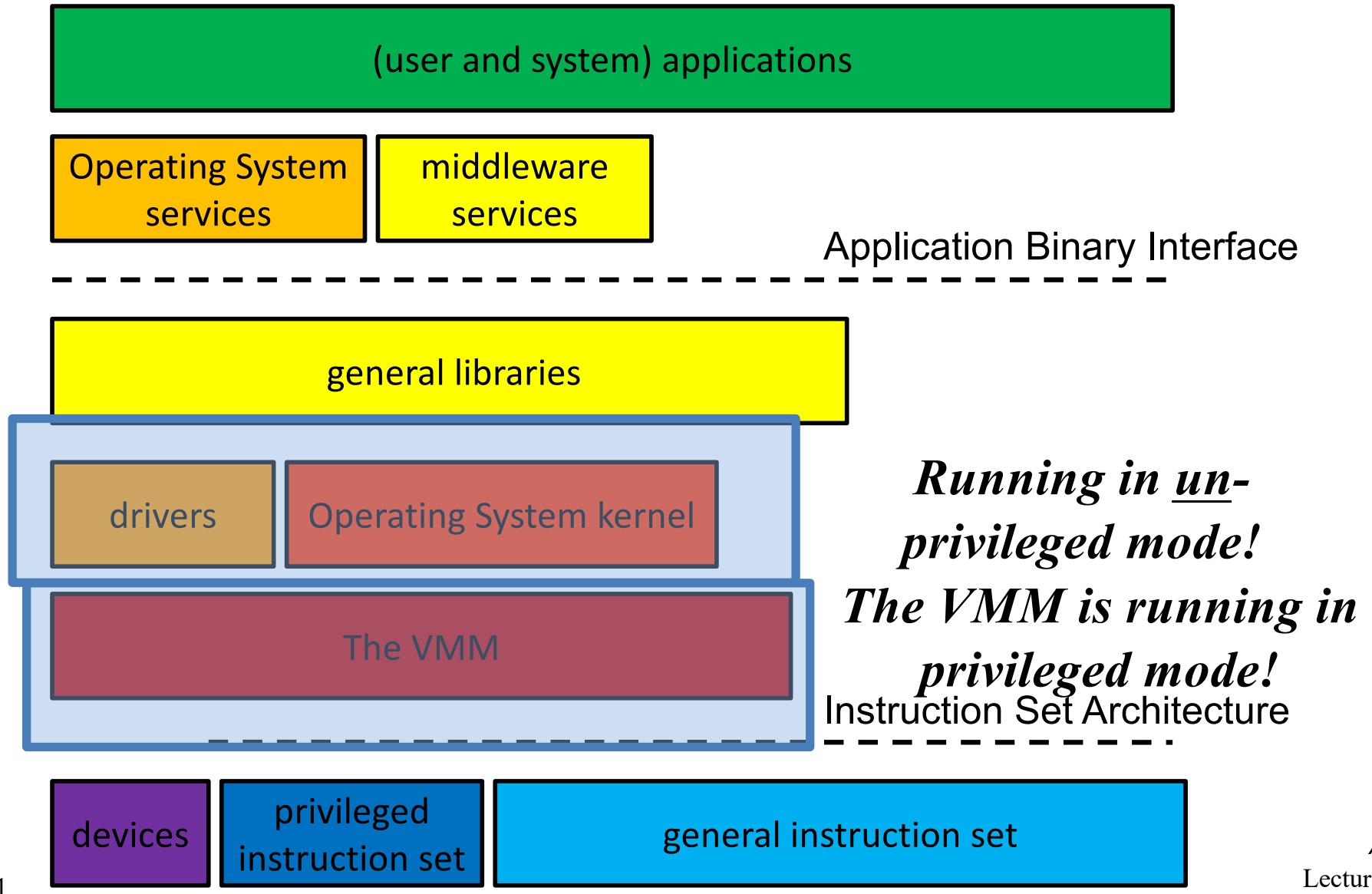
When Is Trapping to the VMM Necessary?

- Whenever the VM does something privileged
 - Kind of like trapping to the OS when a process wants to do something privileged
- The initial system call instruction will trap to the VMM
- Which will typically forward it to the VM's OS
- But subsequent privileged operations trap back to the VMM

The Old Architecture



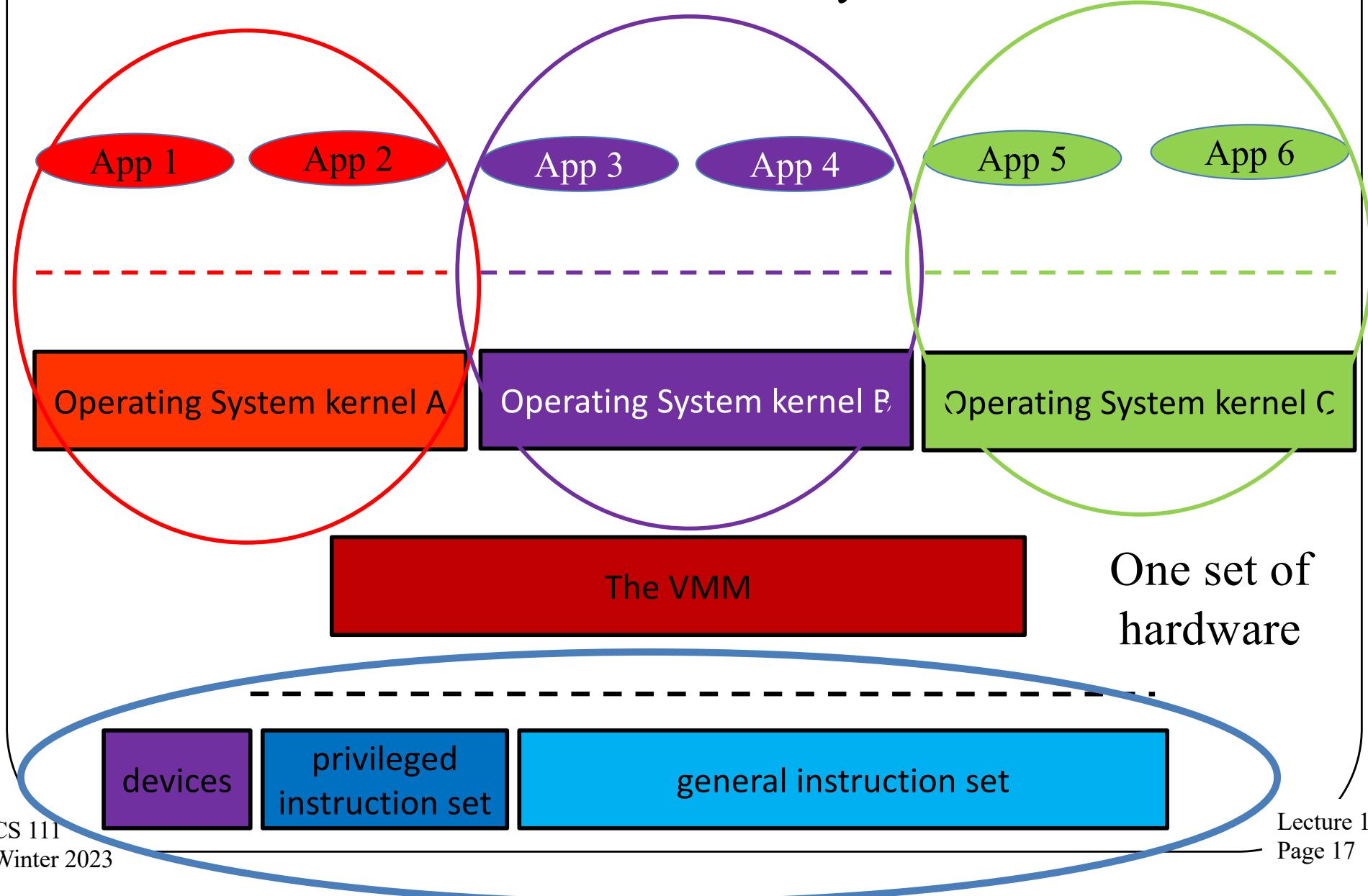
Architecture With a VMM



A More Complex Case

Three virtual machines

With three system call interfaces



How Do System Calls Work Now?

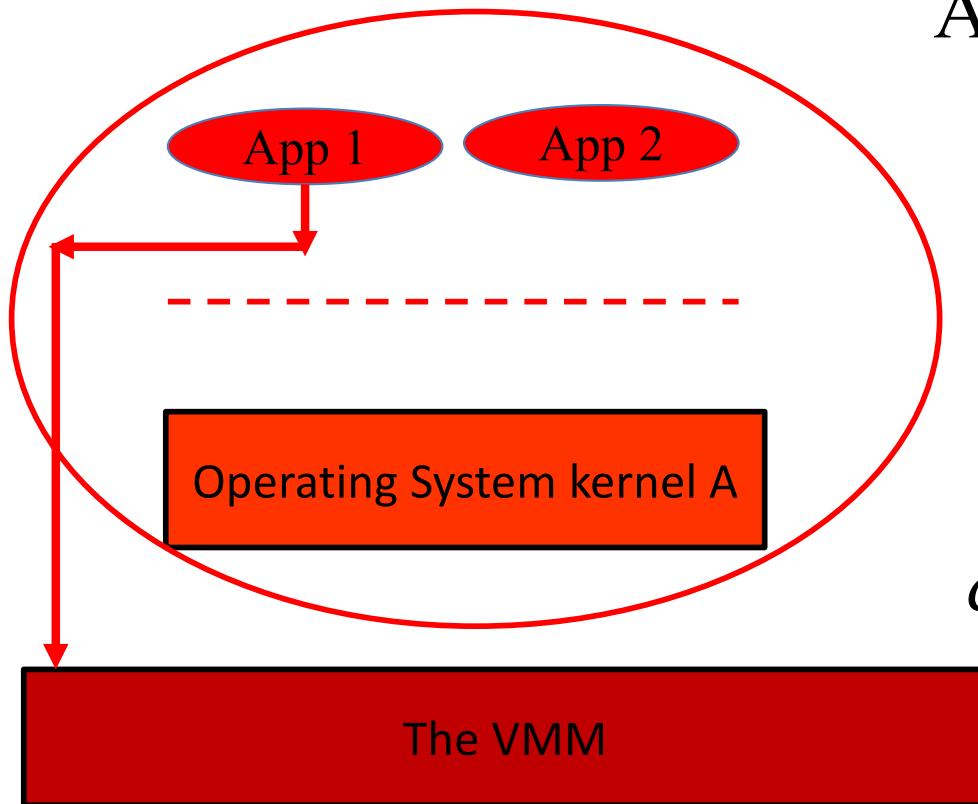
Using a
privileged
instruction

It's sent to
the VMM
instead

App 1 makes a
system call

The virtual
machine

*Which OS A
can't perform*



Yeah, But . . .

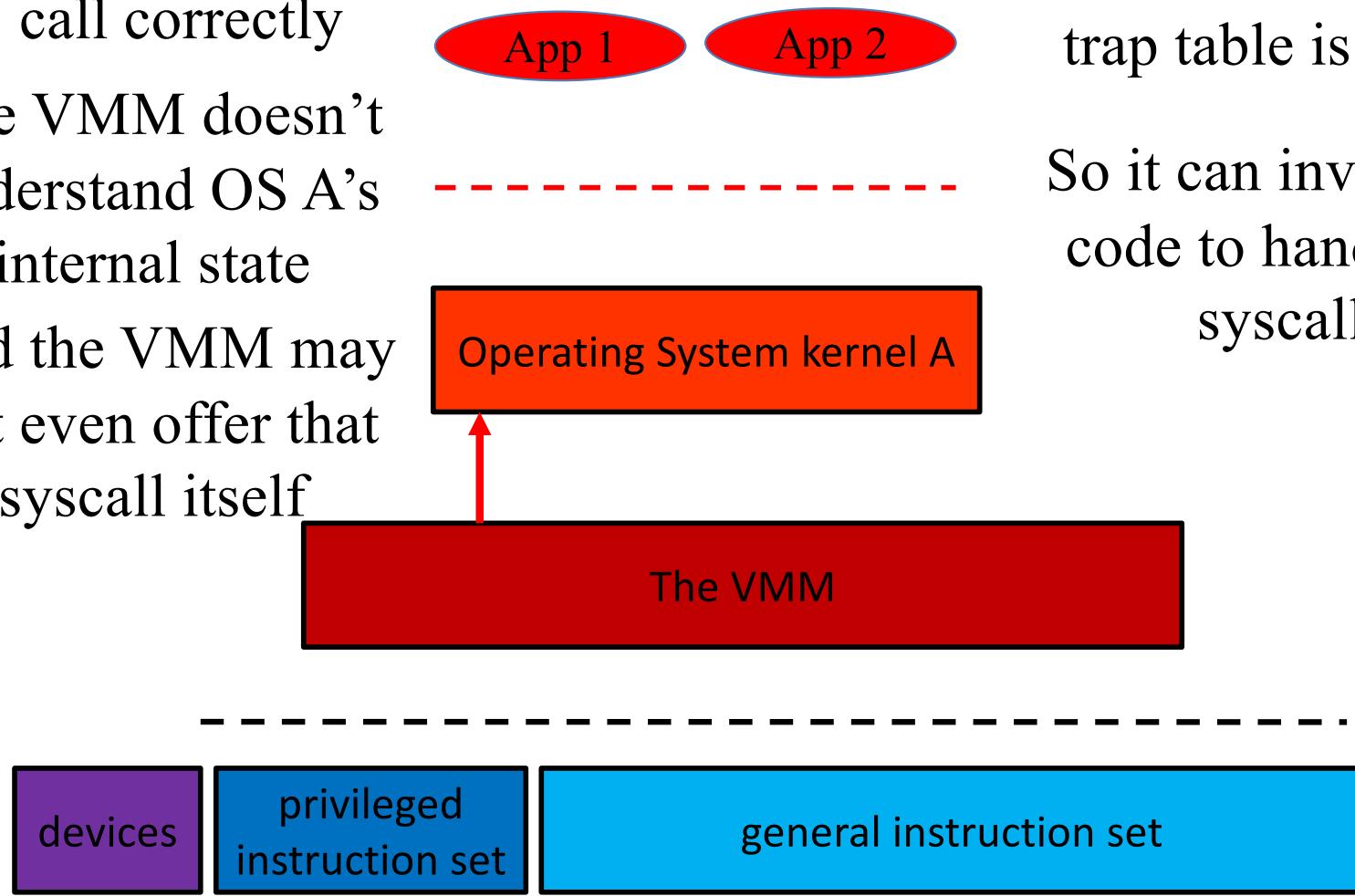
The VMM can't perform the system call correctly

The VMM doesn't understand OS A's internal state

And the VMM may not even offer that syscall itself

But the VMM knows where A's trap table is located

So it can invoke A's code to handle the syscall!



Yeah, But, Again . . .

If it's a syscall, it may need to use privileged instructions to do its work



But OS A can't use privileged instructions

No problem!

OS A traps when it tries to use a privileged instruction

And the VMM catches the trap and does the instruction for A!

Operating System kernel A

The VMM

devices

privileged instruction set

general instruction set

What's the Point of That?

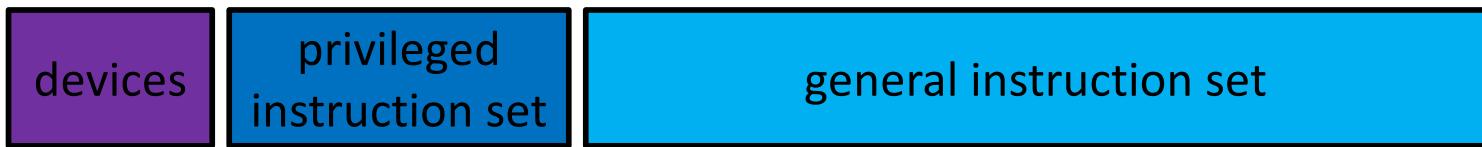
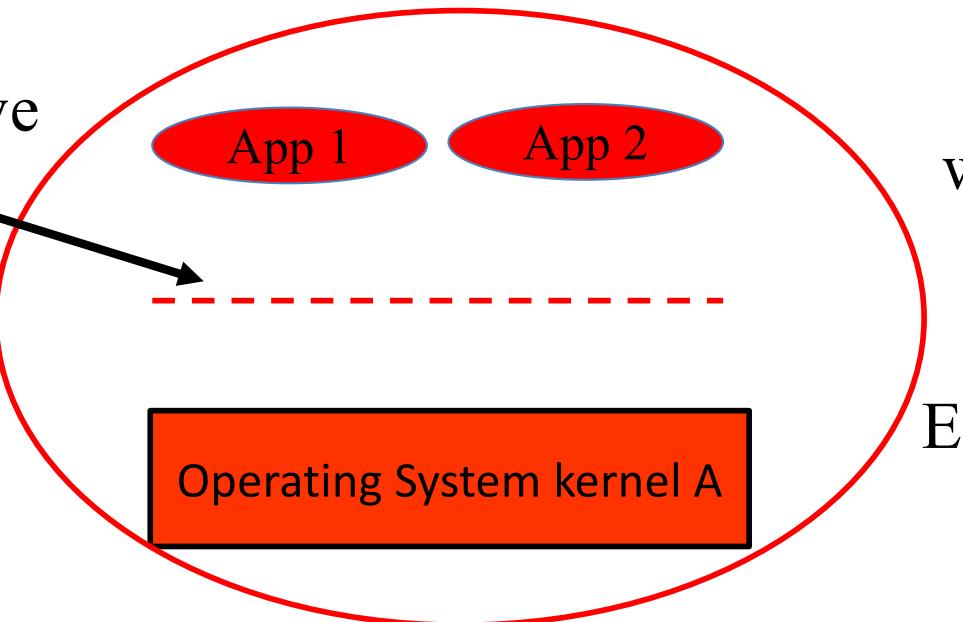
- If the VMM is going to do the instruction, why not just run A with privilege?
 - So it can do its own instructions
- Well, the VMM might decide not to do the instruction
 - If, for example, it tries to access another VM's memory
- Or the VMM might block VM A and run VM B for a while instead
- The key point: the VMM controls what happens
 - Even though the OS in the VM thinks it is in control

A Potential Issue

If A is running in non-privileged mode, how can we enforce this interface?

How can we prevent App 1 from messing with A's internal data?

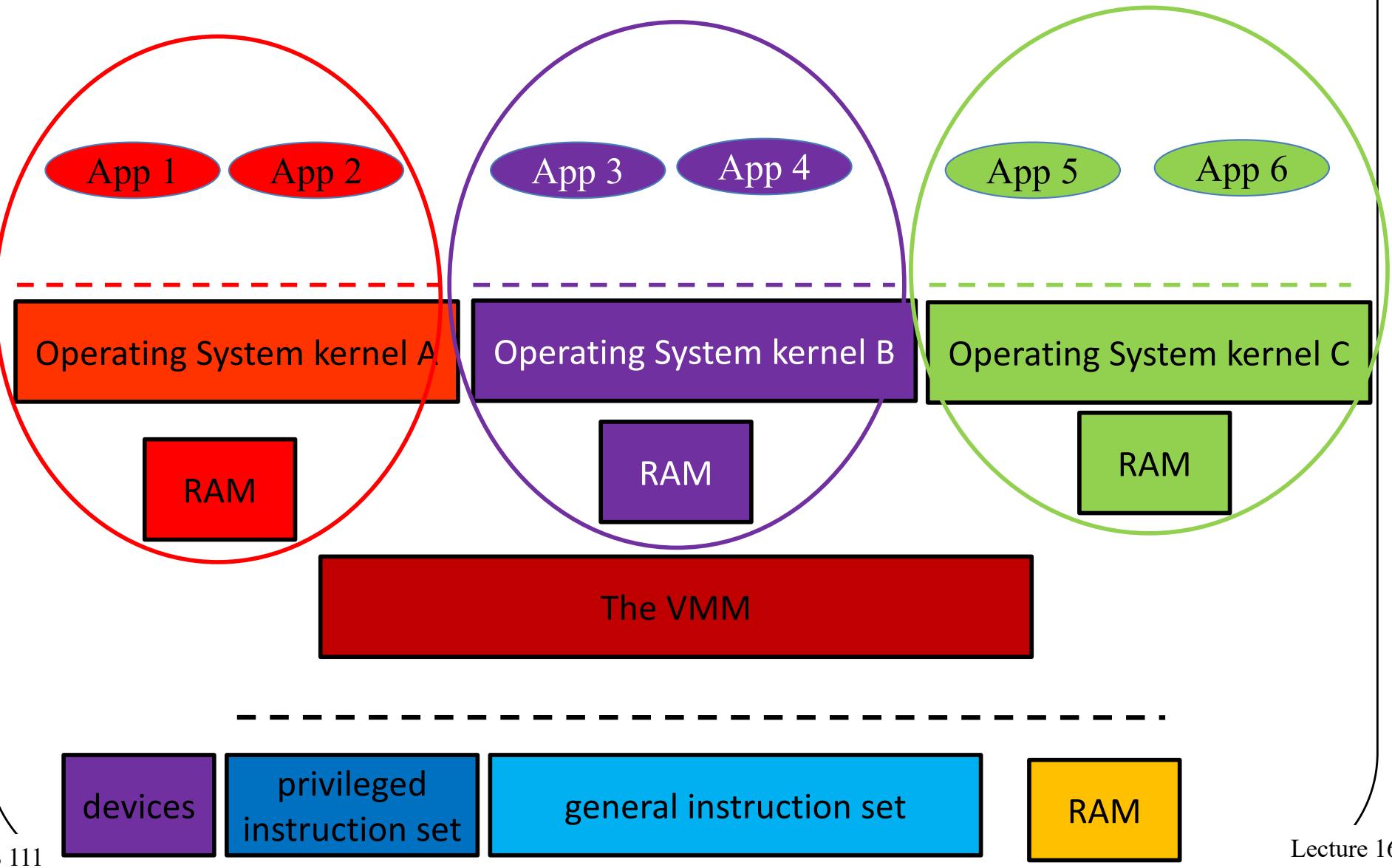
E.g., stop App 1 from killing App 2?



The Core of the Problem

- OS A thinks it's in control
- OS A believes it's providing segregated virtual memories to App 1 and App 2
- The key technology for doing so is managing page tables and CPU registers pointing to them
- But OS A has no control over those registers
 - The VMM does
- But the VMM knows nothing of the page tables OS A “controls”

Virtualizing the Memory



How To Virtualize Memory

- The virtual OS thinks it has physical memory addresses
 - It provides virtual memory addresses to its processes
 - Handling the virtual-to-physical translations
- The VMM has *machine addresses*
 - Which it translates to physical addresses within a single VM
 - Still using the same paging hardware

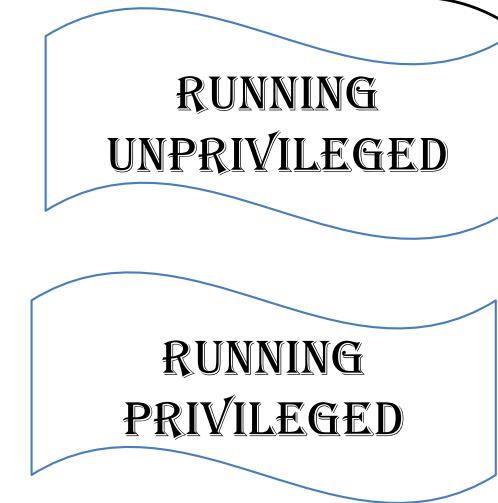
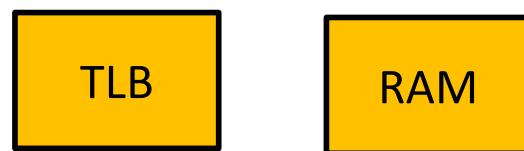
For Example

App 1 issues
virtual address X



Causing a TLB miss
and a trap

The VMM
invokes OS A



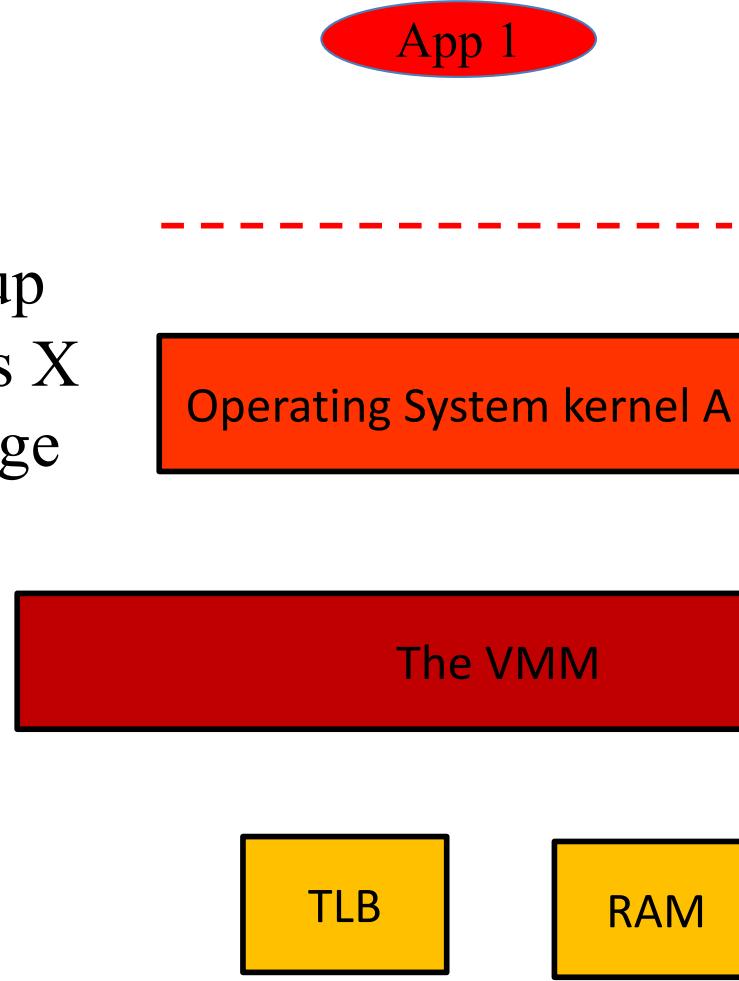
*Since only OS A
understands App
1's page table*

The VMM
catches the
trap

Continuing

And we eventually unwind to run App 1

OS A looks up virtual address X in App 1's page table
And tries to install the physical page number for X in the TLB



RUNNING UNPRIVILEGED

RUNNING PRIVILEGED

The VMM installs the right machine address for X in the TLB Which causes another trap to the VMM

Looked at Another Way

Some page frame
actually contains
page X



Who knows which
page frame?

OS A knows that



So the VMM must
consult OS A to
perform the
translation

But the VMM
doesn't know about
App 1's address
space



The VMM, since it
controls all page
frames

Some Outcomes

- TLB misses are much more expensive
 - Since we'll be moving back and forth from privileged mode to unprivileged
 - Paying overhead costs each time
 - And we'll run more systems code
- We'll need extra paging data structures in the VMM
 - More overhead
- Virtual machines are thus likely to suffer performance penalties

Making VMs Perform Better

- Adding special hardware
 - Some CPUs have features to make issues of virtualizing the CPU and memory cheaper
- Paravirtualization
 - The basic VM approach assumes the guest OSes in VMs don't know about virtualization
 - If you make some changes to those OSes, they can help make virtualization cheaper

Virtual Machines and Cloud Computing

- Cloud computing is about sharing hardware among multiple customers
- The cloud provider sells/rents computing power to customers
 - Handling all the difficult issues for them
 - So they can just run their applications
- Cloud providers need lots of customers, to make money
 - Which implies they need lots of hardware

The Cloud Environment



A warehouse full of vast
numbers of machines
Typically tens of thousands
Packed tightly into racks

Connected by high speed internal networks
And connected to the Internet, to allow customers remote access
The expectation is that the environment will run applications for
many separate customers at a time
Many of which might require multiple computers to run properly
With strong guarantees of isolation between customers

But Why VMs in the Cloud?

- The cloud provider makes the most money by making the most efficient use of the hardware
 - More customers on the same amount of hardware
= more profits
- Often, a customer doesn't need the full power of a machine
 - You make more money by using part of that machine for another customer
- But you need strong isolation
- Like that provided by virtual machines . . .

So . . .

- You run everyone in a virtual machine
- Some customers have many virtual machines to handle their big jobs
- Some customers' virtual machines share physical machines with other customers' VMs
- Customers' work loads fluctuate
- You want the most efficient packing of VMs onto physical machines possible
 - To maximize profits

How To Efficiently Place VMs

- There are many physical nodes and many more VMs
- Which do we put where?
- Often reduces to a bin packing algorithm
- Which tends to be NP-hard
 - Where n may depend on the number of servers and/or VMs considered
 - The more factors considered, the harder to solve
- So estimation techniques are used
- VM migration supported to correct mistakes or adjust to changing workloads

VMs Aren't Just For Cloud Computing

- As you should know, since your projects use them
- They allow experimentation not easily performed on real hardware
- They allow basic servers to safely divide their resources
- They allow greater flexibility in the software your computer can run

Conclusion

- Virtual machines are a critical technology for modern computing
- Virtual machines are implemented on real machines
- The key issue is providing each VM the illusion of complete control
- While also providing good performance
- VMs are of special importance in cloud computing

Operating System Principles: Distributed Systems

CS 111

Winter 2023

Operating System Principles
Peter Reiher

Outline

- Introduction
- Distributed system paradigms
- Remote procedure calls
- Distributed synchronization and consensus
- Distributed system security
- Accessing remote data

Introduction

- Why do we care about distributed systems?
 - Because that's how most modern computing is done
- Why is this an OS topic?
 - Because it's definitely a systems issue
 - And even the OS on a single computer needs to worry about distributed issues
- If you don't know a bit about distributed systems, you're not a modern computer scientist

Why Distributed Systems?

- Better scalability and performance
 - Apps require more resources than one computer has
 - Can we 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
- Enabling new collaboration and business models
 - Collaborations that span system (or national) boundaries
 - A global free market for a wide range of new services

A Few Little Problems

- Different machines don't share memory
 - Or any peripheral devices
 - So one machine can't easily know the state of another
- The only way to interact remotely is to use a network
 - Usually asynchronous, slow, and error prone
 - Usually not controlled by any single machine
- Failures of one machine aren't visible to other machines

Might this cause synchronization problems?

So how can we know what's going on remotely?

How can our computation be reliable if pieces fail?

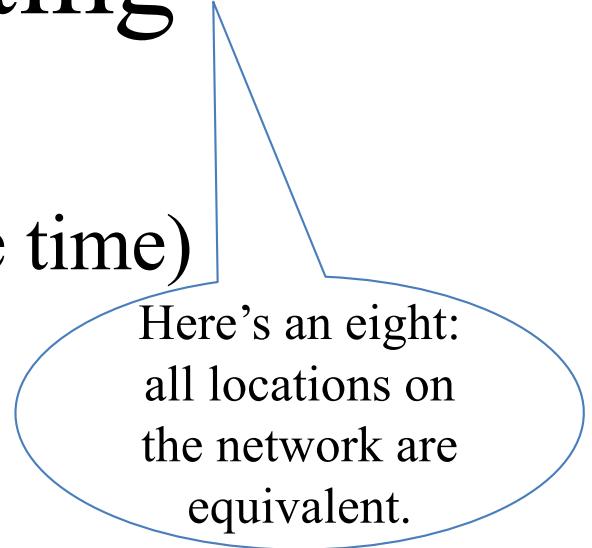
Transparency

- 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



Here's an eight:
all locations on
the network are
equivalent.

Distributed System Paradigms

- Parallel processing.
 - Relying on tightly coupled 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

Not widely used, we won't discuss them.

So these are also not popular, and we won't discuss them.

Loosely Coupled Systems

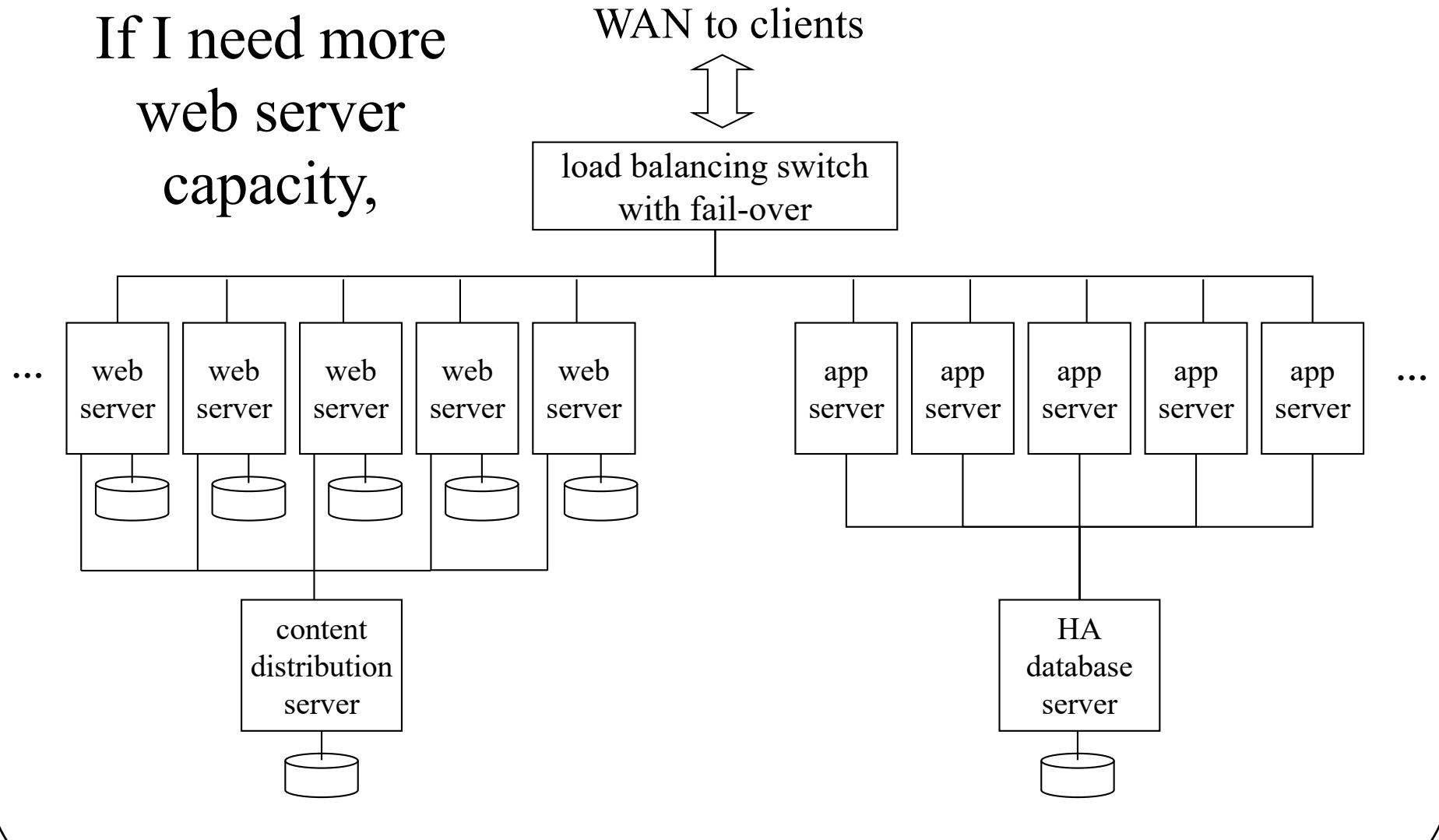
- Characterization:
 - A parallel group of independent computers
 - Connected by a high speed LAN
 - Serving similar but independent requests
 - Minimal coordination and cooperation required
- Motivation:
 - Scalability and price performance
 - Availability – if protocol permits stateless servers
 - 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

Same result if you do it once,
twice, three times, . . . , n times

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

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
 - Using a method like map-reduce or horizontal scaling
- Things where the user need not be a distributed systems expert

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

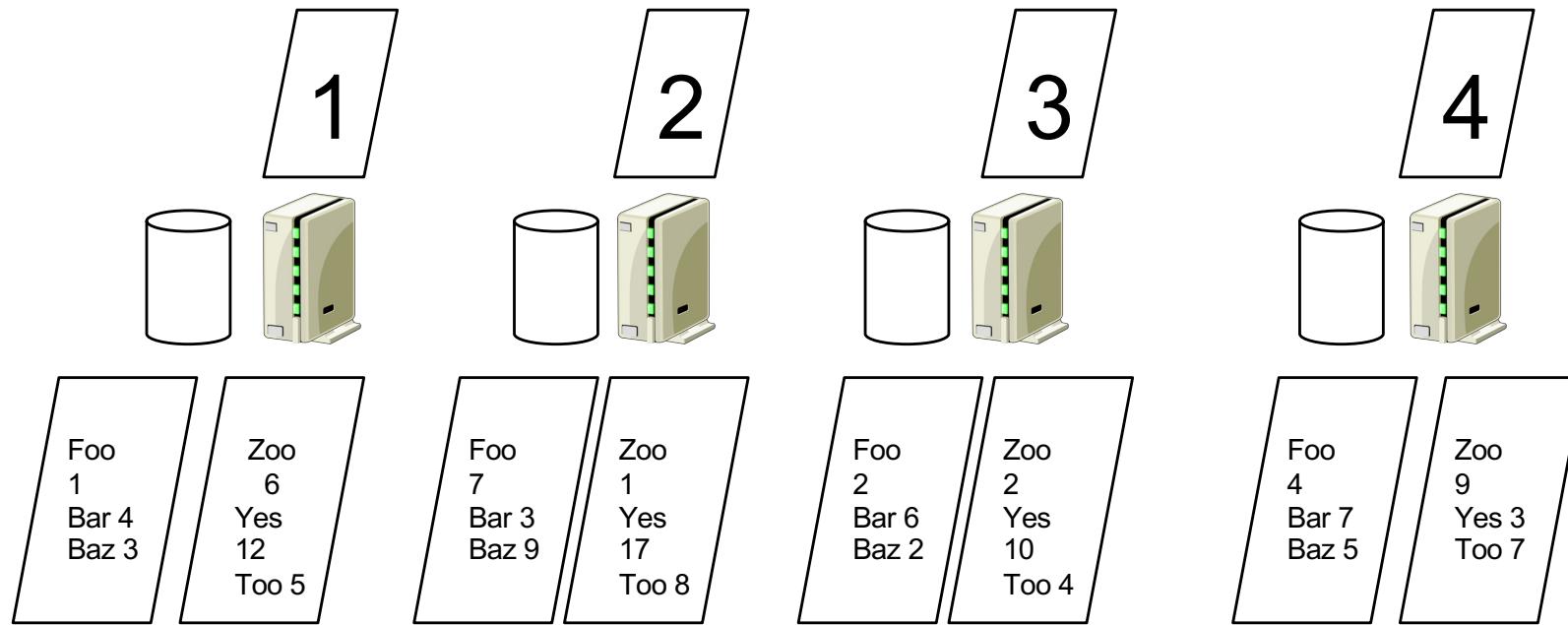
The Idea Behind MapReduce

- There is a single function you want to perform on a lot of data
 - Such as searching it for a particular 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

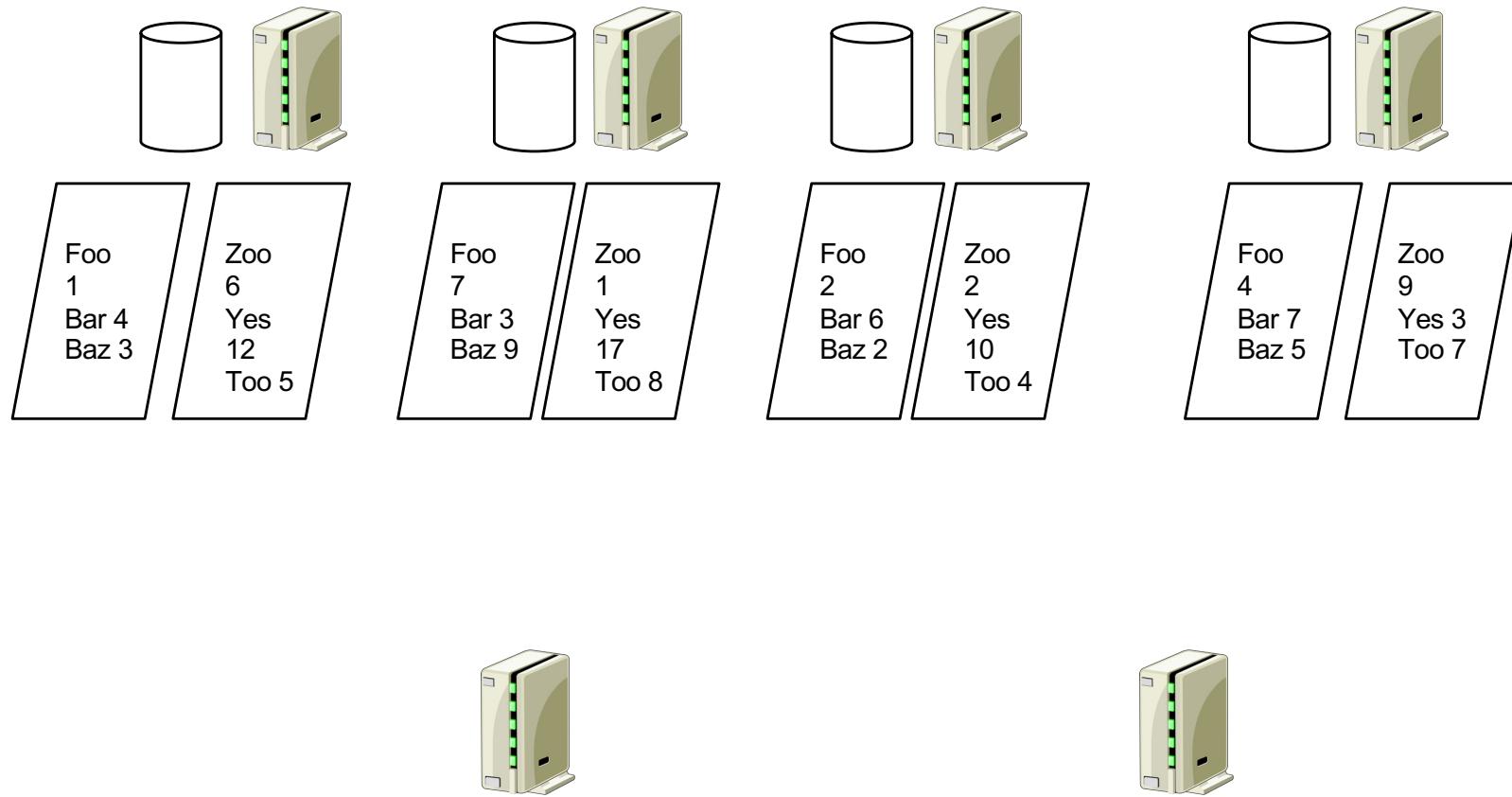


That's the map stage

On To Reduce

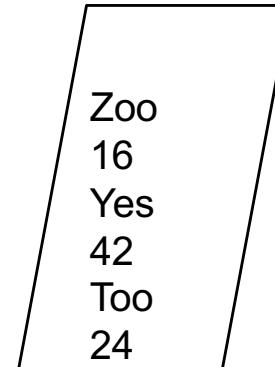
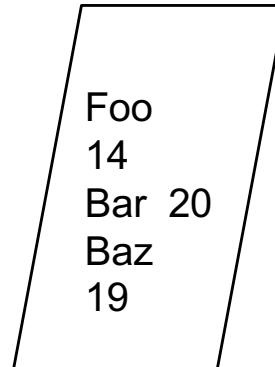
- We might have two more nodes assigned to doing the reduce operation
- They will each receive a share of data from a map node
- The reduce node performs a reduce operation to “combine” the shares
- Outputting its own result

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

Cloud Computing and Horizontal Scaling

- An excellent match
- Rent some cloud nodes to be your web servers
- If load gets heavy, ask the cloud for another web server node
- As load lightens, release unneeded nodes
- No need to buy new machines
- No need to administer your own machines

Cloud Computing and Sysadmin

- Not quite as painless as it sounds
- The cloud provider will take care of lots of the problem
 - Running the hardware
 - Fixing broken hardware
 - Loading your software onto machines
- But they won't take care of internal administration
 - E.g., updating the version of the web server you're running

Actually, they will take care of that, too, but at an extra price and with a loss of control.

Remote Procedure Calls

- RPC, for short
- One way of building a distributed program
- 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)

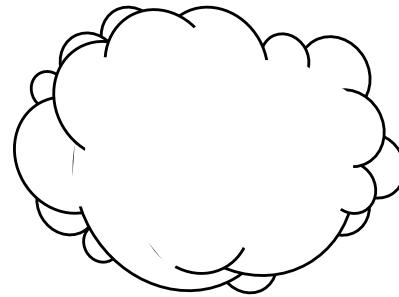
Remote Procedure Call Concepts

- Interface Specification
 - Methods, parameter types, return types
- eXternal Data Representation (XDR)
 - Machine independent data-type representations
 - May have optimizations for similar client/server
- Client stub
 - Client-side proxy for a method in the API
- Server stub (or skeleton)
 - Server-side recipient for API invocations

Key Features of RPC

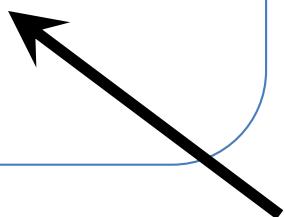
- Client application links against local procedures
 - Calls local procedures, gets results
- All RPC implementation inside those procedures
- Client application does not know about RPC
 - Does not know about formats of messages
 - Does not worry about sends, timeouts, resends
 - Does not know about external data representation
- All of this is generated automatically by RPC tools
- The key to the tools is the interface specification

RPC At Work, Step 1



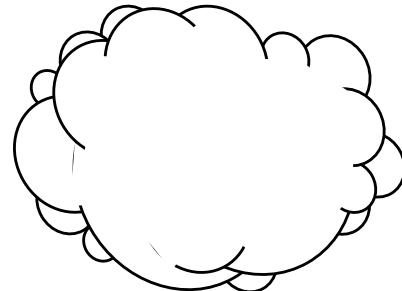
Process_list <return>

```
    . . .
list[0] = 10;
list[1] = 20;
list[2] = 17;
max = list_max(list);
```



list_max () is a remote procedure call!

RPC At Work, Step 2



```
Process_list <return>  
    . . .  
    list[0] = 10;  
    list[1] = 20;  
    list[2] = 17;  
    max = list_max(list);
```



Format RPC message

RPC message: list_max(),
parameter list

Send the message



```
local_max =  
list_max(list);
```

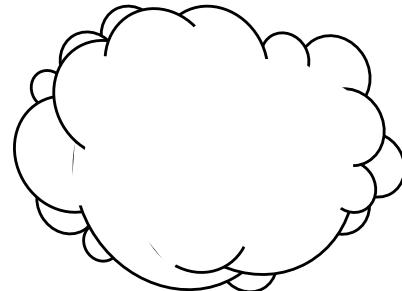
Extract RPC info

list_max()

list

Call local procedure

RPC At Work, Step 3



```
    . . .
list[0] = 10;
list[1] = 20;
list[2] = 17;
max = list_max(list);
If (max > 10) {
    max 20
```



Extract the return value

Resume the local program

```
local_max =
list_max(list);

local_max 20
```

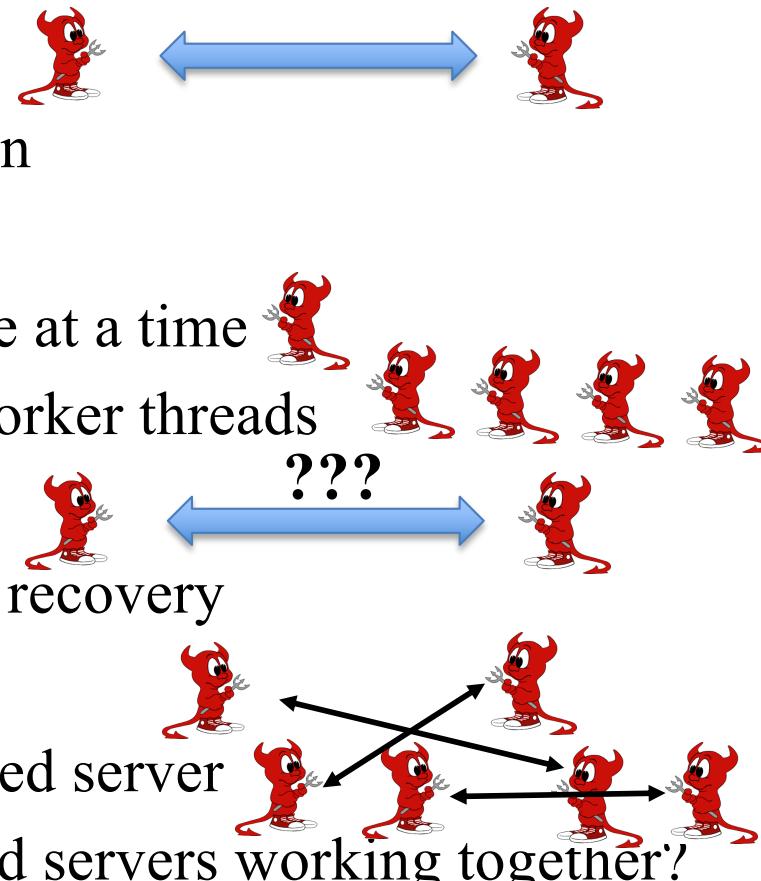


Format RPC response

RPC response: list_max(),
return value 20

Send the message

RPC Is Not a Complete Solution

- Requires client/server binding model
 - Expects to be given a live connection
 - Threading model implementation
 - A single thread services requests one at a time
 - So use numerous one-per-request worker threads
 - Limited failure handling
 - Client must arrange for timeout and recovery
 - Limited consistency support
 - Only between calling client and called server
 - What if there are multiple clients and servers working together?
 - Higher level abstractions improve RPC
 - e.g. Microsoft DCOM, Java RMI, DRb, Pyro
- 

Distributed Synchronization

- Why is it hard to synchronize distributed systems?
- What tools do we use to synchronize them?

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
 - But 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

Security for Distributed Systems

- Security is hard in single machines
- It's even harder in distributed systems
- Why?

Why Is Distributed Security Harder?

- Your OS cannot guarantee privacy and integrity
 - Network activities happen outside of the OS
 - Should you trust where they happen?
- 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

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 the hash be transmitted securely?
 - Cryptographic hashes aren't keyed, so anyone can produce them (including a bad guy)
- How to transmit hash securely?
 - Encrypt it
 - 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?

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

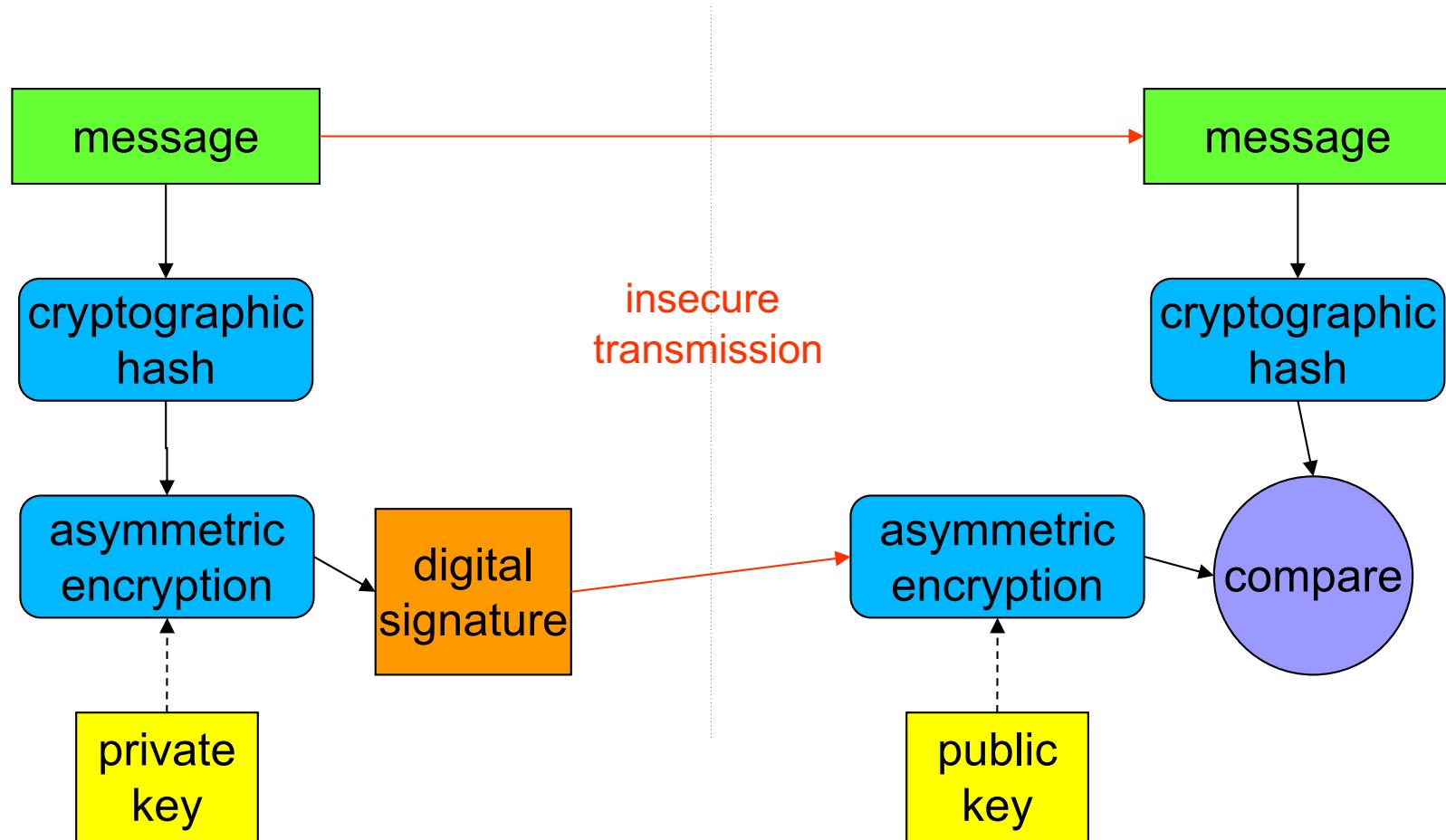
Putting It Together: Secure Socket Layer (SSL)

- 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

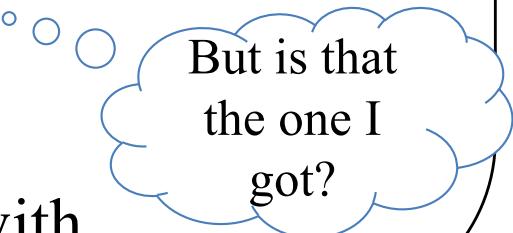
- Encrypting a message with private key signs it
 - Only you could have encrypted it, it must be from you
 - It has not been tampered with after you wrote it
- Encrypting everything with your private key is a bad idea
 - Asymmetric encryption is extremely slow
- If you only care about integrity, you 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

Digital Signatures



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 owns the private key?
 - How do I know that this is really my bank's public key?
 - Could some swindler have sent me his public 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 that the signature is correct
 - And that 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

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

Winter 2023

Operating System Principles
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, robustness, and synchronization
 - Is remote data as fast and reliable as local data?
 - Is synchronized access to remote data like local?
- Architecture
 - How is solution integrated into clients and servers?
- Protocol and work partitioning
 - How do client and server cooperate?

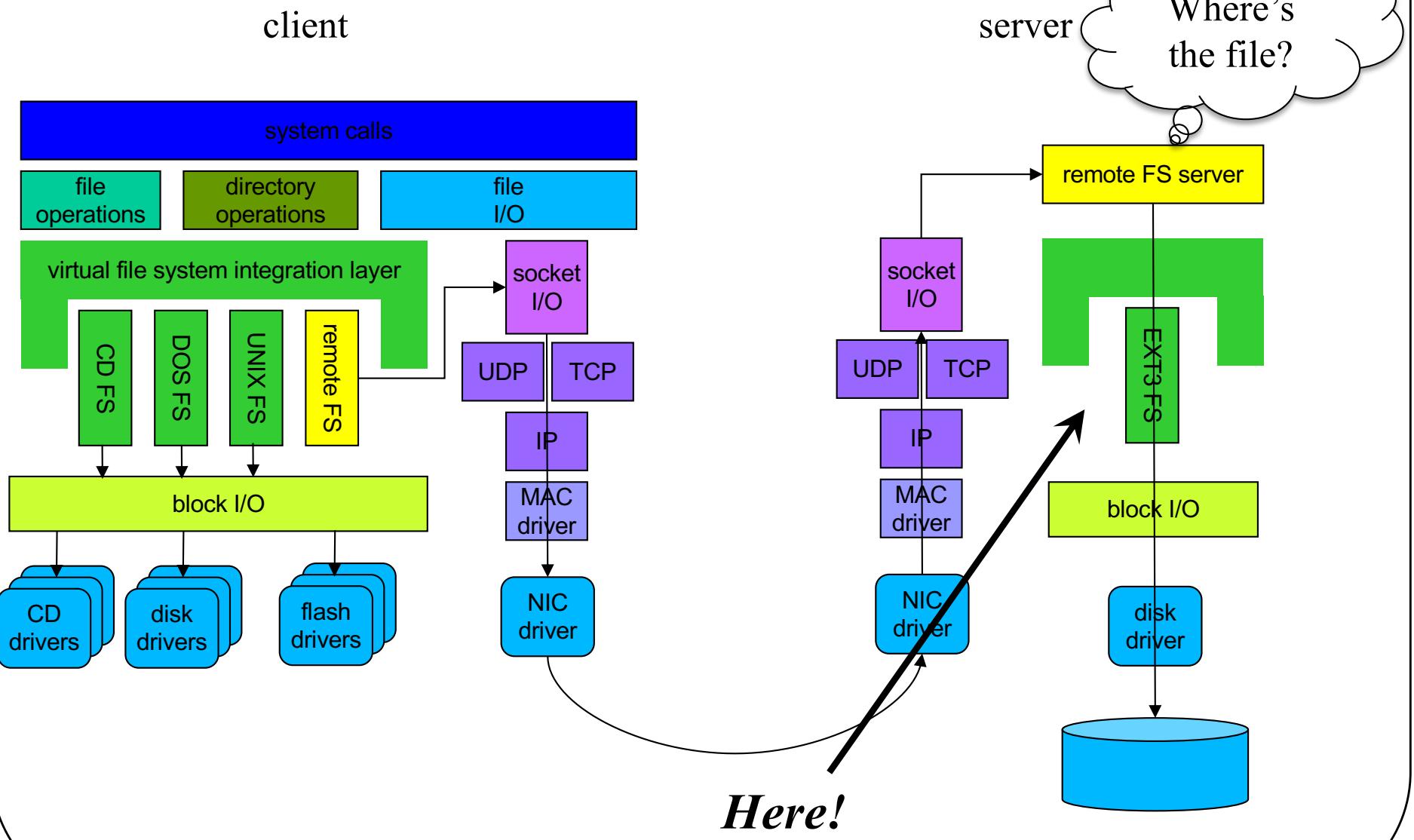
Remote File Access Architectures

- Remote file transfer
- Remote disk access
- **Remote file access**
- Distributed file system
- Cloud model

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 Storage

- A logical extension of client/server model
 - All data services accessed via standard protocols
- Opaque encapsulation of servers/resources
 - Resources are abstract/logical
 - E.g., a file system or object store
 - 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
- Clouds are not just for storage

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

Peer-to-Peer Authentication

- 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: Login-based FTP, SCP, 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

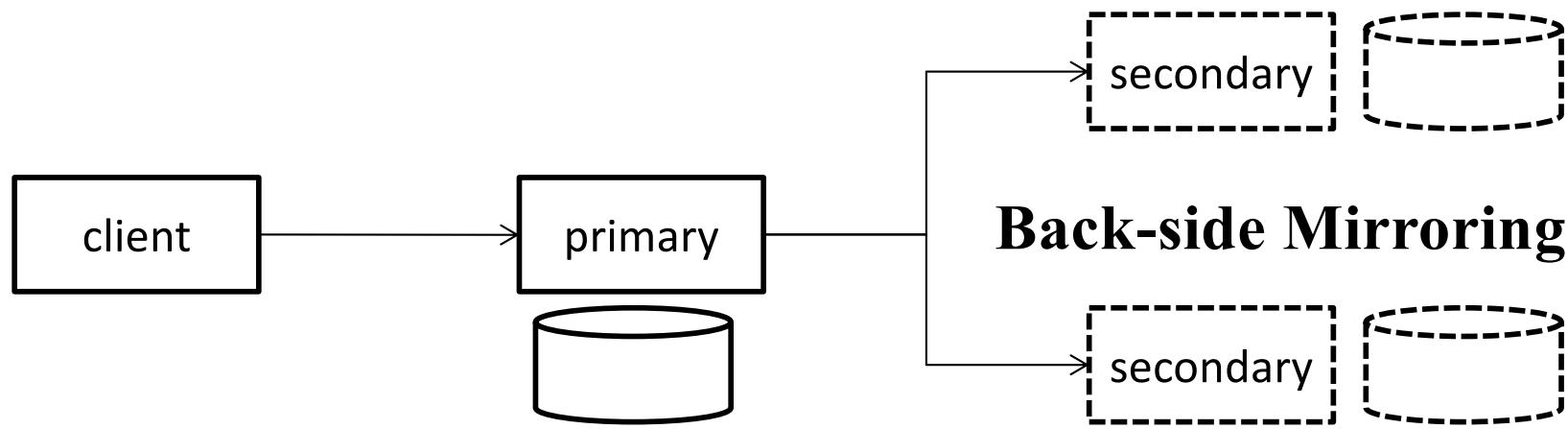
Reliability and Availability

- *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

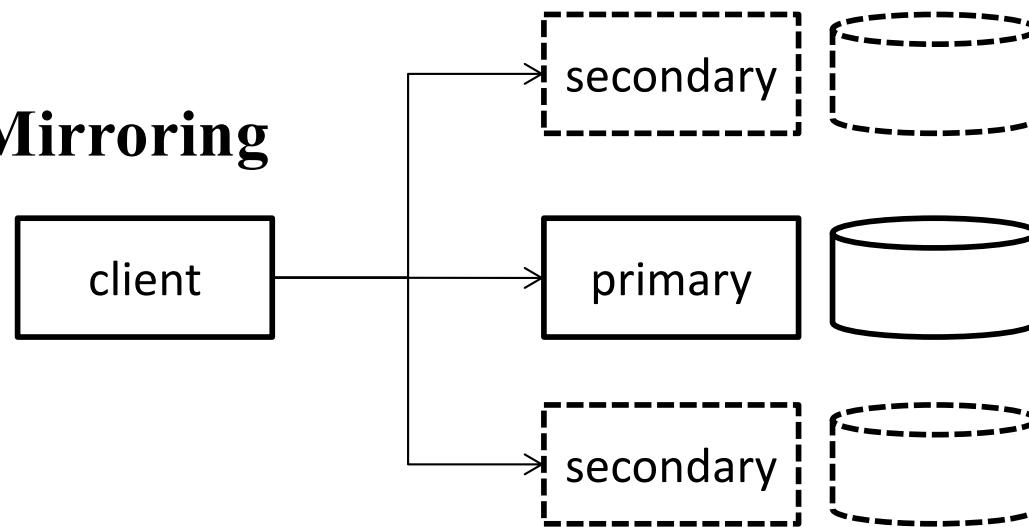
- Must reduce probability of data loss
- Typically by some form of redundancy
 - Disk/server failures must not result in data loss
 - RAID (mirroring, parity, erasure coding)
 - Copies on multiple servers
 - Backup, at the worst
- Also important to automatically recover after failure
 - Remote copies of data become available again
 - Redundancy loss due to failure must be made up

Reliability: Data Mirroring



Back-side Mirroring

Front-side Mirroring



Availability and Fail-Over

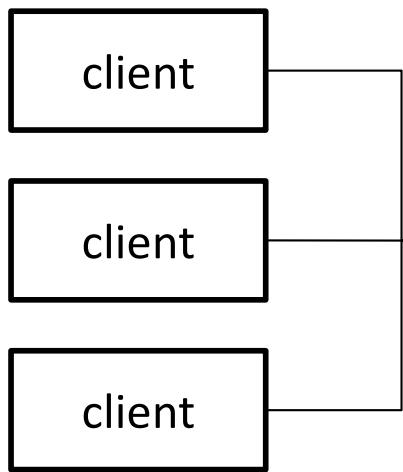
- Fail-over means transferring work/requests from failed server to some other server
- Data must be mirrored to secondary server
- Failure of primary server must be detected
- Client must be failovered to secondary
- Session state must be reestablished
 - Client authentication/credentials
 - Session 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

Idempotency
helps a lot
here.

Remote File System Performance

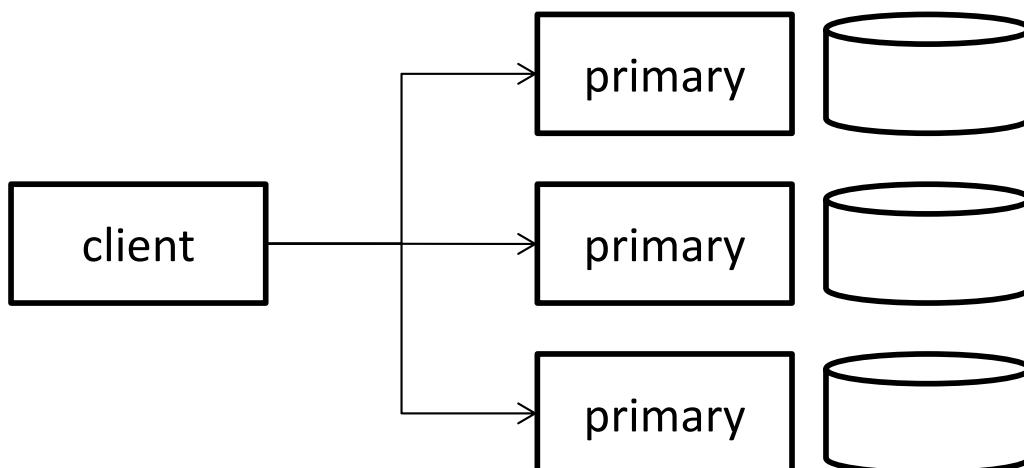
- Storage device bandwidth and performance
- Performance for reads
- Performance for writes
- Overheads particular to remote file systems

Storage Device Bandwidth Implications



a single server has limited throughput

Since any storage device has bandwidth limitations



Striping files across multiple servers provides scalable throughput

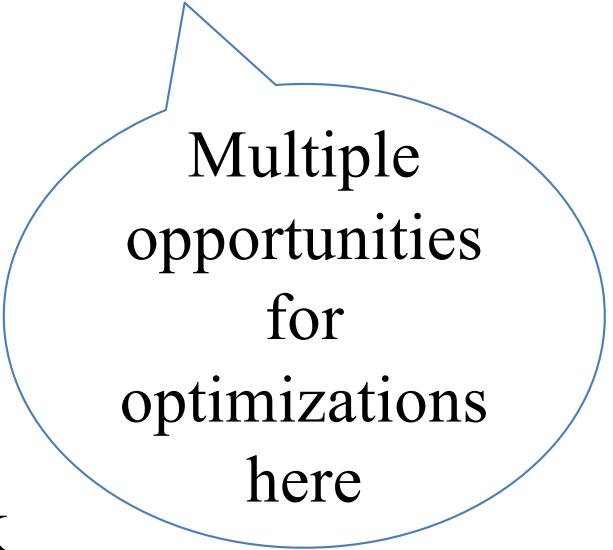
By using the bandwidth of multiple storage devices

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

Performance of Reads

- Most file system operations are reads, so read performance is critical
- Network read consists of several steps:
 - Client application requests read
 - Read request is sent via network
 - Server receives read request
 - Server fetches requested data
 - Server sends data across network
 - Client receives data and gives it to application



Multiple opportunities for optimizations here

Read Caching in Remote File Systems

- Common way to improve read performance is through caching
- Can use read-ahead, but costs of being wrong are higher than for local disk
 - Though benefits are also higher
 - Client can speculatively request reads
 - Or server can speculatively request them
 - And either send them or cache them locally

Performance 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
 - File might 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
 - What do you do when one local cached copy is updated?
 - Must you know about other cached copies?
 - Will they see the update? When?
 - What if two local cached copies are updated?

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?
 - Fewer failures
 - Quicker recovery from failures that occur

Scalability and Performance: Network Traffic

- Network messages are expensive
 - They use NIC and network capacity to carry them
 - And 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

Conclusion

- Accessing data on remote machines is key to most distributed processing
- There are major challenges to doing so:
 - Performance
 - Consistency
 - Scalability
 - Reliability
- Solutions are available, but have associated costs and drawbacks
 - None of them are perfect for all purposes