

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?

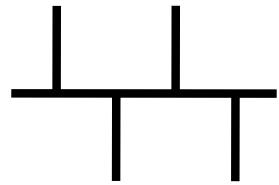
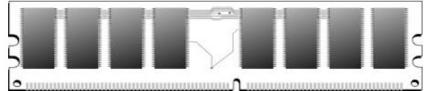
- Build on concepts from other courses
 - Data structures, programming languages, assembly language programming, computer architectures, ...
- Prepare you for advanced courses
 - Data bases, data mining, and distributed computing
 - Security, fault-tolerance, high availability
 - Network protocols, computer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

Why Study Operating Systems?

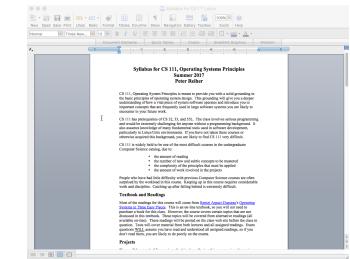
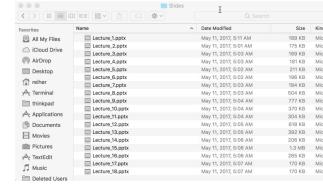
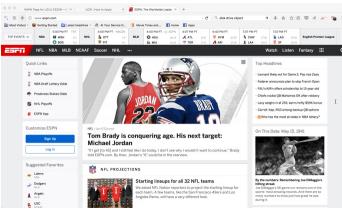
- Why do we have them, in the first place?
- Why are they important?
- What do they do for us?

Starting From the Bottom

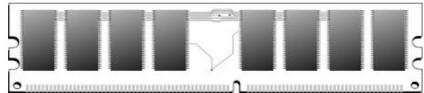
Here's
what
you've
got



Here's
what
you
want



What Can You Do With What You've Got?



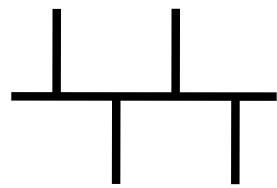
Read or write some binary words



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

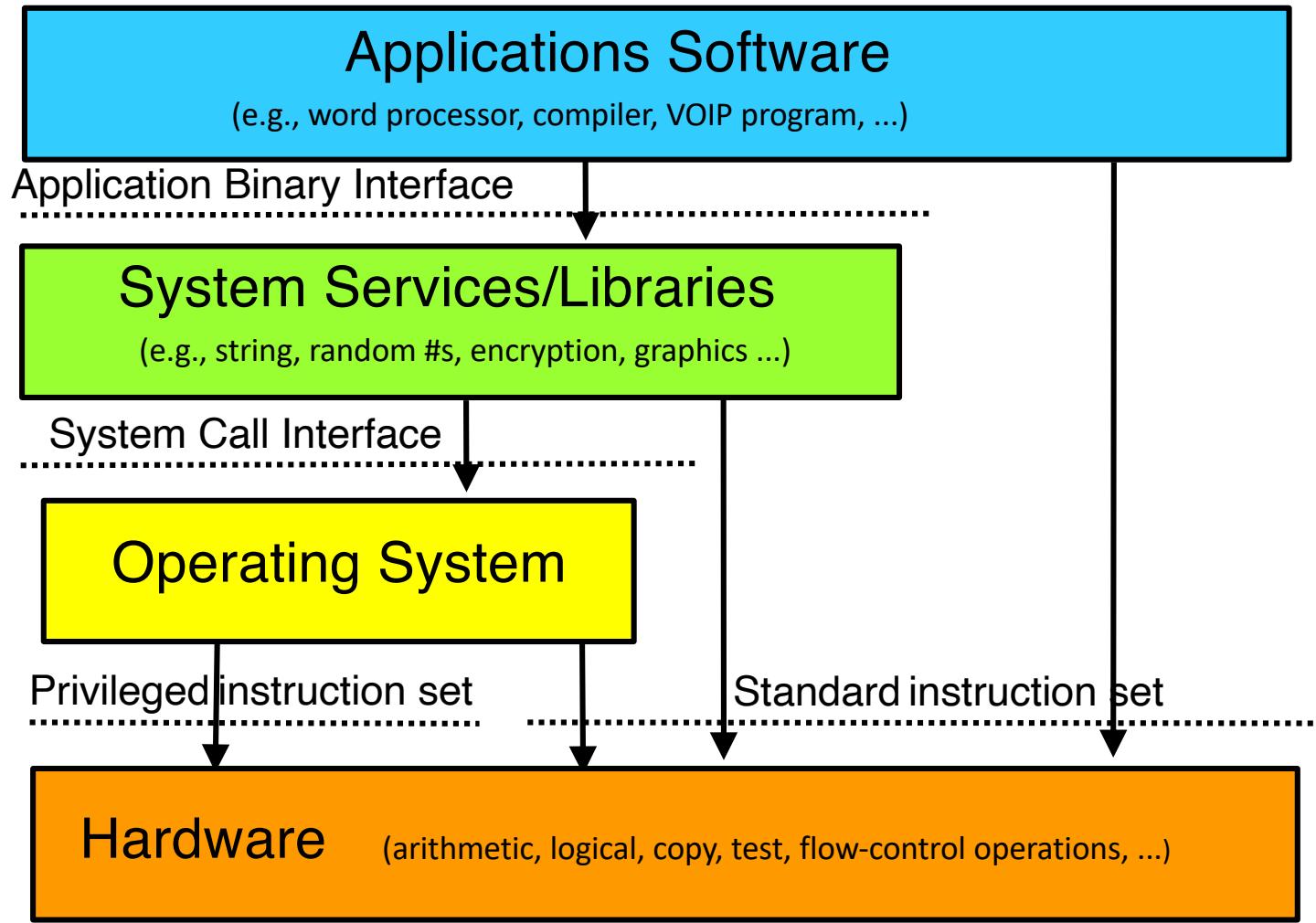
What Does an OS Do?

- 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

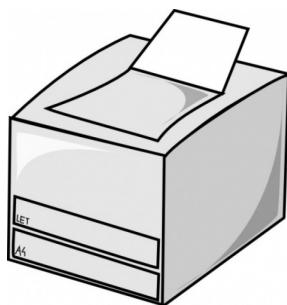
- Lots of variations in machines' HW and SW
- Make many different types appear the same
 - So applications can deal with single common class
- Usually involves a common unifying model
 - E.g., portable document format (pdf) for printers
 - Or SCSI standard for disks, CDs and tapes
- For example:
 - Printer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

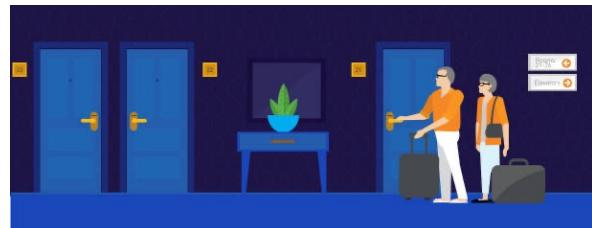
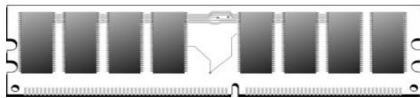


What Is A Graceful Transition?

- 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?

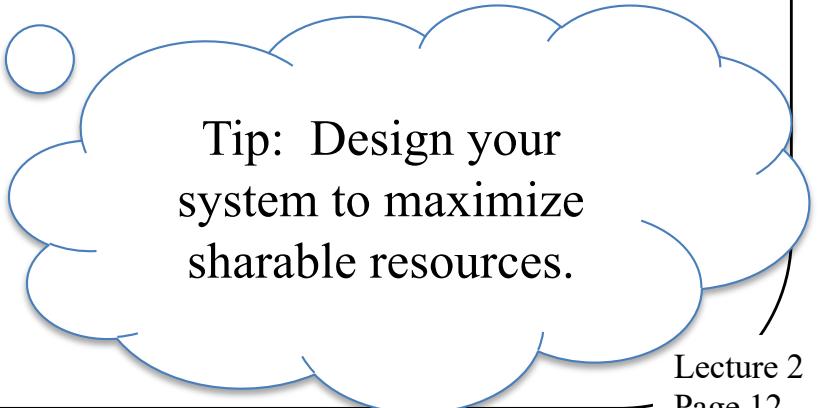
- Yes
- Most partitionable resources aren't permanently allocated
 - The piece of RAM you're using now will belong to another process later
- As long as it's “yours,” no transition required
- But sooner or later it's likely to become someone else's

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

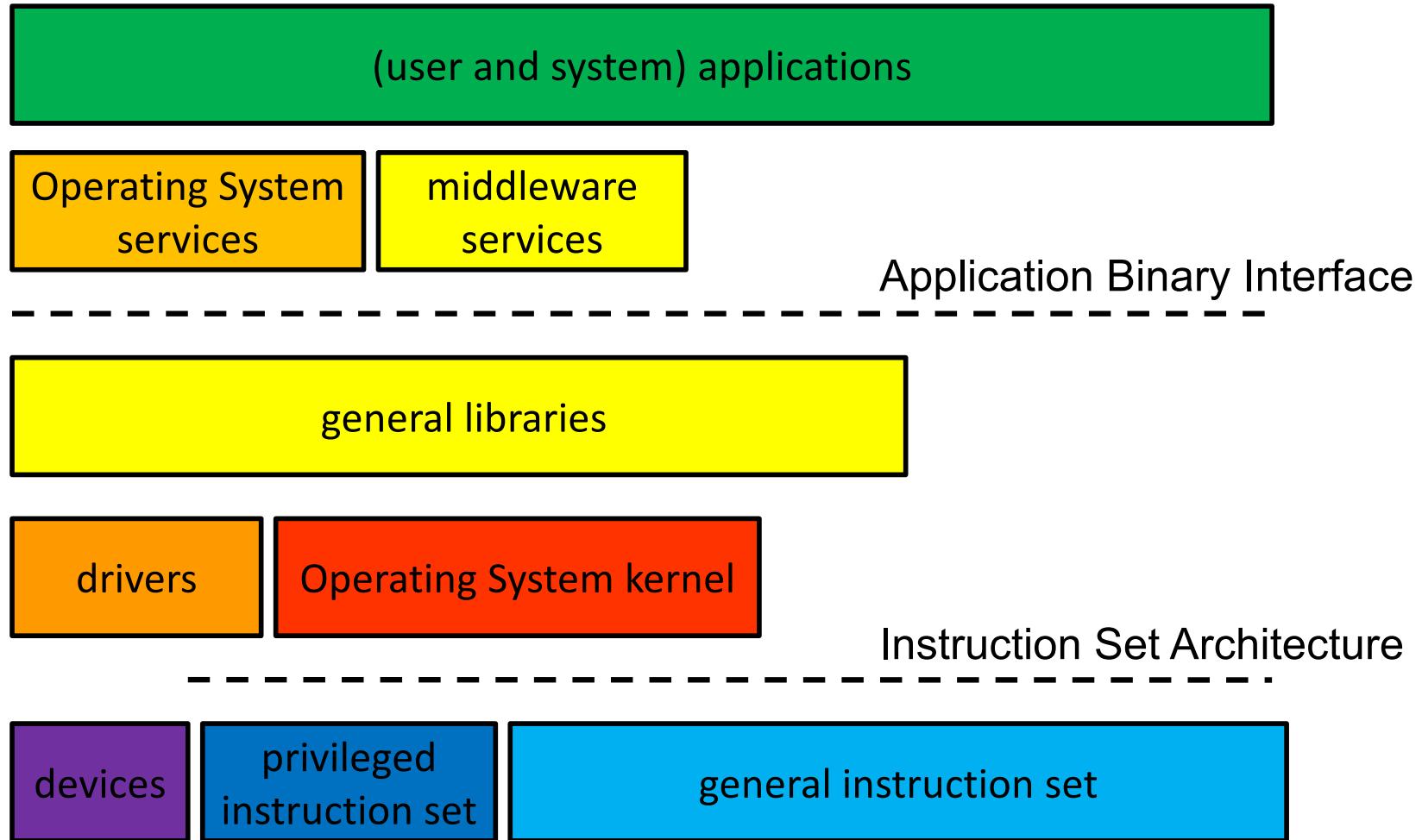
How Can the OS Deliver These Services?

- 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



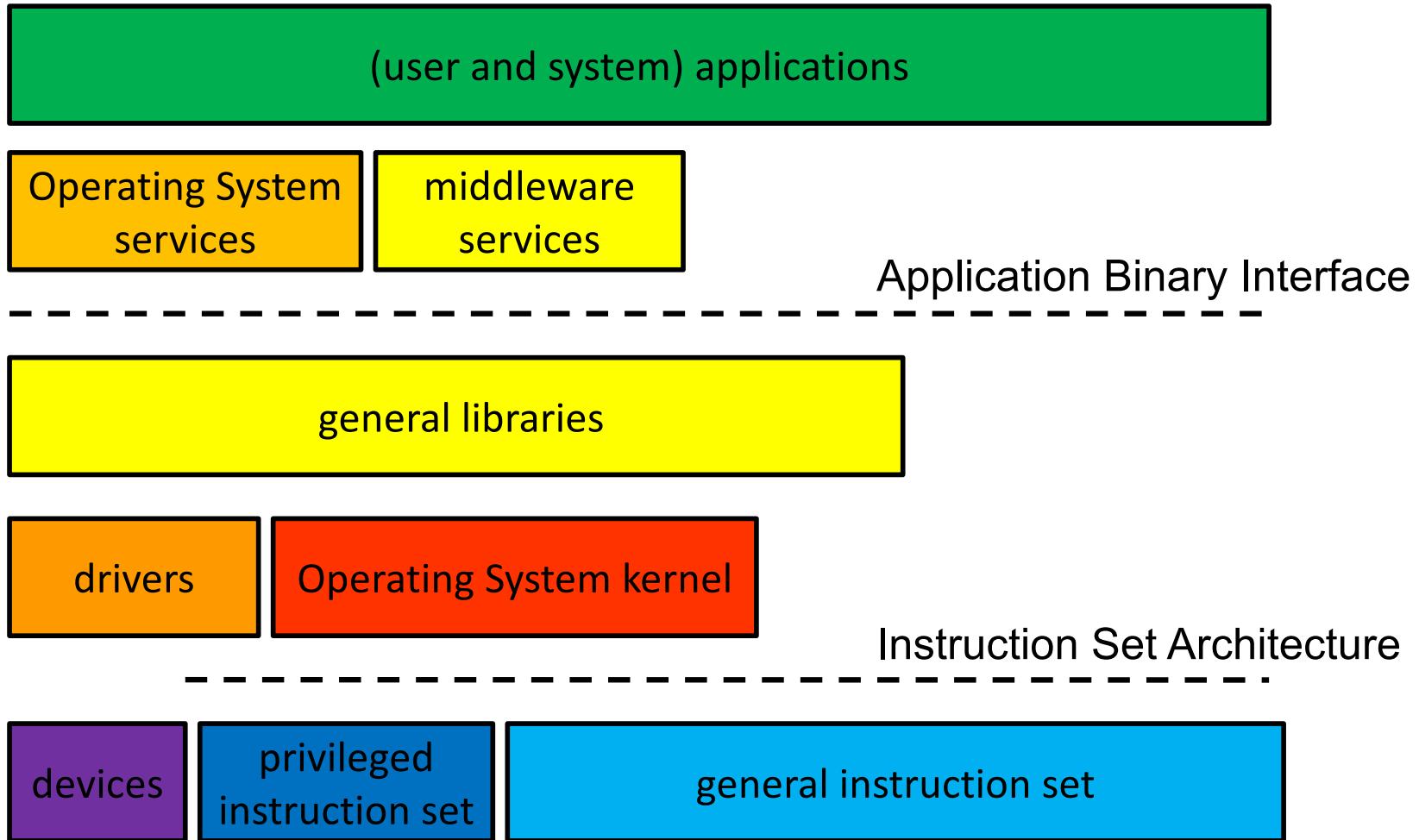
Service Delivery via Subroutines

- 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

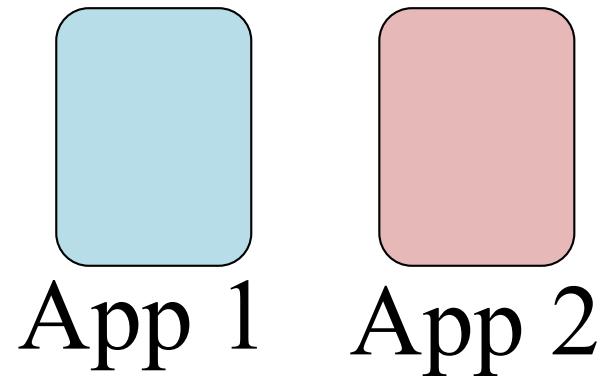
Advantages of Shared Libraries

- 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

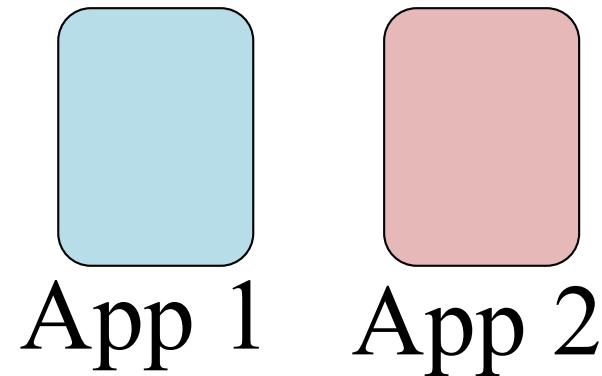
- Not all modules will work in a shared library
 - They cannot define/include global data storage
- They are added into program memory
 - Whether they are actually needed or not
- Called routines must be known at compile-time
 - Only the fetching of the code is delayed 'til run-time
 - Symbols known 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

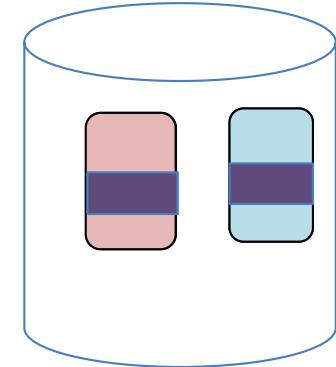
Static Libraries



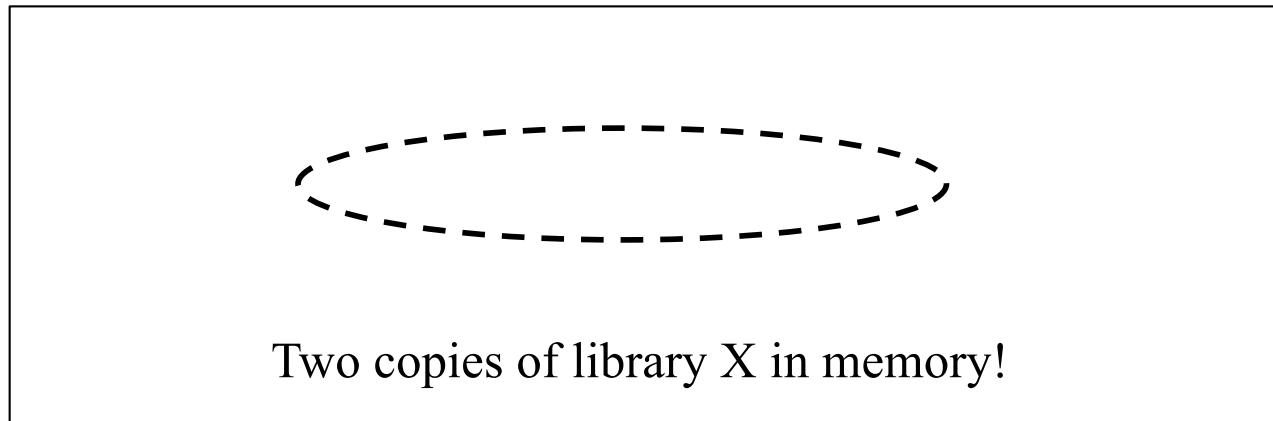
Compile App 1 Compile App 2
Run App 1 Run App 2



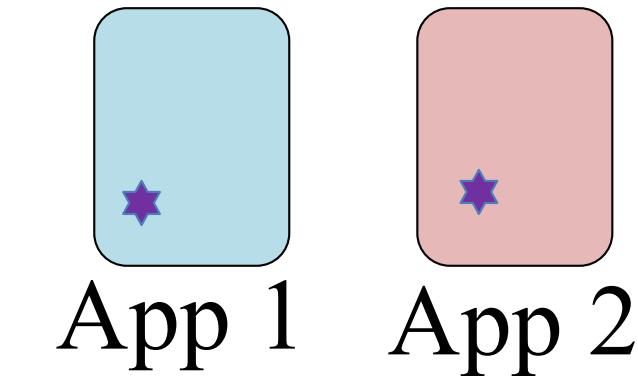
Library X



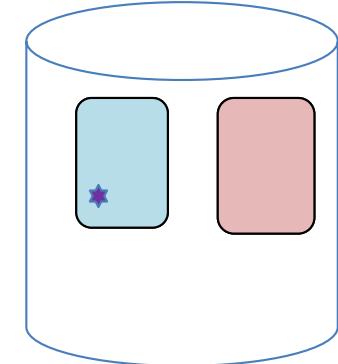
RAM



Shared Libraries



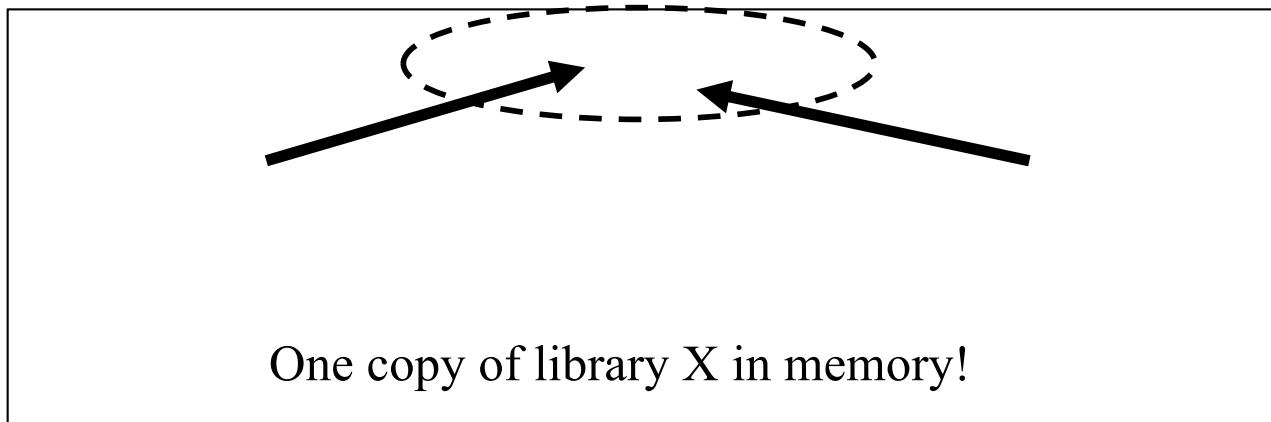
Library X



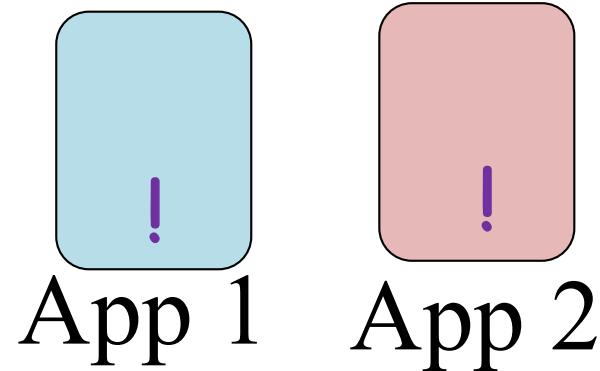
Secondary
Storage

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

RAM

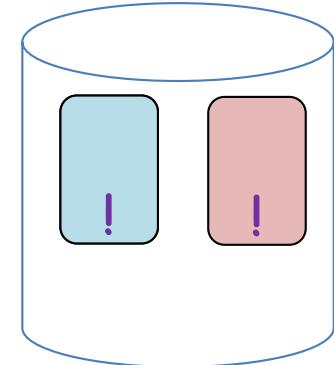


Dynamic Libraries

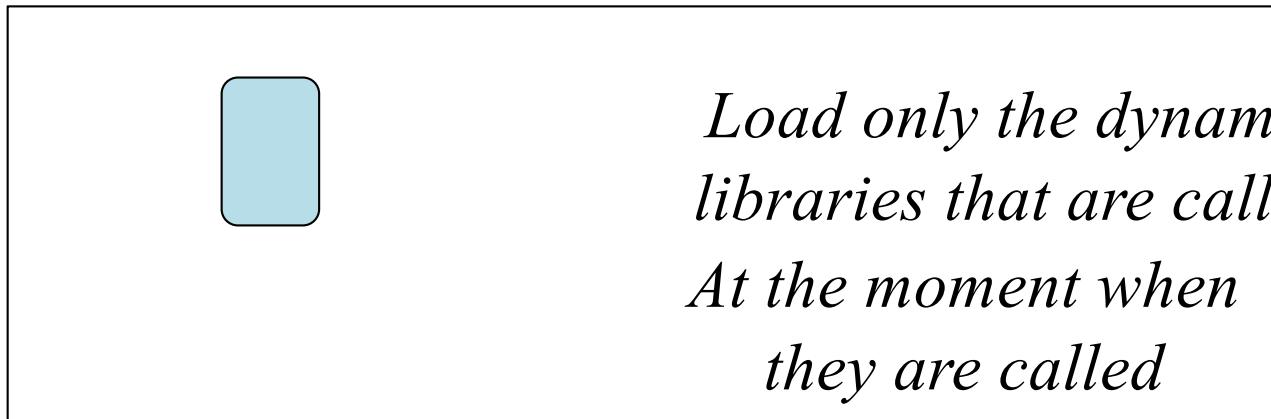


Compile App 1 Compile App 2

Run App 1 App 1 calls library function



RAM



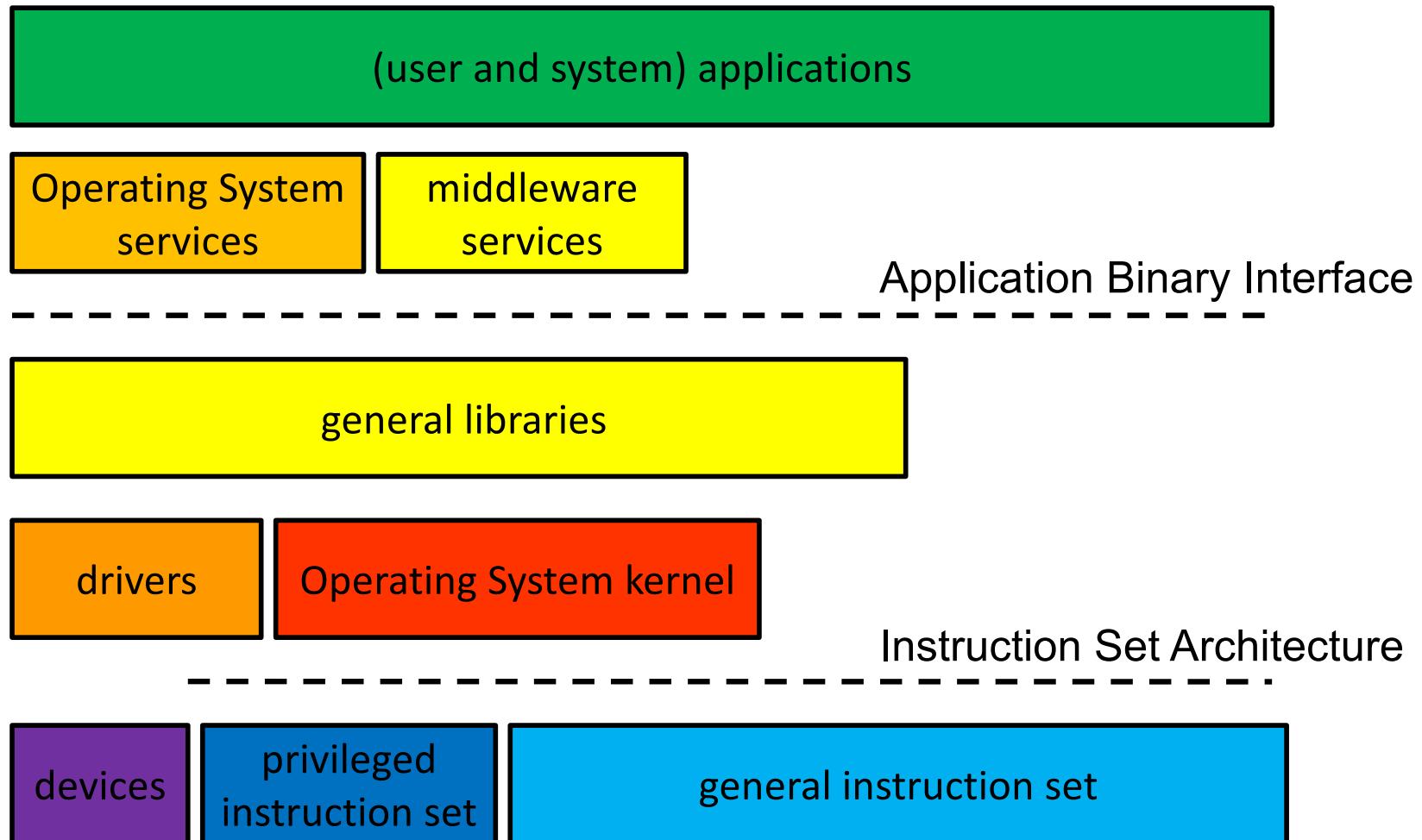
Service Delivery via System Calls

- 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

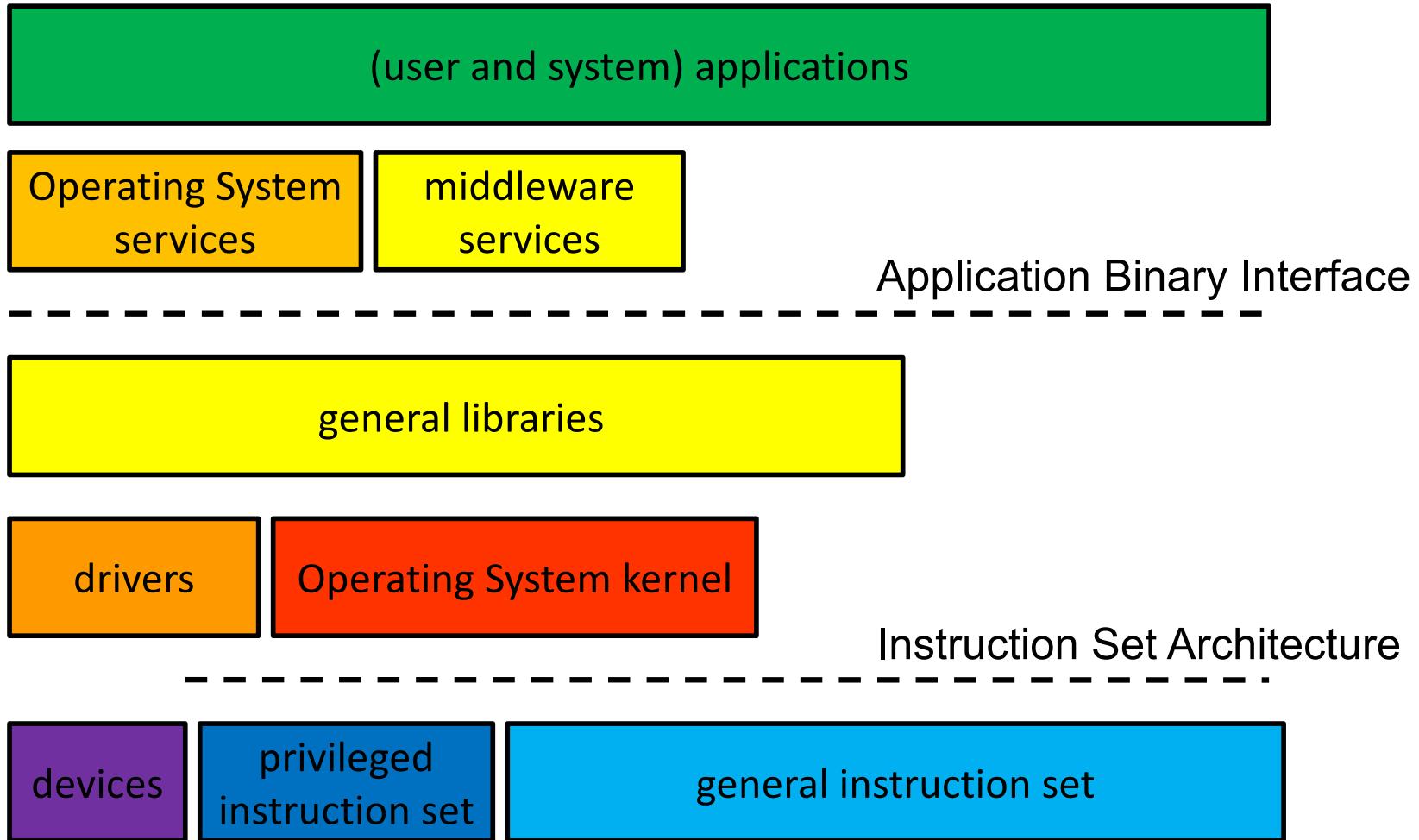
The Kernel Layer



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



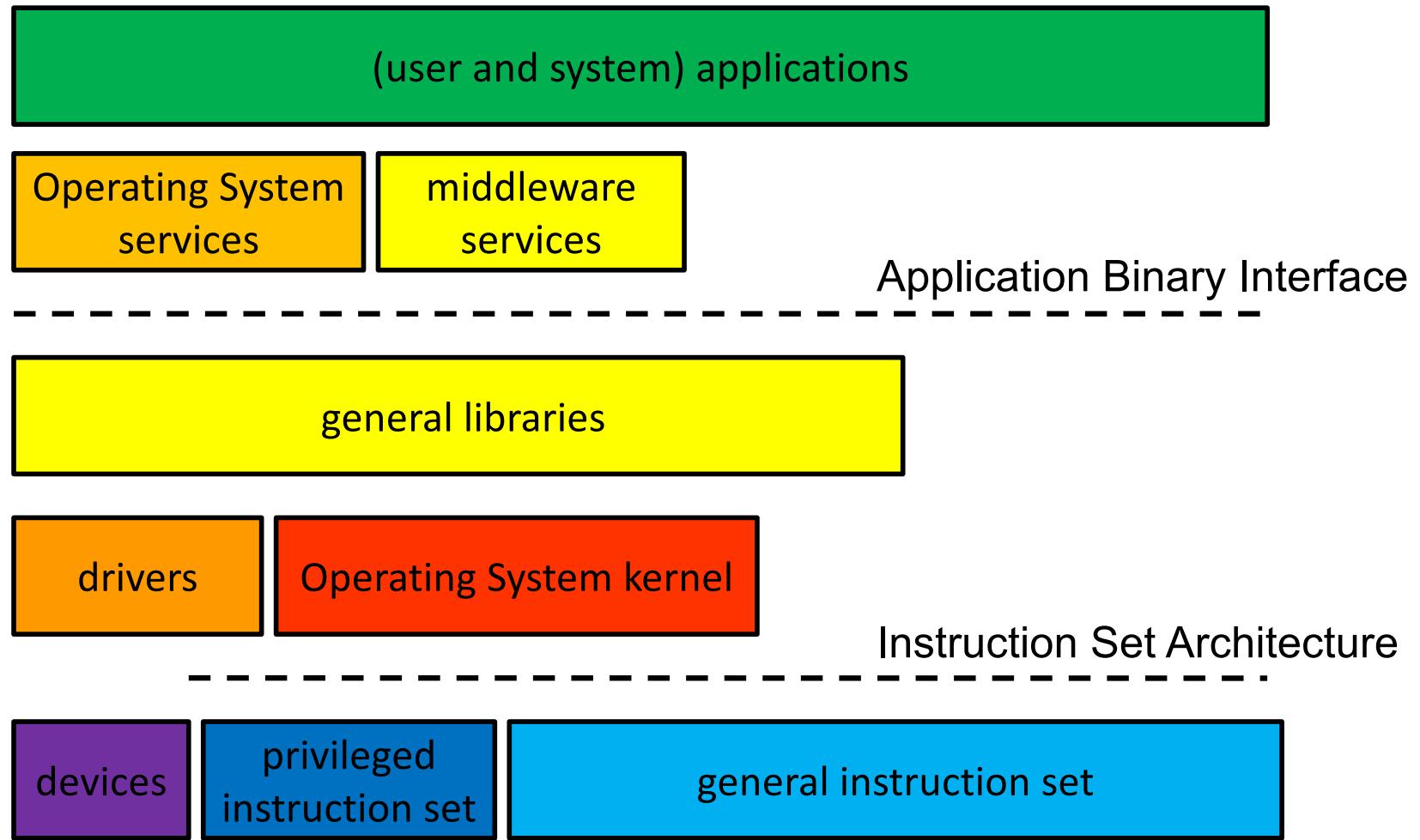
Service Delivery via Messages

- 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)

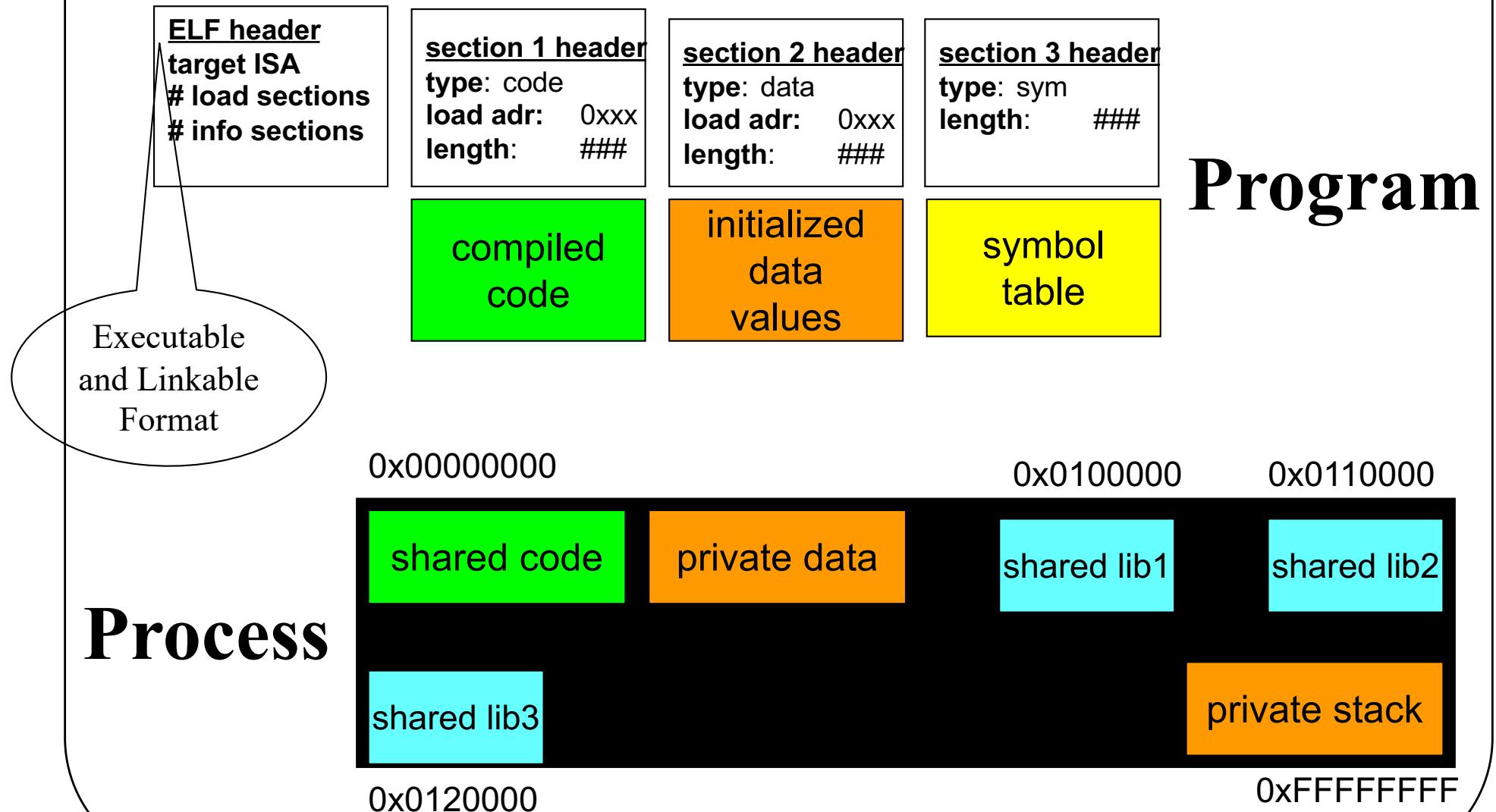
Examples Of OS Object State

- 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

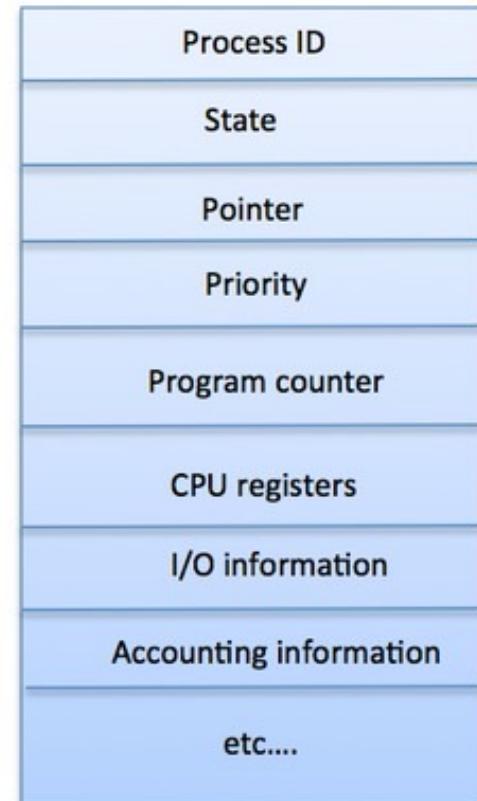
- Registers
 - General registers
 - Program counter, processor status, stack pointer, frame pointer
- Process' own OS resources
 - Open files, current working directory, locks
- But also OS-related state information
- The OS needs some data structure to keep track of all this information

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.

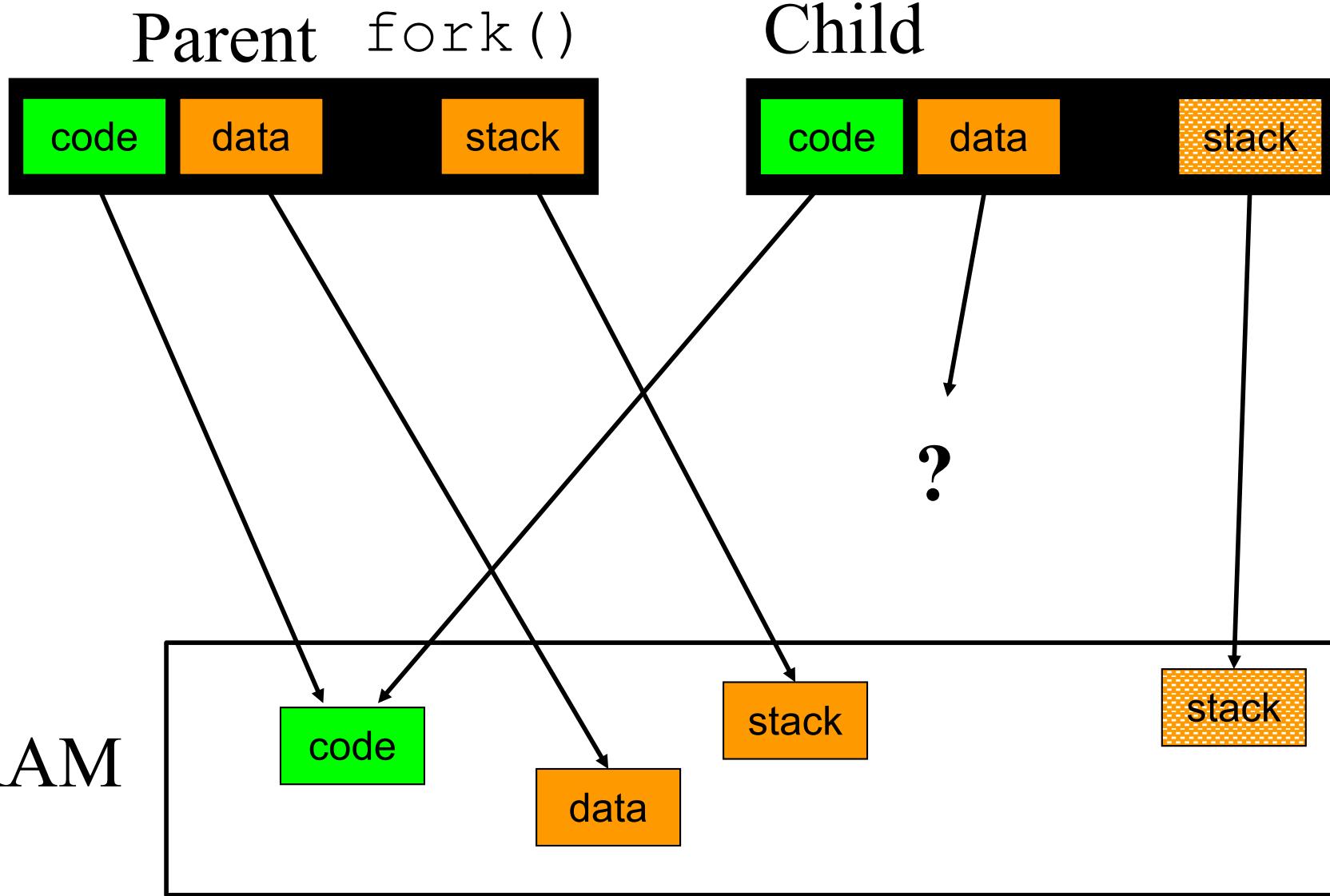
Process Forking

- 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



Forking and the Data Segments

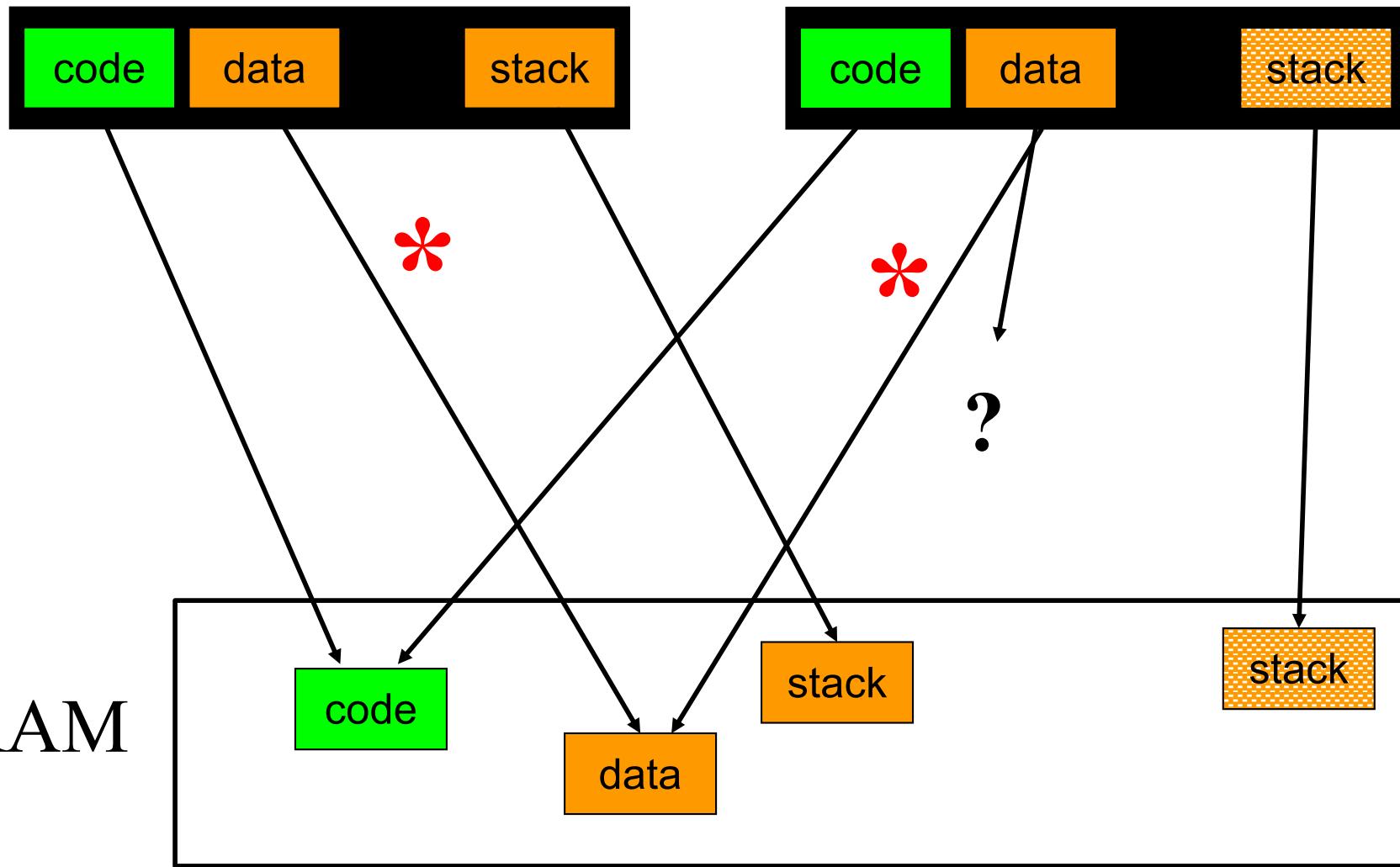
- Forked child shares the parent's code
- But not its stack
 - It has its own stack, initialized to match the parent's
 - Just as if a second process running the same program had reached the same point in its run
- Child should also have its own data segment
 - Forked processes do not share their data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

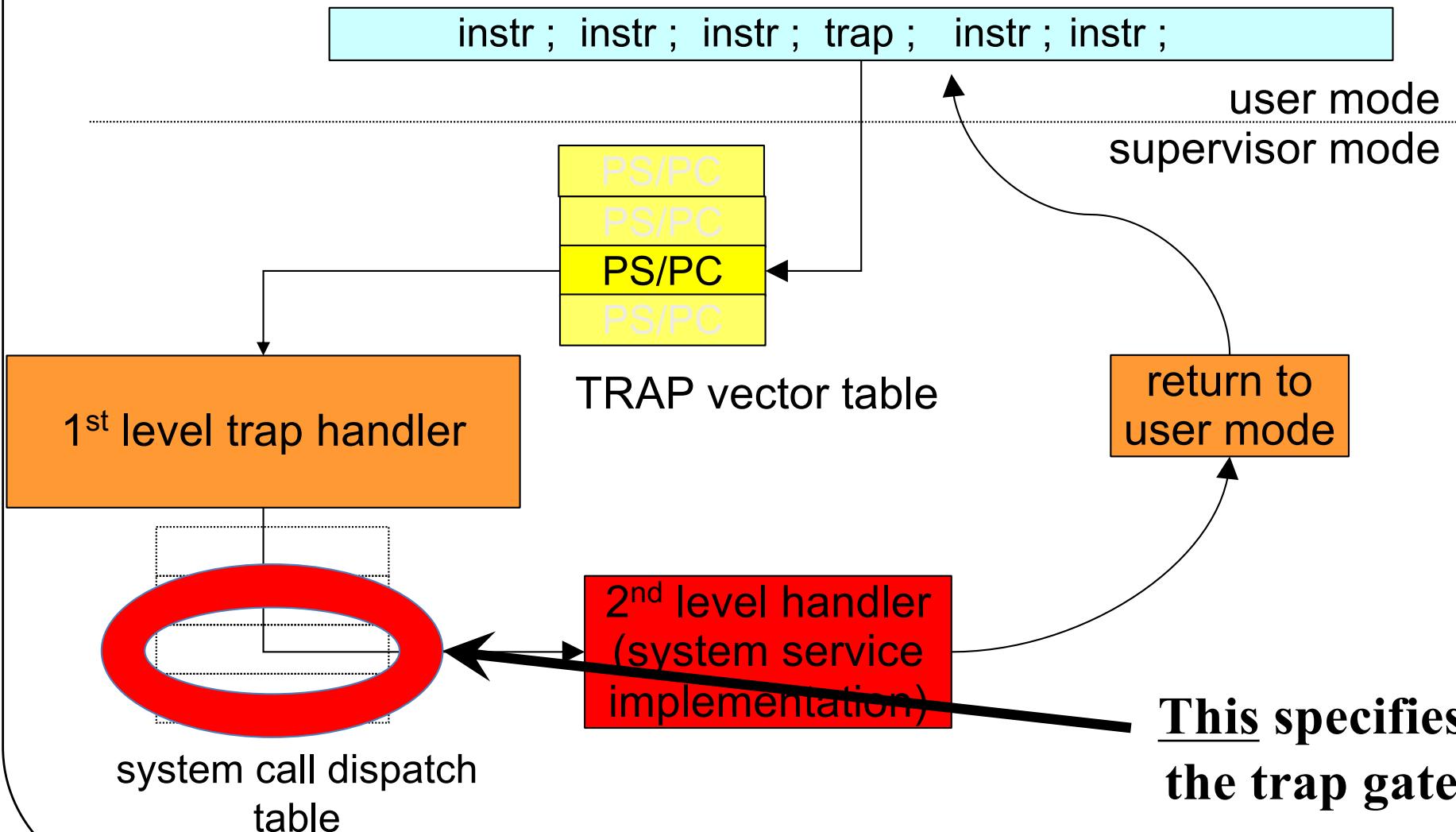
- Inherently unpredictable
- Programs can't check for them, since no way of knowing when and if they happen
- Some languages support try/catch operations
- Hardware and OS support traps
 - Which catch these exceptions and transfer control to the OS
- Operating systems also use these for *system calls*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

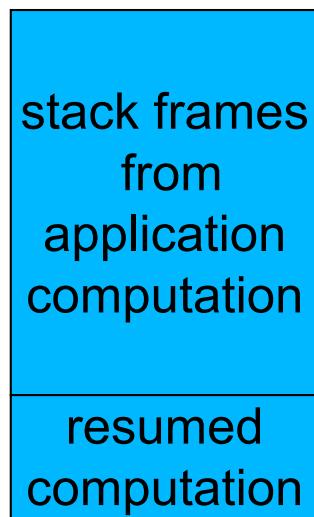
Where do you
think this table
came from?

Traps and the Stack

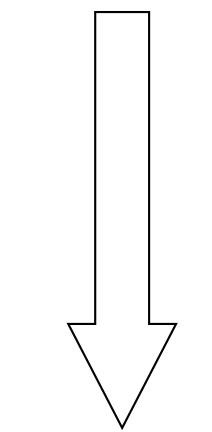
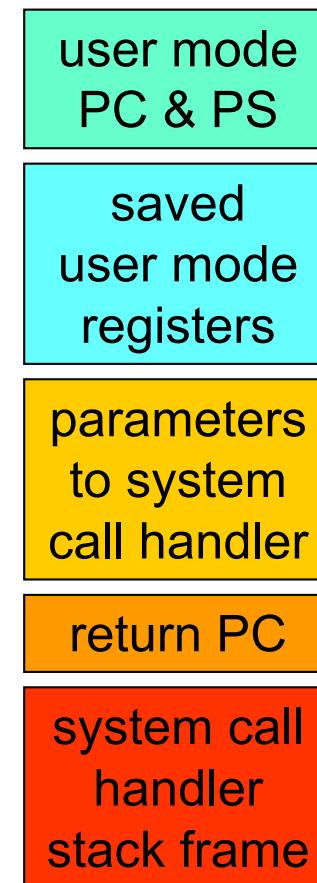
- The code to handle a trap is just code
 - Although run in privileged mode
- It requires a stack to run
 - Since it might call many routines
- How does the OS provide it with the necessary stack?
- While not losing track of what the user process was doing?
- Or leaving sensitive data in the user's stack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

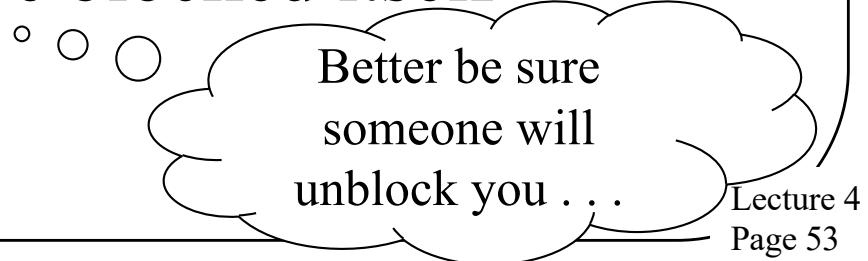
Which implies that they can screw these up if they aren't careful

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

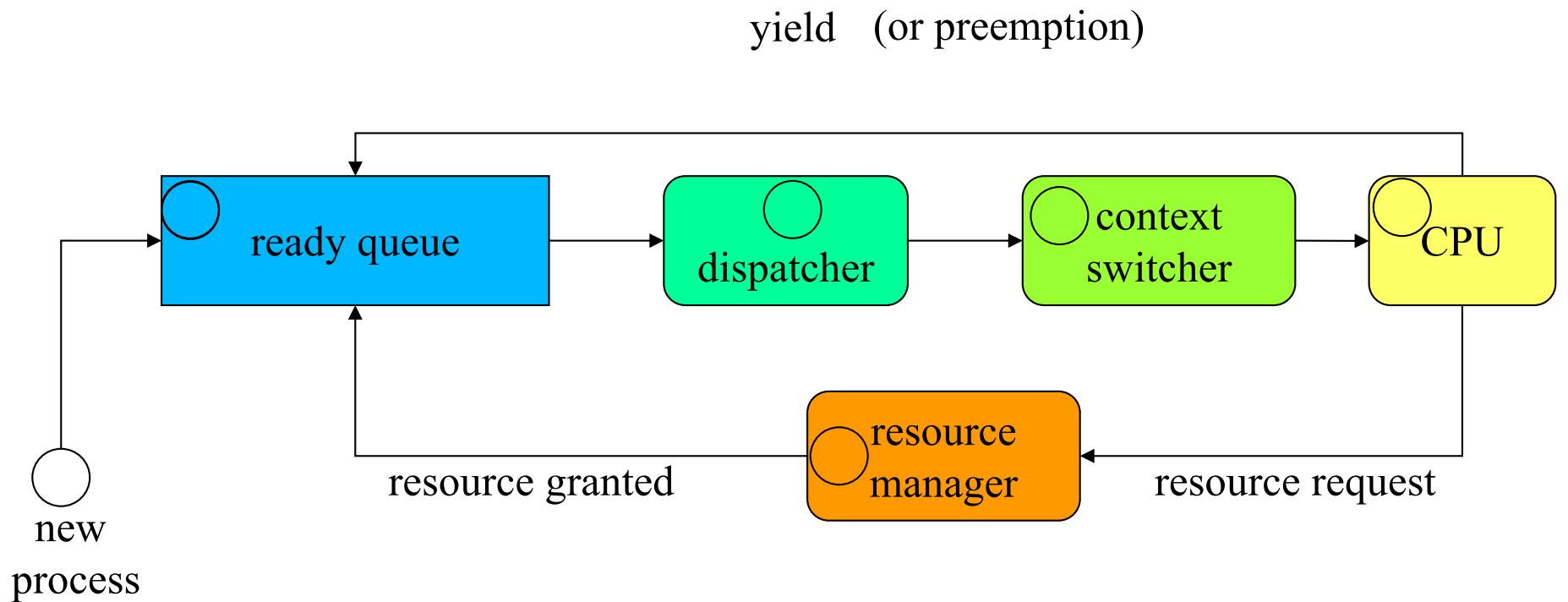
Pros and Cons of Non-Preemptive Scheduling

- + 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

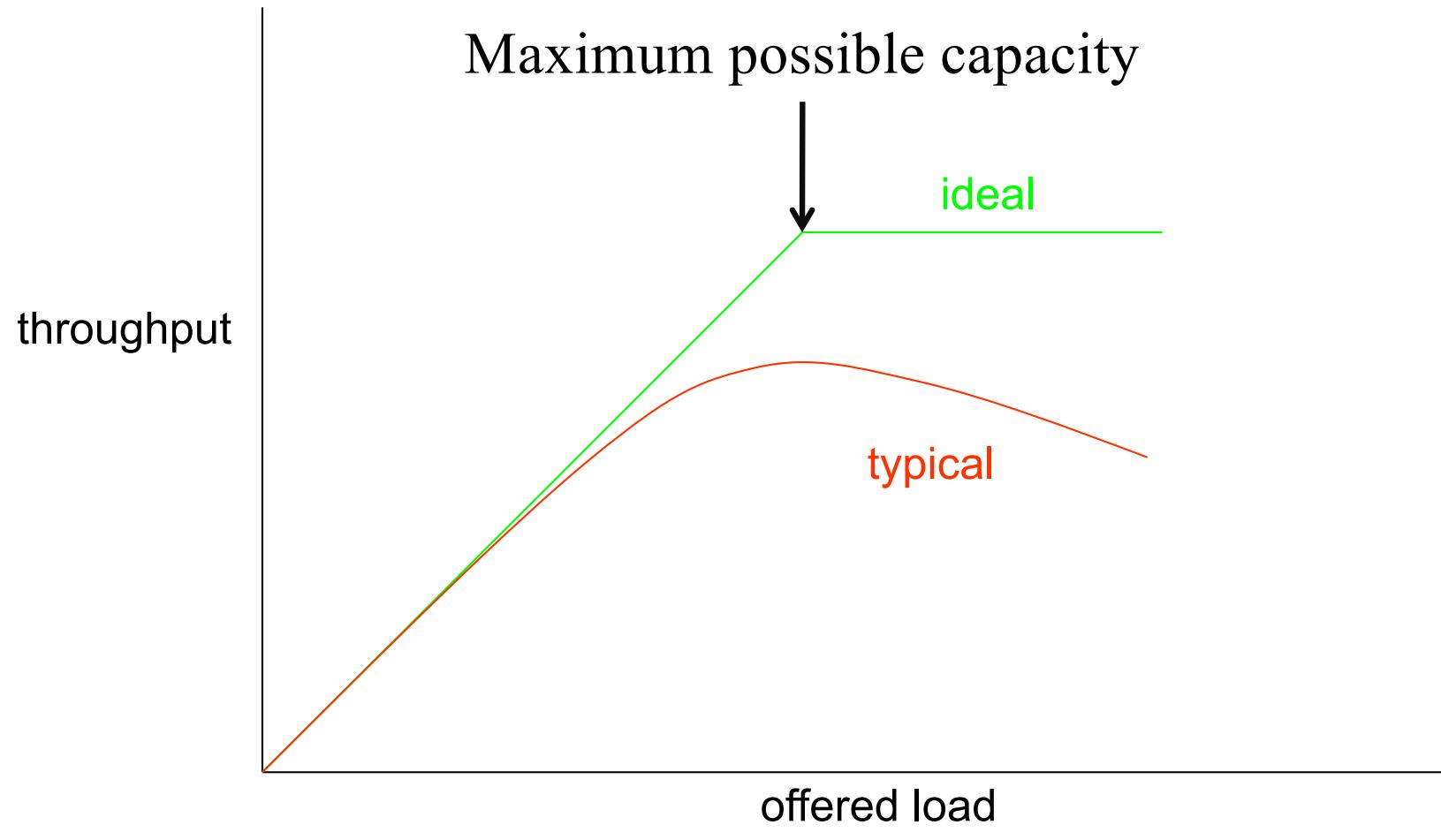
- Mean time to completion (seconds)
 - For a particular job mix (benchmark)
- Throughput (operations per second)
 - For a particular activity or job mix (benchmark)
- Mean response time (milliseconds)
 - Time spent on the ready queue
- Overall “goodness”
 - Requires a customer-specific weighting function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

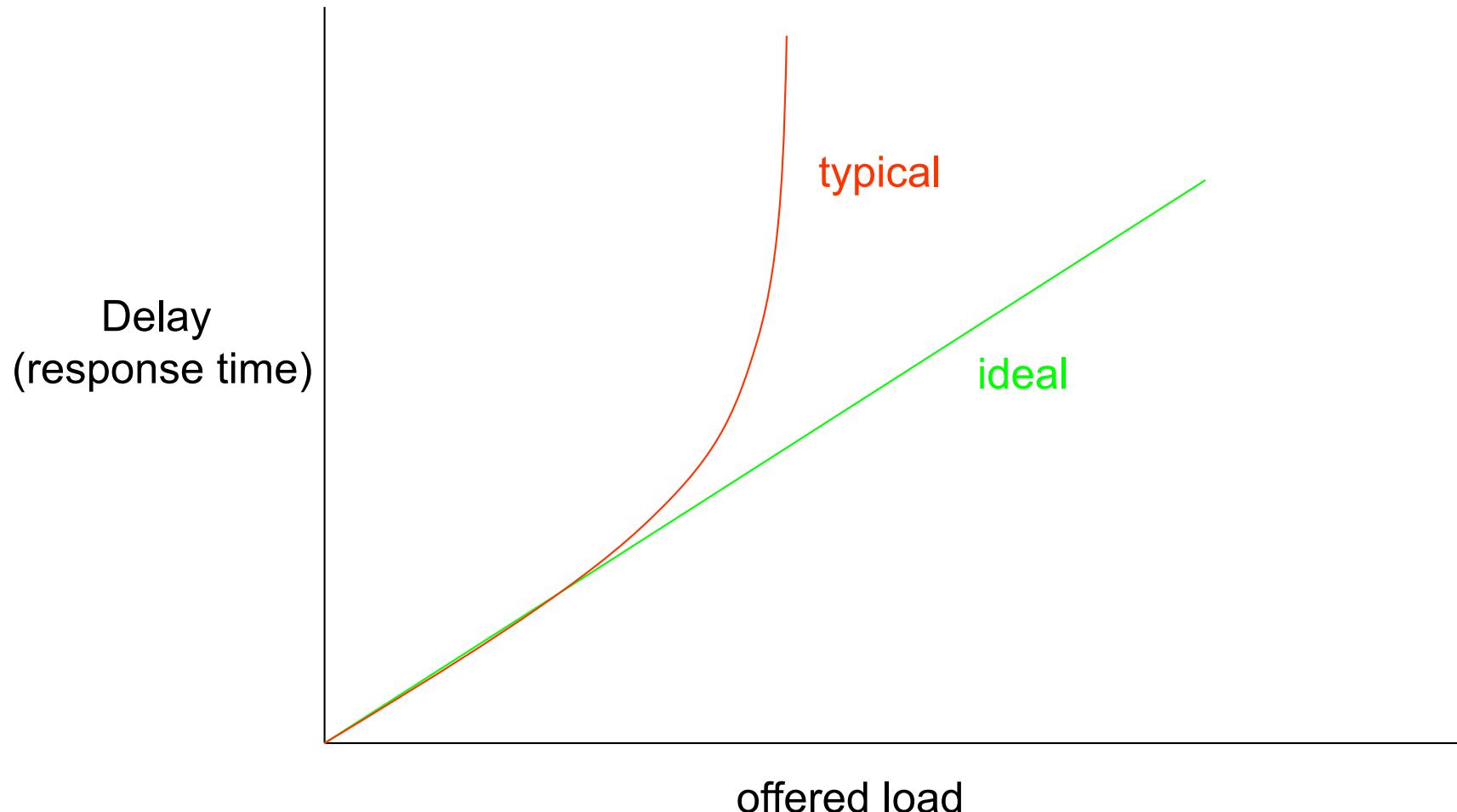
Typical Throughput vs. Load Curve



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	

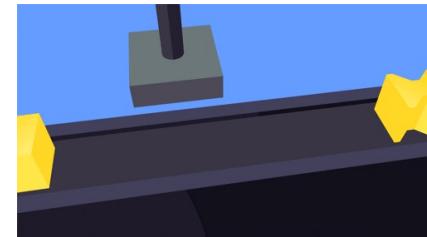
Note: Average is worse than total/5 because four other processes had to wait for the slow-poke who ran first.

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

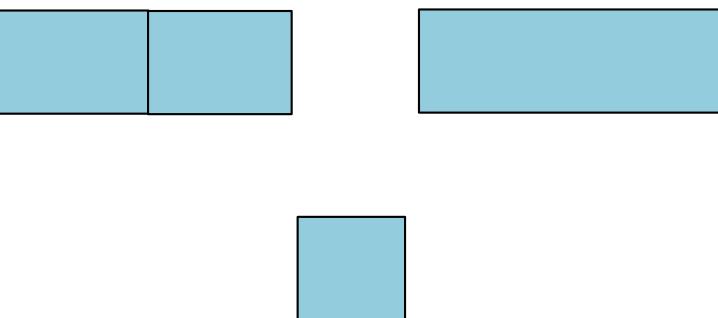
- If non-preemptive, priority scheduling is just about ordering processes
- Much like shortest job first, but ordered by priority instead
- But what if scheduling is preemptive?
- In that case, when new process is created, it might preempt running process
 - If its priority is higher

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

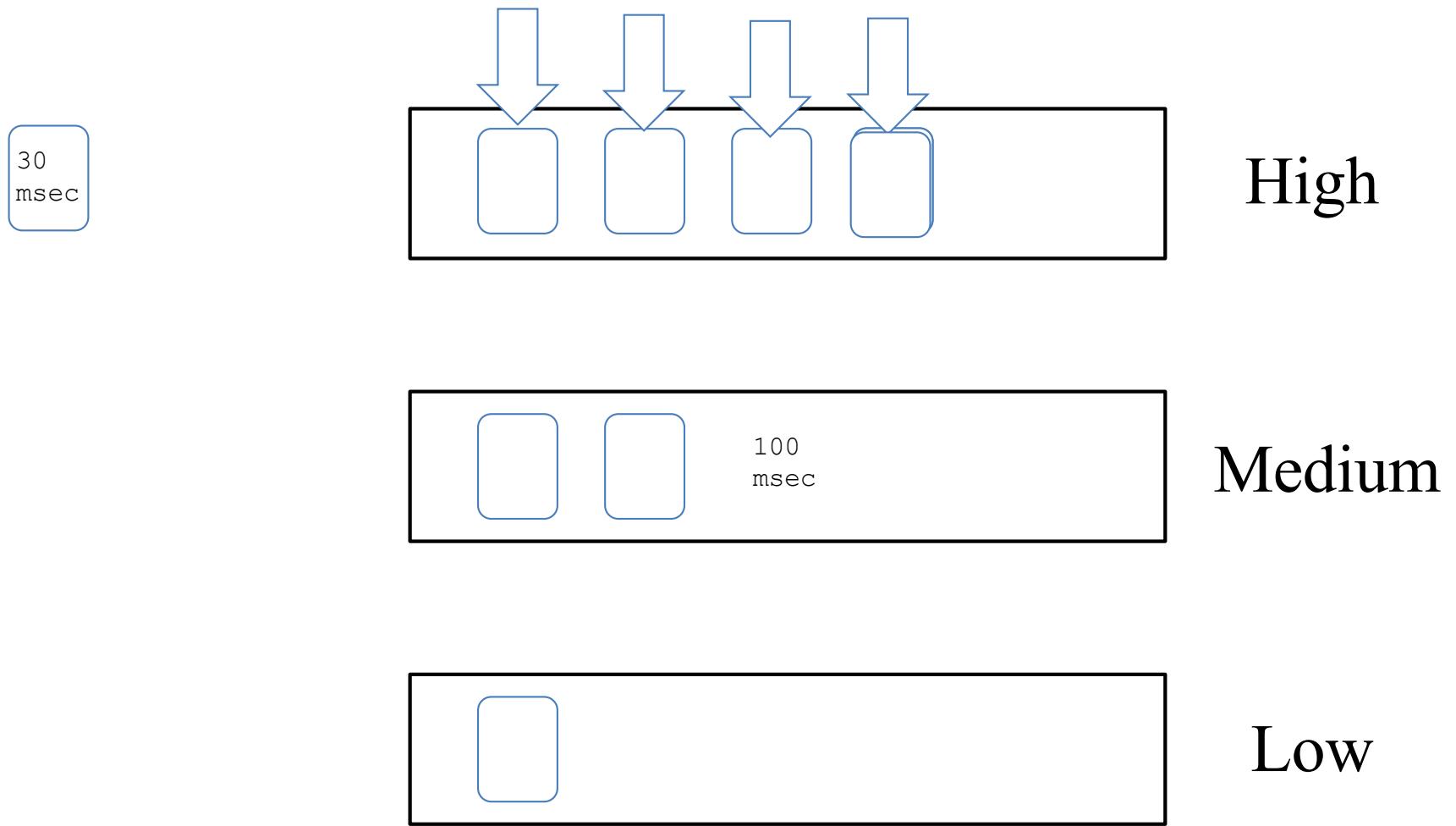
Multi-Level Feedback Queue (MLFQ) Scheduling

- 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

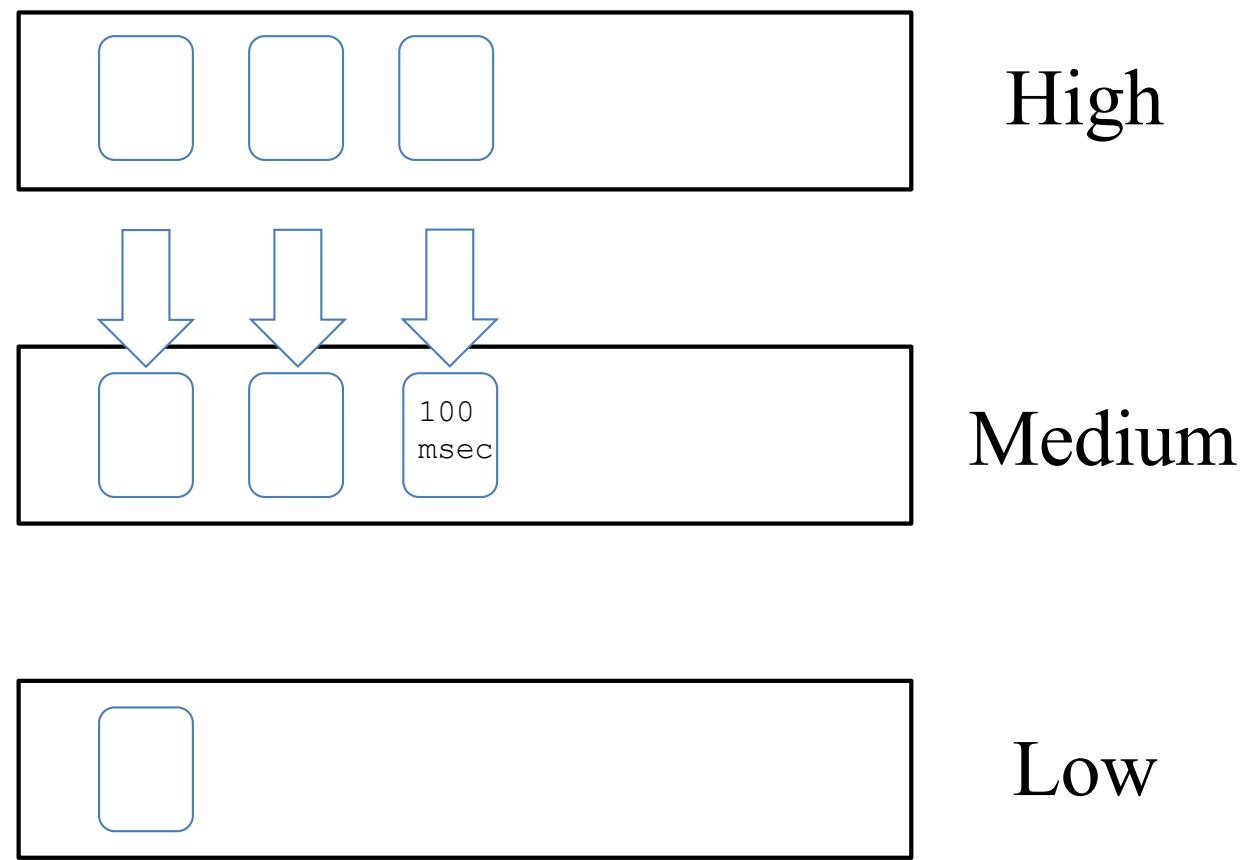
How Do I Know What Queue To Put New Process Into?

- 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

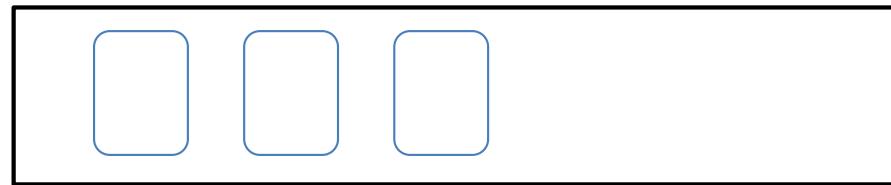


MLFQ Continuing



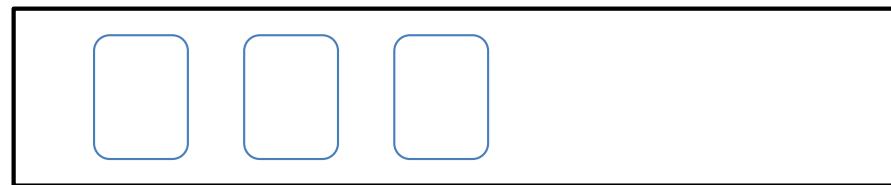
What About Fairness?

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

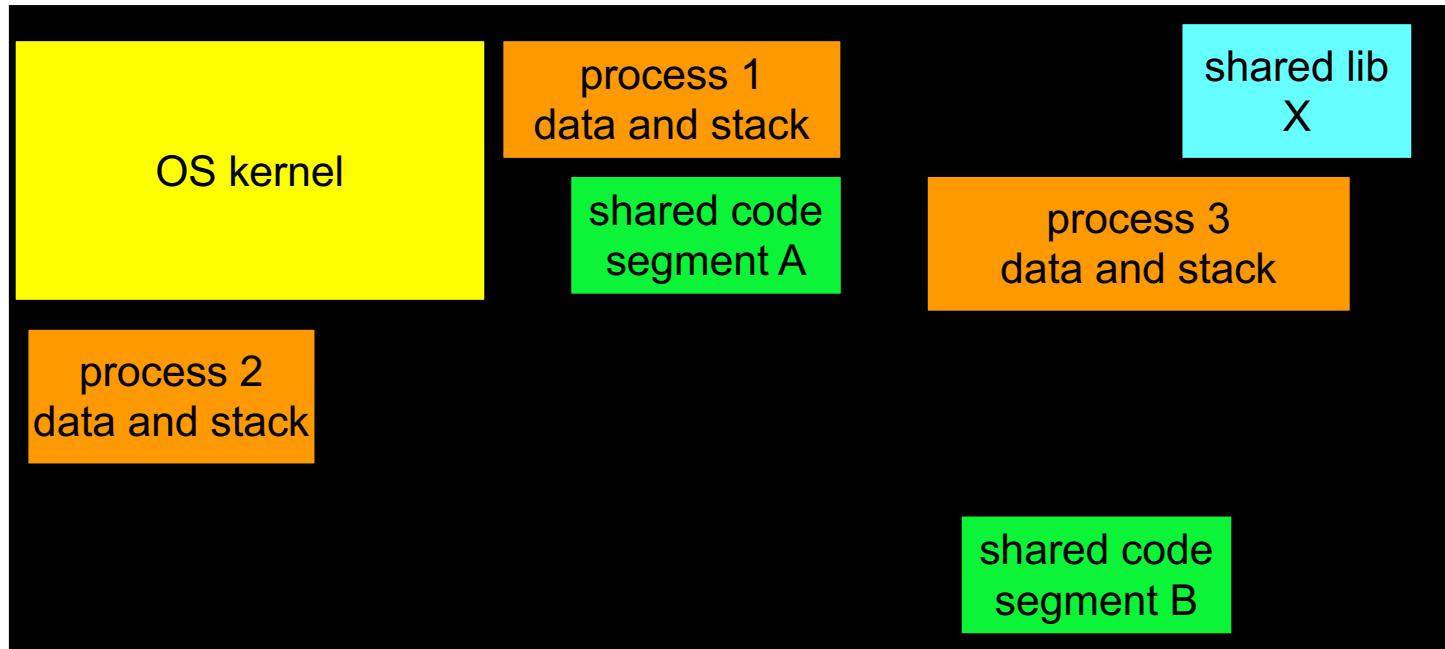
2. Efficiency

- 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

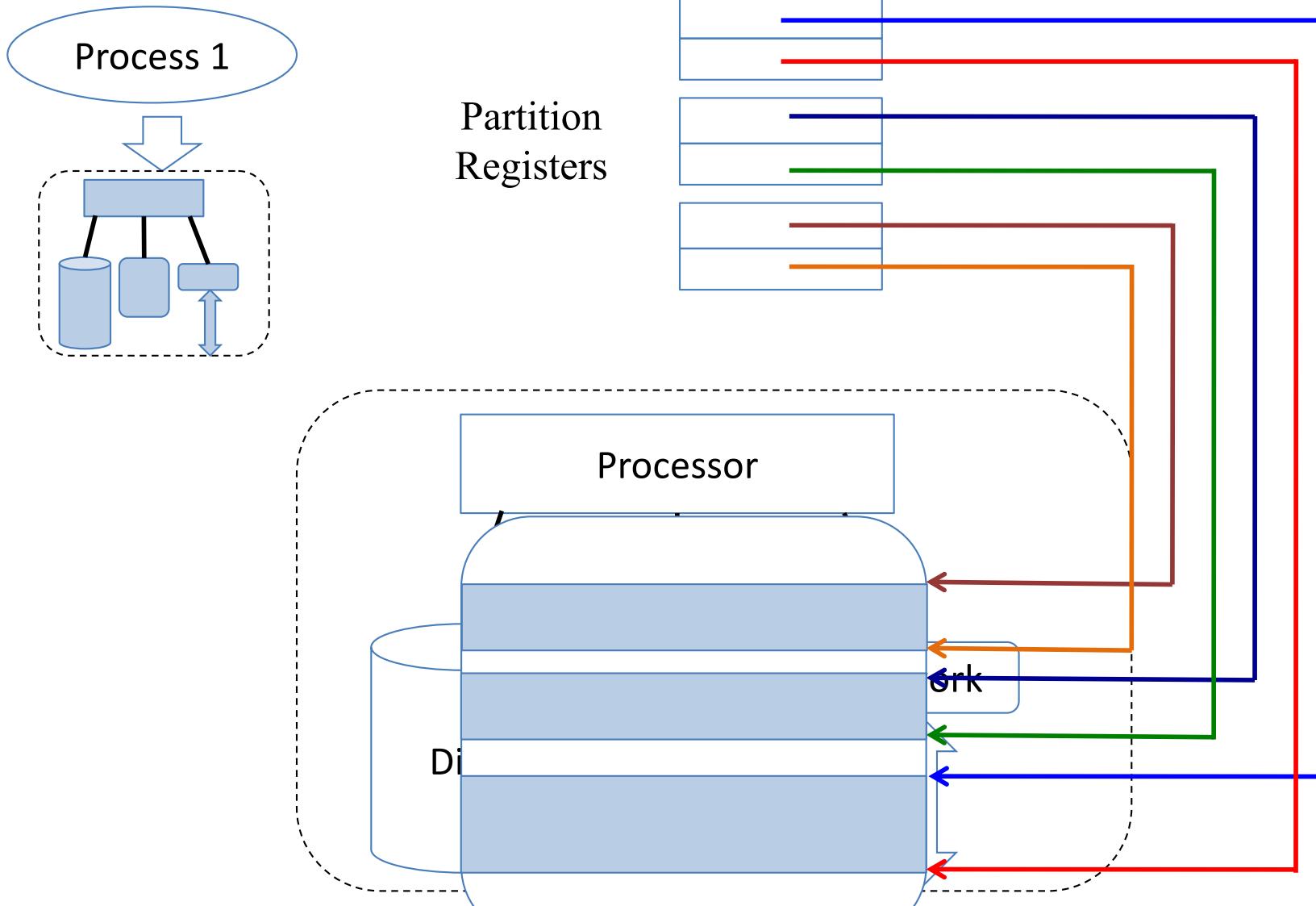
Fixed Partition Allocation

- 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

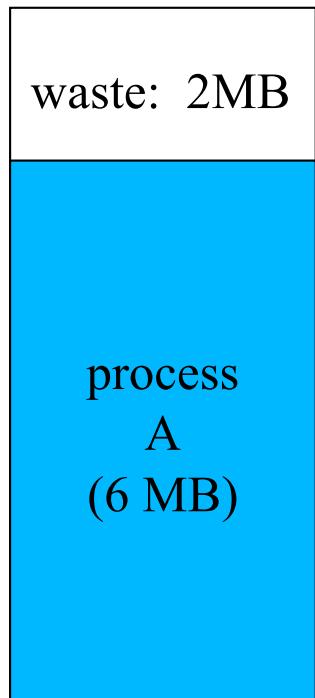
Available partition sizes:

8 Mbytes

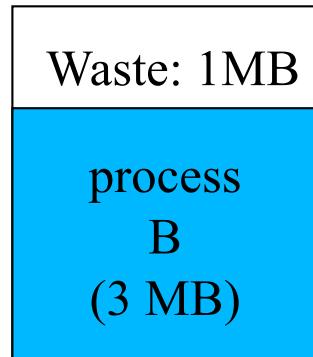
4 Mbytes

4 Mbytes

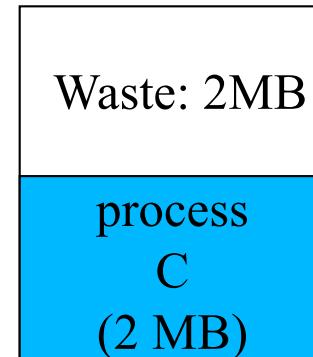
$$\begin{aligned}\text{Total waste} &= 2\text{MB} + 1\text{MB} + 2\text{MB} = \\ &5/16\text{MB} = 31\%\end{aligned}$$



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

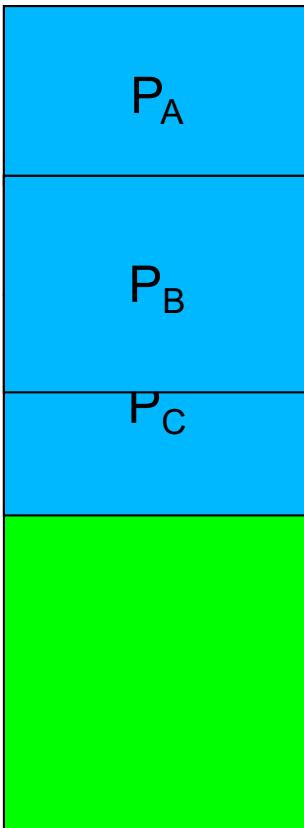
Problems With Dynamic Partitions

- 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

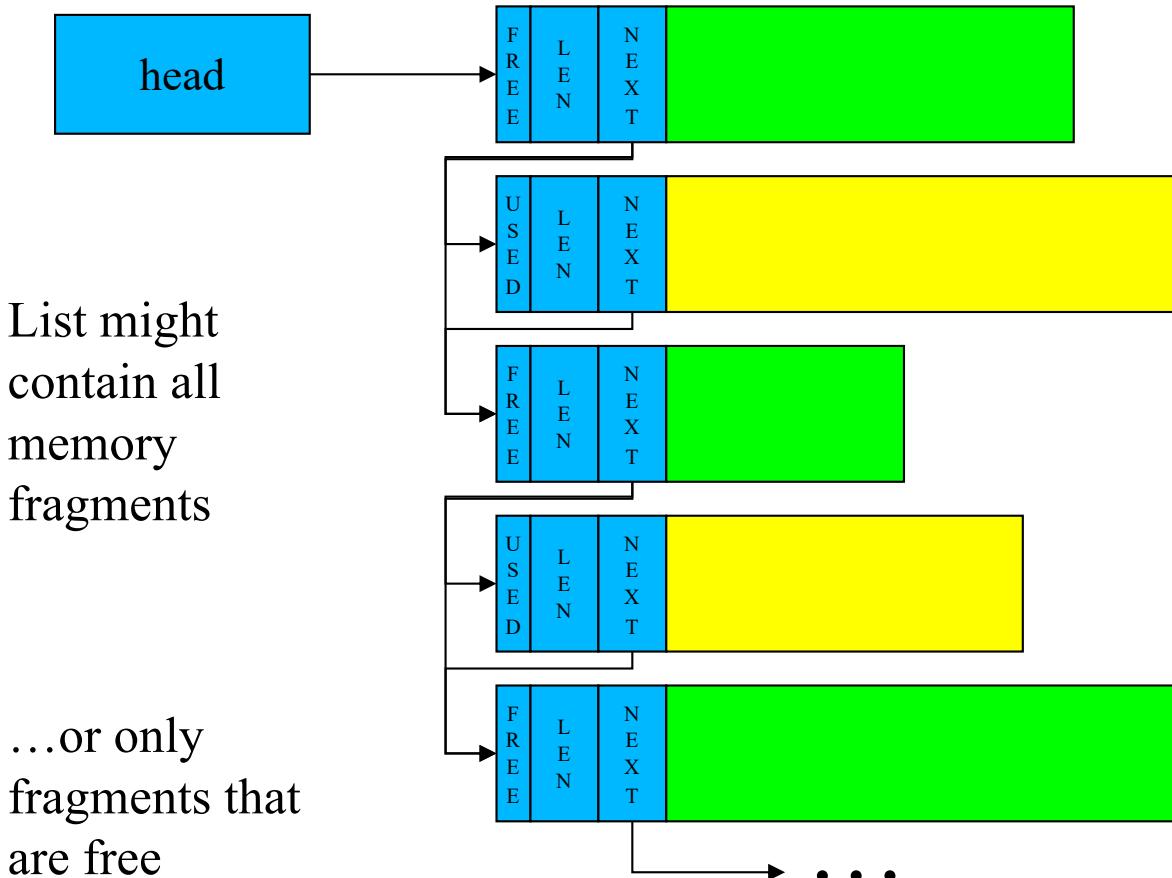
How To Keep Track of Variable Sized Partitions?

- 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

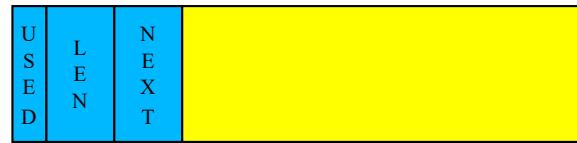
- Fixed sized blocks are easy to track
 - A bit map indicating which blocks are free
- Variable chunks require more information
 - A linked list of descriptors, one per chunk
 - Each descriptor lists the size of the chunk and whether it is free
 - Each has a pointer to the next chunk on list
 - Descriptors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

The Free List



Free Chunk Carving

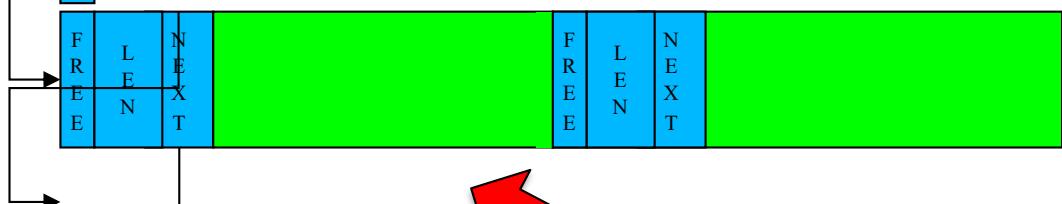
1. Find a large enough free chunk



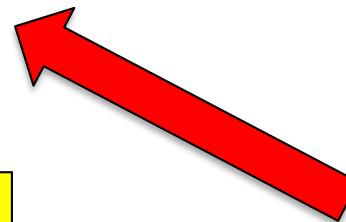
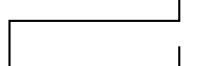
2. Reduce its len to requested size



3. Create a new header for residual chunk



4. Insert the new chunk into the list



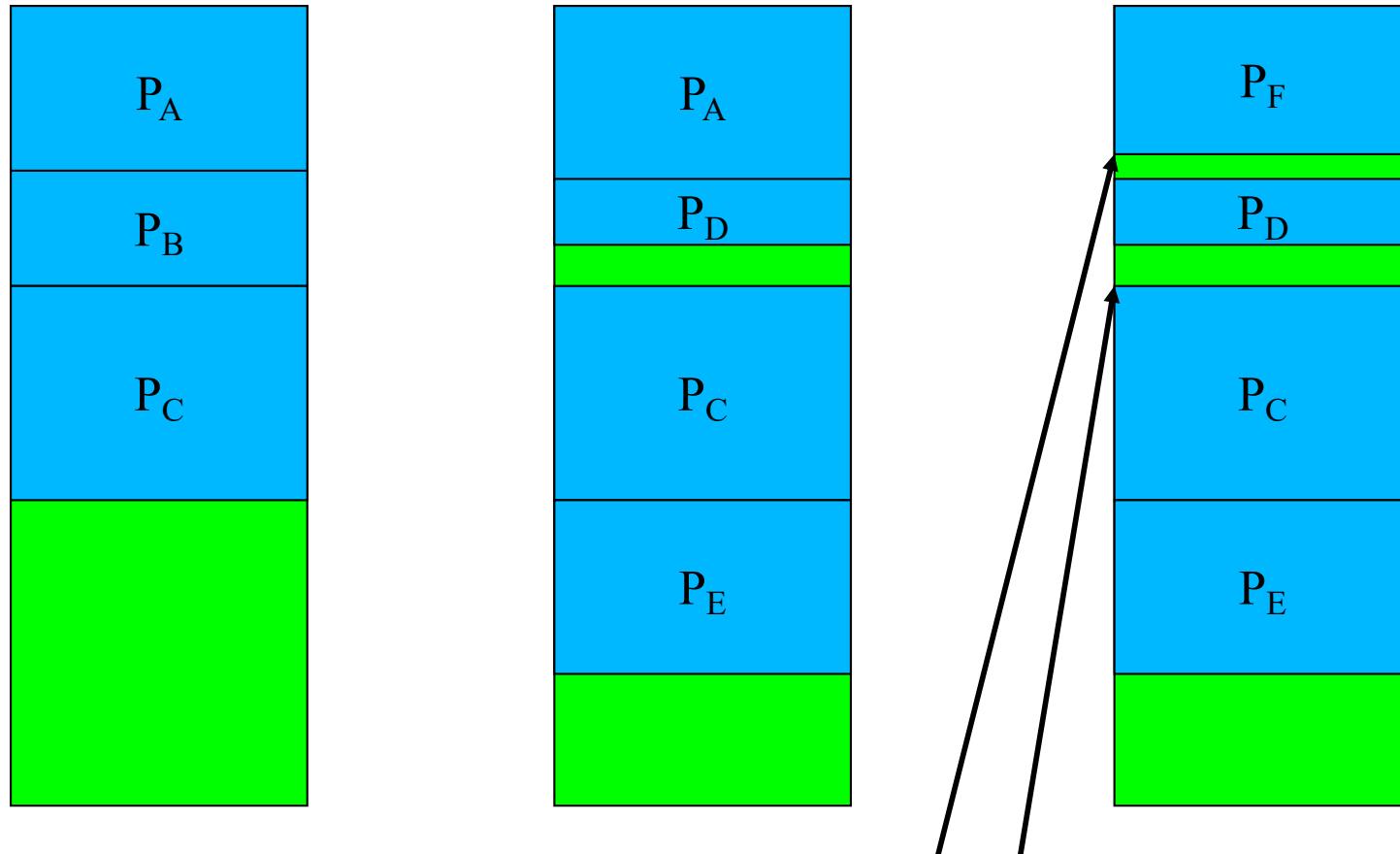
5. Mark the carved piece as in use



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

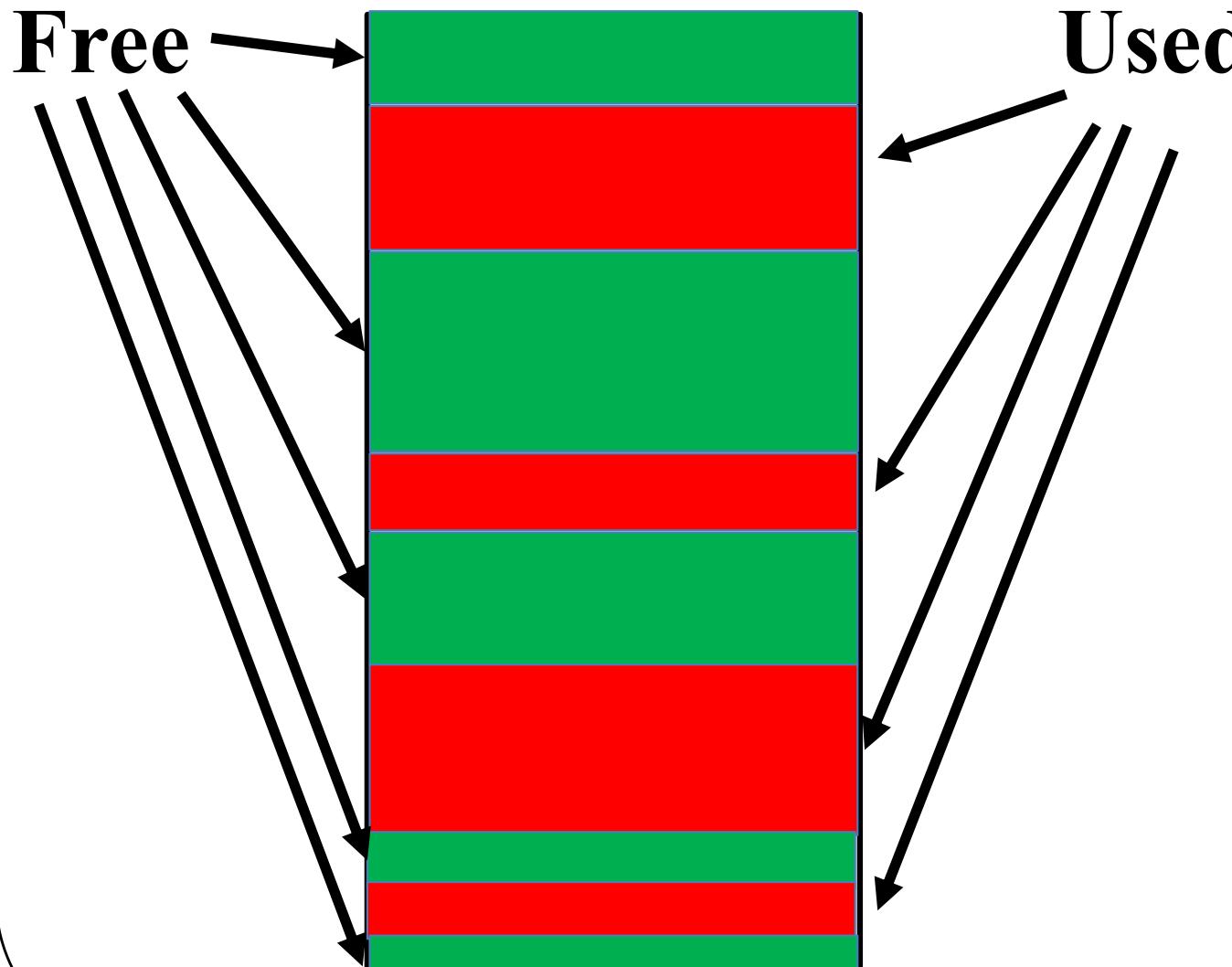
External Fragmentation: Causes and Effects

- 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

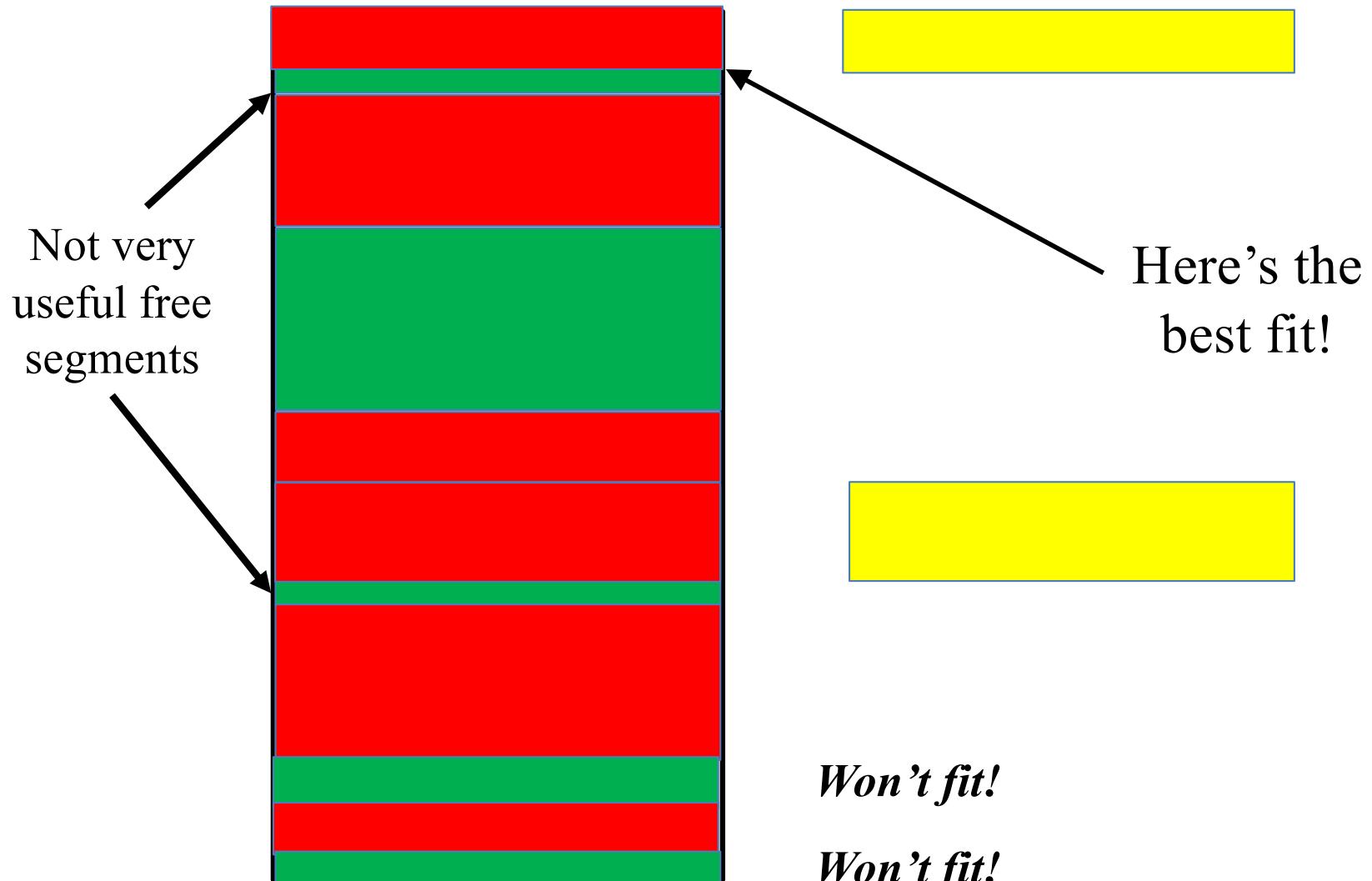
Allocating Partitions in Memory



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



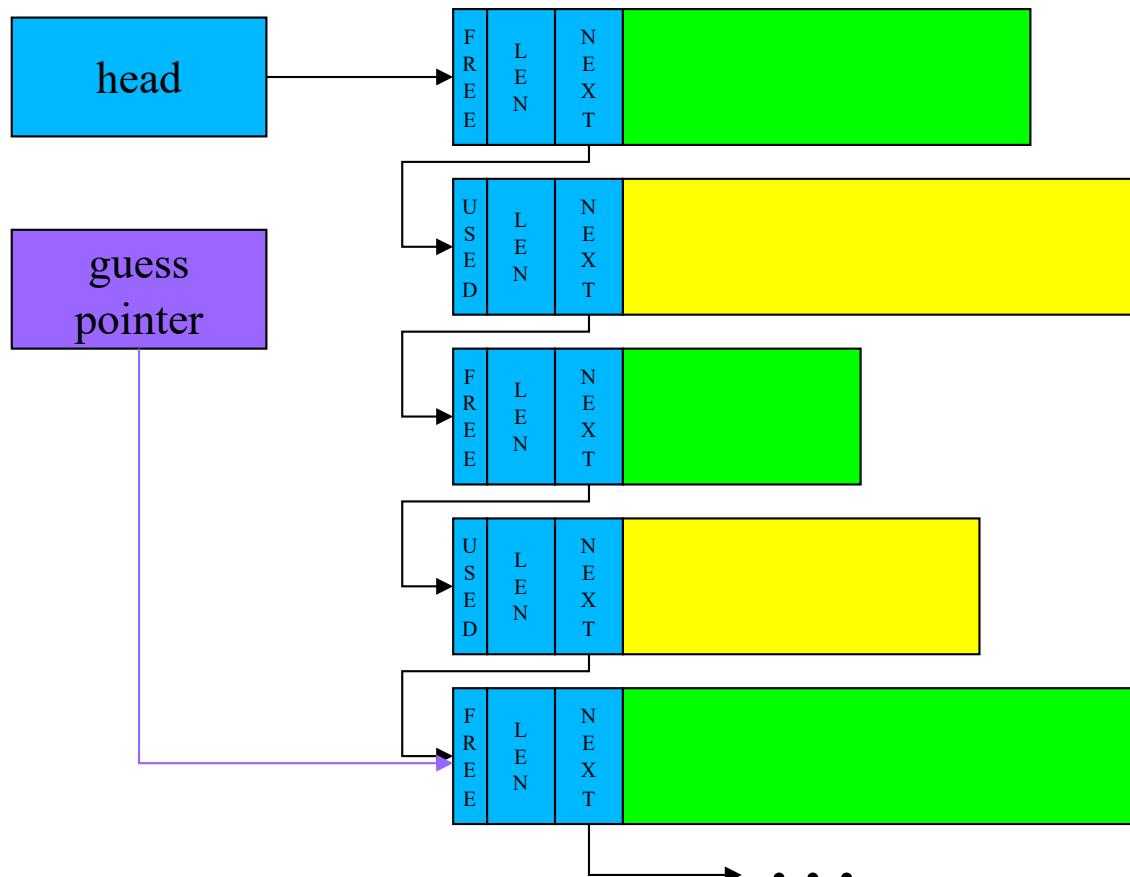
First Fit

- 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.



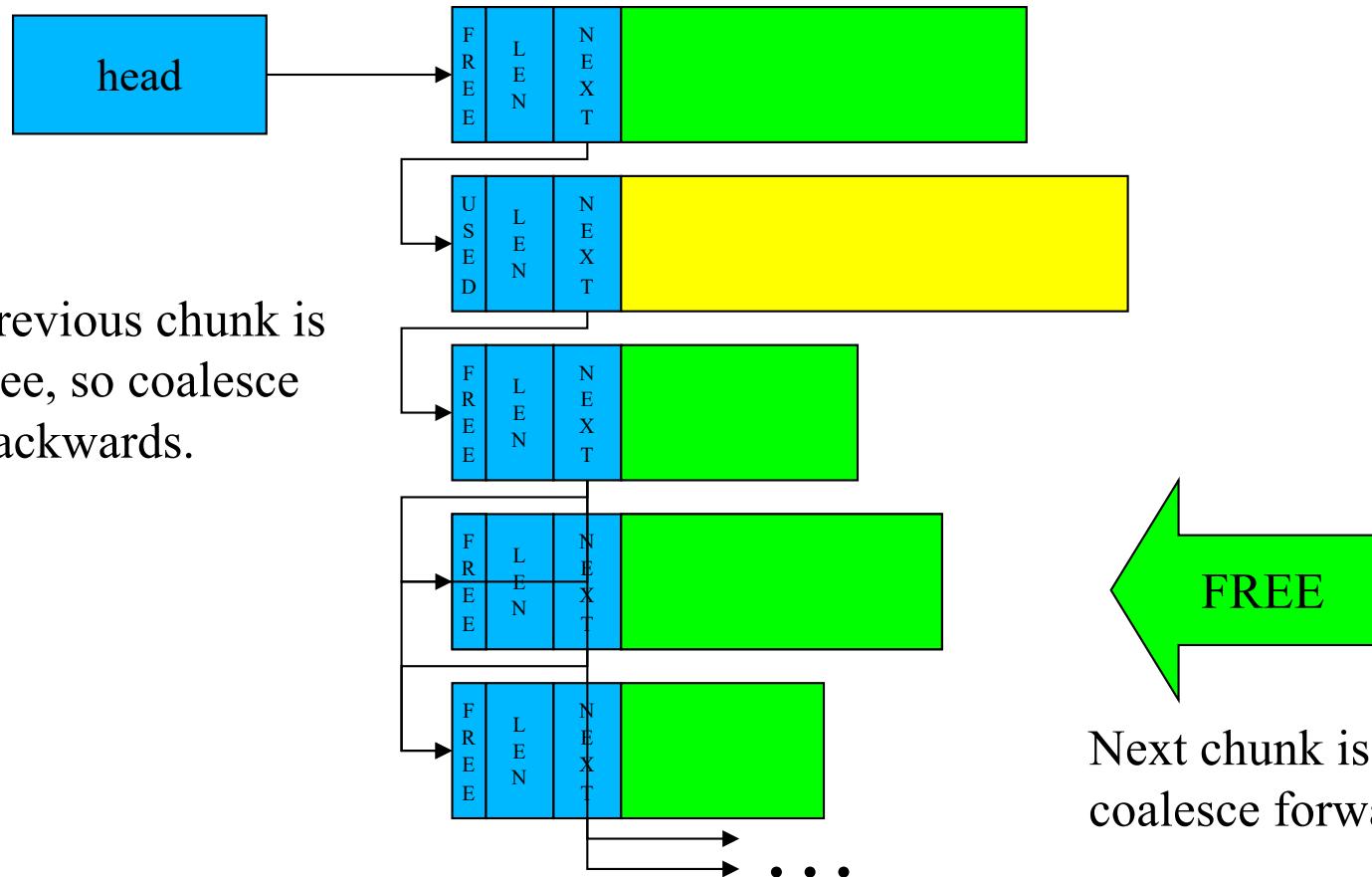
Next Fit Properties

- 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



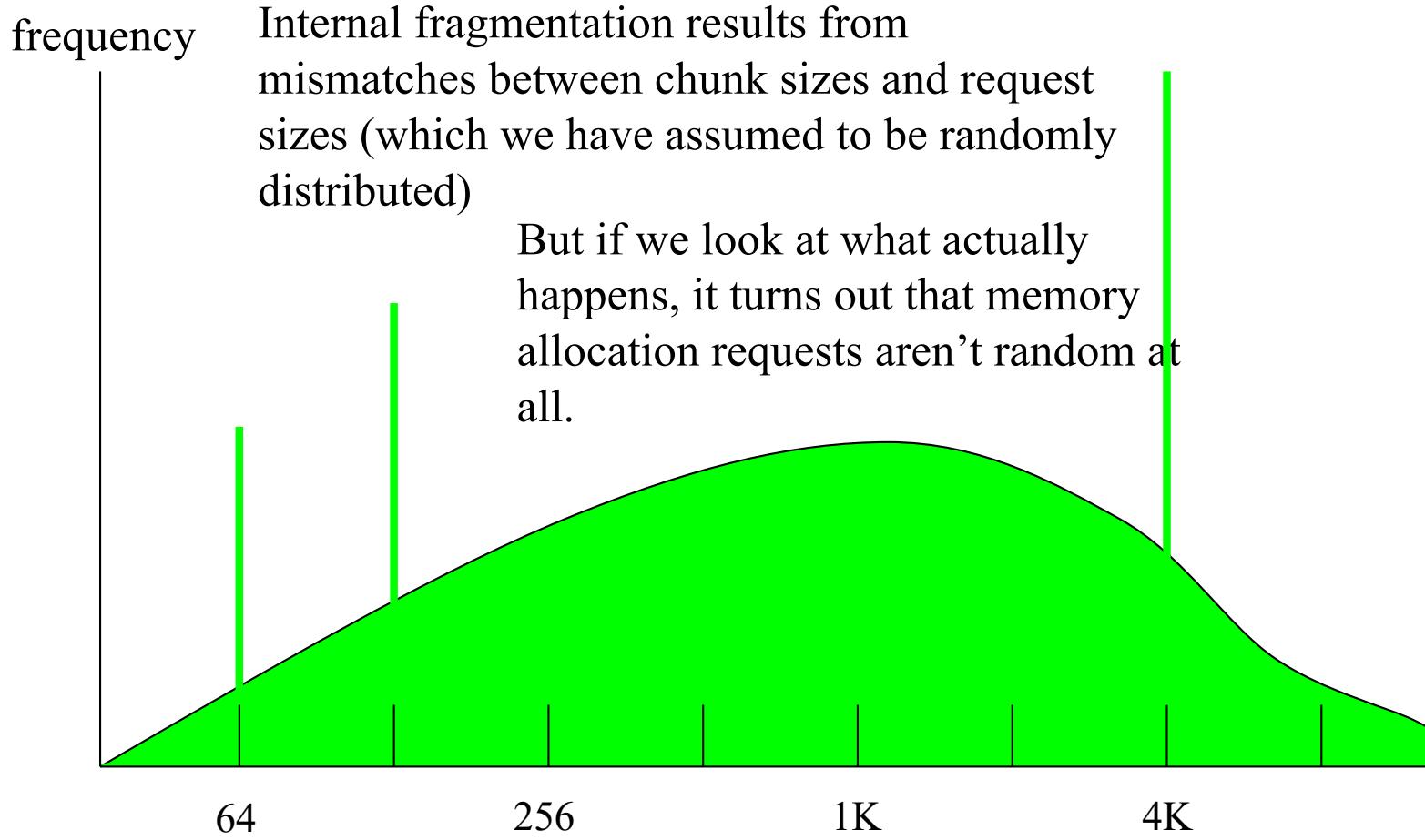
Fragmentation and Coalescing

- 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?

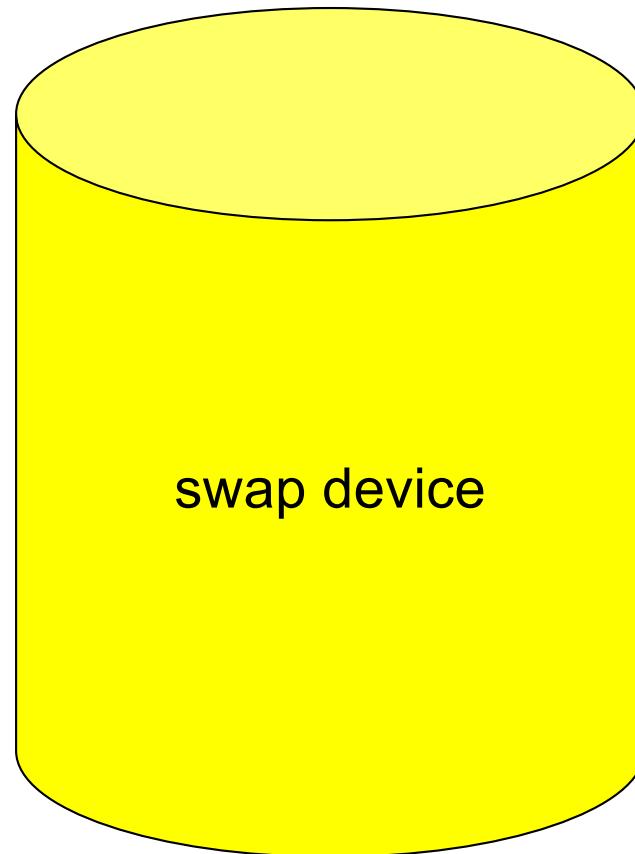
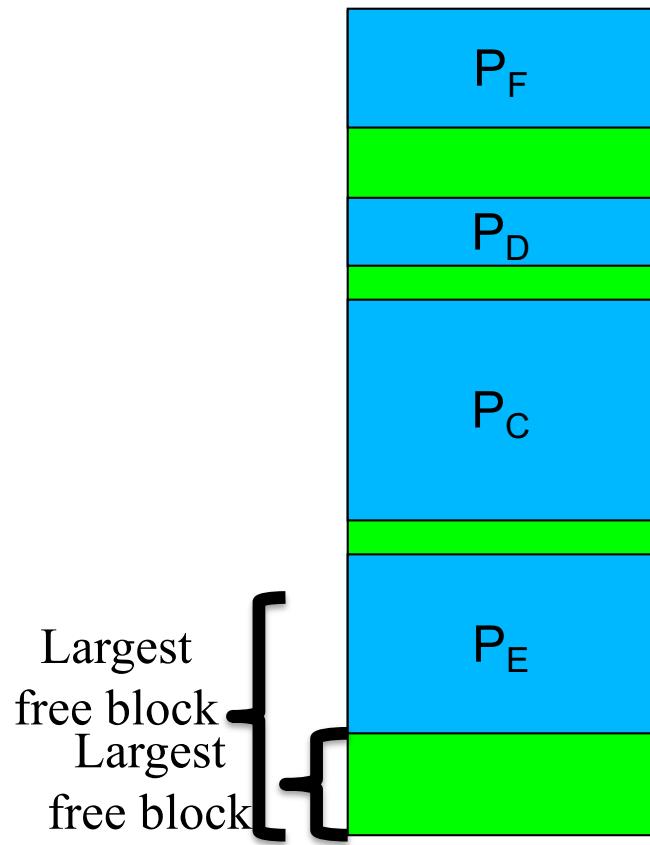
Problems With General Garbage Collection

- 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

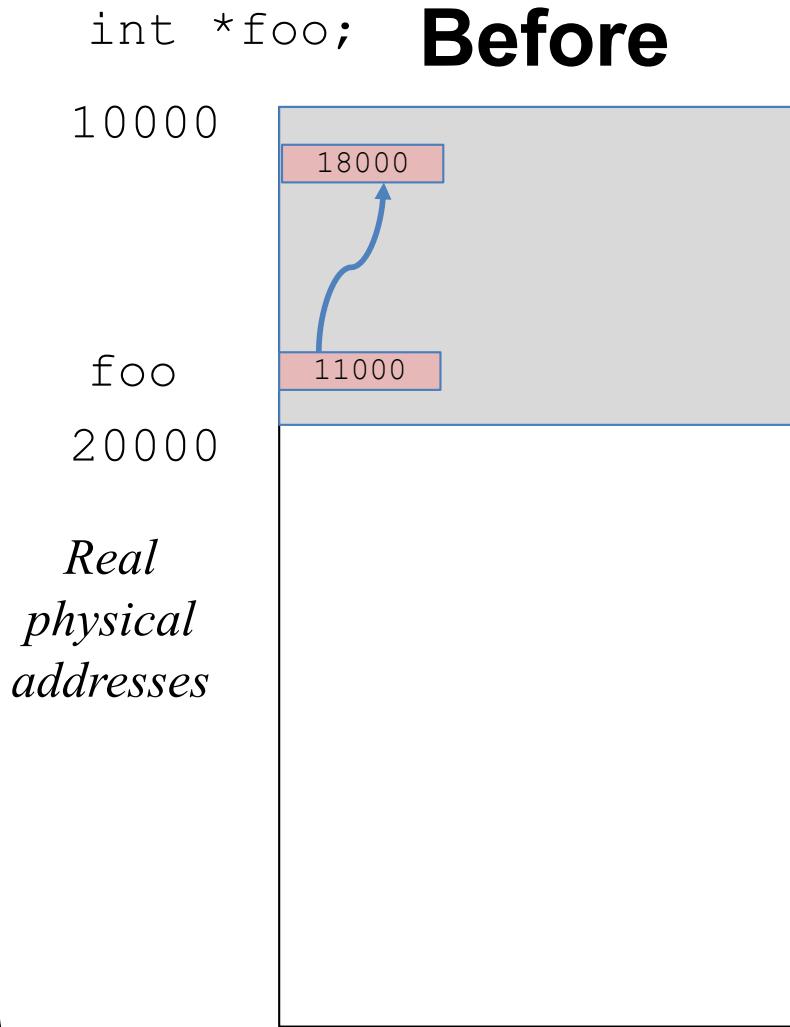


*An obvious
improvement!*

All This Requires Is Relocation . . .

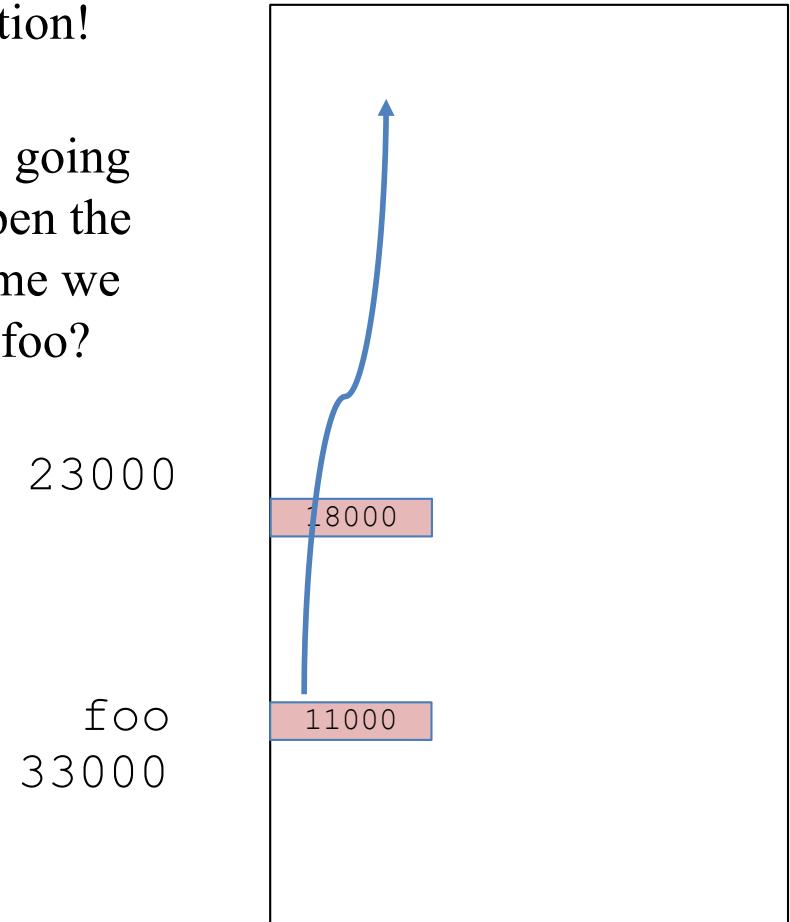
- The ability to move a process' data
 - From region where it was initially loaded
 - Into a new and different region of memory
- What's so hard about that?
- All addresses in the program will be wrong
 - References in the code segment
 - Calls and branches to other parts of the code
 - References to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

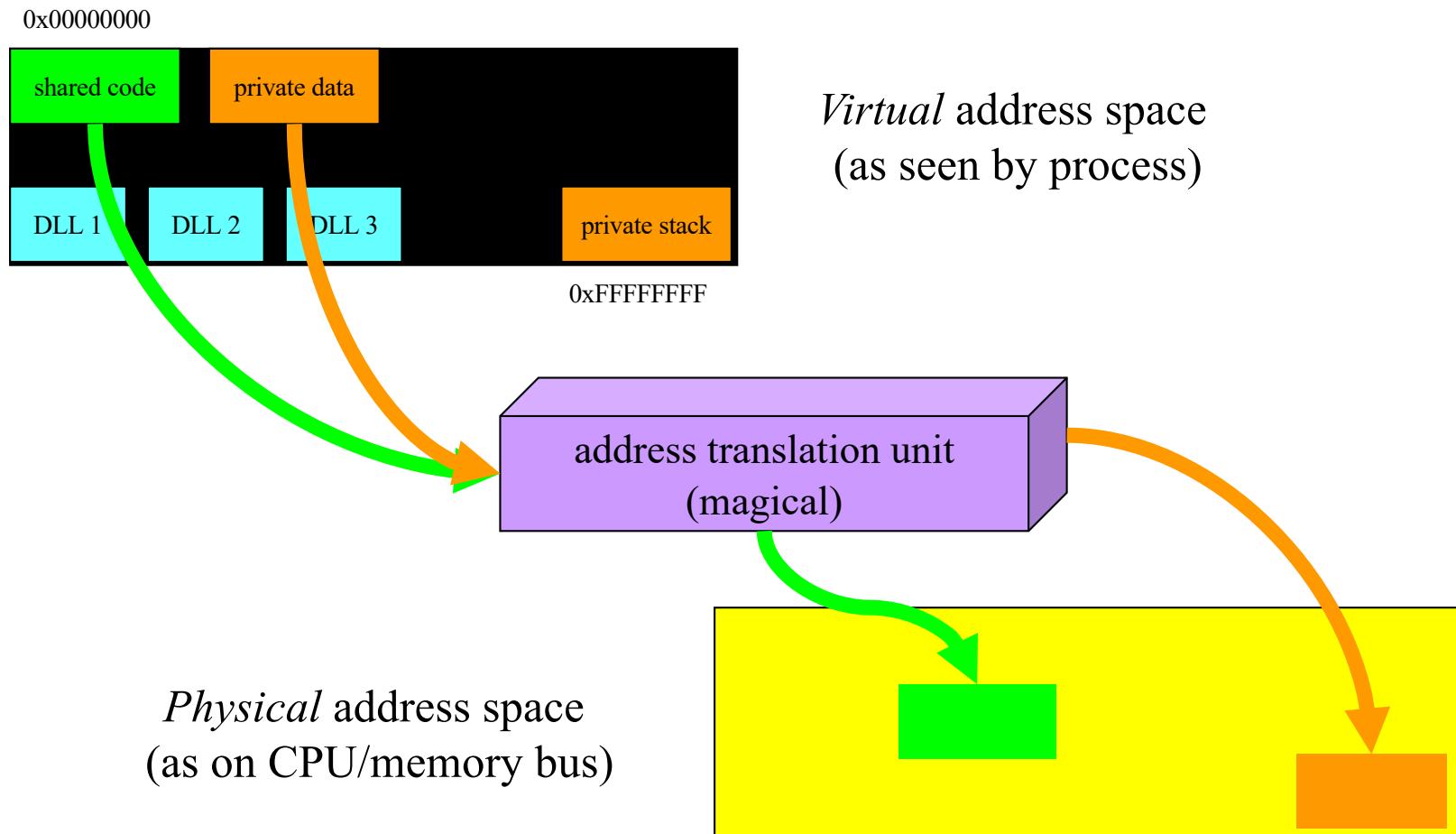


Of course, we copy the partition's contents when we move it

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?

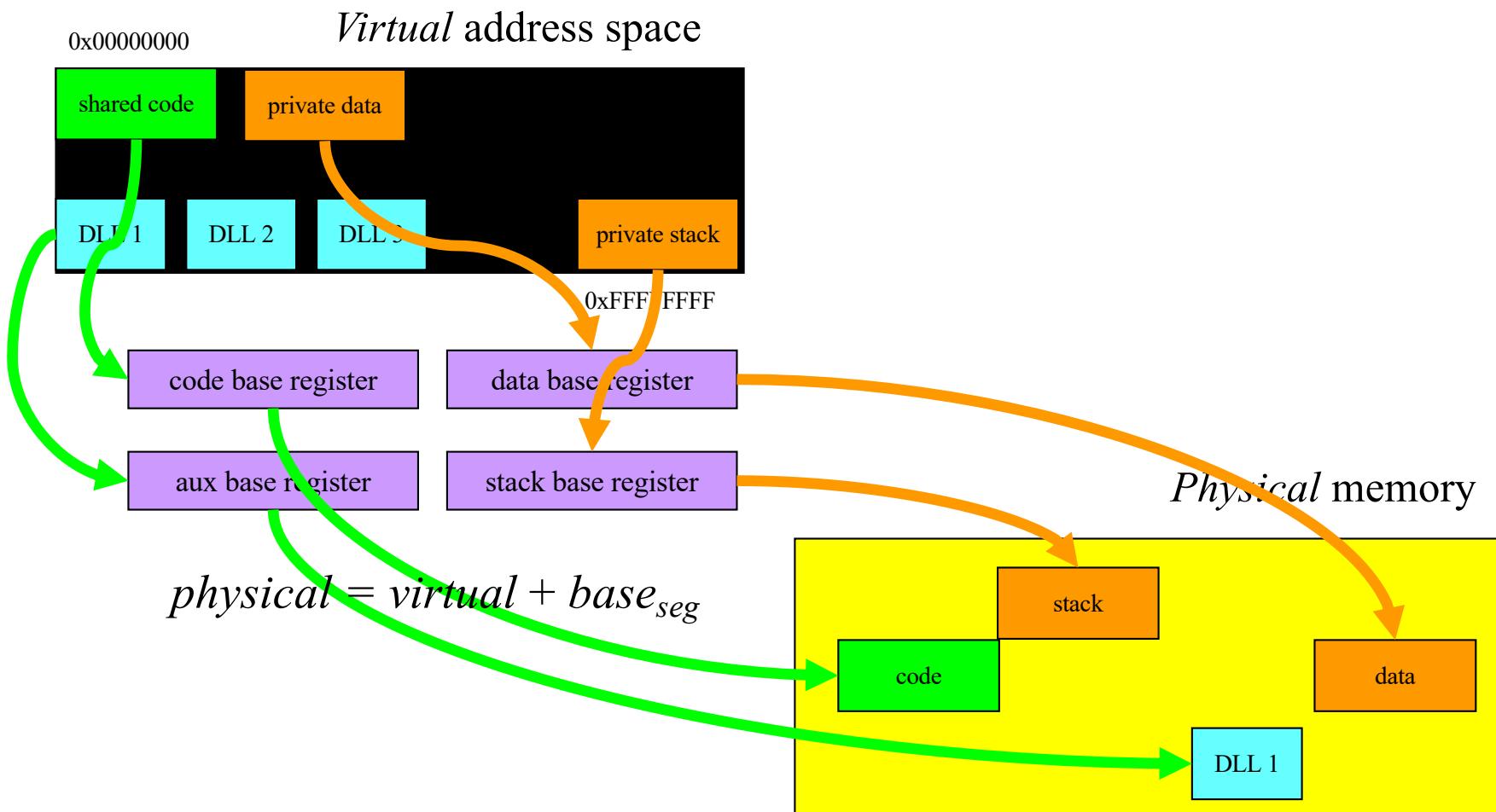
Virtual Address Spaces



Memory Segment Relocation

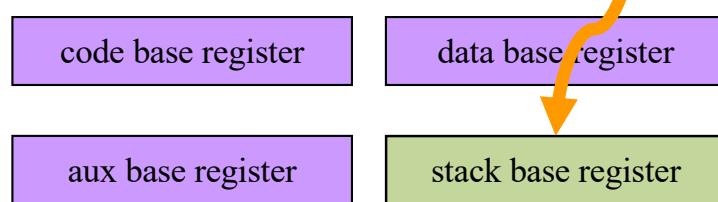
- A natural model
 - Process address space is made up of multiple segments
 - Use the segment as the unit of relocation
 - Long tradition, from the IBM system 360 to Intel x86 architecture
- Computer has special relocation registers
 - They are called *segment base registers*
 - They point to the start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

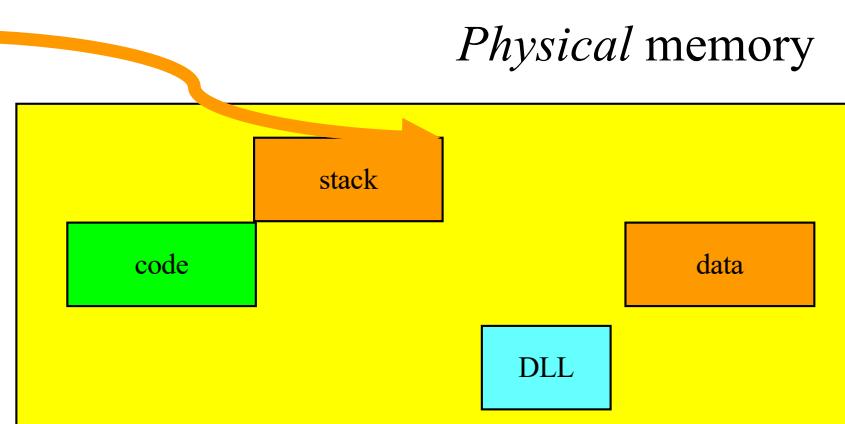
The virtual address of the stack doesn't change



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

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

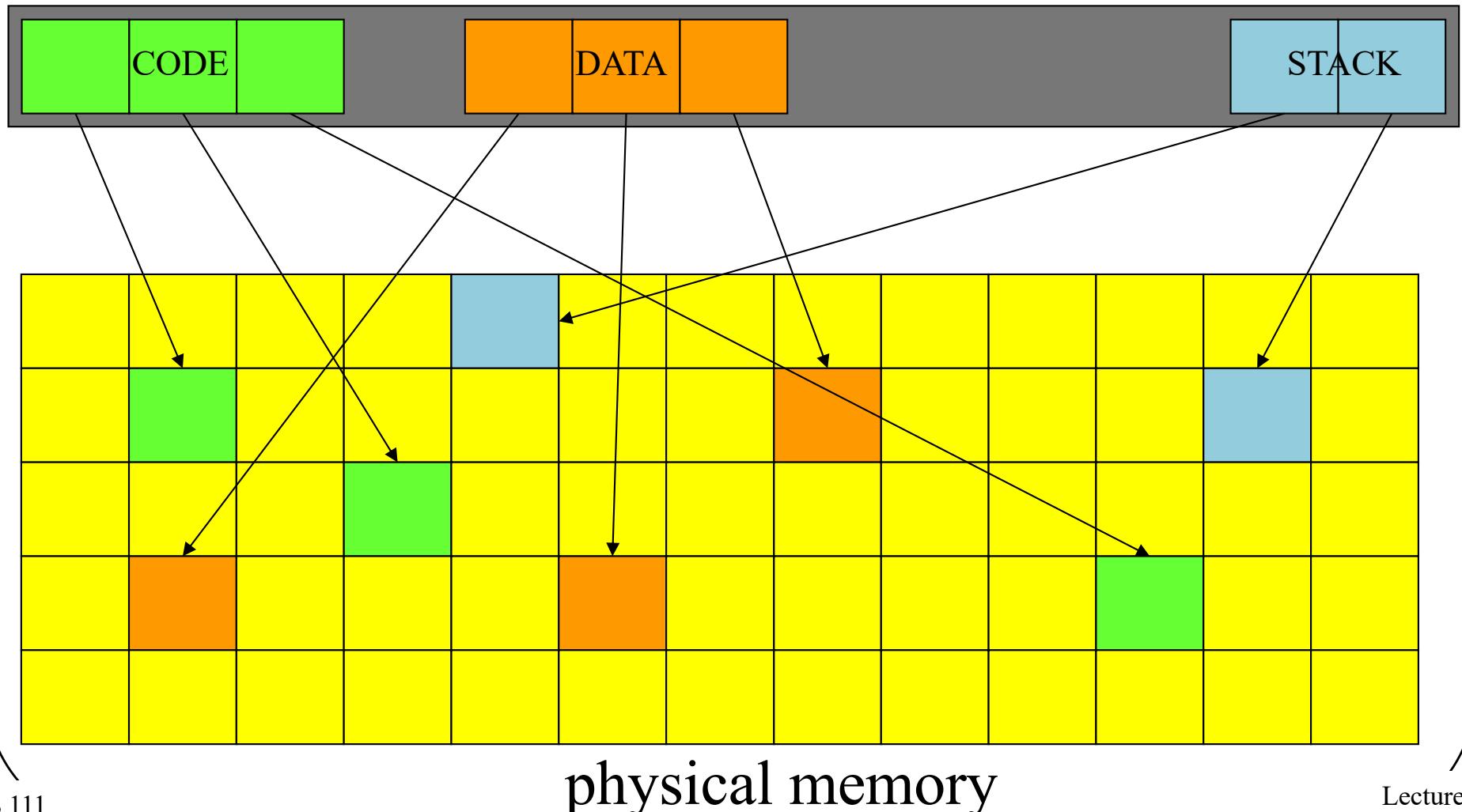
- Segment relocation solved the relocation problem for us
- It used base registers to compute a physical address from a virtual address
 - Allowing us to move data around in physical memory
 - By only updating the base register
- It did nothing about external fragmentation
 - Because segments are still 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

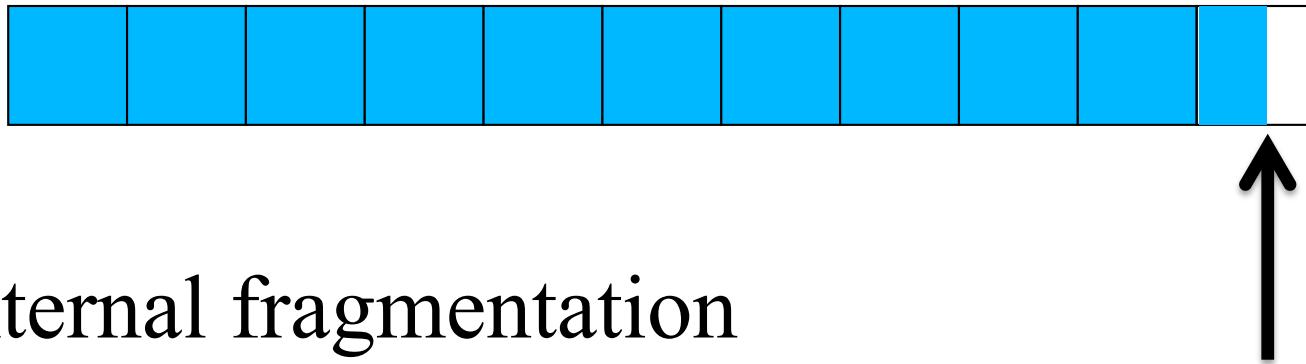
Paged Address Translation

process virtual address space



Paging and Fragmentation

- A segment is implemented as a set of virtual pages



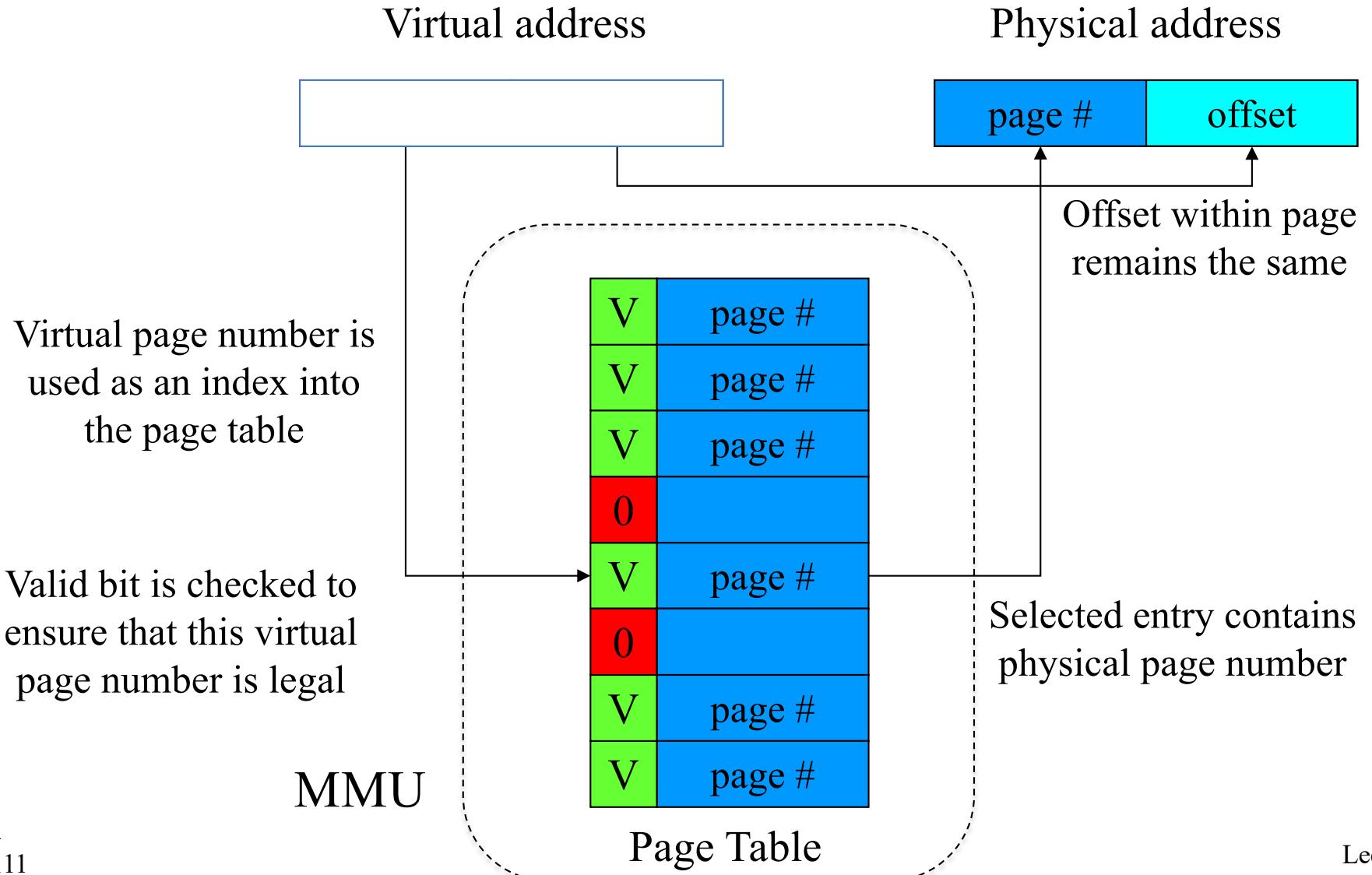
- Internal fragmentation
 - Averages only $\frac{1}{2}$ page (half of the last one)
- External fragmentation
 - Completely non-existent
 - We never carve up pages

Tremendous
reduction in
fragmentation costs!

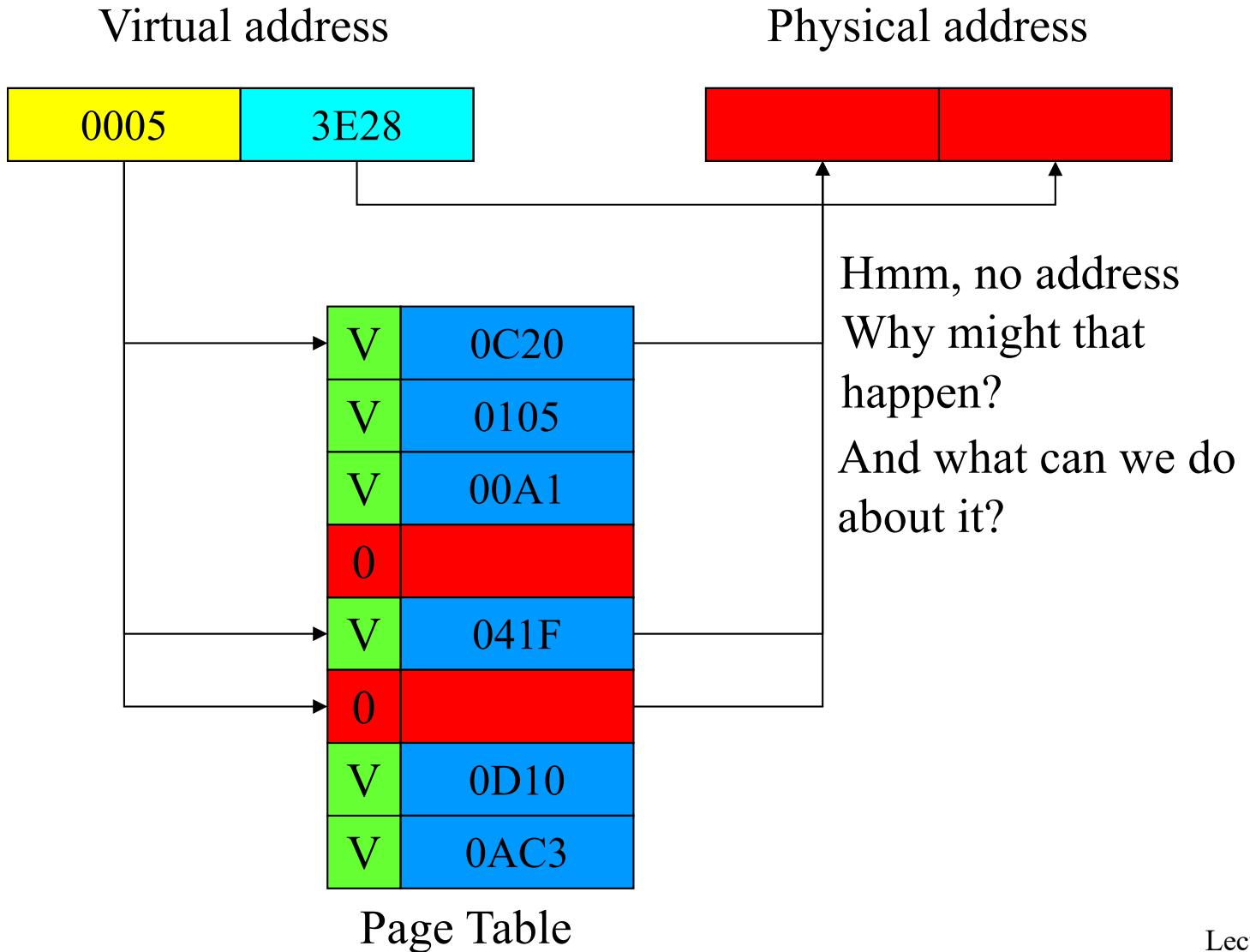
Providing the Magic Translation Mechanism

- 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!

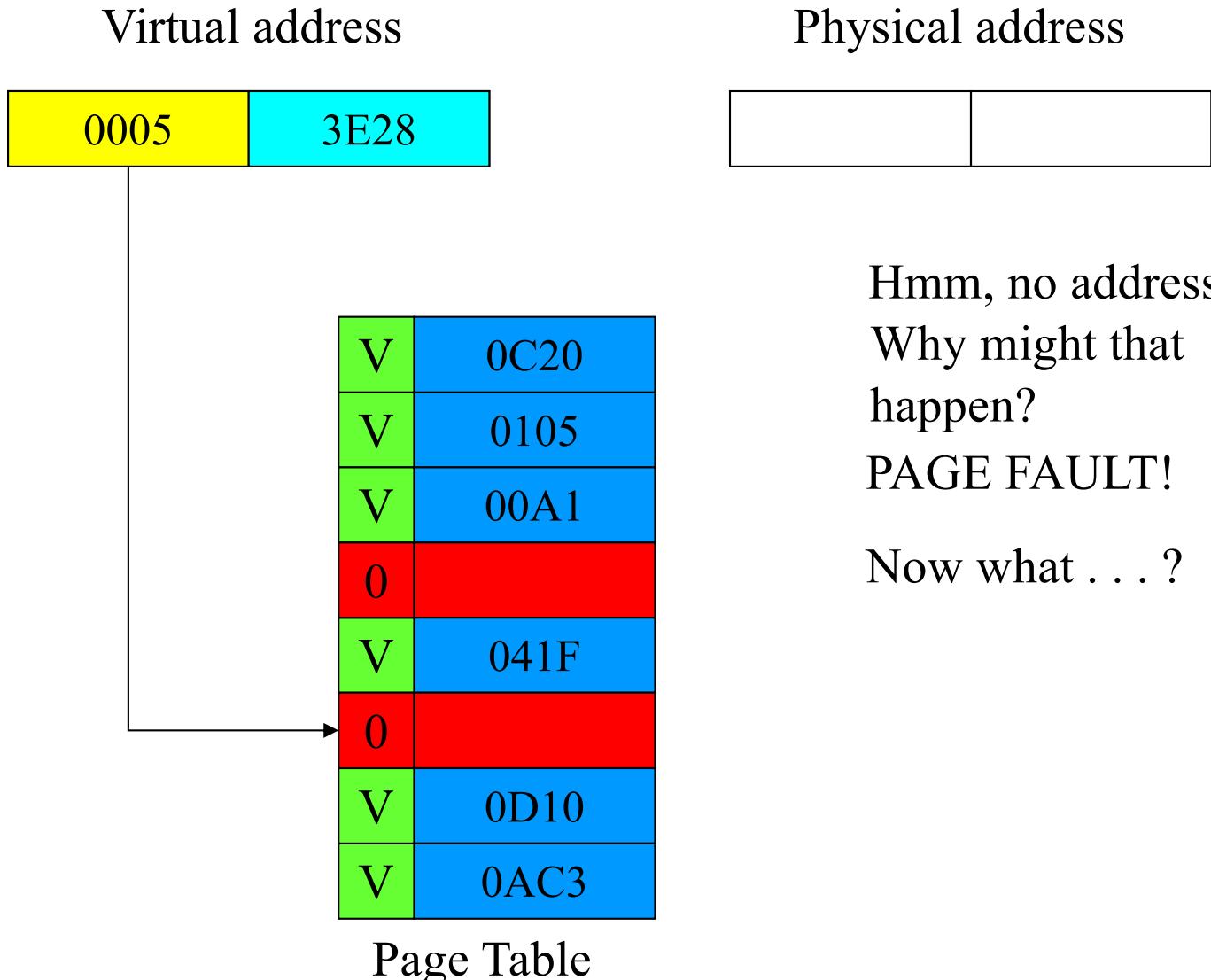
Why is Locality of Reference Usually Present?

- 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?

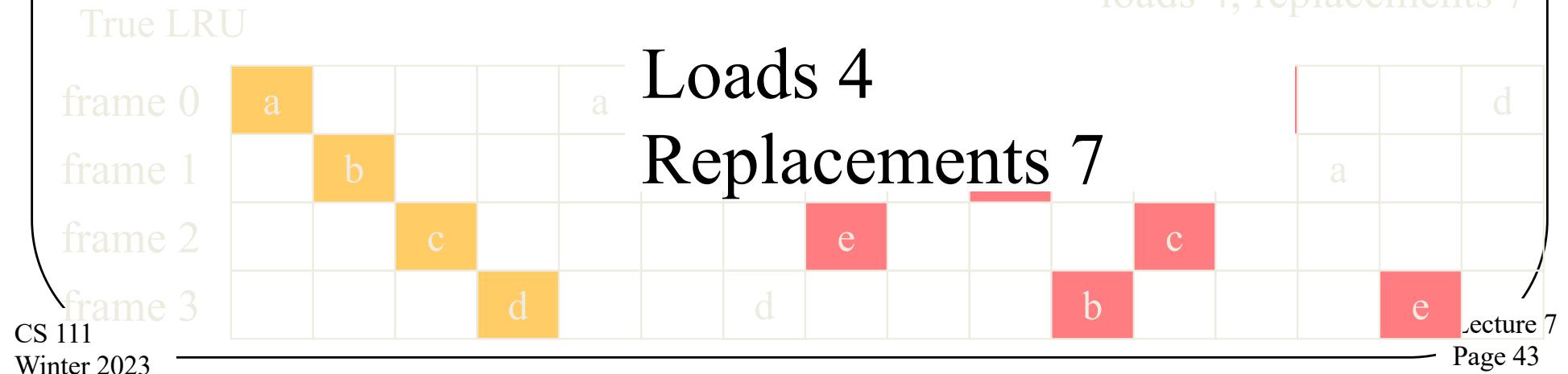
Clock Algorithm Page Replacement

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

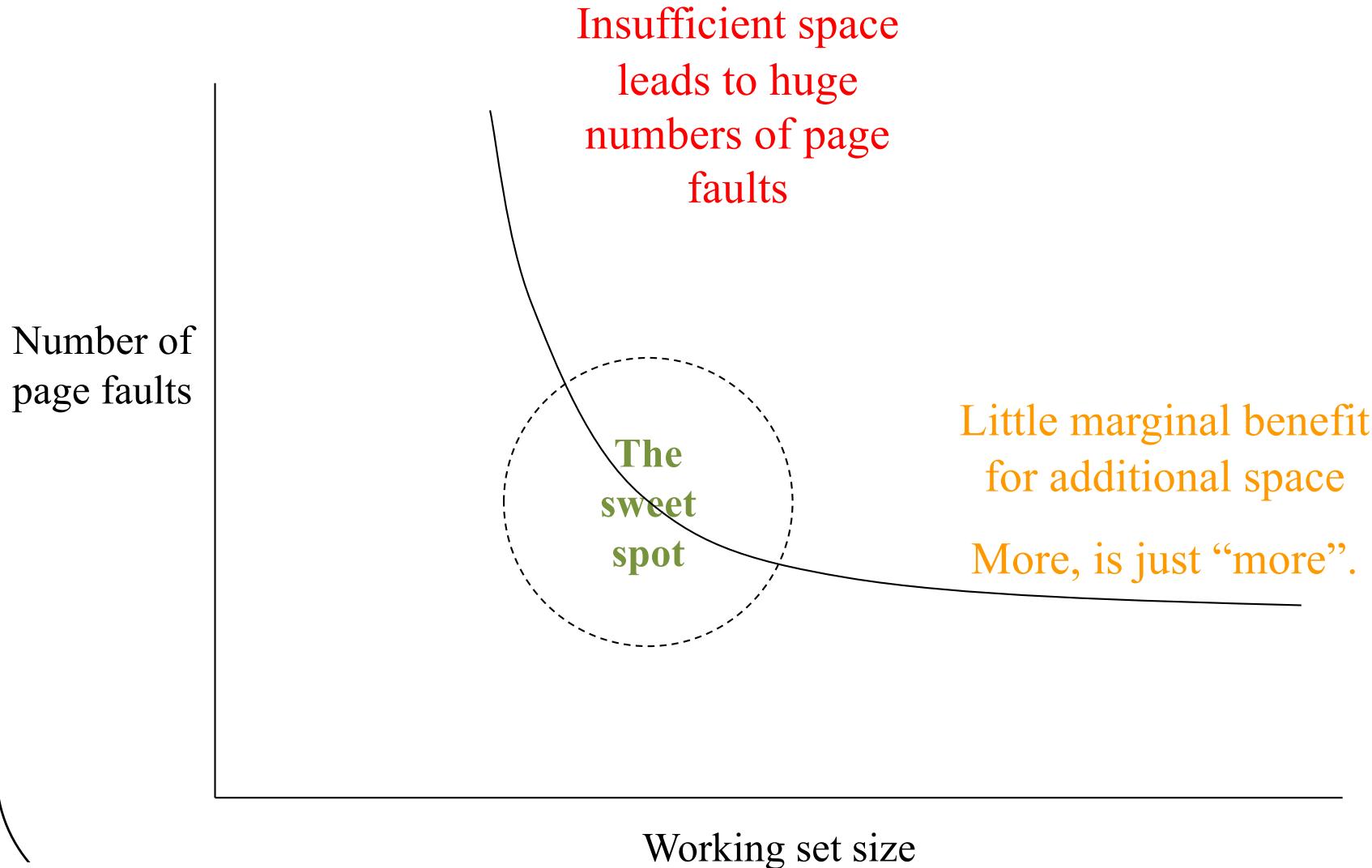
- Set aside some number of page frames for each running process
 - Use an LRU approximation separately for each
- How many page frames per process?
- Fixed number of pages per process is bad
 - Different processes exhibit different locality
 - Which pages are needed changes over time
 - Number of pages 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 ~~com~~ into simpler pieces
- Improved reliability
 - The failure ~~reliability~~ 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

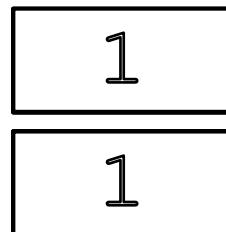
```
load r1,numsigs // = 0  
add r1,=1 // = 1  
store r1,numsigs // =1
```

```
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

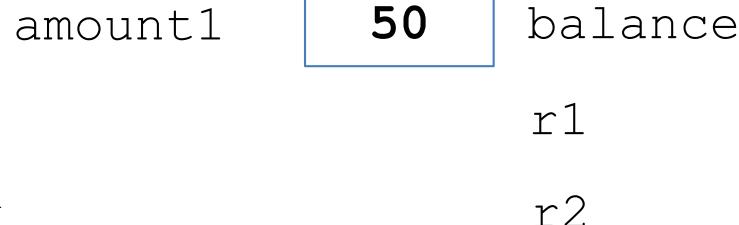
Thread 1

```
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

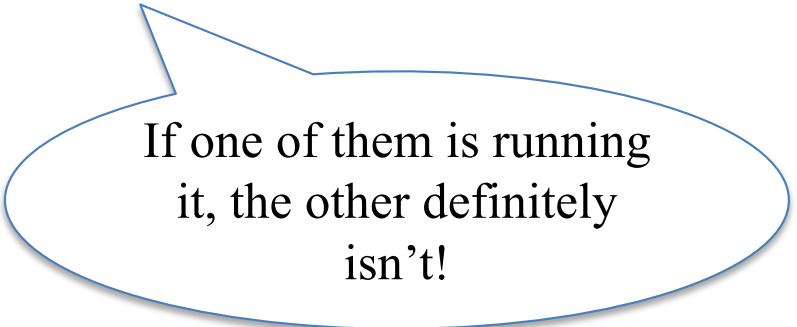
If counter started at 1, it should end at 3
In this execution, it ends at 2

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

thread #2

counter = counter + 1; counter = counter + 1;

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

mov counter, %eax

add \$0x1, %eax

mov %eax, counter

Three instructions . . .

- How can we solve this with locks?

Using Locks For Mutual Exclusion

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



Now the three assembly instructions are mutually exclusive

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?

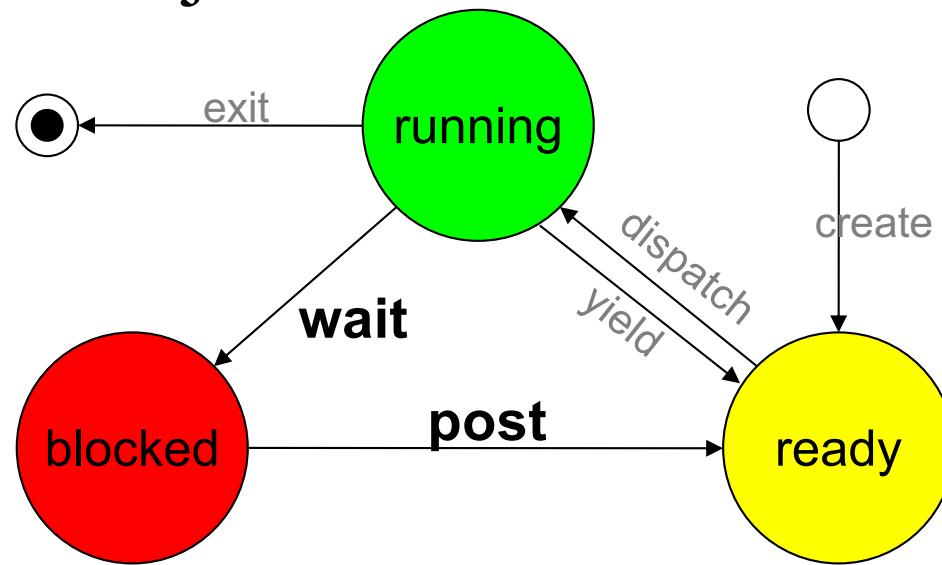
- Spin locking/busy waiting
- Yield and spin ...
- Either spin option may still require mutual exclusion
 - And any time spent spinning is wasted
- And fairness may be an issue
- *Completion events*

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



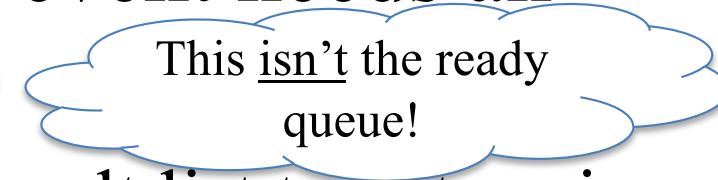
Condition Variables and the OS

- 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

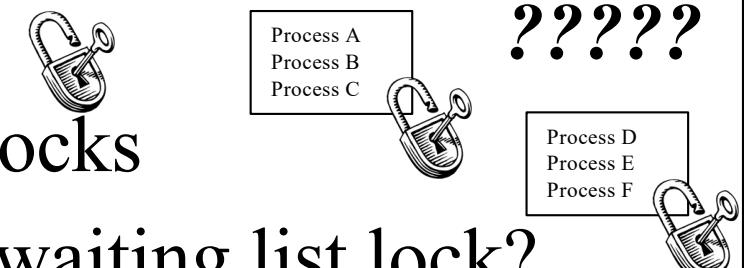
- Suggests each completion event needs an associated waiting list 
 - When posting an event, consult list to determine who's waiting for that event
 - Then what?
 - Wake up everyone on that event's waiting list?
 - One-at-a-time in FIFO order?
 - One-at-a-time in priority order (possible starvation)?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