

Operating Systems

2019

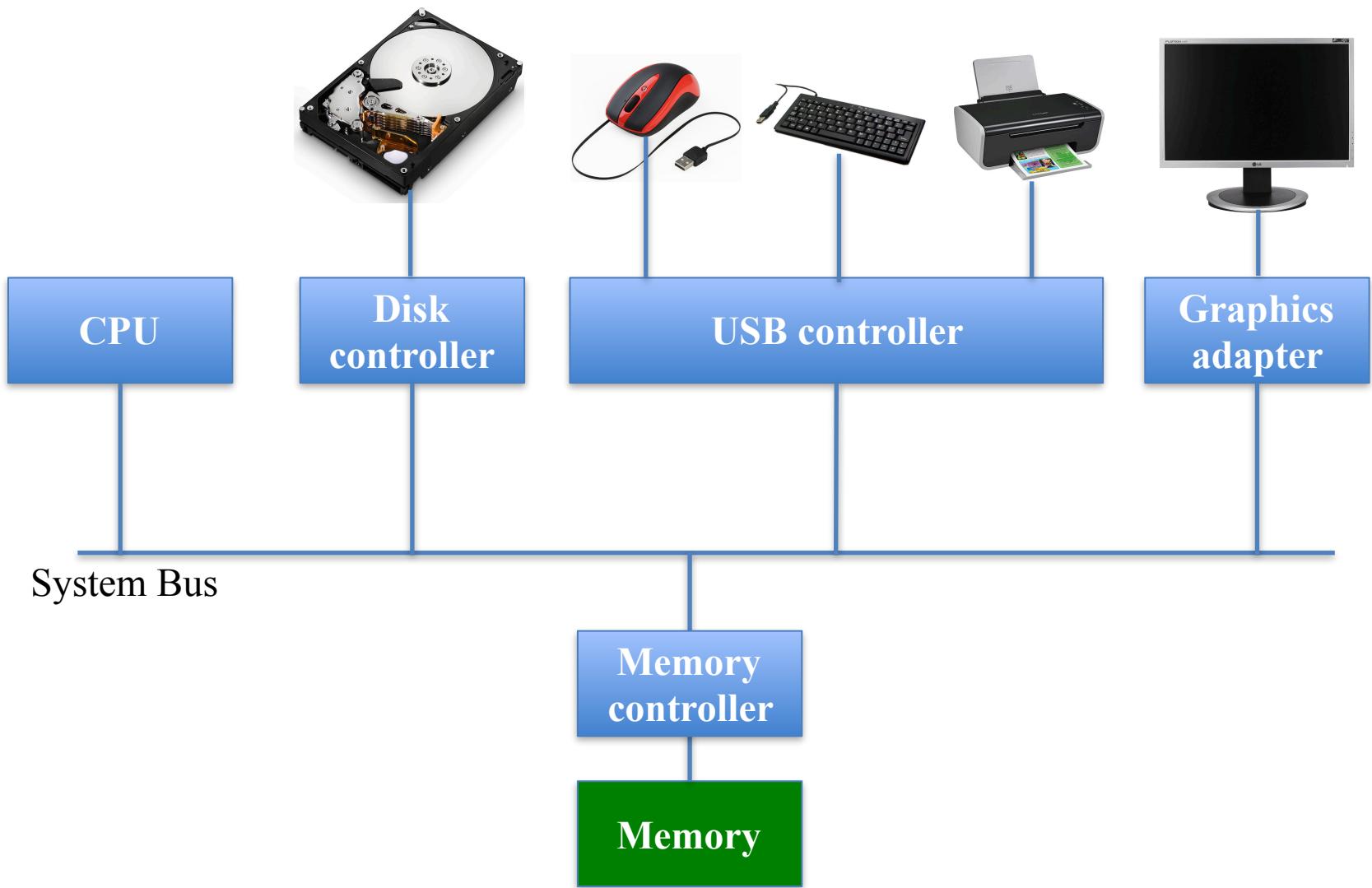
Introduction

Michael O'Boyle
mob@inf.ed.ac.uk

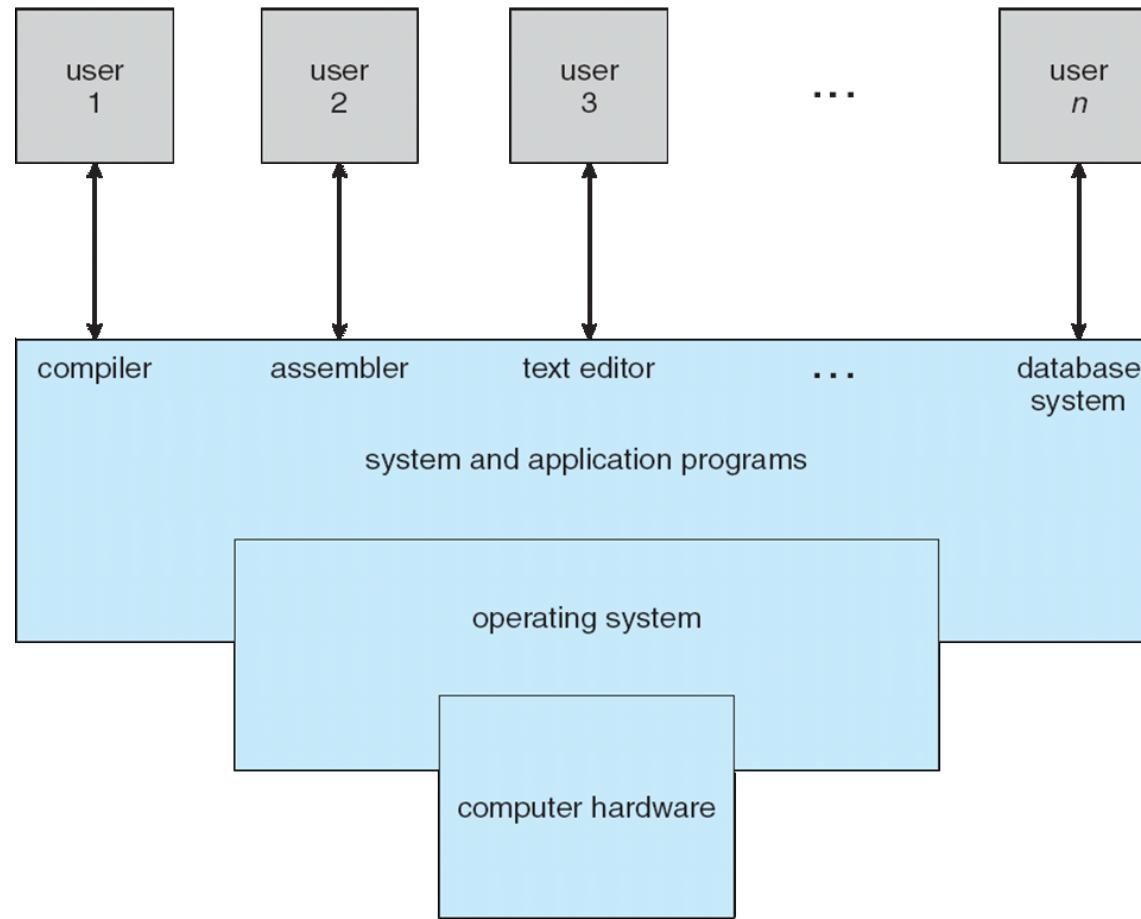
Overview

- Introduction
- Definition of an operating system
 - Hard to pin down
- Historical look
- Key functions
 - Timesharing
 - Multitasking
- Various types of OS
 - Depends on platform and scenario

Modern computer system



Four Components of a Computer System

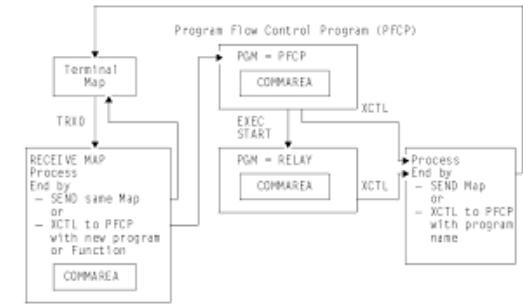


What is an Operating System?

- A big program
 - Linux kernel has 20M lines of code
- A program that
 - **manages** a computer's hardware
- A program that
 - acts as an **intermediary** between the user of a computer and computer hardware

Operating System Definition

- OS is a **resource allocator**
 - Manages all resources
 - Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
 - Controls execution of programs to prevent errors and improper use of the computer



Operating System Definition (Cont.)

- No universally accepted definition
- “Everything a vendor ships when you order an operating system” is a good approximation
 - But varies wildly
- “The one program running at all times on the computer” is the **kernel**.
 - **Not the case in bare-metal embedded systems**
- Everything else is either
 - a system program (ships with the operating system) , or
 - an application program.

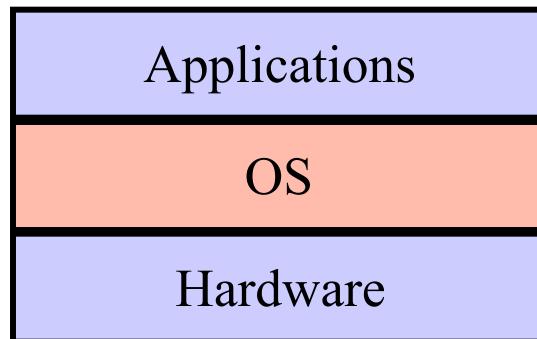


Some goals of operating systems

- Simplify the execution of user programs and make **solving user problems easier**
- Use computer hardware **efficiently**
 - Allow sharing of hardware and software resources
- Make application software **portable and versatile**
- Provide isolation, security and protection among user programs
- Improve overall system reliability
 - error confinement, fault tolerance, reconfiguration



The traditional Picture



- “The OS is everything you don’t need to write in order to run your application”
 - This depiction invites you to think of the OS as a library; we’ll see that
- In some ways, it is:
 - all operations on I/O devices require OS calls (*syscalls*)
- In other ways, it isn’t:
 - you use the CPU/memory without OS calls
 - it intervenes without having been explicitly called

The OS and Hardware

- An OS **mediates** programs' access to hardware resources (*sharing and protection*)
 - computation (CPU)
 - volatile storage (memory) and persistent storage (disk, etc.)
 - network communications (TCP/IP stacks, Ethernet cards, etc.)
 - input/output devices (keyboard, display, sound card, etc.)
- The OS **abstracts** hardware into **logical resources** and well-defined **interfaces** to those resources (*ease of use*)
 - processes (CPU, memory)
 - files (disk)
 - programs (sequences of instructions)
 - sockets (network)

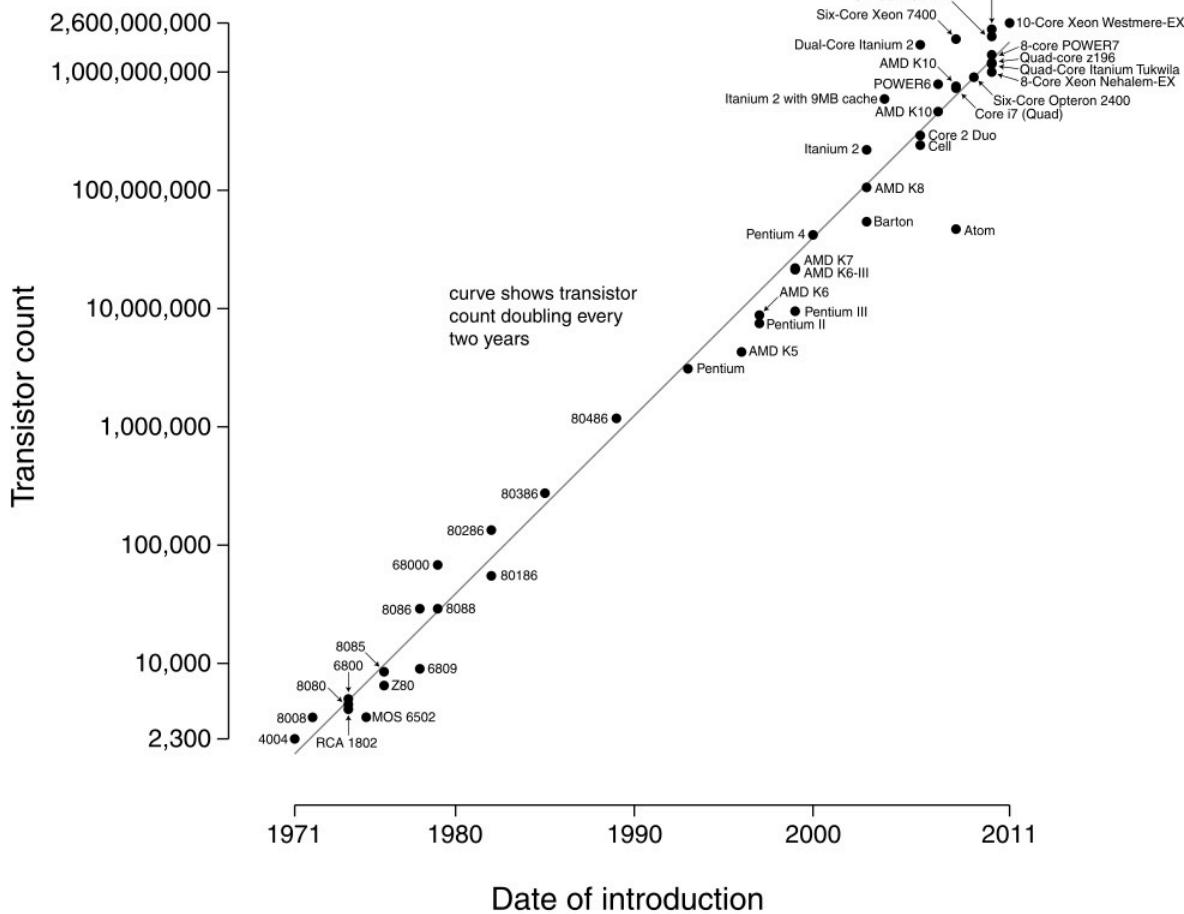
Why Bother with an OS?

- Application benefits
 - programming **simplicity**
 - see high-level abstractions (files) instead of low-level hardware details (device registers)
 - abstractions are **reusable** across many programs
 - **portability** (across machine configurations or architectures)
 - device independence: 3com card or Intel card?
- User benefits
 - **safety**
 - program “sees” its own virtual machine, thinks it “owns” the computer
 - OS **protects** programs from each other
 - OS **fairly multiplexes** resources across programs
 - **efficiency** (cost and speed)
 - **share** one computer across many users
 - **concurrent** execution of multiple programs

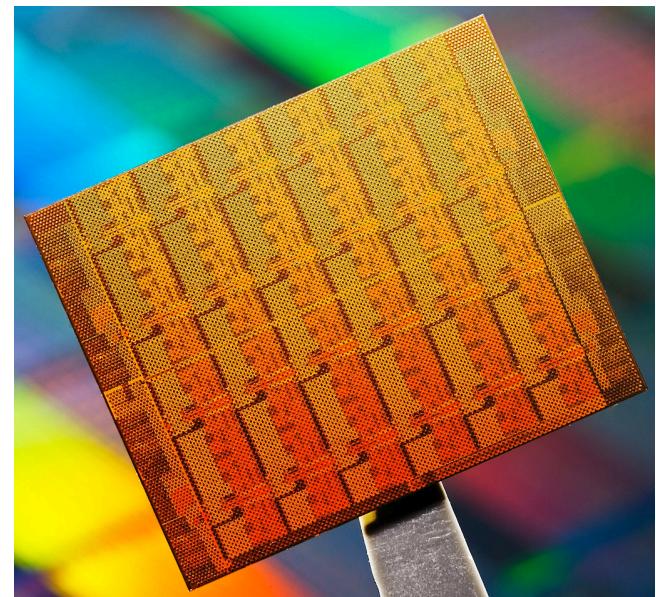
Hardware/Software Changes with Time

Hardware Complexity Increases

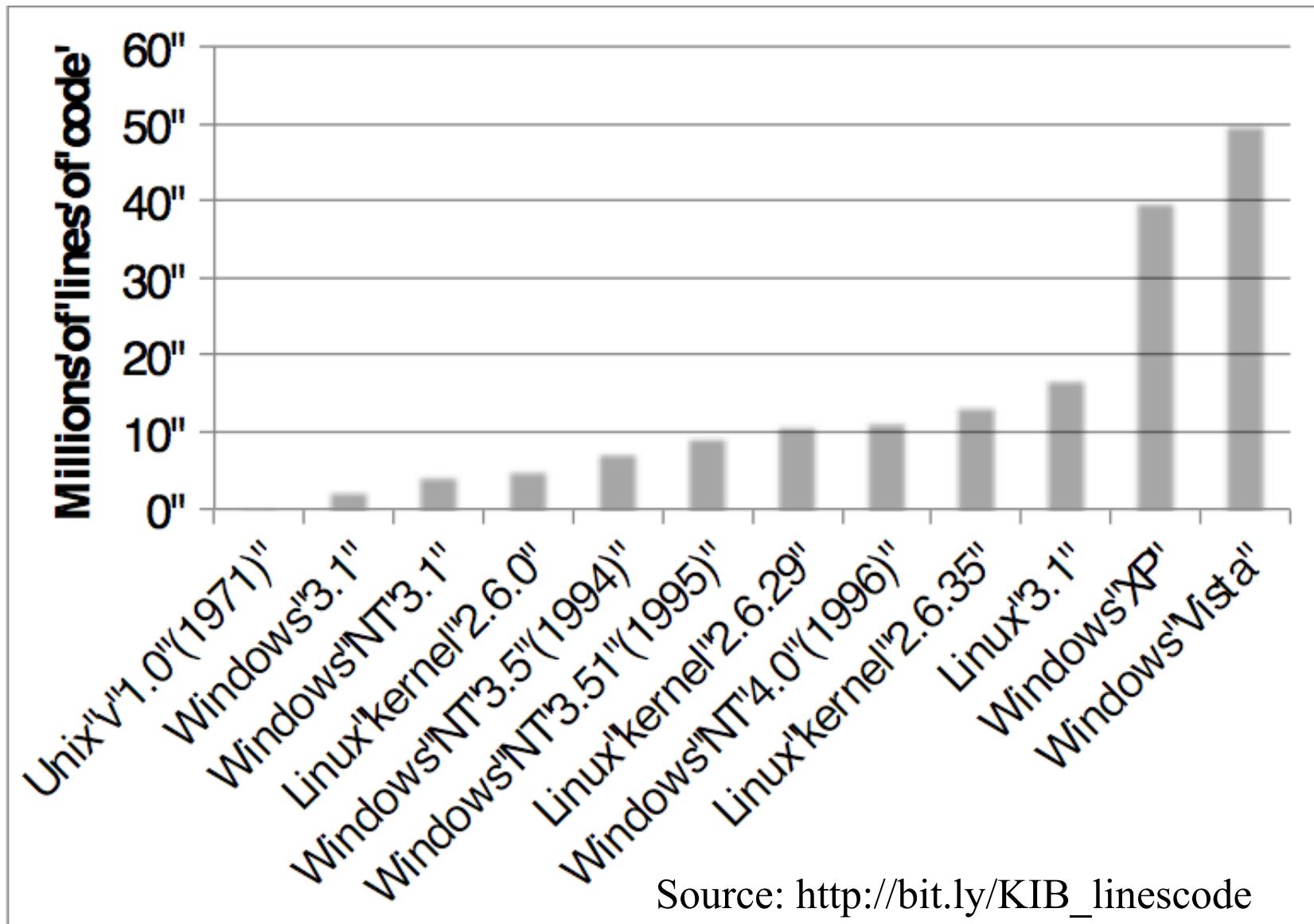
Microprocessor Transistor Counts 1971-2011 & Moore's Law



Moore's Law: 2X transistors/Chip Every 1.5 years



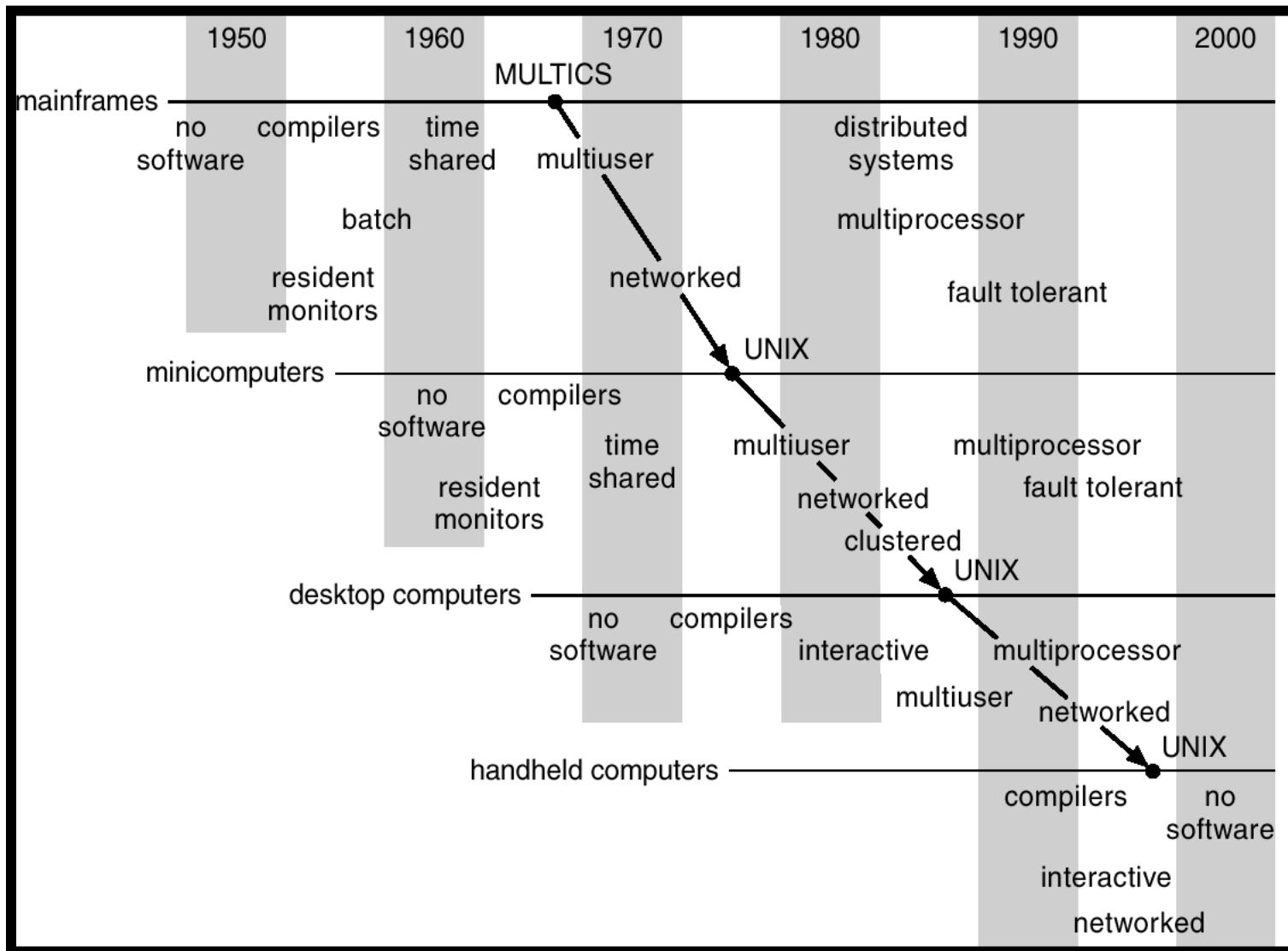
Software Complexity Increases



Hardware/Software Changes with Time

- 1960s: mainframe computers (IBM)
- 1970s: minicomputers (DEC)
- 1980s: microprocessors and workstations (SUN), local-area networking, the Internet
- 1990s: PCs (rise of Microsoft, Intel, Dell), the Web
- 2000s:
 - Internet Services / Clusters (Amazon)
 - General Cloud Computing (Google, Amazon, Microsoft)
 - Mobile/ubiquitous/embedded computing (iPod, iPhone, iPad, Android)
- 2010s: sensor networks, “data-intensive computing,” computers and the physical world
- 2020: exascale, IoT ??

Progression of Concepts and Form Factors



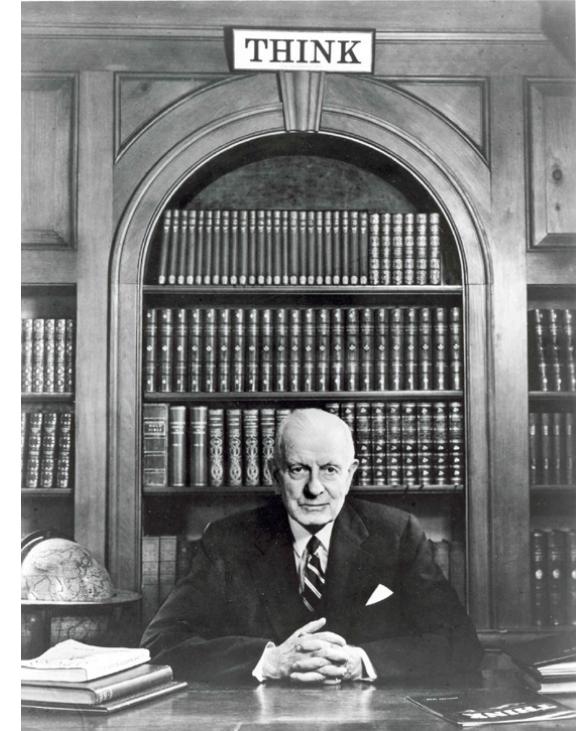
An OS History Lesson

- Operating systems are the result of a 60 year long evolutionary process
 - They were born out of need
- Examine their evolution
- Explains what some of their functions are, and why

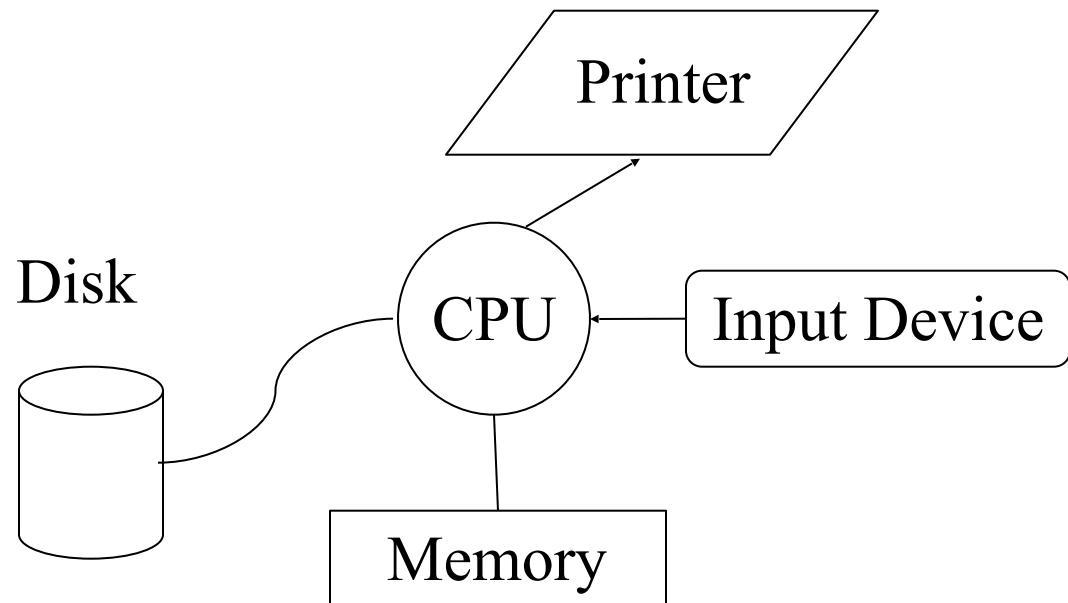


Early days

- 1943
 - T.J. Watson (created IBM):
“I think there is a world market for maybe five computers.”
- Fast forward ... 1950
 - There are maybe 20 computers in the world
 - They were unbelievably expensive
 - Machine time is considerably more valuable than person time!
 - Ergo: efficient use of the hardware is paramount
 - Operating systems are born
 - They carry with them the vestiges of these economic assumptions



Simplified early computer

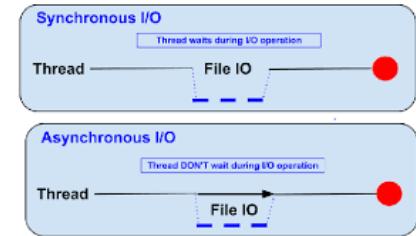


The OS as a linked library

- In the very beginning...
 - OS was just a library of code that you linked into your program; programs were loaded in their entirety into memory, and executed
 - “OS” had an “API” that let you control the disk, control the printer, etc.
 - Interfaces were literally switches and blinking lights
 - When you were done running your program, you’d leave and turn the computer over to the next person
- Not so very different from some embedded devices today

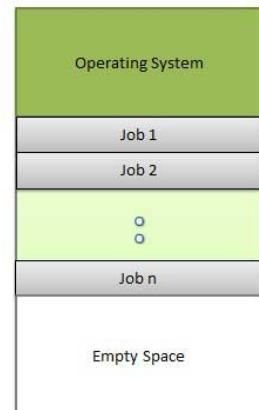
Asynchronous I/O

- The disk was really slow
- Add hardware so that the disk could operate without tying up the CPU
 - Disk controller
- Programmers could now write code that:
 - Starts an I/O
 - Goes off and does some computing
 - Checks if the I/O is done at some later time
- Upside
 - Helps increase (expensive) CPU utilization
- Downsides
 - It's hard to get right
 - The benefits are job specific



Multiprogramming

- To further increase system utilization, **multiprogramming** OSs were invented
 - keeps multiple runnable jobs loaded in memory at once
 - overlaps I/O of one job with computing of another
 - while one job waits for I/O completion, another job uses the CPU
- Can get rid of asynchronous I/O within individual jobs
 - Life of application programmer becomes simpler; only the OS programmer needs to deal with asynchronous events
- How do we tell when devices are done?
 - Interrupts
 - Polling
- What new requirements does this impose?



Timesharing



- To support interactive use, create a **timesharing OS**:
 - multiple terminals into one machine
 - each user has illusion of entire machine to him/herself
 - optimize response time, perhaps at the cost of throughput
- Timeslicing
 - divide CPU equally among the users
 - if job is truly interactive (e.g., editor), then can jump between programs and users faster than users can generate work
 - permits users to interactively view, edit, debug running programs
- Multics system (operational 1968) was the first large timeshared system
 - nearly all OS concepts can be traced back to Multics

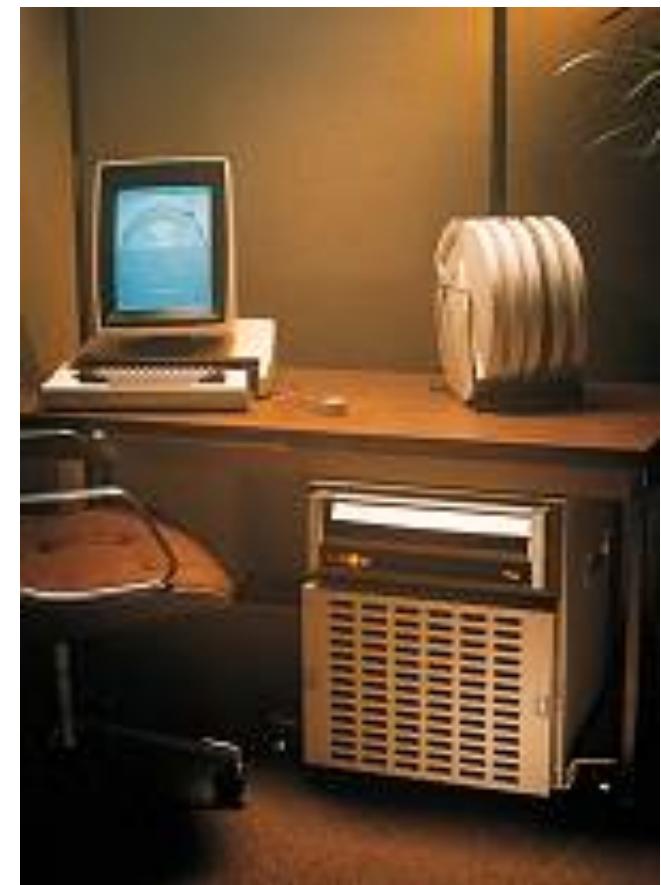
Parallel Systems

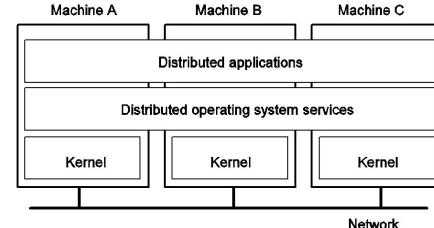


- Some applications can be written as multiple parallel **threads** or **processes**
 - can speed up the execution by running multiple threads/processes simultaneously on multiple CPUs [Burroughs D825, 1962]
 - need OS and language primitives for dividing program into multiple parallel activities
 - need OS primitives for fast communication among activities
 - degree of speedup dictated by communication/computation ratio
- Many flavors of parallel computers today
 - Multi-cores – all(ish) processors are parallel
 - SMPs (symmetric multi-processors)
 - MPPs (massively parallel processors)
 - NOWs (networks of workstations) –less common
 - Massive clusters (Google, Amazon.com, Microsoft)
 - Heterogeneous accelerators eg GPUs

Personal Computing

- Primary goal was to enable new kinds of applications
- Bit mapped display [Xerox Alto, 1973]
 - new classes of applications
 - new input device (the mouse)
- Move computing near the display
 - why?
- Window systems
 - the display as a managed resource
- Local area networks [Ethernet]
 - why?
- Effect on OS?



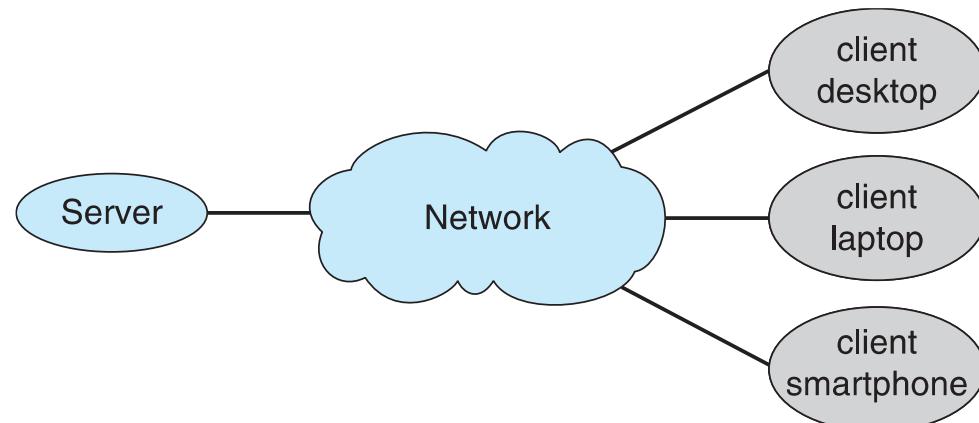


Distributed OS

- Distributed systems to facilitate use of geographically distributed resources
 - workstations on a LAN
 - servers across the Internet
- Supports communications between programs
 - interprocess communication
 - message passing, shared memory
 - networking stacks
- Sharing of distributed resources (hardware, software)
 - load balancing, authentication and access control, ...
- Speedup isn't the issue
 - access to diversity of resources is goal

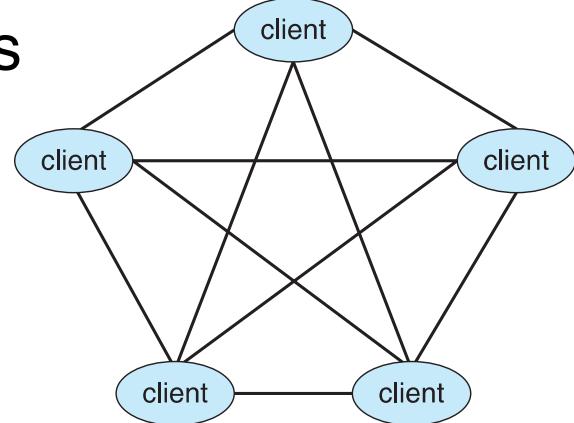
Client/Server Computing

- Dumb terminals supplanted by smart PCs
 - Many systems now servers, responding to requests generated by clients
- Compute-server system
 - provides an interface to client to request services (i.e., database)
- File-server system
 - provides interface for clients to store and retrieve files
- Mail server/service
- Print server/service
- Game server/service
- Music server/service
- Web server/service
- etc.

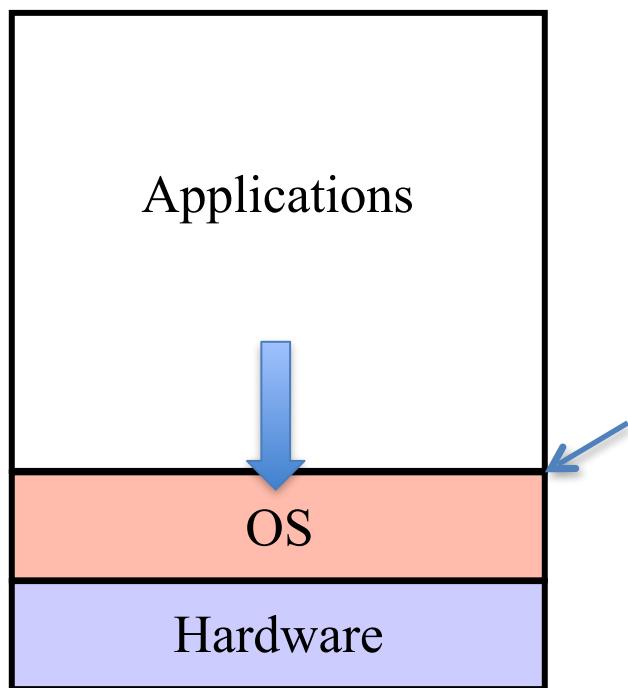


Peer-to-Peer (p2p) Systems

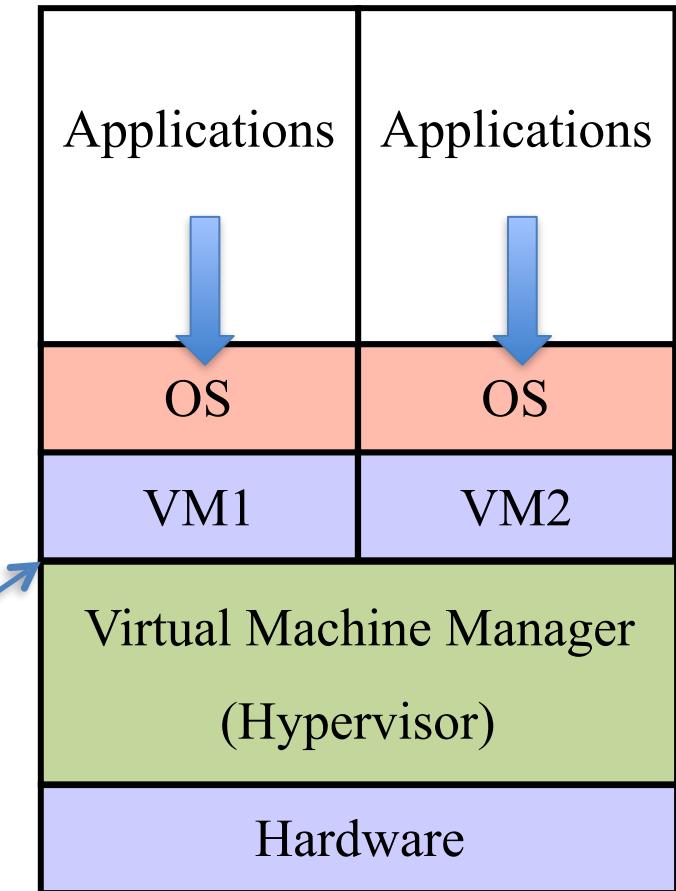
- Another model of distributed system
 - Does not distinguish clients and servers
 - All nodes are considered peers
 - Each may act as client or server
-
- Node must join P2P network
 - Registers its service with central lookup service on network, or
 - Broadcast request for service and respond to requests for service via ***discovery protocol***
 - Examples include Napster and Gnutella, **Voice over IP (VoIP)** such as Skype



Virtualization

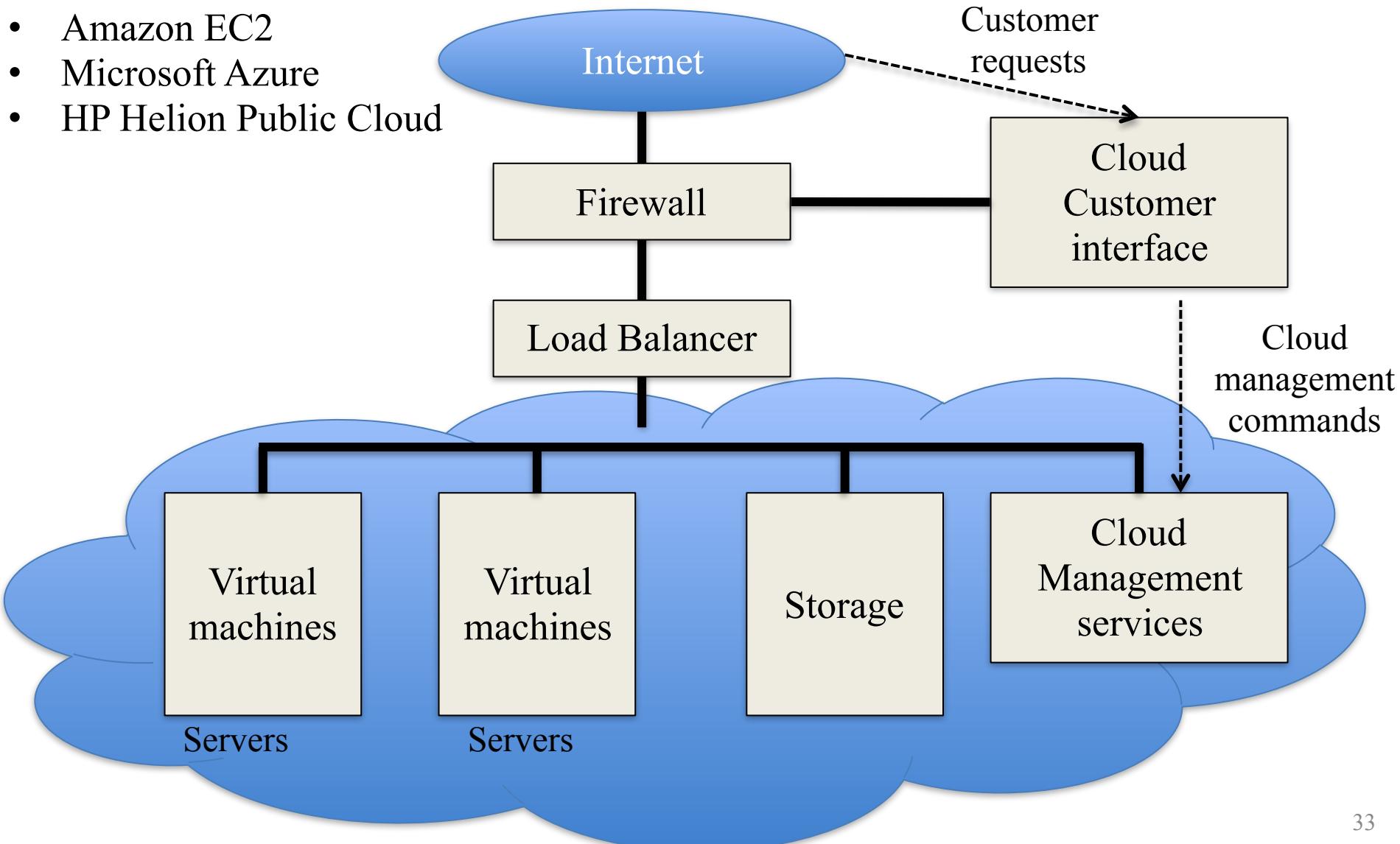


Programming
Interface



Cloud Computing

- Amazon EC2
- Microsoft Azure
- HP Helion Public Cloud



The major OS issues

- **structure**: how is the OS organized?
- **sharing**: how are resources shared across users?
- **naming**: how are resources named (by users or programs)?
- **security**: how is the integrity of the OS and its resources ensured?
- **protection**: how is one user/program protected from another?
- **performance**: how do we make it all go fast?
- **reliability**: what happens if something goes wrong (either with hardware or with a program)?
- **extensibility**: can we add new features?
- **communication**: how do programs exchange information, including across a network?

More OS issues...

- **concurrency**: how are parallel activities (computation and I/O) created and controlled?
- **scale**: what happens as demands or resources increase?
- **persistence**: how do you make data last longer than program executions?
- **distribution**: how do multiple computers interact with each other?
- **accounting**: how do we keep track of resource usage, and perhaps charge for it?

There are tradeoffs, solution depends on scenario

Summary

- Introduction
- Definition of an operating system
 - Hard to pin down
- Historical look
- Key functions
 - Timesharing
 - Multitasking
- Various types of OS
 - Depends on platform and scenario
- Next lecture: structure and organisation

Operating Systems

Operating System Structure

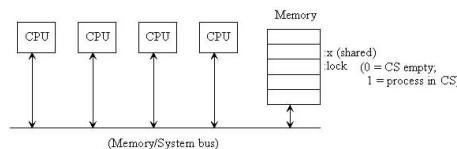
Lecture 2
Michael O'Boyle

Overview

- Architecture impact
- User operating interaction
 - User vs kernel
 - Syscall
- Operating System structure
 - Layers
 - Examples

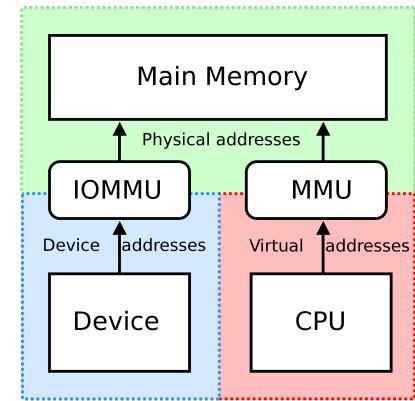
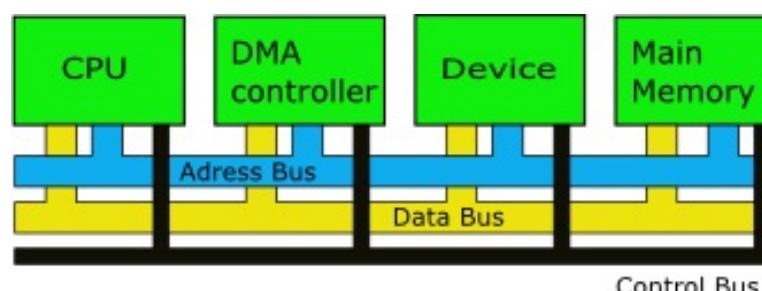
Lower-level architecture affects (is affected by) the OS

- The operating system supports sharing and protection
 - multiple applications can run concurrently, sharing resources
 - a buggy or malicious application cannot disrupt other applications or the system
- There are many approaches to achieving this
- The architecture determines which approaches are viable (reasonably efficient, or even possible)
 - includes instruction set (synchronization, I/O, ...)
 - also hardware components like MMU or DMA controllers



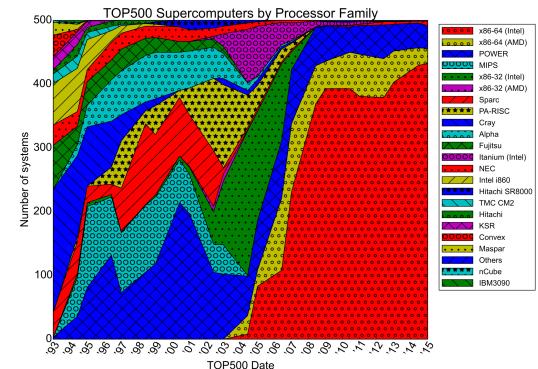
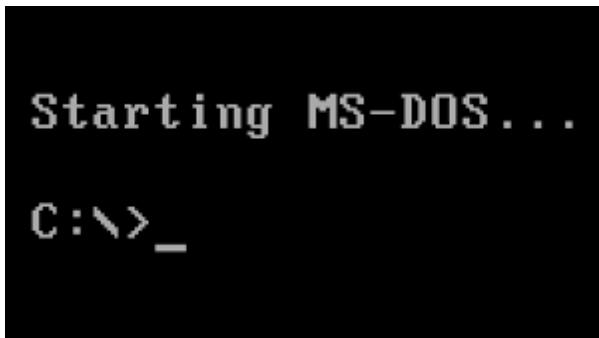
```
Functionality of Test-and-Set Instruction  
boolean Test-and-Set ( boolean & lock ) {  
    boolean value = lock;  
    lock = TRUE;  
    return value;  
}
```

```
Simple Mutual Exclusion  
lock = FALSE; // initialization  
Entry Section  
do ( // loop forever  
    while Test-and-Set(lock) {  
        no-op;  
    } // end while  
    critical section  
    Exit Section  
    lock = FALSE;  
    remainder section  
    while (TRUE)
```



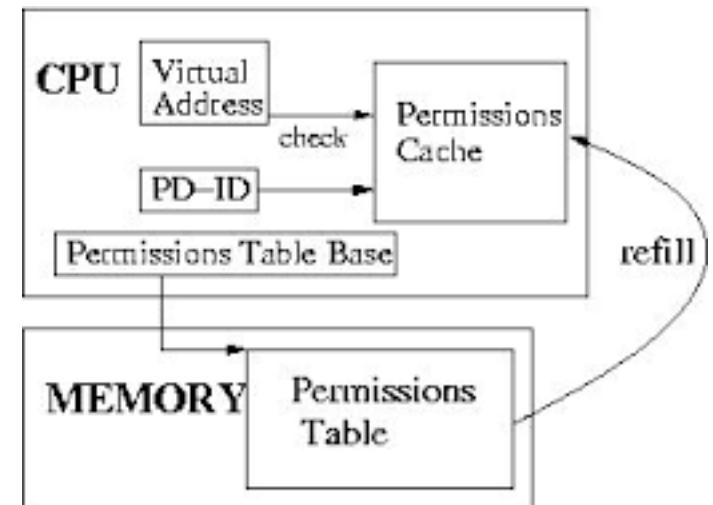
Architecture support for the OS

- Architectural support can simplify OS tasks
 - e.g.: early PC operating systems (DOS, MacOS) lacked support for virtual memory, in part because at that time PCs lacked necessary hardware support
- Until recently, Intel-based PCs still lacked support for 64-bit addressing
 - has been available for a decade on other platforms: MIPS, Alpha, IBM, etc...
 - Change driven by AMD's 64-bit architecture



Architectural features affecting OS's

- These features were built primarily to support OS's:
 - timer (clock) operation
 - synchronization instructions
 - e.g., atomic test-and-set
 - memory protection
 - I/O control operations
 - interrupts and exceptions
 - protected modes of execution
 - kernel vs. user mode
 - privileged instructions
 - system calls
 - Including software interrupts
 - virtualization architectures
- ASPLOS



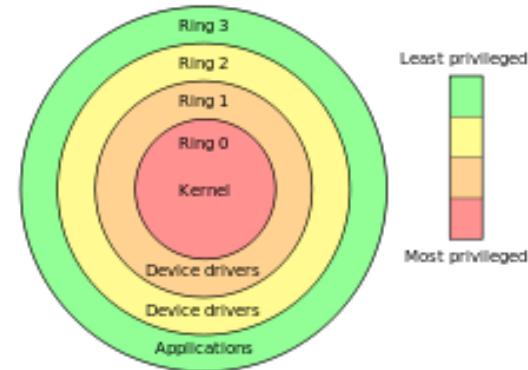
Privileged instructions

- Some instructions are restricted to the OS
 - known as **privileged** instructions
- Only the OS can:
 - directly access I/O devices (disks, network cards)
 - manipulate memory state management
 - page table pointers, TLB loads, etc.
 - manipulate special ‘mode bits’
 - interrupt priority level
- Restrictions provide safety and security



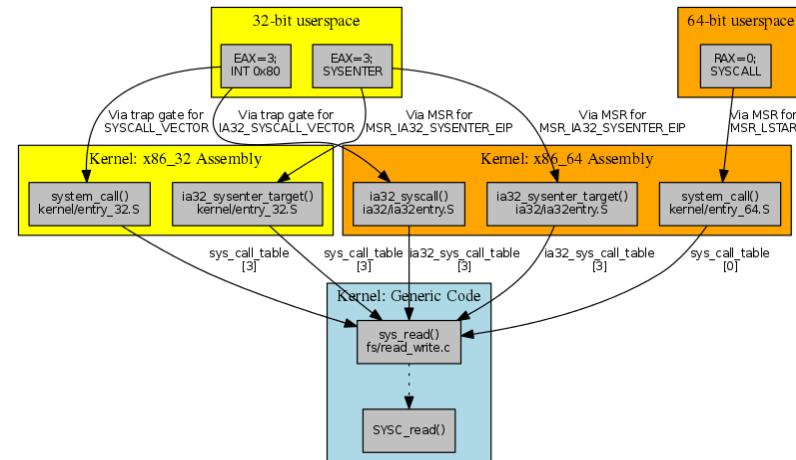
OS protection

- So how does the processor know if a privileged instruction should be executed?
 - the architecture must support at least two modes of operation: kernel mode and user mode
 - x86 support 4 protection modes
 - mode is set by status bit in a protected processor register
 - user programs execute in user mode
 - OS executes in kernel (privileged) mode (OS == kernel)
- Privileged instructions can only be executed in kernel (privileged) mode
 - if code running in user mode attempts to execute a privileged instruction the Illegal execution trap



Crossing protection boundaries

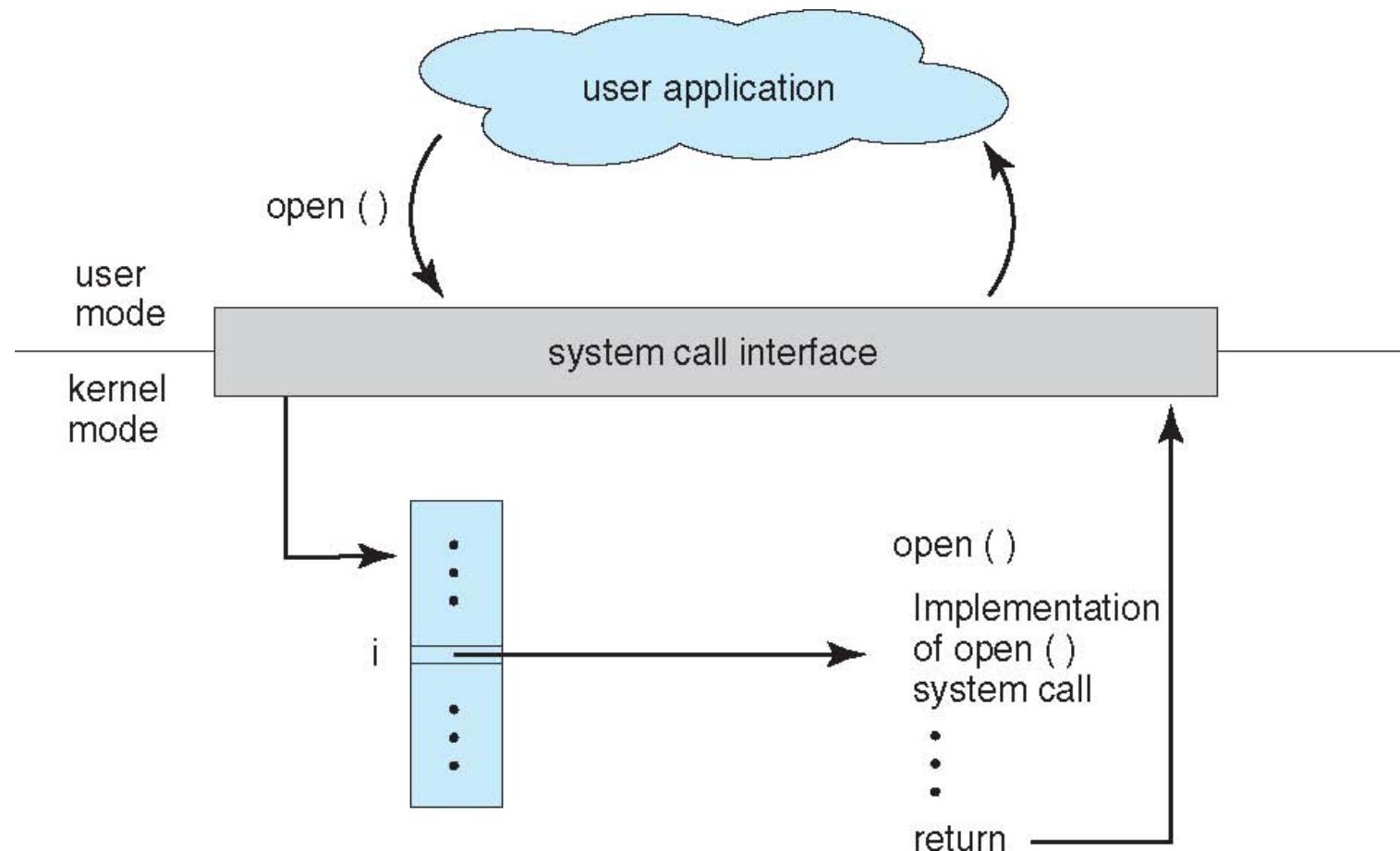
- So how do user programs do something privileged?
 - e.g., how can you write to a disk if you can't execute an I/O instructions?
- User programs must call an OS procedure – that is ask the OS to do it for them
 - OS defines a set of system calls
 - User-mode program executes system call instruction
- Syscall instruction
 - Like a protected procedure call



Syscall

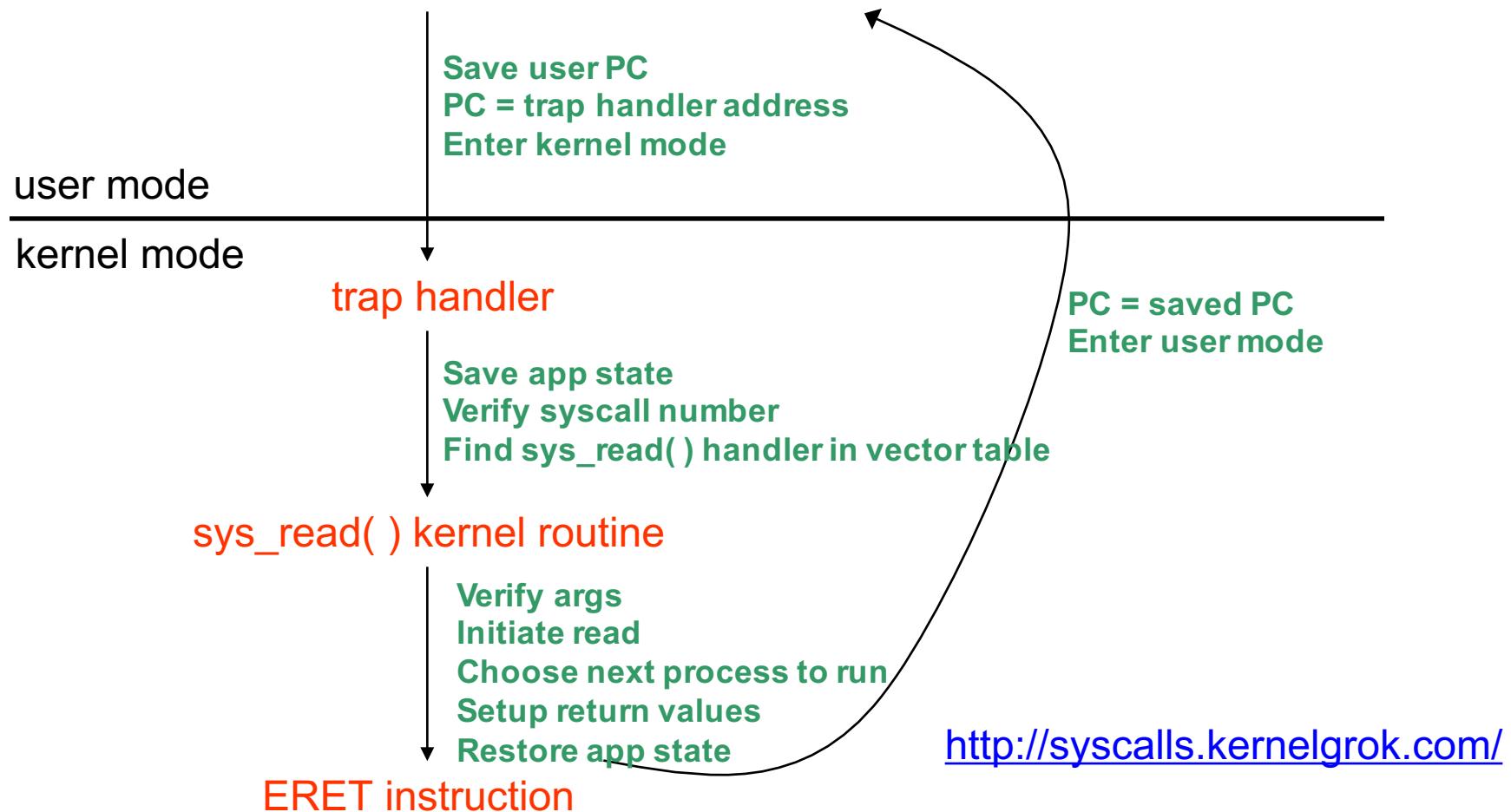
- The syscall instruction atomically:
 - Saves the current PC
 - Sets the execution mode to privileged
 - Sets the PC to a handler address
- Similar to a procedure call
 - Caller puts arguments in a place callee expects (registers or stack)
 - One of the args is a syscall number, indicating which OS function to invoke
 - Callee (OS) saves caller's state (registers, other control state) so it can use the CPU
 - OS function code runs
 - **OS must verify caller's arguments** (e.g., pointers)
 - OS returns using a special instruction
 - Automatically sets PC to return address and sets execution mode to user

API – System Call – OS Relationship



A kernel crossing illustrated

Firefox: `read(int fileDescriptor, void *buffer, int numBytes)`

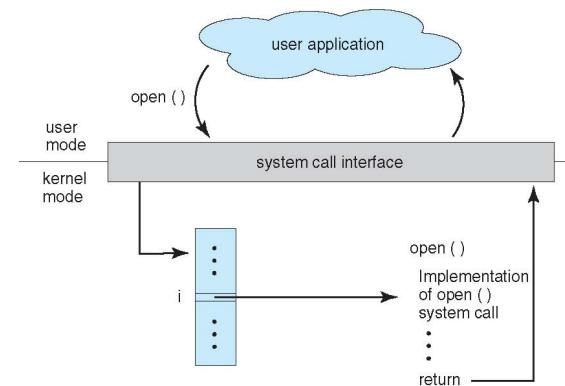


Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

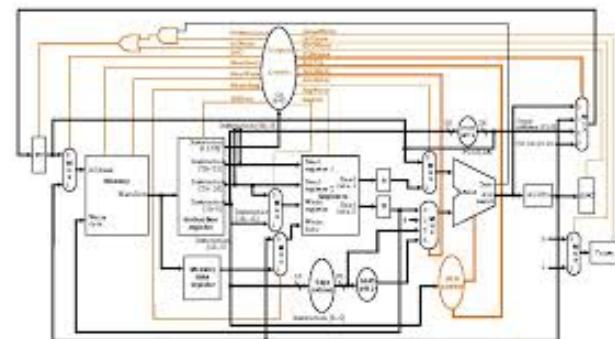
System call issues

- A syscall is not subroutine call, with the caller specifying the next PC.
 - the caller knows where the subroutines are located in memory; therefore they can be target of attack.
- The kernel saves state?
 - Prevents overwriting of values
- The kernel verify arguments
 - Prevents buggy code crashing system
- Referring to kernel objects as arguments
 - Data copied between user buffer and kernel buffer



Exception Handling and Protection

- All entries to the OS occur via the mechanism just shown
 - Acquiring privileged mode and branching to the trap handler are inseparable
- Terminology:
 - **Interrupt**: asynchronous; caused by an external device
 - **Exception**: synchronous; unexpected problem with instruction
 - **Trap**: synchronous; intended transition to OS due to an instruction
- Privileged instructions and resources are the basis for most everything: memory protection, protected I/O, limiting user resource consumption, ...

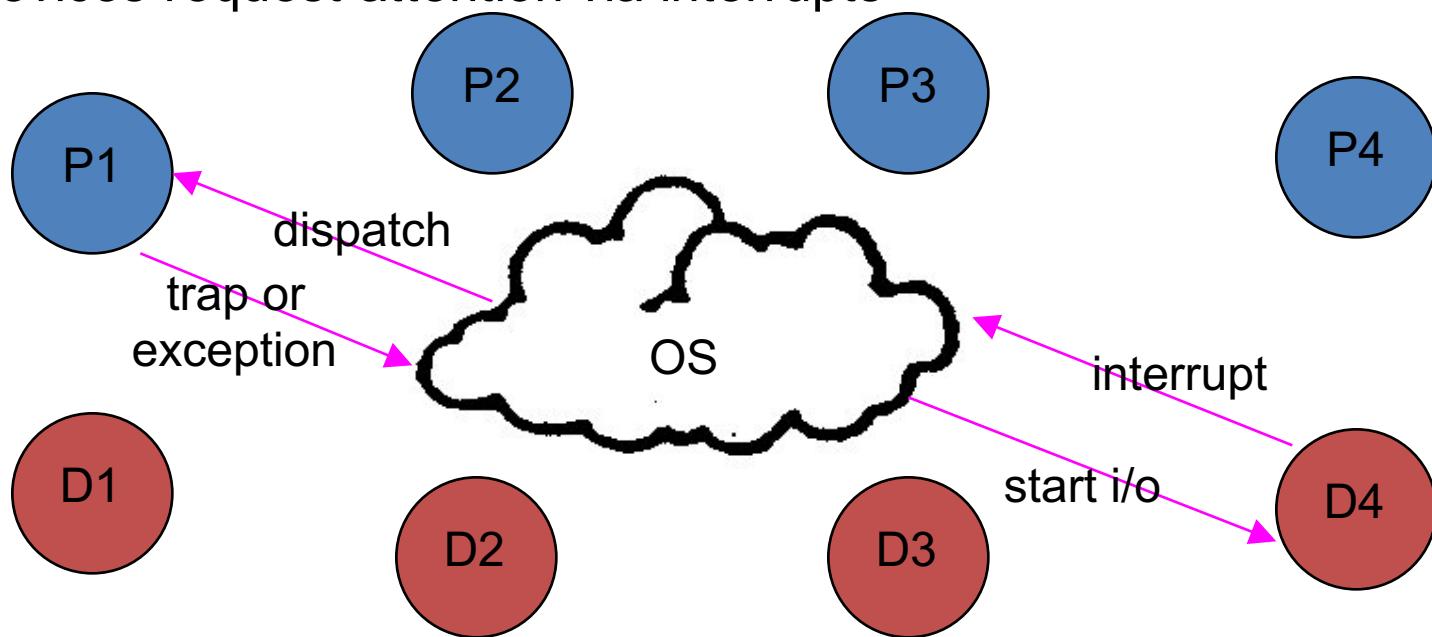


Overview

- *Architecture impact*
- *User operating interaction*
 - *User vs kernel*
 - *Syscall*
- **Operating System structure**
 - Layers
 - Examples

OS structure

- The OS sits between application programs and the hardware
 - it mediates access and abstracts away ugliness
 - programs request services via traps or exceptions
 - devices request attention via interrupts



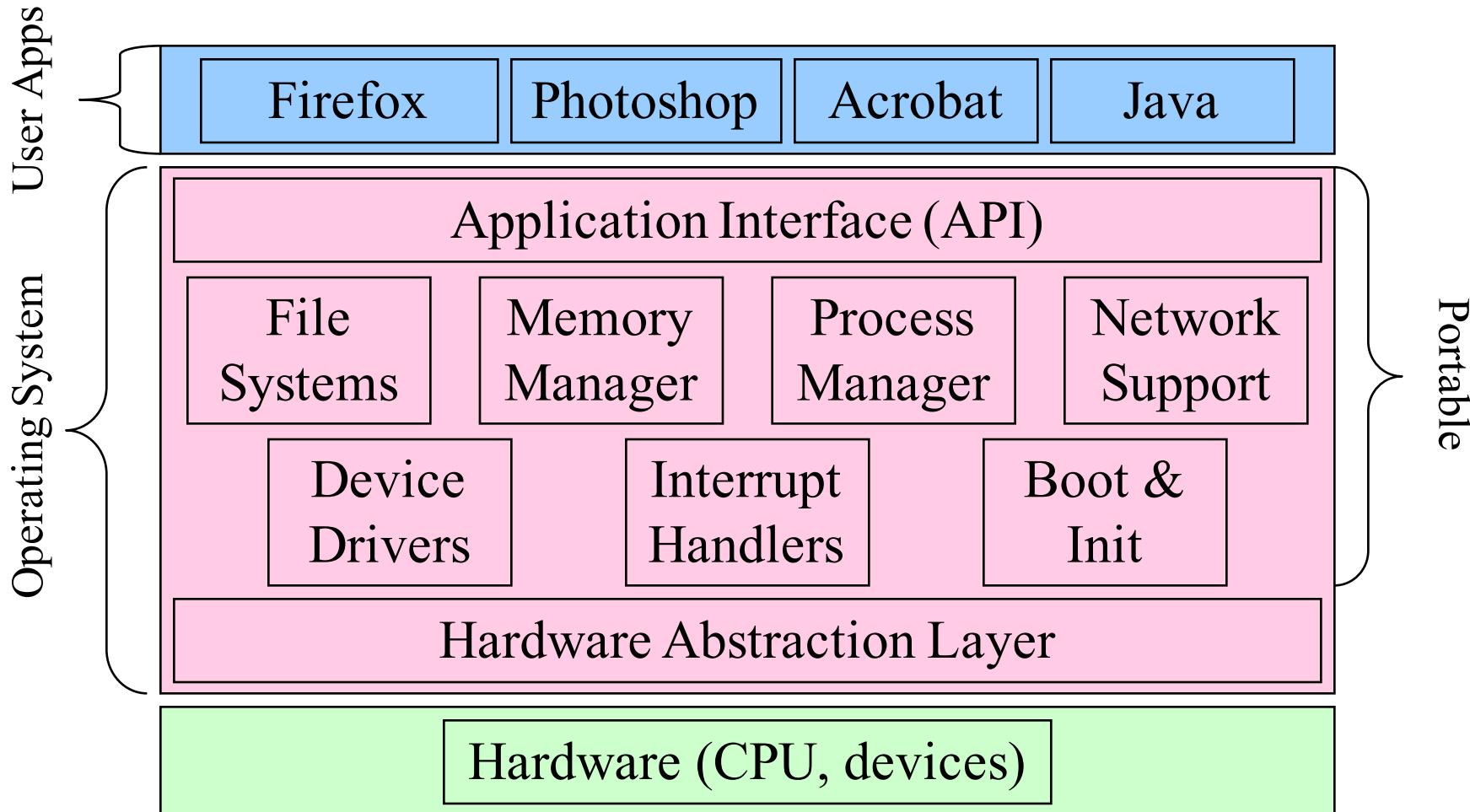
Operating System Design and Implementation

- Design and Implementation of OS not “solvable”, but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system
- **User** goals and **System** goals
 - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
 - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Operating System Design and Implementation

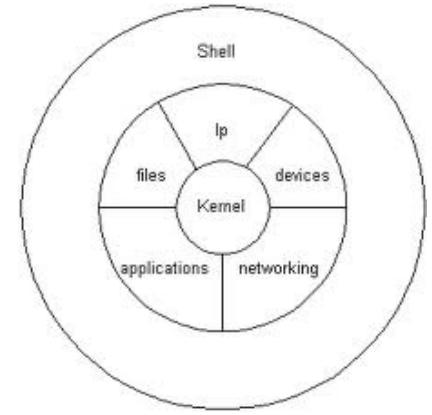
- Important principle to separate
Policy: *What* will be done?
Mechanism: *How* to do it?
- Mechanisms determine how to do something, policies decide what will be done
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)
- Specifying and designing an OS is highly creative task of
software engineering

System layers



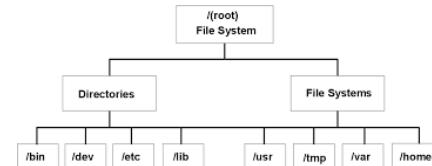
Major OS components

- processes
- memory
- I/O
- secondary storage
- file systems
- protection
- shells (command interpreter, or OS UI)
- GUI
- Networking



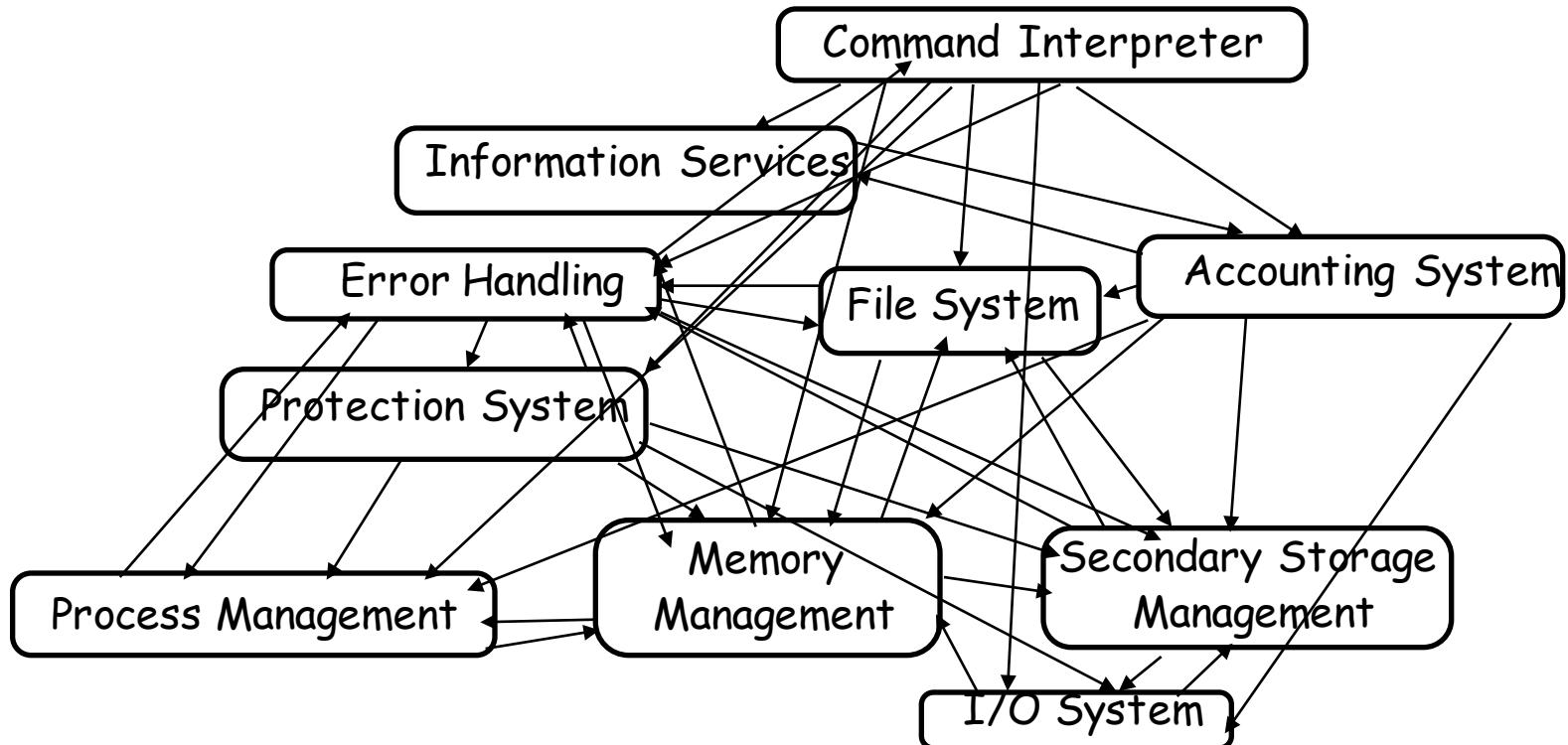
```
#!/bin/bash

<root> env X*() { :; } ; echo shellshock" /bin/sh ><echo completed"
> shellshock
> completed
```

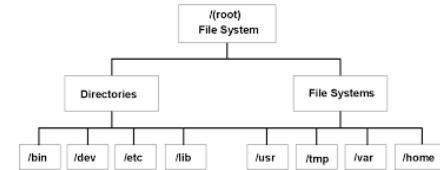


OS structure

- It's not always clear how to stitch OS modules together:



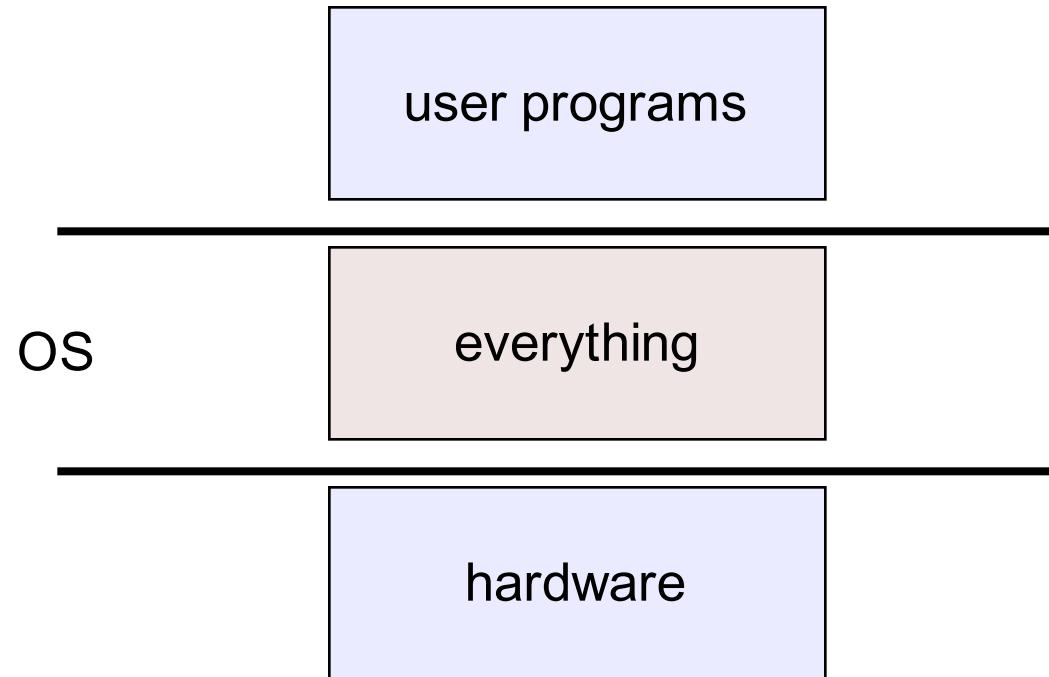
OS structure



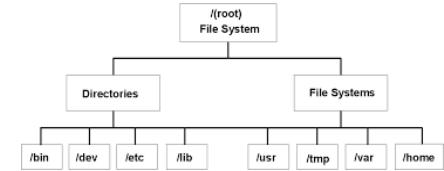
- An OS consists of all of these components, plus:
 - many other components
 - system programs (privileged and non-privileged)
 - e.g., bootstrap code, the init program, ...
- Major issue:
 - how do we organize all this?
 - what are all of the code modules, and where do they exist?
 - how do they cooperate?
- Massive software engineering and design problem
 - design a large, complex program that:
 - performs well, is reliable, is extensible, is backwards compatible,

Early structure: Monolithic

- Traditionally, OS's (like UNIX) were built as a **monolithic** entity:



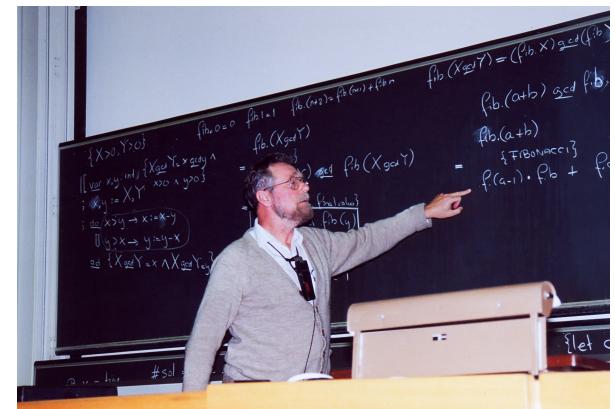
Monolithic design



- Major advantage:
 - cost of module interactions is low (procedure call)
- Disadvantages:
 - hard to understand
 - hard to modify
 - unreliable (no isolation between system modules)
 - hard to maintain
- What is the alternative?
 - find a way to organize the OS in order to simplify its design and implementation

Layering

- The traditional approach is layering
 - implement OS as a set of layers
 - each layer presents an enhanced ‘virtual machine’ to the layer above
- The first description of this approach was Dijkstra’s THE system
 - Layer 5: **Job Managers**
 - Execute users’ programs
 - Layer 4: **Device Managers**
 - Handle devices and provide buffering
 - Layer 3: **Console Manager**
 - Implements virtual consoles
 - Layer 2: **Page Manager**
 - Implements virtual memories for each process
 - Layer 1: **Kernel**
 - Implements a virtual processor for each process
 - Layer 0: **Hardware**
- Each layer can be tested and verified independently



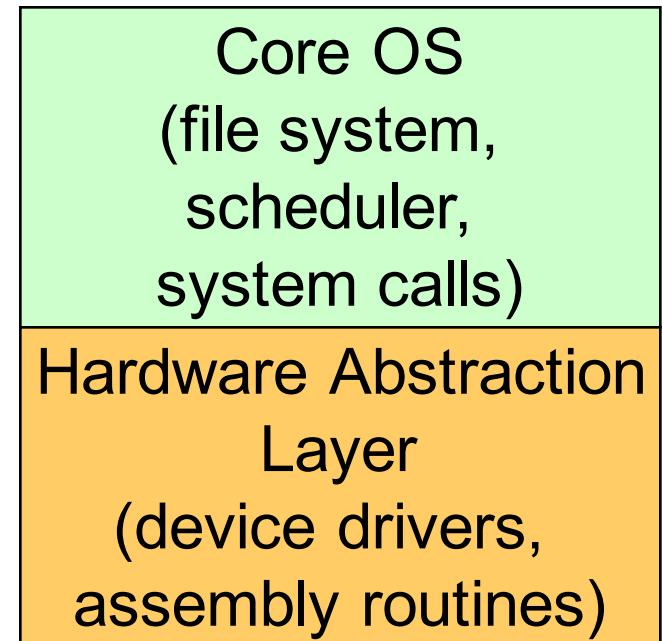
Problems with layering

- Imposes hierarchical structure
 - but real systems are more complex:
 - file system requires VM services (buffers)
 - VM would like to use files for its backing store
 - strict layering isn't flexible enough
 - Poor performance
 - each layer crossing has **overhead** associated with it
 - Disjunction between model and reality
 - systems modeled as layers, but not really built that way



Hardware Abstraction Layer

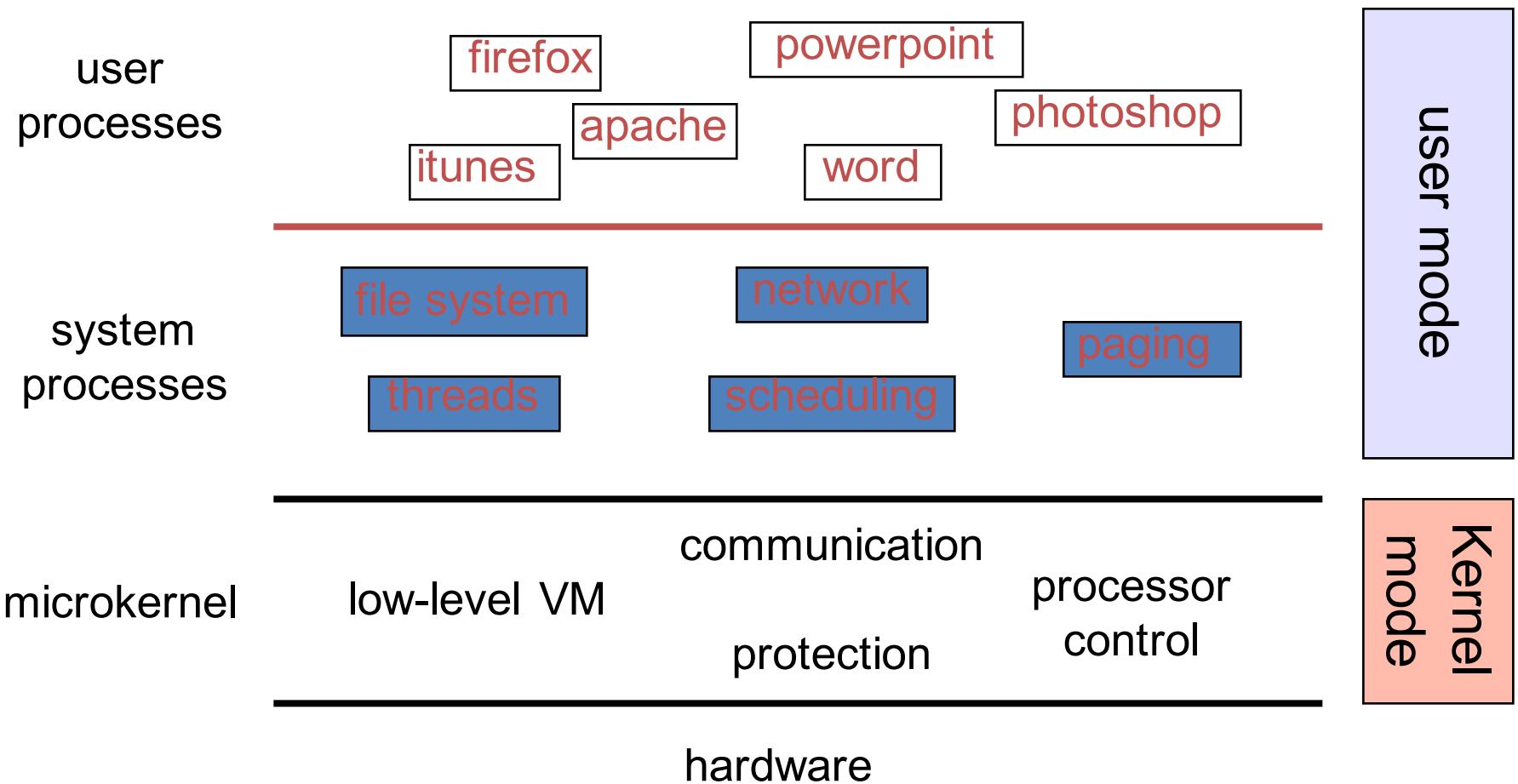
- An example of layering in modern operating systems
- Goal: separates hardware-specific routines from the “core” OS
 - Provides portability
 - Improves readability



Microkernels

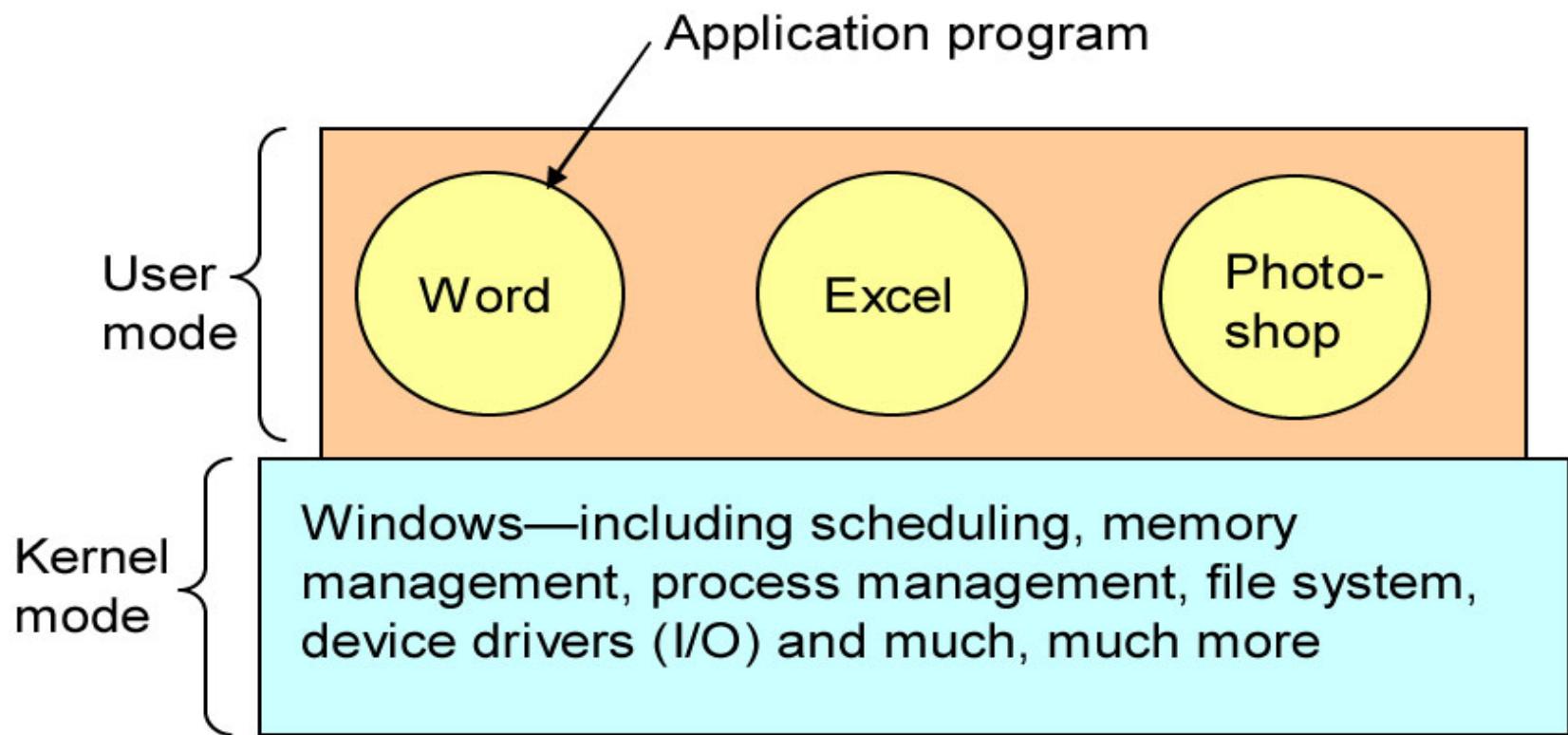
- Popular in the late 80's, early 90's
 - recent resurgence of popularity
- Goal:
 - minimize what goes in kernel
 - organize rest of OS as user-level processes
- This results in:
 - better reliability (isolation between components)
 - ease of extension and customization
 - poor performance (user/kernel boundary crossings)
- First microkernel system was Hydra (CMU, 1970)
 - Follow-ons: Mach (CMU), Chorus (French UNIX-like OS), OS X (Apple), in some ways NT (Microsoft)

Microkernel structure illustrated

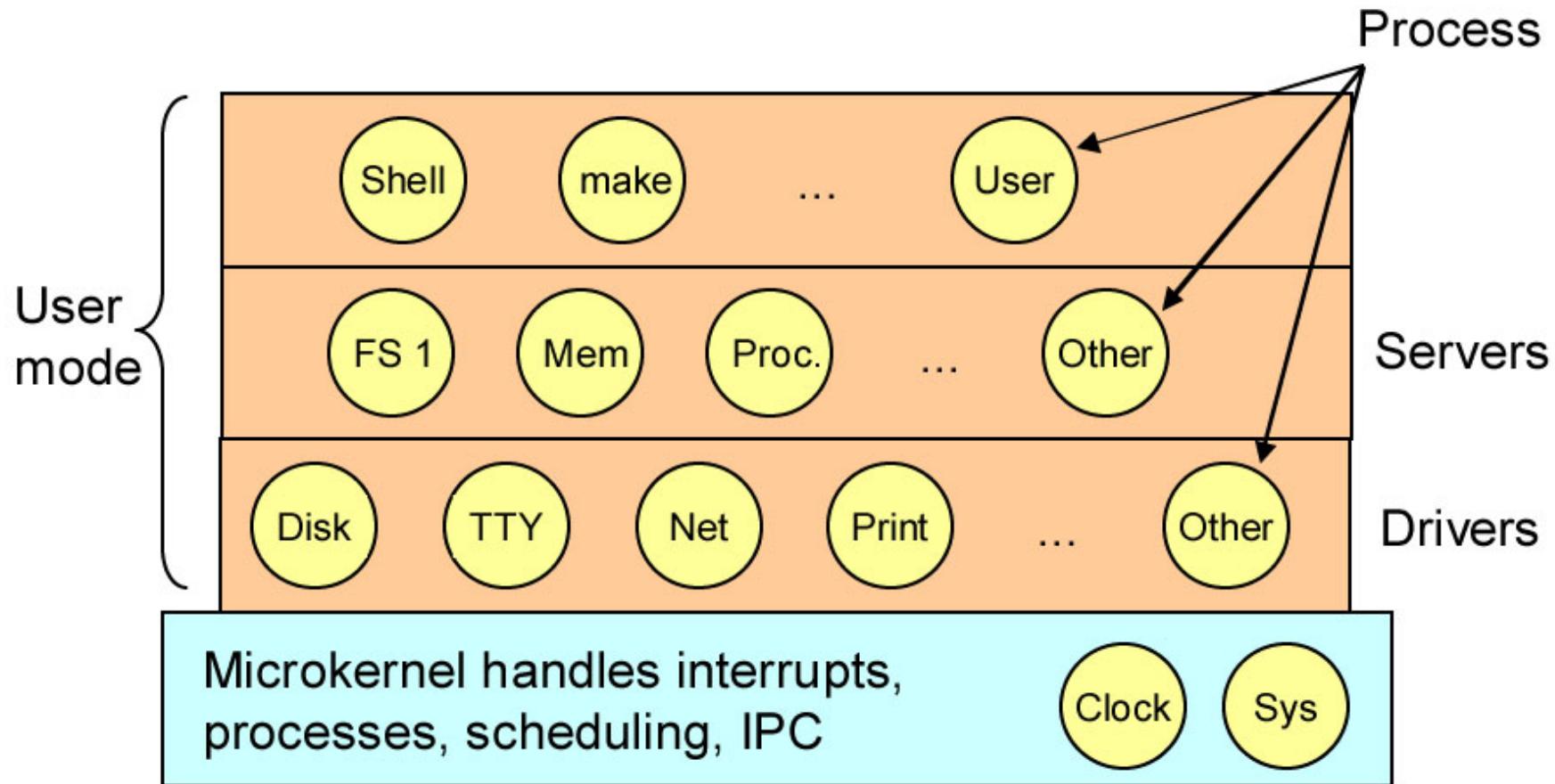


Monolithic

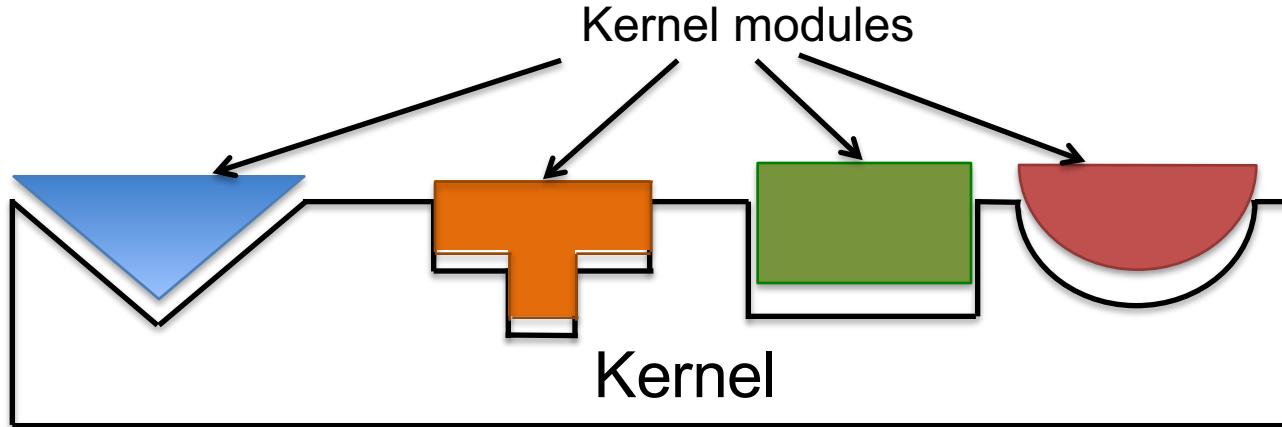
EXAMPLE: WINDOWS



ARCHITECTURE OF MINIX 3



Loadable Kernel Modules



- (Perhaps) the best practice for OS design
- Core services in the kernel and others dynamically loaded
- Common in modern implementations
 - Solaris, Linux, etc.
- Advantages
 - convenient: no need for rebooting for newly added modules
 - efficient: no need for message passing unlike microkernel
 - flexible: any module can call any other module unlike layered model

Summary

- Fundamental distinction between user and privileged mode supported by most hardware
- OS design has been an evolutionary process of trial and error. Probably more error than success
- Successful OS designs have run the spectrum from monolithic, to layered, to micro kernels
- The role and design of an OS are still evolving
- It is impossible to pick one “correct” way to structure an OS

Operating Systems

Processes

Lecture 3

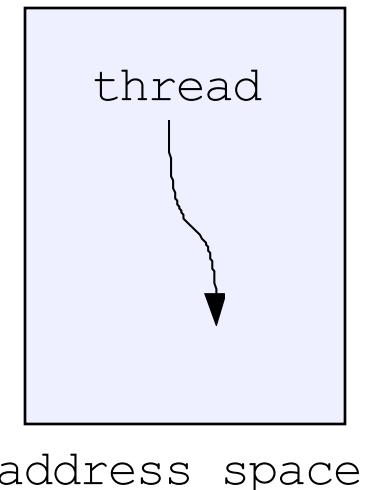
Michael O'Boyle

Overview

- Process
- Process control block
- Process state
- Context switch
- Process creation and termination

What is a “process”?

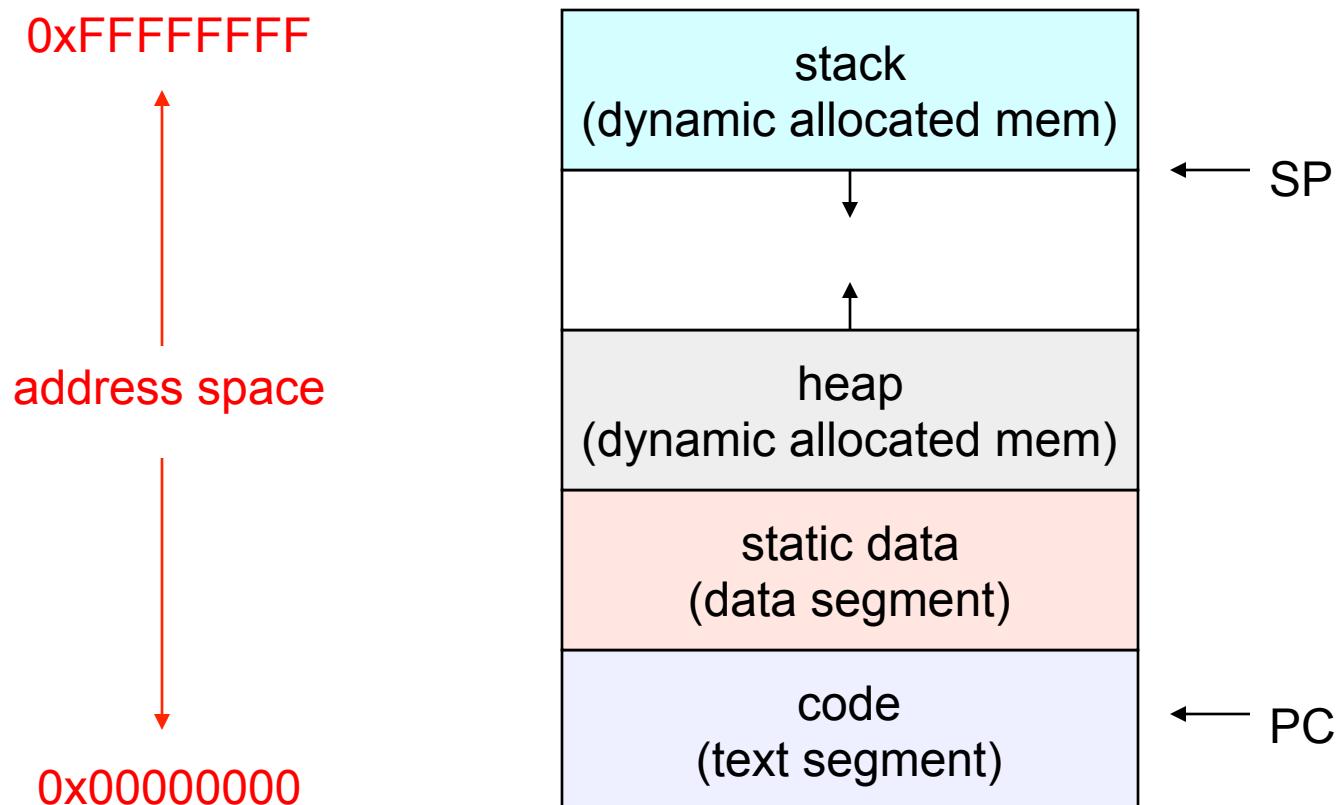
- The process is the OS’s abstraction for execution
 - A process is a program in execution
- Simplest (classic) case: a **sequential process**
 - An address space (an abstraction of memory)
 - A single thread of execution (an abstraction of the CPU)
- A sequential process is:
 - The unit of execution
 - The unit of scheduling
 - The dynamic (active) execution context
 - vs. the program – static, just a bunch of bytes



What's “in” a process?

- A process consists of (at least):
 - An **address space**, containing
 - the code (instructions) for the running program
 - the data for the running program (static data, heap data, stack)
 - **CPU state**, consisting of
 - The program counter (PC), indicating the next instruction
 - The stack pointer
 - Other general purpose register values
 - A set of **OS resources**
 - open files, network connections, sound channels, ...
- In other words, everything needed to run the program
 - or to re-start, if interrupted

A process's address space (idealized)



The OS process namespace

- The particulars depend on the specific OS, but the principles are general
- The name for a process is called a **process ID (PID)**
 - An integer
- The PID namespace is global to the system
 - Only one process at a time has a particular PID
- Operations that create processes return a PID
 - E.g., fork()
- Operations on processes take PIDs as an argument
 - E.g., kill(), wait(), nice()

Representation of processes by the OS

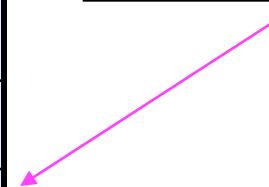
- The OS maintains a data structure to keep track of a process's state
 - Called the **process control block** (PCB) or **process descriptor**
 - Identified by the PID
- OS keeps all of a process's execution state in (or linked from) the PCB when the process isn't running
 - PC, SP, registers, etc.
 - when a process is unscheduled, the state is transferred out of the hardware into the PCB
 - (when a process is running, its state is spread between the PCB and the CPU)

The PCB

- The PCB is a data structure with many, many fields:
 - process ID (PID)
 - parent process ID
 - execution state
 - program counter, stack pointer, registers
 - address space info
 - UNIX user id, group id
 - scheduling priority
 - accounting info
 - pointers for state queues
- In Linux:
 - defined in `task_struct` ([include/linux/sched.h](#))
 - over 95 fields!!!

Process ID
Pointer to parent
List of children
Process state
Pointer to address space descriptor
Program counter stack pointer (all) register values
uid (user id) gid (group id) euid (effective user id)
Open file list
Scheduling priority
Accounting info
Pointers for state queues
Exit ("return") code value

This is (a simplification of) what each of those PCBs looks like inside



PCBs and CPU state

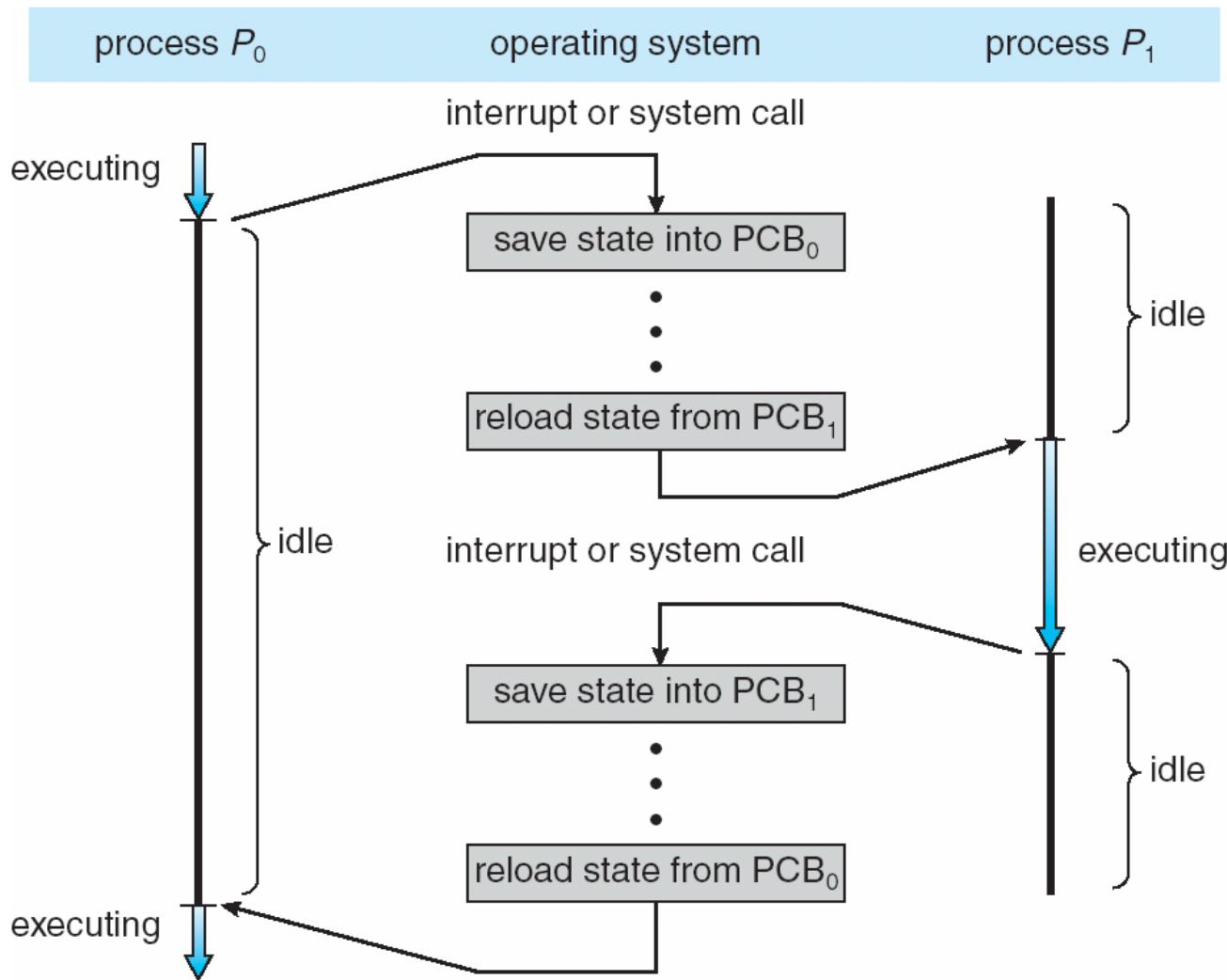
- When a process is running, its CPU state is inside the CPU
 - PC, SP, registers
 - CPU contains current values
- When the OS gets control because of a ...
 - Trap: Program executes a syscall
 - Exception: Program does something unexpected (e.g., page fault)
 - Interrupt: A hardware device requests service

the OS saves the CPU state of the running process in that process's PCB

PCBs and CPU state

- When the OS returns the process to the running state
 - it loads the hardware registers with values from that process's PCB
 - eg general purpose registers, stack pointer, instruction pointer
- The act of switching the CPU from one process to another is called a **context switch**
 - systems may do 100s or 1000s of switches/sec.
 - takes a few microseconds on today's hardware
 - Still expensive relative to thread based context switches
- Choosing which process to run next is called **scheduling**

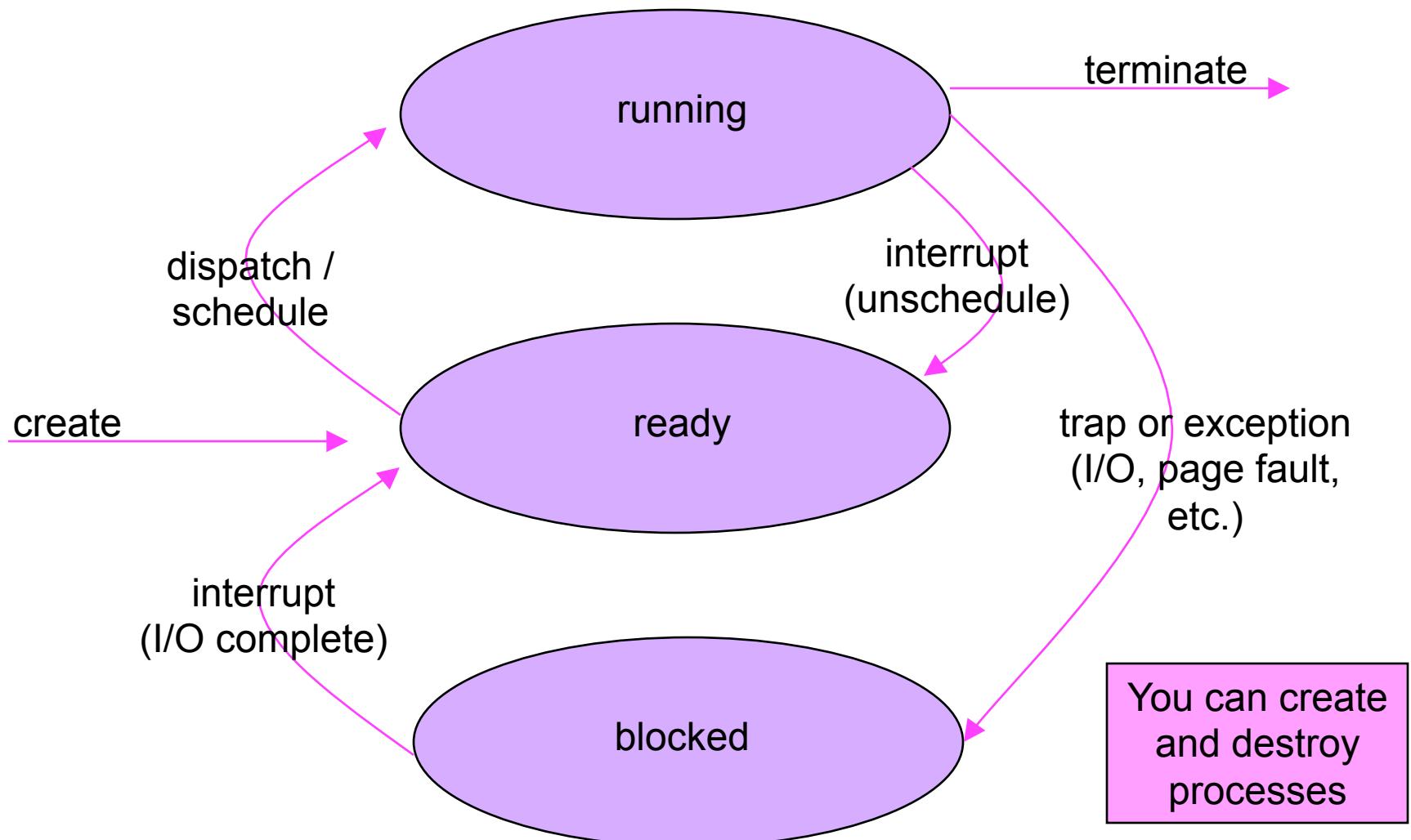
Process context switch



Process execution states

- Each process has an **execution state**, which indicates what it's currently doing
 - **ready**: waiting to be assigned to a CPU
 - could run, but another process has the CPU
 - **running**: executing on a CPU
 - it's the process that currently controls the CPU
 - **waiting** (aka “blocked”): waiting for an event, e.g., I/O completion, or a message from (or the completion of) another process
 - cannot make progress until the event happens
- As a process executes, it moves from state to state
 - UNIX: run **top**, STAT column shows current state
 - which state is a process in most of the time?

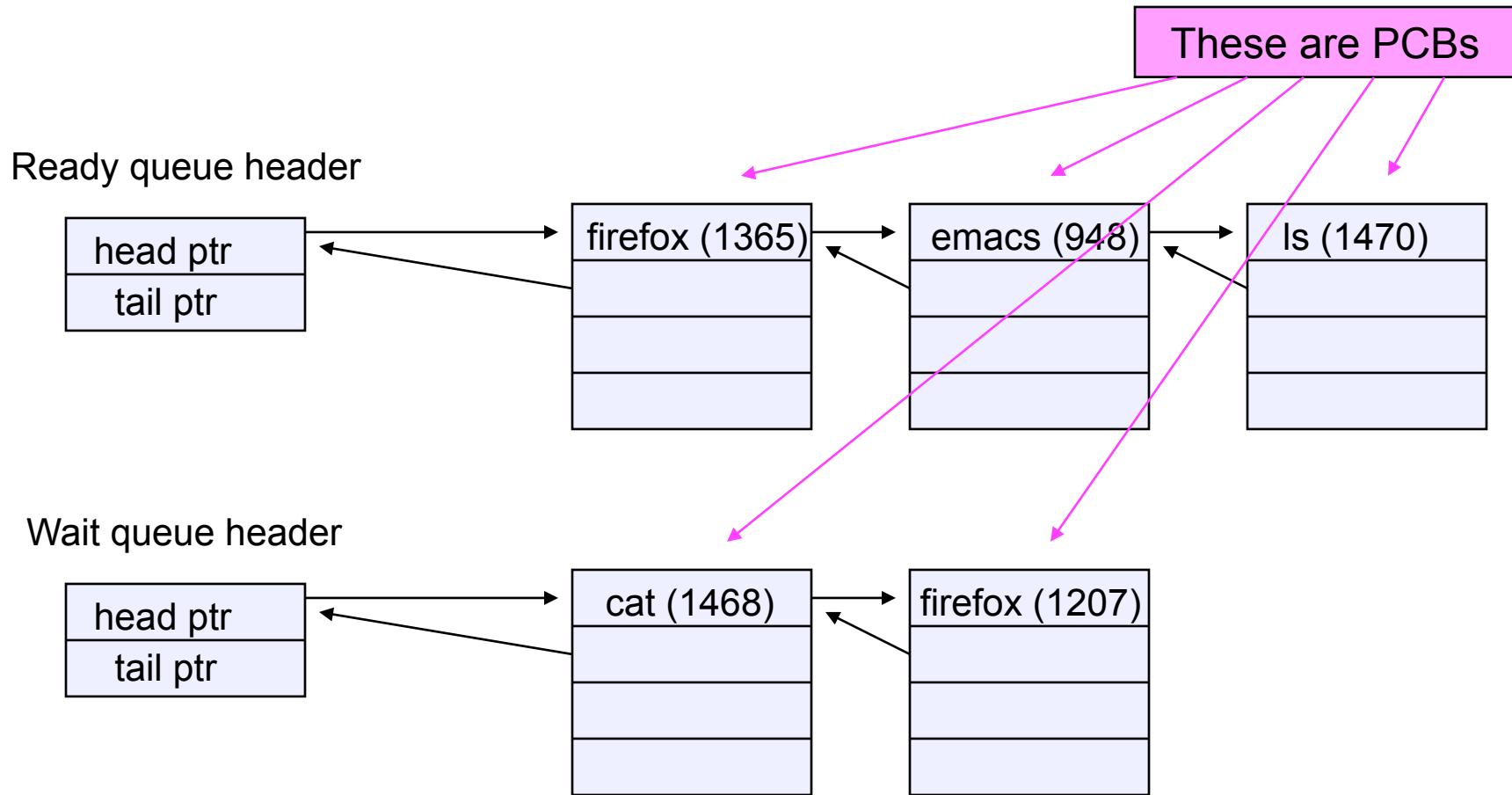
Process states and state transitions



State queues

- The OS maintains a collection of queues that represent the state of all processes in the system
 - typically one queue for each state
 - e.g., ready, waiting, ...
 - each PCB is queued onto a state queue according to the current state of the process it represents
 - as a process changes state, its PCB is unlinked from one queue, and linked onto another
- The PCBs are moved between queues, which are represented as linked lists

State queues



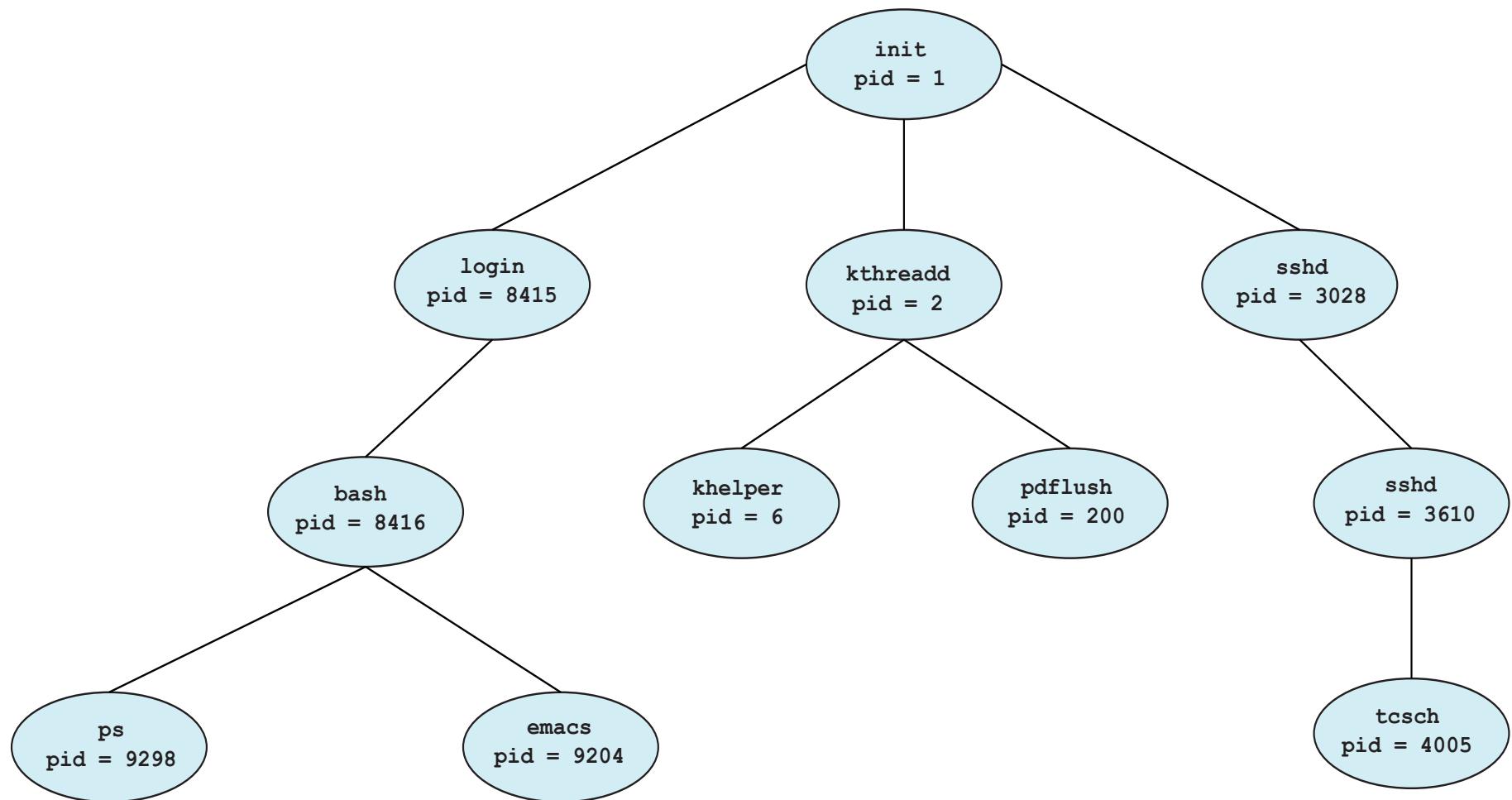
- There may be many wait queues, one for each type of wait (particular device, timer, message, ...)

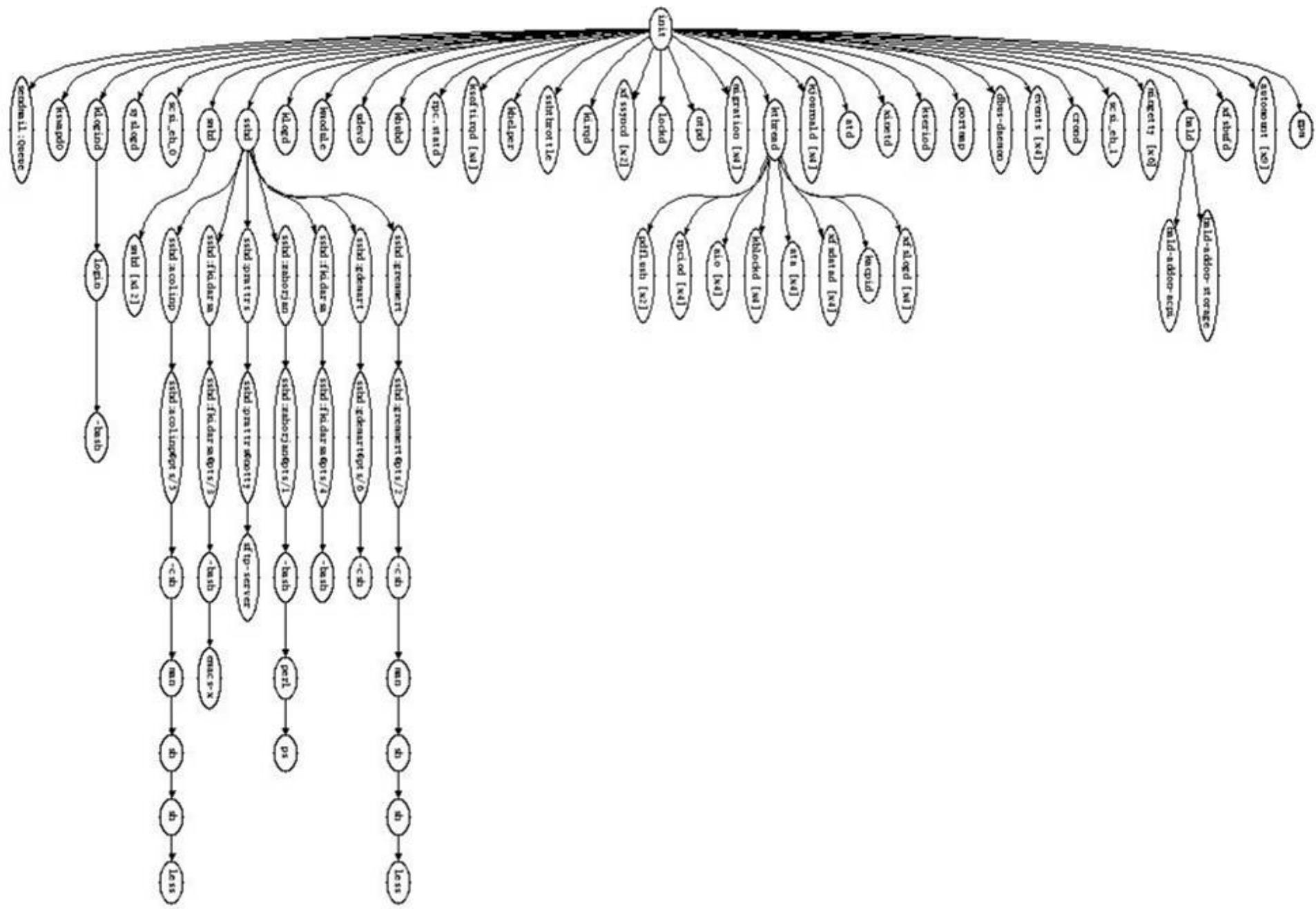
PCBs and state queues

- PCBs are data structures
 - dynamically allocated inside OS memory
- When a process is created:
 - OS allocates a PCB for it
 - OS initializes PCB
 - (OS does other things not related to the PCB)
 - OS puts PCB on the correct queue
- As a process computes:
 - OS moves its PCB from queue to queue
- When a process is terminated:
 - PCB may be retained for a while (to receive signals, etc.)
 - eventually, OS deallocates the PCB

Process creation

- New processes are created by existing processes
 - creator is called the **parent**
 - created process is called the **child**
 - UNIX: do `ps -ef`, look for PPID field
 - what creates the first process, and when?



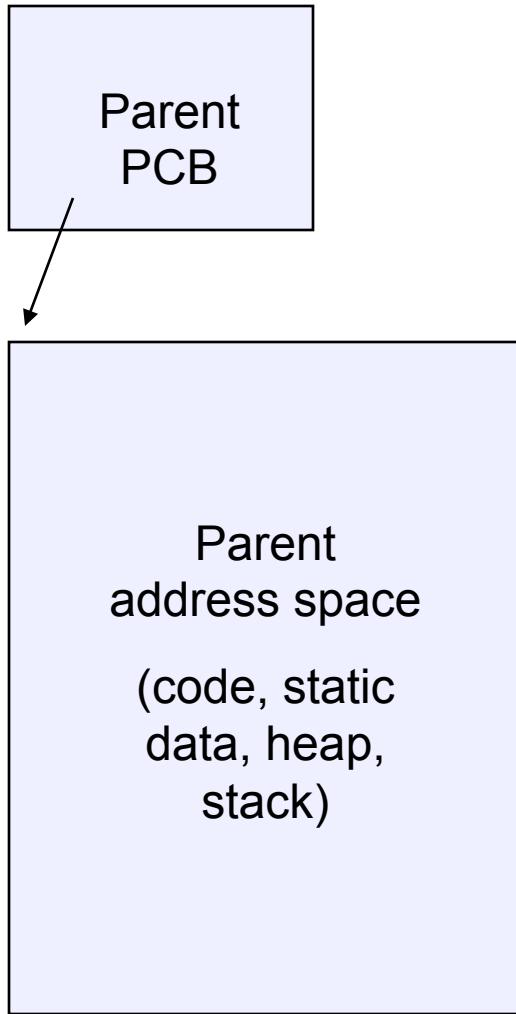


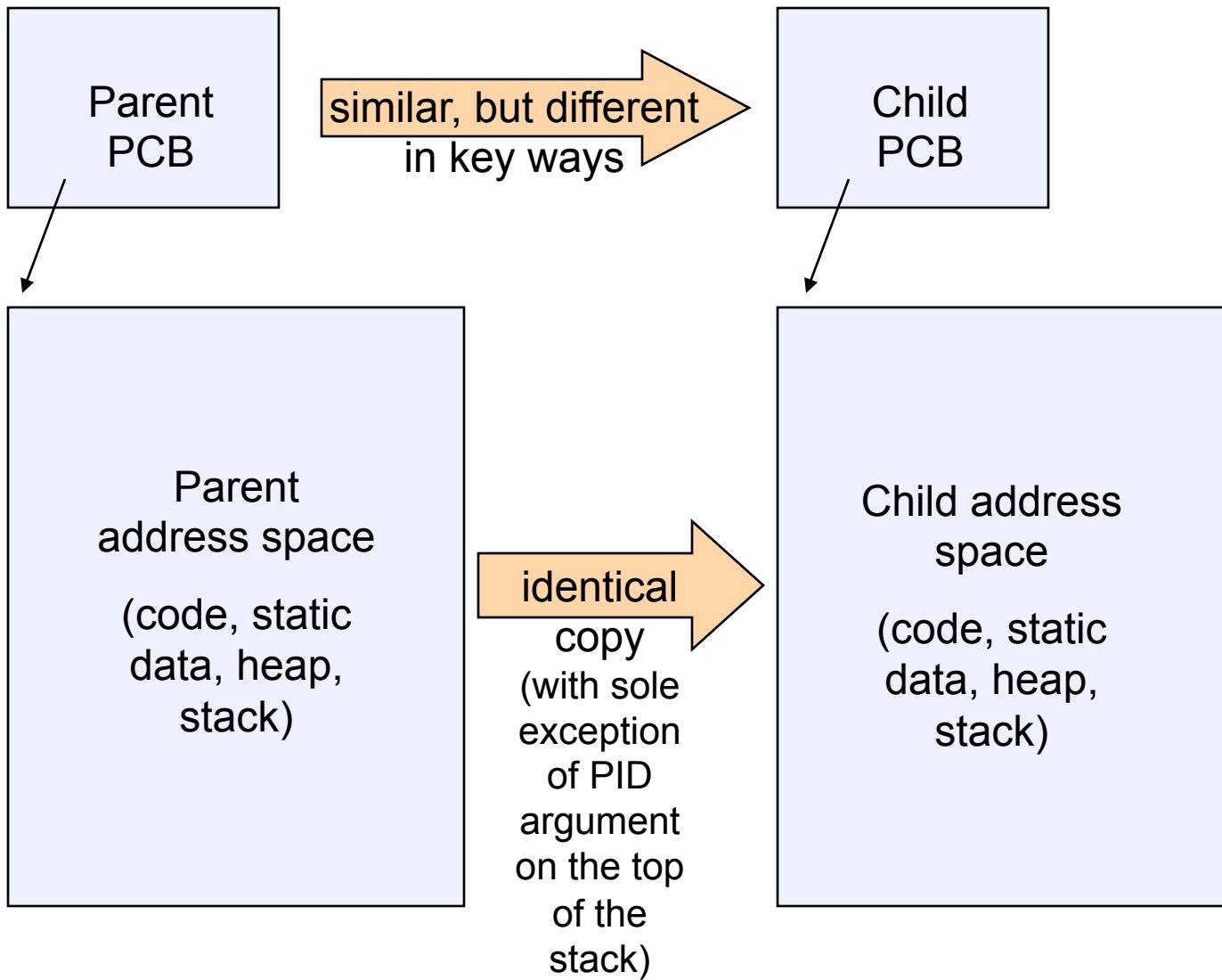
Process creation semantics

- (Depending on the OS) child processes inherit certain attributes of the parent
 - Examples:
 - Open file table: implies stdin/stdout/stderr
 - On some systems, resource allocation to parent may be divided among children
- (In Unix) when a child is created, the parent may either wait for the child to finish, or continue in parallel

UNIX process creation details

- UNIX process creation through **fork()** system call
 - creates and initializes a new PCB
 - initializes kernel resources of new process with resources of parent (e.g., open files)
 - initializes PC, SP to be same as parent
 - creates a new address space
 - initializes new address space with a copy of the entire contents of the address space of the parent
 - places new PCB on the ready queue
- the **fork()** system call “returns twice”
 - once into the parent, and once into the child
 - returns the child’s PID to the parent
 - returns 0 to the child
- **fork()** = “clone me”





testparent – use of fork()

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    char *name = argv[0];
    int pid = fork();
    if (pid == 0) {
        printf("Child of %s is %d\n", name, pid);
        return 0;
    } else {
        printf("My child is %d\n", pid);
        return 0;
    }
}
```

testparent output

```
spinlock% gcc -o testparent testparent.c
```

```
spinlock% ./testparent
```

```
My child is 486
```

```
Child of testparent is 0
```

```
spinlock% ./testparent
```

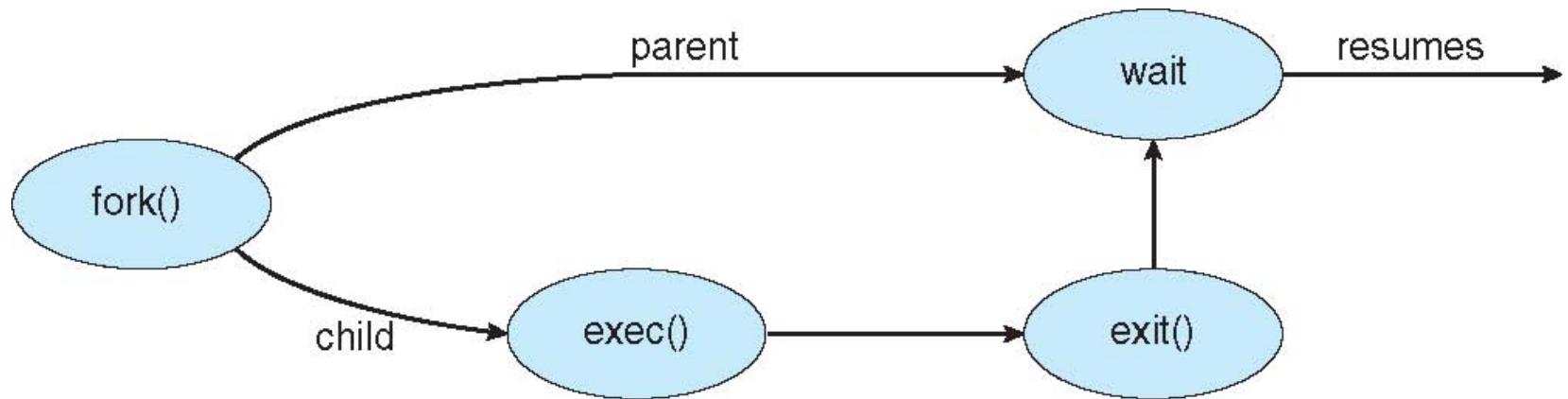
```
Child of testparent is 0
```

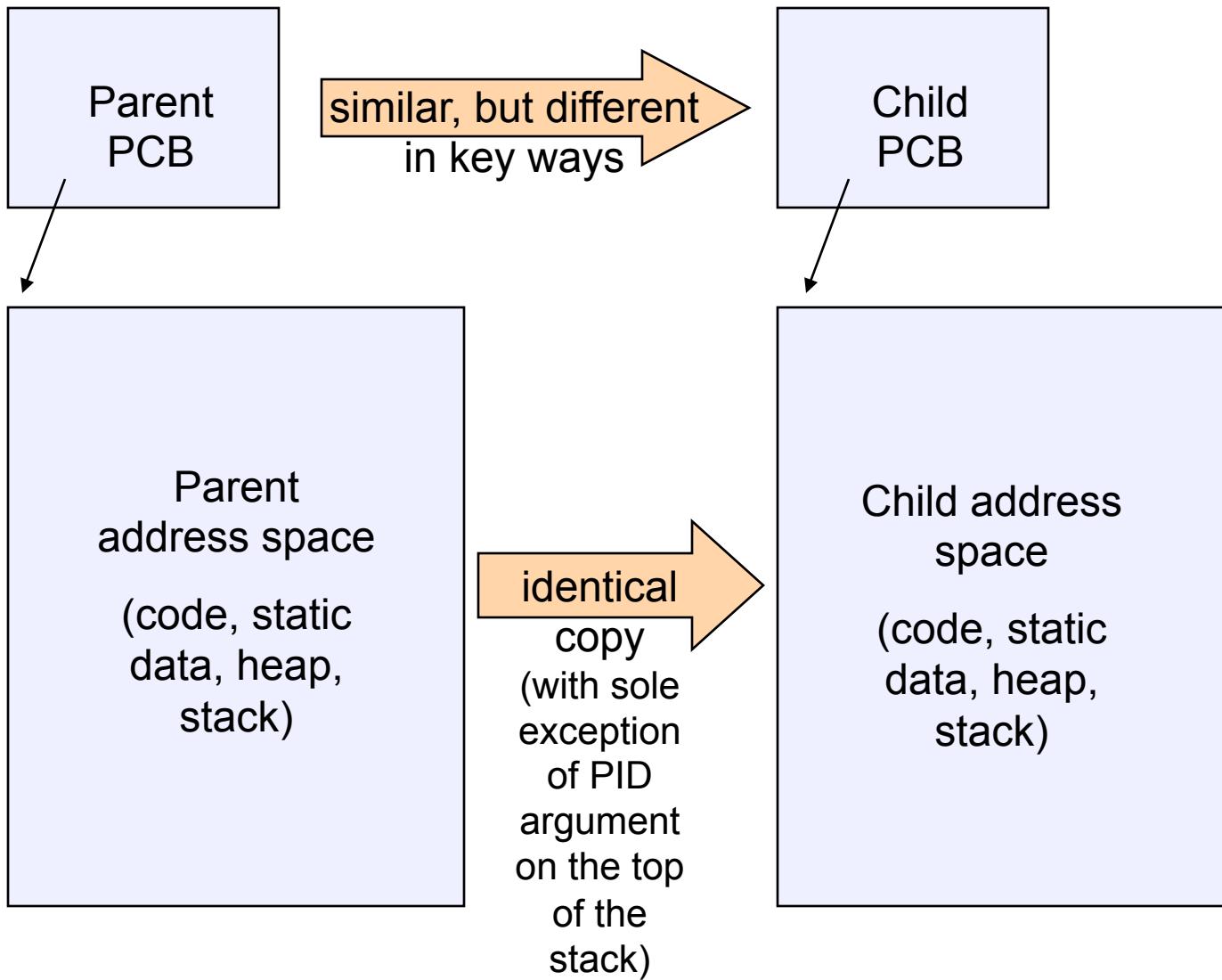
```
My child is 571
```

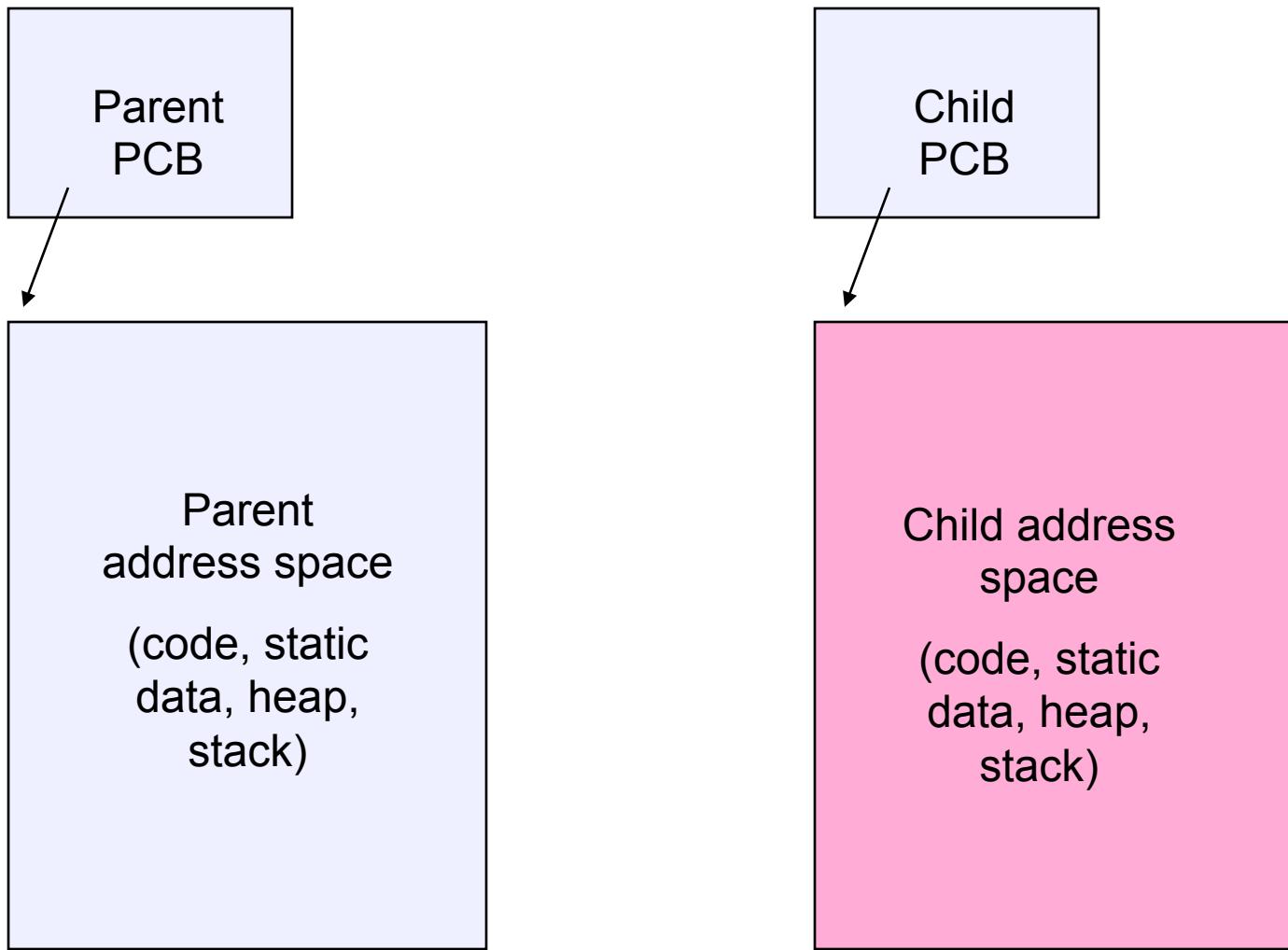
exec() vs. fork()

- Q: So how do we start a new program, instead of just forking the old program?
- A: First fork, then **exec**
 - `int exec(char * prog, char * argv[])`
- **exec()**
 - stops the current process
 - loads program ‘prog’ into the address space
 - i.e., over-writes the existing process image
 - initializes hardware context, args for new program
 - places PCB onto ready queue
 - note: does not create a new process!

exec() and fork()





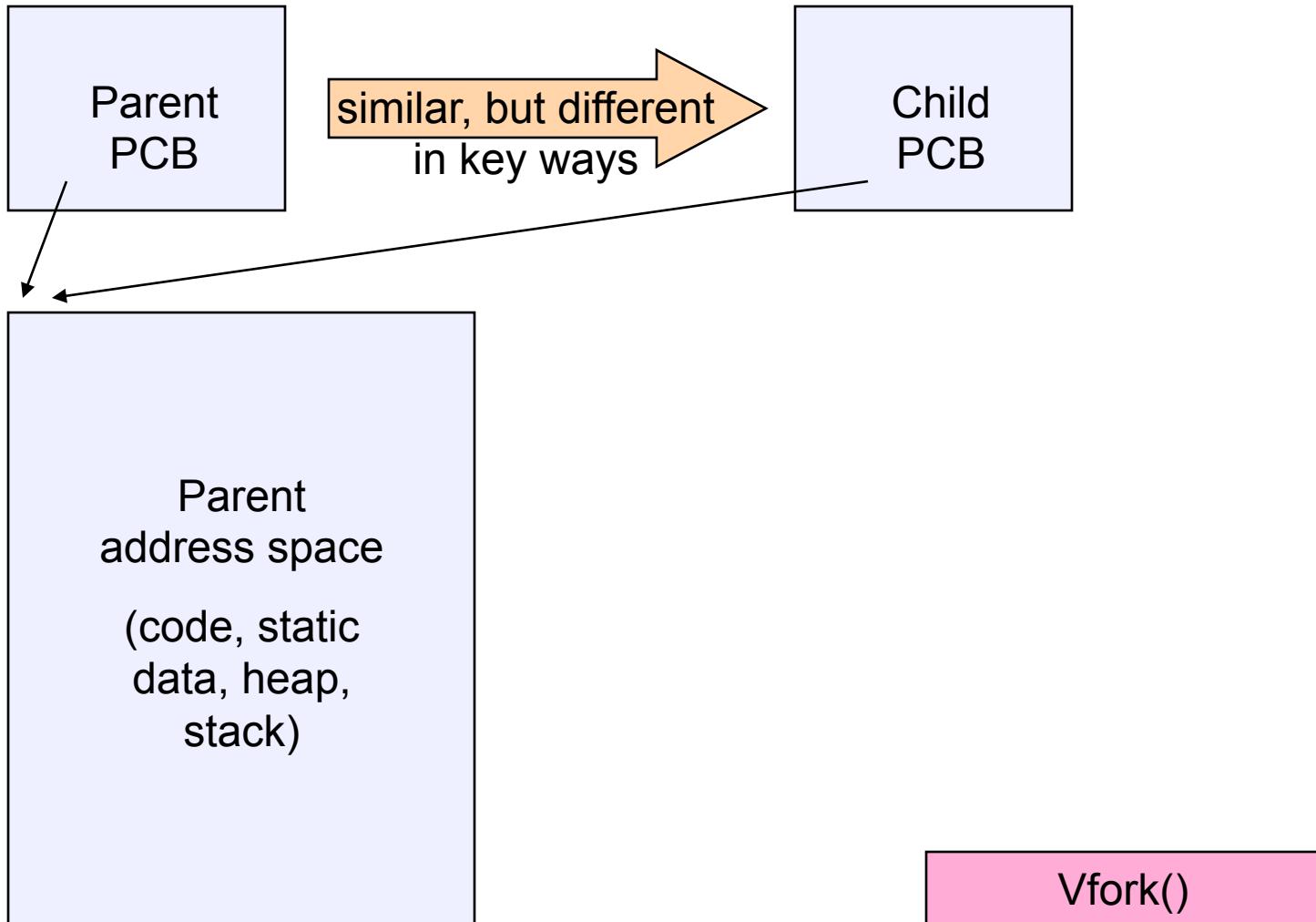


Making process creation faster

- The semantics of fork() say the child's address space is a copy of the parent's
- Implementing fork() that way is slow
 - Have to allocate physical memory for the new address space
 - Have to set up child's page tables to map new address space
 - Have to copy parent's address space contents into child's address space
 - Which you are likely to destroy with an exec()

Method 1: vfork()

- vfork() is the older (now uncommon) of the two approaches we'll discuss
- Instead of “child’s address space is a copy of the parent’s,” the semantics are “child’s address space *is* the parent’s”
 - With a “promise” that the child won’t modify the address space before doing an execve()
 - Unenforced! You use vfork() at your own peril
 - When execve() is called, a new address space is created and it’s loaded with the new executable
 - Parent is blocked until execve() is executed by child
 - Saves wasted effort of duplicating parent’s address space



Method 2: copy-on-write

- Retains the original semantics, but copies “only what is necessary” rather than the entire address space
- On fork():
 - Create a new address space
 - Initialize page tables with same mappings as the parent’s (i.e., they both point to the same physical memory)
 - No copying of address space contents have occurred at this point – with the sole exception of the top page of the stack
 - Set both parent and child page tables to make all pages read-only
 - If either parent or child writes to memory, an exception occurs
 - When exception occurs, OS copies the page, adjusts page tables, etc.

UNIX shells

```
int main(int argc, char **argv)
{
    while (1) {
        printf ("$ ");
        char *cmd = get_next_command();
        int pid = fork();
        if (pid == 0) {
            exec(cmd);
            panic("exec failed!");
        } else {
            wait(pid);
        }
    }
}
```

Summary

- Process
- Process control block
- Process state
- Context switch
- Process creation and termination
- Next time
 - threads

Operating Systems

Threads

Lecture 4
Michael O'Boyle

Overview

- Process vs threads
 - how related
- Concurrency
 - why threads
- Design space of process/threads
 - a simple taxonomy
- Kernel threads
 - more efficient
- User-level threads
 - even faster

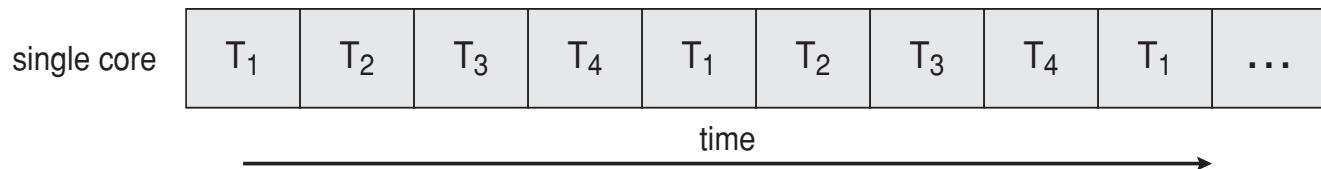
What's “in” a process?

- A process consists of (at least):
 - An **address space**, containing
 - the code (instructions) for the running program
 - the data for the running program
 - **Thread state**, consisting of
 - The program counter (PC), indicating the next instruction
 - The stack pointer register (implying the stack it points to)
 - Other general purpose register values
 - A set of **OS resources**
 - open files, network connections, sound channels, ...
- Decompose ...
 - address space
 - **thread of control** (stack, stack pointer, program counter, registers)
 - OS resources

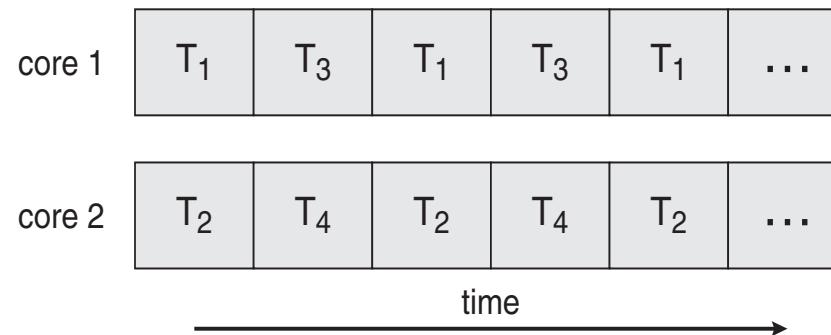
Thread: Concurrency vs. Parallelism

- Threads are about **concurrency** and **parallelism**

- Concurrent execution on single-core system:**



- Parallelism on a multi-core system:**



Motivation

- Threads are about **concurrency** and **parallelism**
- One way to get concurrency and parallelism is to use multiple processes
 - The programs (code) of distinct processes are isolated from each other
- Threads are another way to get concurrency and parallelism
 - Threads “share a process” – same address space, same OS resources
 - Threads have private stack, CPU state – are schedulable

What's needed?

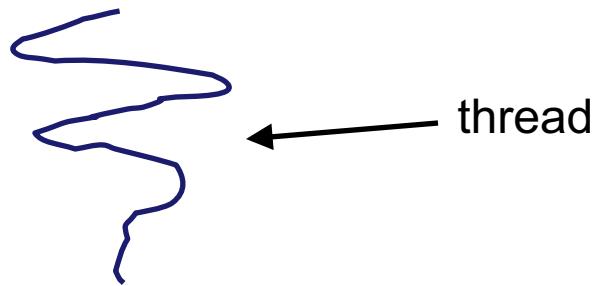
- In many cases
 - Everybody wants to run the same code
 - Everybody wants to access the same data
 - Everybody has the same privileges
 - Everybody uses the same resources (open files, network connections, etc.)
- But you'd like to have multiple hardware execution states:
 - an execution stack and stack pointer (SP)
 - traces state of procedure calls made
 - the program counter (PC), indicating the next instruction
 - a set of general-purpose processor registers and their values

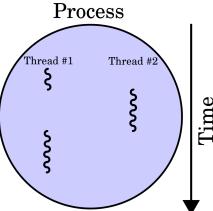
How could we achieve this?

- Given the process abstraction as we know it:
 - fork several processes
 - cause each to *map* to the **same** physical memory to share data
 - see the `shmget()` system call for one way to do this
- This is really inefficient
 - space: PCB, page tables, etc.
 - time: creating OS structures, fork/copy address space, etc.

Can we do better?

- Key idea:
 - separate the concept of a **process** (address space, OS resources)
 - ... from that of a minimal “**thread of control**” (execution state: stack, stack pointer, program counter, registers)
- This execution state is usually called a **thread**, or sometimes, a **lightweight process**

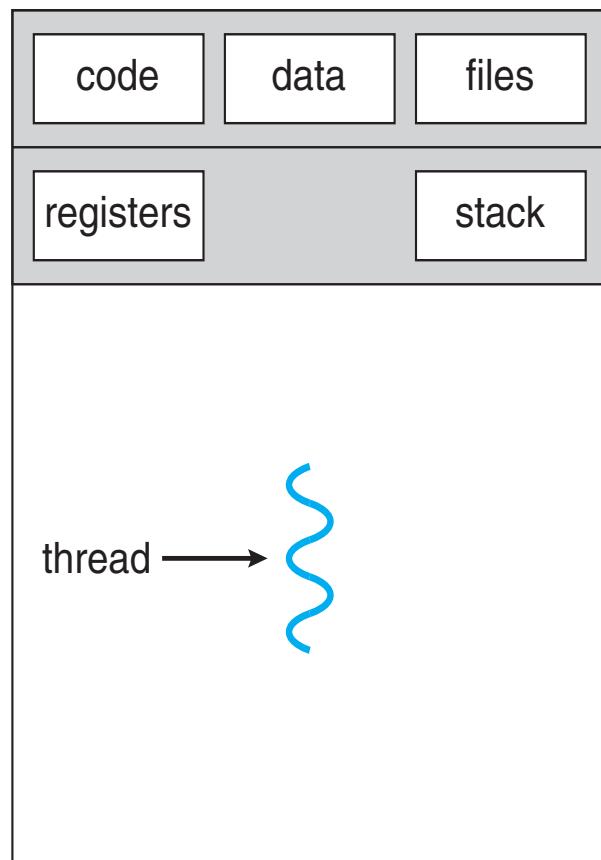




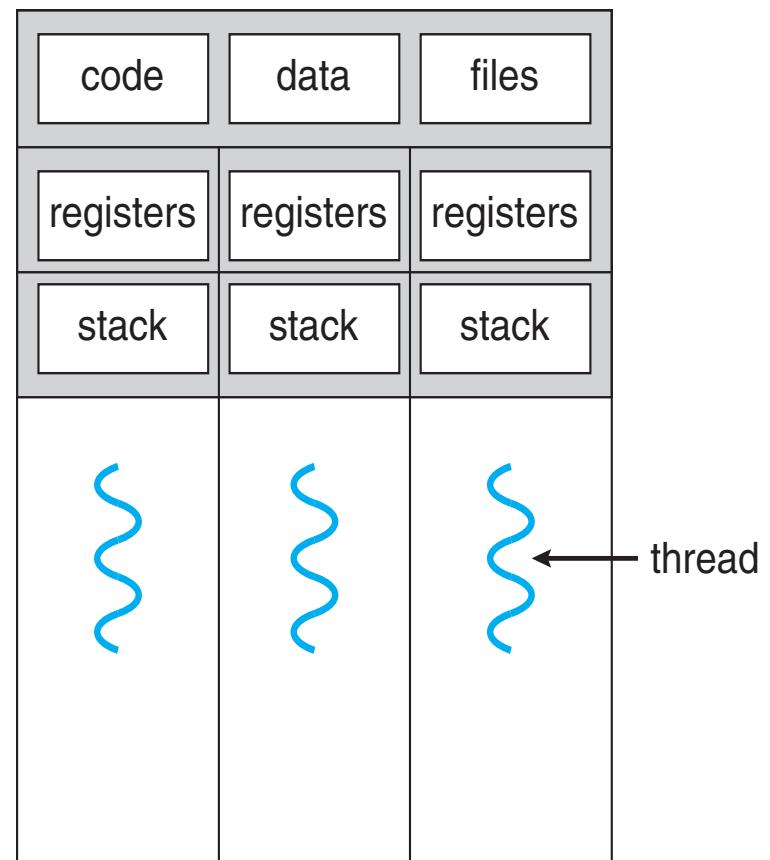
Threads and processes

- Most modern OS's (Mach (Mac OS), Chorus, Windows, UNIX) therefore support two entities:
 - the **process**, which defines the address space and general process attributes (such as open files, etc.)
 - the **thread**, which defines a sequential execution stream within a process
- A thread is bound to a single process / address space
 - address spaces, however, can have multiple threads executing within them
 - sharing data between threads is cheap: all see the same address space
 - creating threads is cheap too!
- Threads become the unit of scheduling
 - processes / address spaces are just **containers** in which threads execute

Single and Multithreaded Processes



single-threaded process



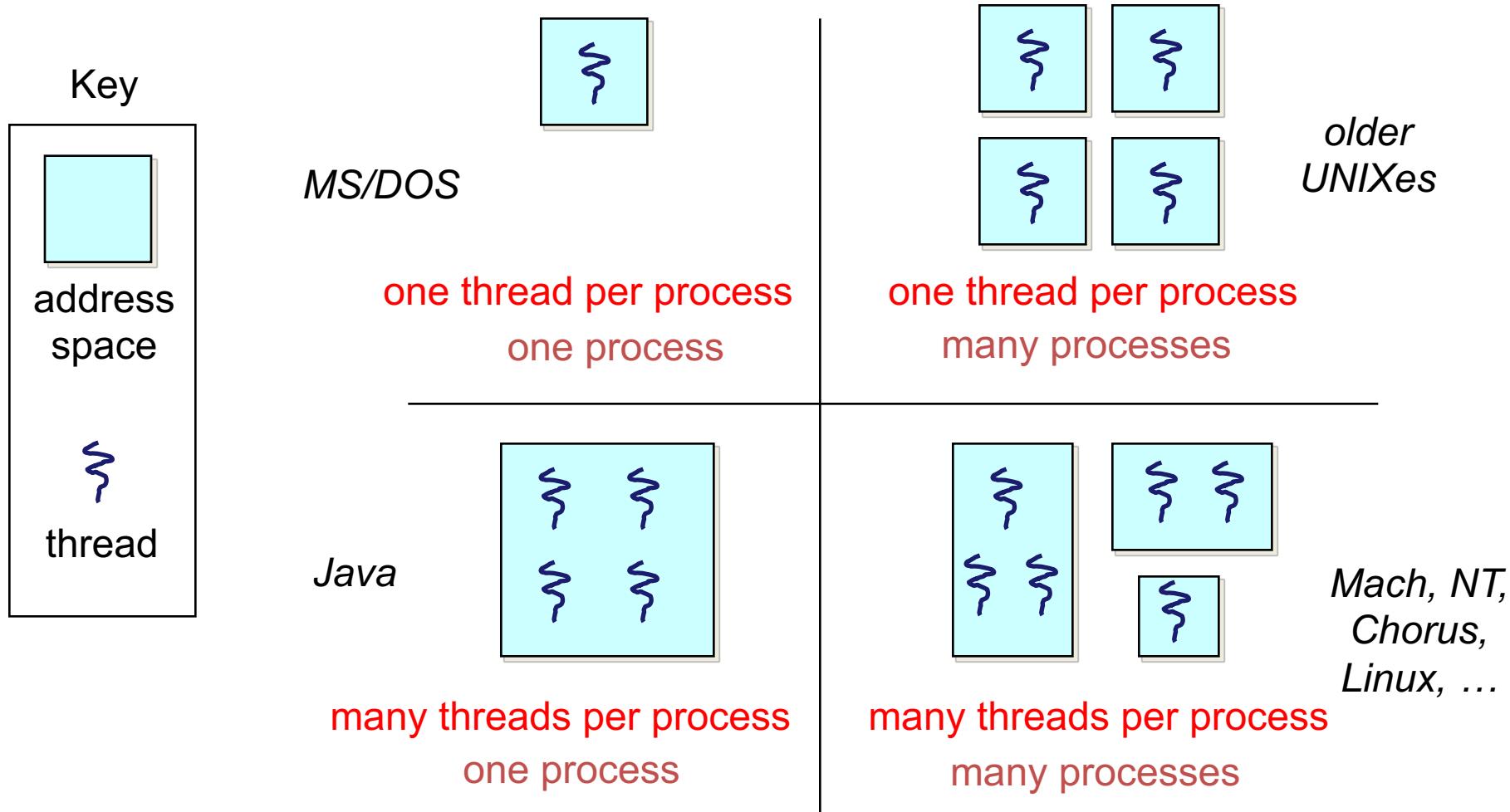
multithreaded process

Communication

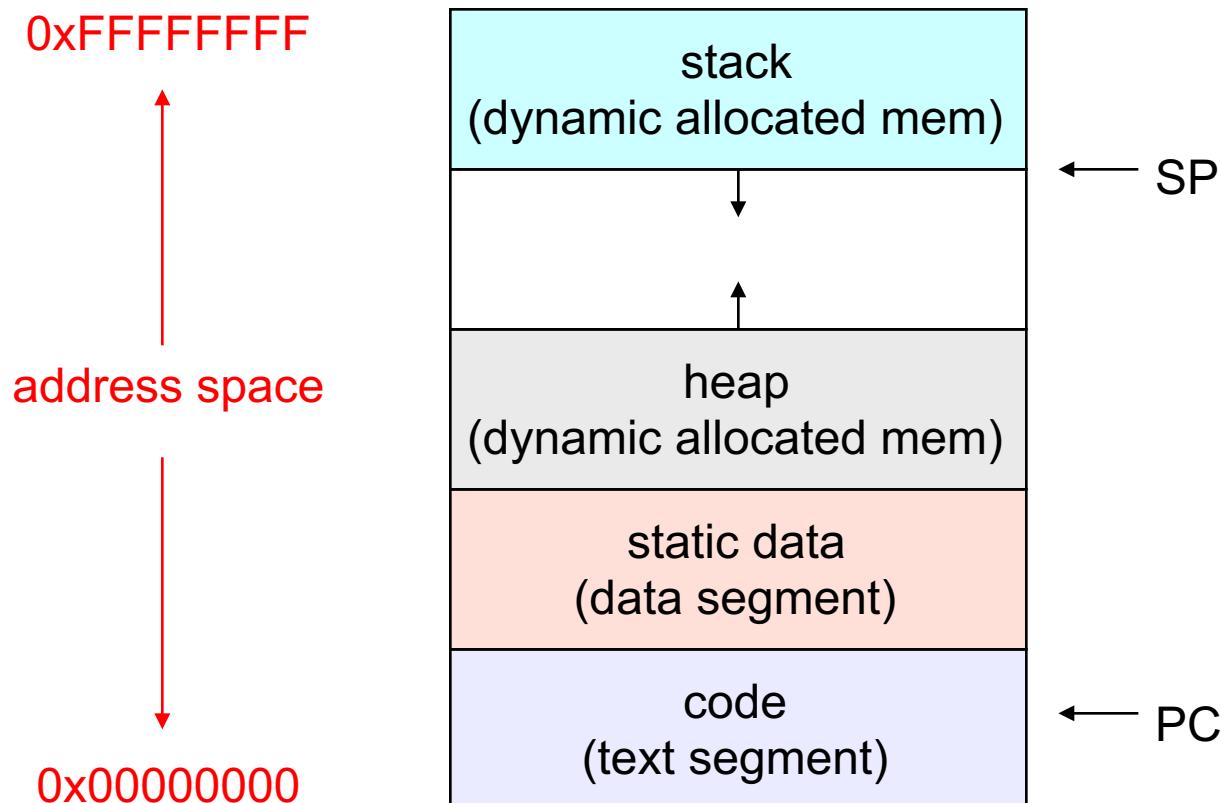
- Threads are concurrent executions sharing an address space (and some OS resources)
- Address spaces provide isolation
 - If you can't name it, you can't read or write it
- Hence, communicating between processes is expensive
 - Must go through the OS to move data from one address space to another
- Because threads are in the same address space, communication is simple/cheap
 - Just update a shared variable!



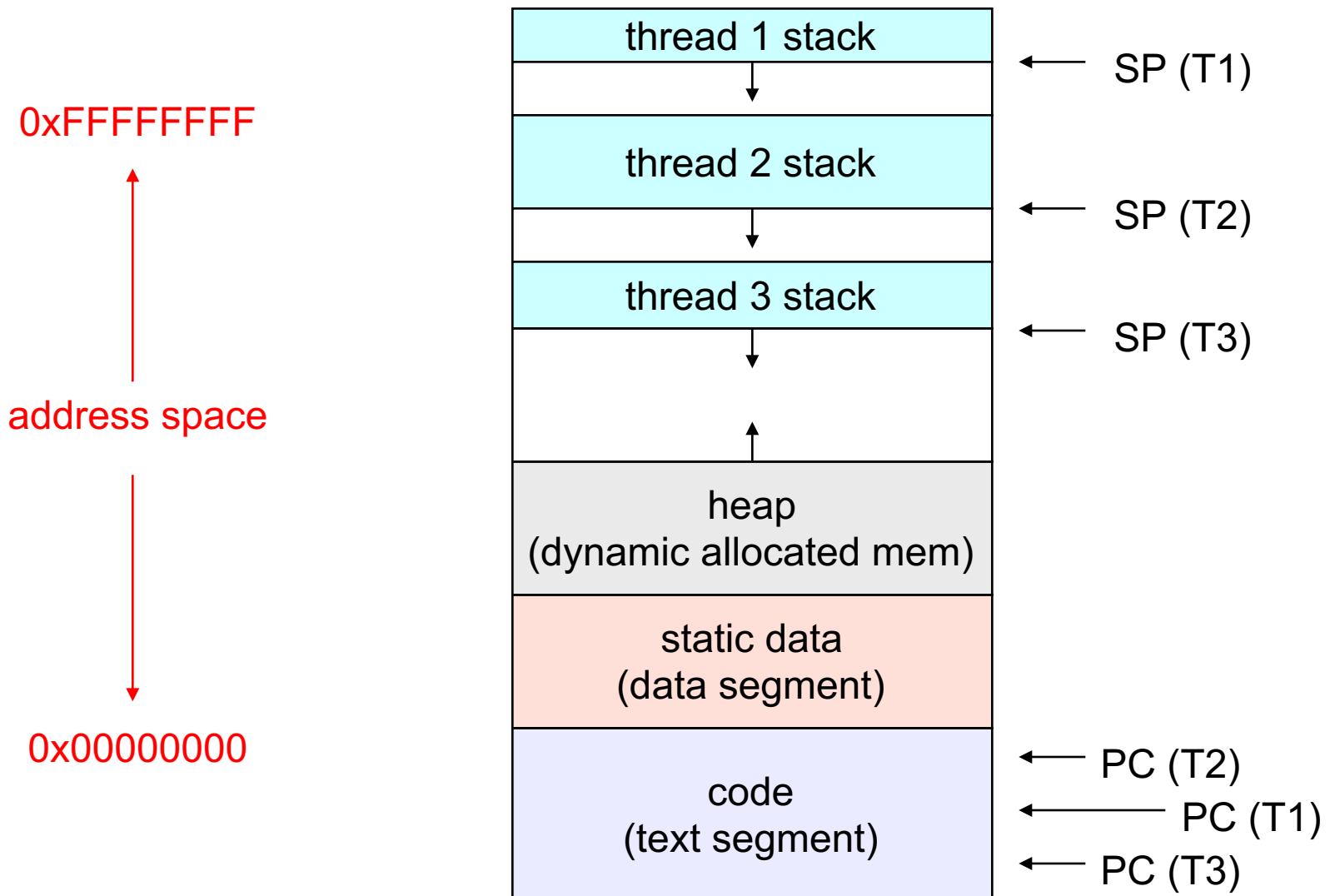
The design space



(old) Process address space



(new) Address space with threads



Process/thread separation

- Concurrency (multithreading) is useful for:
 - handling concurrent events (e.g., web servers and clients)
 - building parallel programs (e.g., matrix multiply, ray tracing)
 - improving program structure (the Java argument)
- Multithreading is useful even on a uniprocessor
 - even though only one thread can run at a time
- Supporting multithreading – that is, separating the concept of a **process** (address space, files, etc.) from that of a minimal **thread of control** (execution state), is a big win
 - creating concurrency does not require creating new processes
 - “faster / better / cheaper”

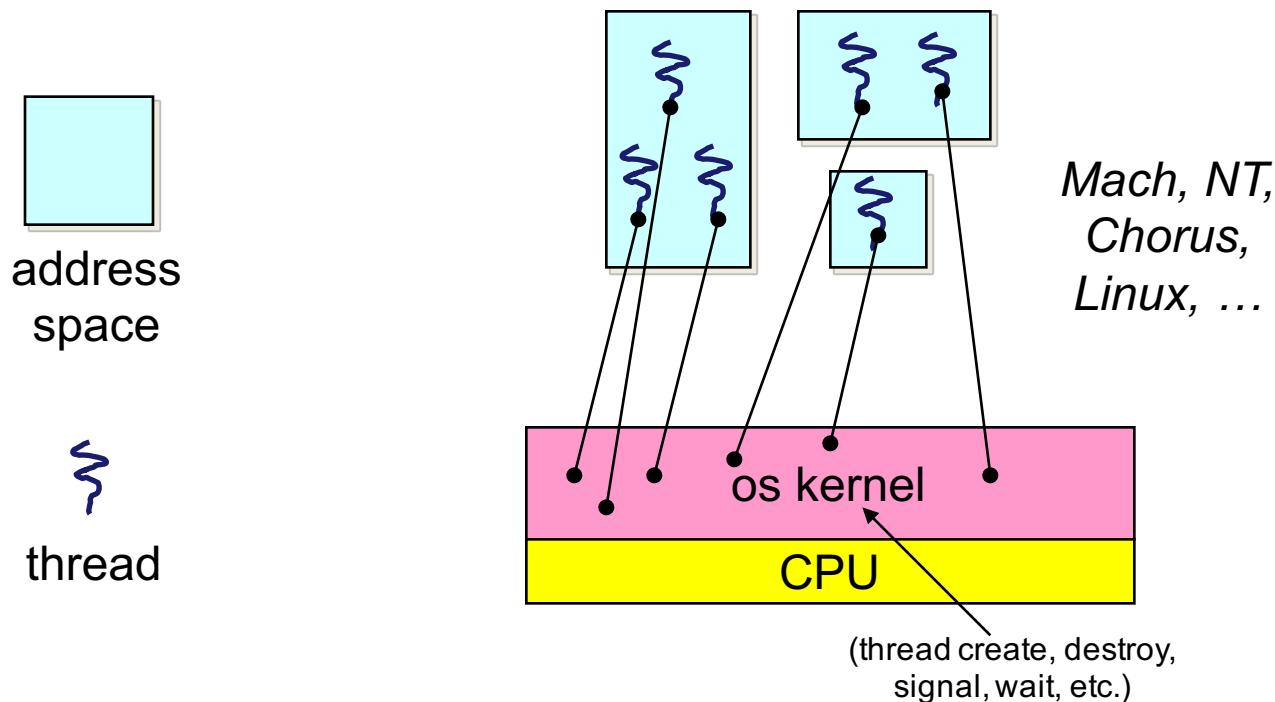
Terminology

- Just a note that there's the potential for some confusion ...
 - Old : “process” == “address space + OS resources + single thread”
 - New: “process” typically refers to an address space + system resources + all of its threads ...
 - When we mean the “address space” we need to be explicit “thread” refers to a single thread of control within a process / address space
- A bit like “kernel” and “operating system” ...
 - Old: “kernel” == “operating system” and runs in “kernel mode”
 - New: “kernel” typically refers to the microkernel; lots of the operating system runs in user mode

Where do threads come from?

- Natural answer: the OS is responsible for creating/managing threads
 - For example, the kernel call to create a new thread would
 - allocate an execution stack within the process address space
 - create and initialize a Thread Control Block
 - stack pointer, program counter, register values
 - stick it on the ready queue
- We call these **kernel threads**
 - There is a “thread name space”
 - Thread id's (TID's)
 - TID's are integers

Kernel threads



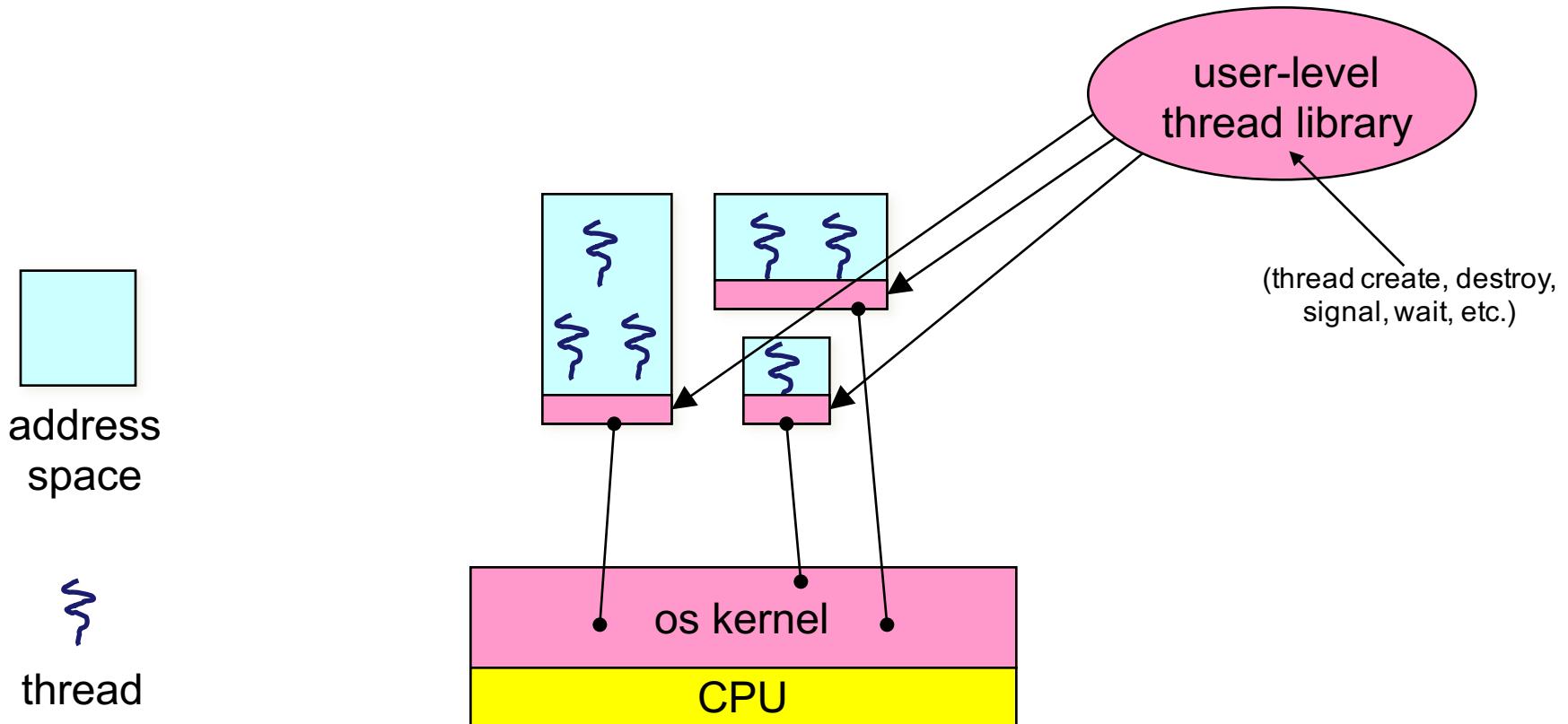
Kernel threads

- OS now manages threads *and* processes / address spaces
 - all thread operations are implemented in the kernel
 - OS schedules all of the threads in a system
 - if one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process
 - possible to overlap I/O and computation **inside** a process
- Kernel threads are cheaper than processes
 - less state to allocate and initialize
- But, they're still pretty expensive for fine-grained use
 - orders of magnitude more expensive than a procedure call
 - thread operations are all **system calls**
 - context switch
 - argument checks
 - must maintain kernel state for each thread

Cheaper alternative

- There is an alternative to kernel threads
- Threads can also be managed at the user level (within the process)
 - a library linked into the program manages the threads
 - the thread manager doesn't need to manipulate address spaces (which only the kernel can do)
 - threads differ (roughly) only in hardware contexts (PC, SP, registers), which can be manipulated by user-level code
 - the **thread package** multiplexes user-level threads on top of kernel thread(s)
 - each kernel thread is treated as a “virtual processor”
 - we call these **user-level threads**

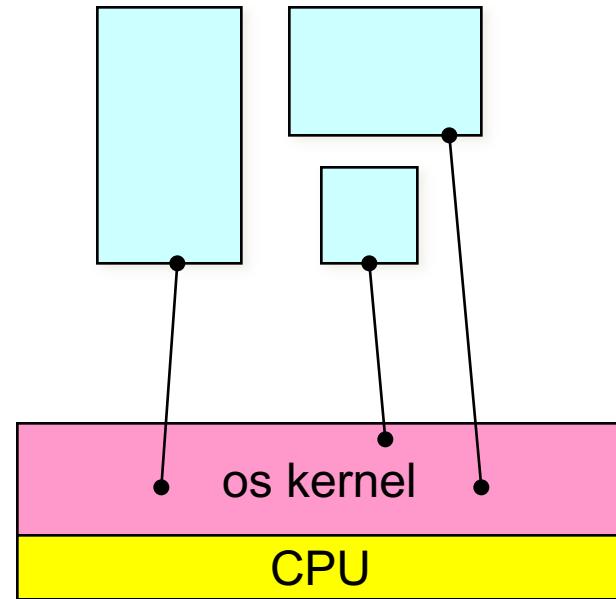
User-level threads



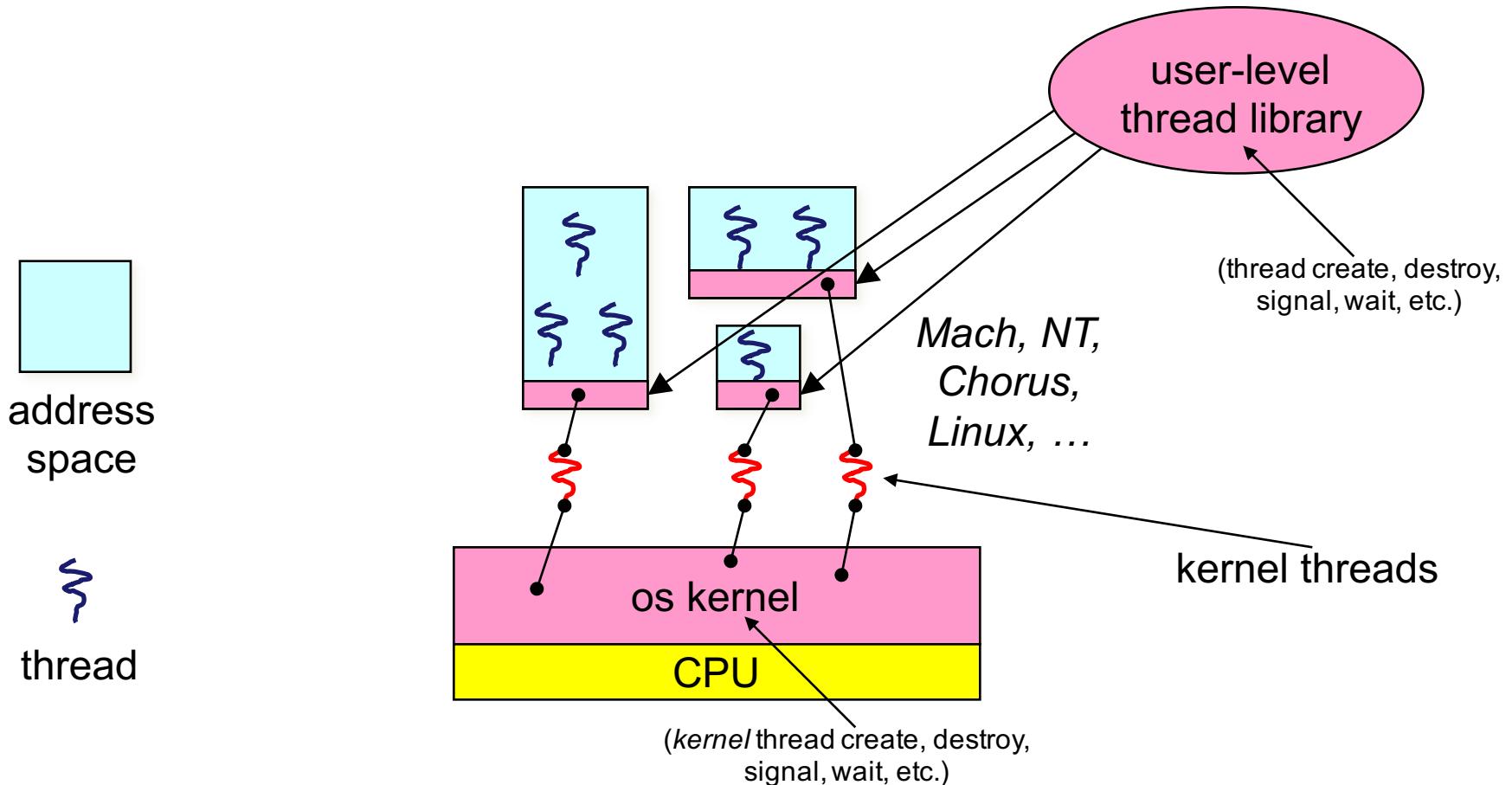
Now thread id is unique within the context of a process, not unique system-wide

User-level threads: what the kernel sees

address space
thread



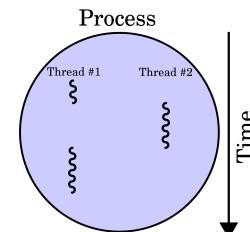
User-level threads



One problem: If a user-level thread blocked due to I/O, all other blocked

User-level threads

- User-level threads are small and fast
 - managed entirely by user-level library
 - E.g., **pthreads** (**libpthreads.a**)
 - each thread is represented simply by a PC, registers, a stack, and a small **thread control block** (TCB)
 - creating a thread, switching between threads, and synchronizing threads are done via procedure calls
 - no kernel involvement is necessary!
- User-level thread operations can be 10-100x faster than kernel threads as a result

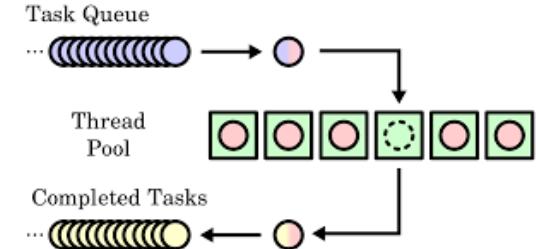
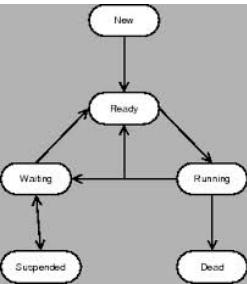


OLD Performance example

- On a 700MHz Pentium running Linux 2.2.16 (only the relative numbers matter; ignore the ancient CPU!):
 - Processes
 - `fork/exit`: 251 µs
 - Kernel threads
 - `pthread_create() / pthread_join()`: 94 µs **(2.5x faster)**
Why?
 - User-level threads
 - `pthread_create() / pthread_join`: 4.5 µs **(another 20x faster)**
Why?

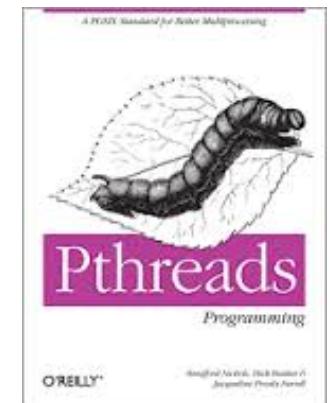
User-level thread implementation

- The OS schedules the kernel thread
- The kernel thread executes user code, including the thread support library and its associated thread scheduler
- The thread scheduler determines when a user-level thread runs
 - it uses queues to keep track of what threads are doing: run, ready, wait
 - just like the OS and processes
 - but, implemented at user-level as a library



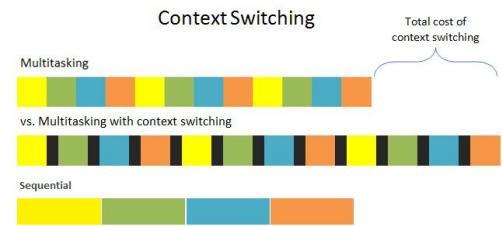
Thread interface

- This is taken from the **POSIX pthreads API**:
 - `rcode = pthread_create(&t, attributes, start_procedure)`
 - creates a new thread of control
 - new thread begins executing at `start_procedure`
 - `pthread_cond_wait(condition_variable, mutex)`
 - the calling thread blocks, sometimes called `thread_block()`
 - `pthread_signal(condition_variable)`
 - starts a thread waiting on the condition variable
 - `pthread_exit()`
 - terminates the calling thread
 - `pthread_join(t)`
 - waits for the named thread to terminate



Thread context switch

- Very simple for user-level threads:
 - save context of currently running thread
 - push CPU state onto thread stack
 - restore context of the next thread
 - pop CPU state from next thread's stack
 - return as the new thread
 - execution resumes at PC of next thread
 - Note: no changes to memory mapping required
- This is all done in assembly language
 - it works at the level of the procedure calling convention



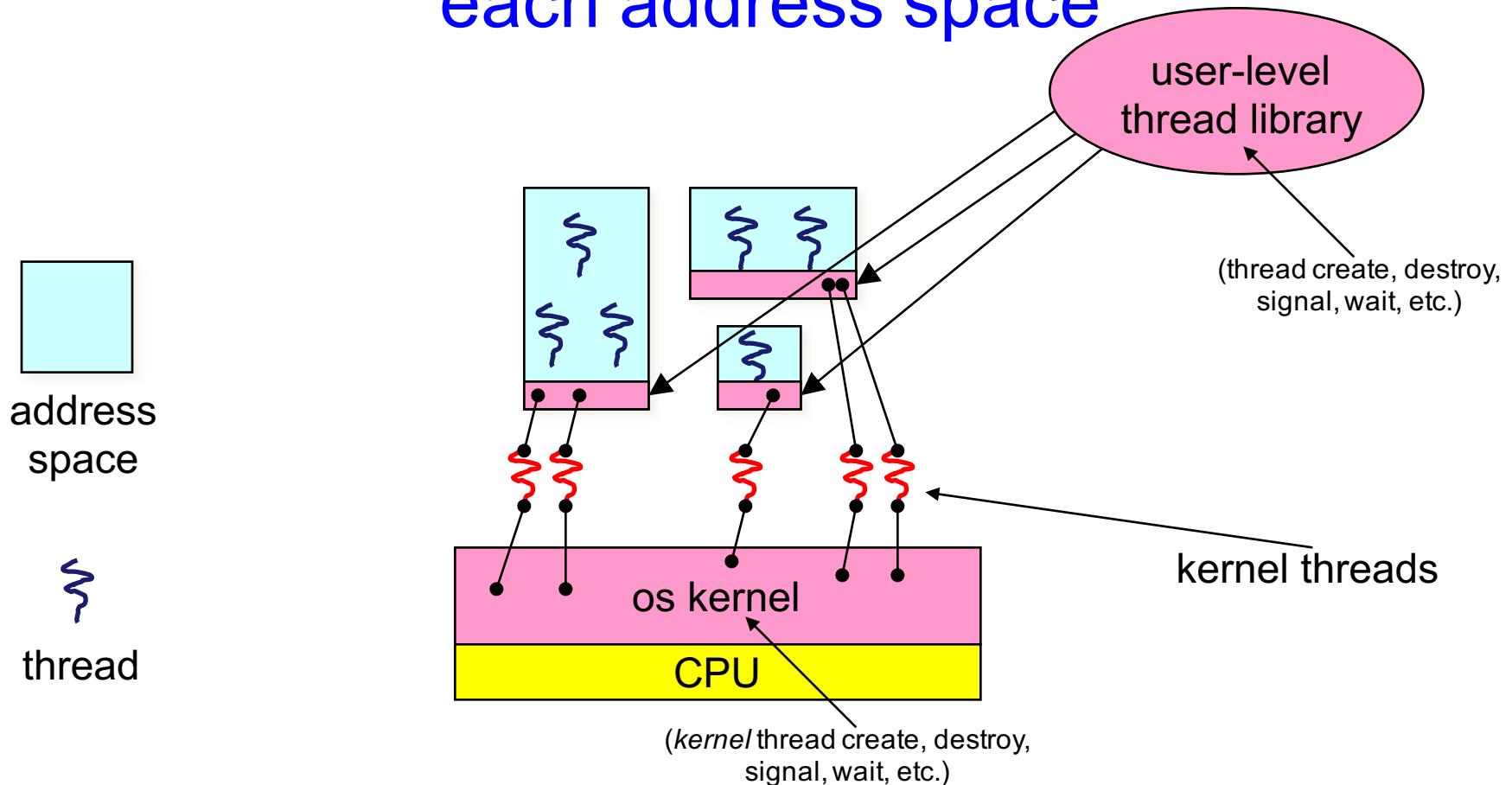
How to keep a user-level thread from hogging the CPU?

- Strategy 1: force everyone to cooperate
 - a thread willingly gives up the CPU by calling `yield()`
 - `yield()` calls into the scheduler, which context switches to another ready thread
 - what happens if a thread never calls `yield()`?
- Strategy 2: use preemption
 - scheduler requests that a timer interrupt be delivered by the OS periodically
 - usually delivered as a UNIX signal (`man signal`)
 - signals are just like software interrupts, but delivered to user-level by the OS instead of delivered to OS by hardware
 - at each timer interrupt, scheduler gains control and context switches as appropriate

What if a thread tries to do I/O?

- The kernel thread “powering” it is lost for the duration of the (synchronous) I/O operation!
 - The kernel thread blocks in the OS, as always
 - It maroons with it the state of the user-level thread
- Could have one kernel thread “powering” each user-level thread
 - “common case” operations (e.g., synchronization) would be quick
- Could have a limited-size “pool” of kernel threads “powering” all the user-level threads in the address space
 - the kernel will be scheduling these threads, obliviously to what’s going on at user-level

Multiple kernel threads “powering” each address space



Summary

- Multiple threads per address space
- Kernel threads are much more efficient than processes, but still expensive
 - all operations require a kernel call and parameter validation
- User-level threads are:
 - much cheaper and faster
 - great for common-case operations
 - creation, synchronization, destruction
 - can suffer in uncommon cases due to kernel obliviousness
 - I/O
 - preemption of a lock-holder

Operating Systems

Synchronization

Lecture 5
Michael O'Boyle

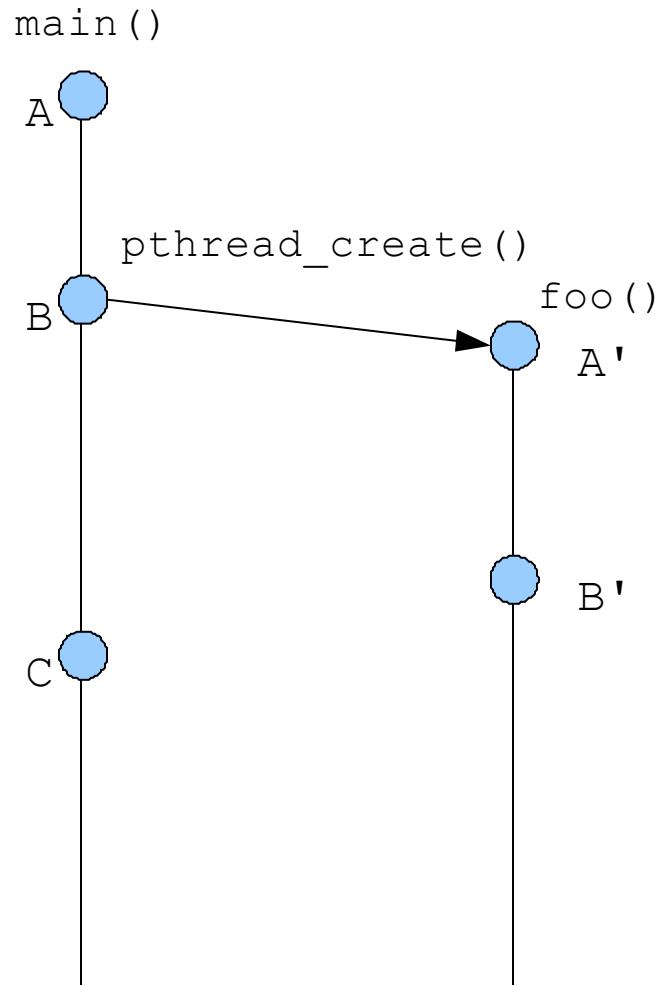
Temporal relations

User view of parallel threads

- Instructions executed by a single thread are totally ordered
 - $A < B < C < \dots$
- In absence of **synchronization**,
 - instructions executed by distinct threads must be considered unordered / simultaneous
 - Not $X < X'$, and not $X' < X$

Hardware largely supports this

Example



Y-axis is “time.”

*Could be one CPU, could
be multiple CPUs (cores).*

- $A < B < C$
- $A' < B'$
- $A < A'$
- $C == A'$
- $C == B'$

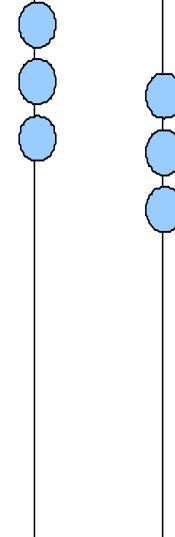
Critical Sections / Mutual Exclusion

- Sequences of instructions that may get incorrect results if executed simultaneously are called **critical sections**
- **Race condition** results depend on timing
- **Mutual exclusion** means “not simultaneous”
 - $A < B$ or $B < A$
 - We don’t care which
- Forcing mutual exclusion between two critical section executions
 - is sufficient to ensure correct execution
 - guarantees ordering

Critical sections

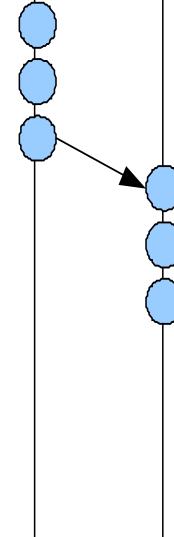
→ is the "happens-before" relation

T1 T2



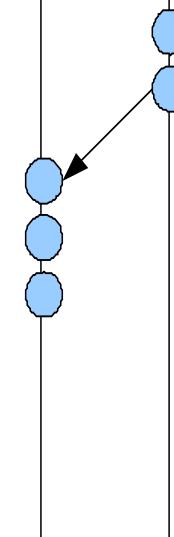
Possibly incorrect

T1 T2



Correct

T1 T2



Correct

When do critical sections arise?

- One common pattern:
 - read-modify-write of
 - a shared value (variable)
 - in code that can be executed by concurrent threads
- Shared variable:
 - Globals and heap-allocated variables
 - NOT local variables (which are on the stack)

Race conditions

- A program has a **race condition** (data race) if the result of an executing depends on timing
 - i.e., is non-deterministic
- Typical symptoms
 - I run it on the same data, and sometimes it prints 0 and sometimes it prints 4
 - I run it on the same data, and sometimes it prints 0 and sometimes it crashes

Example: shared bank account

- Suppose we have to implement a function to withdraw money from a bank account:

```
int withdraw(account, amount) {  
    int balance = get_balance(account);      // read  
    balance -= amount;                      // modify  
    put_balance(account, balance);          // write  
    spit out cash;  
}
```

- Now suppose that you and your partner share a bank account with a balance of £100.00
 - what happens if you both go to separate CashPoint machines, and simultaneously withdraw £10.00 from the account?

- Assume the bank's application is multi-threaded
- A random thread is assigned a transaction when that transaction is submitted

```
int withdraw(account, amount) {  
    int balance = get_balance(account);  
    balance -= amount;  
    put_balance(account, balance);  
    spit out cash;  
}
```

```
int withdraw(account, amount) {  
    int balance = get_balance(account);  
    balance -= amount;  
    put_balance(account, balance);  
    spit out cash;  
}
```

Interleaved schedules

- The problem is that the execution of the two threads can be interleaved:

Execution sequence
as seen by CPU

```
balance = get_balance(account);  
balance -= amount;
```

context switch

```
balance = get_balance(account);  
balance -= amount;  
put_balance(account, balance);  
spit out cash;
```

context switch

```
put_balance(account, balance);  
spit out cash;
```

- What's the account balance after this sequence?
 - who's happy, the bank or you?
- How often is this sequence likely to occur?

Other Execution Orders

- Which interleavings are ok? Which are not?

```
int withdraw(account, amount) {  
    int balance = get_balance(account);  
    balance -= amount;  
    put_balance(account, balance);  
    spit out cash;  
}
```

```
int withdraw(account, amount) {  
    int balance = get_balance(account);  
    balance -= amount;  
    put_balance(account, balance);  
    spit out cash;  
}
```

How About Now?

```
int xfer(from, to, machine) {  
    withdraw( from, machine );  
    deposit( to, machine );  
}
```

```
int xfer(from, to, machine) {  
    withdraw( from, machine );  
    deposit( to, machine );  
}
```

- Moral:
 - Interleavings are hard to reason about
 - We make lots of mistakes
 - Control-flow analysis is hard for tools to get right
 - Identifying critical sections and ensuring mutually exclusive access can make things easier

Another example

```
i++;
```

```
i++;
```

Correct critical section requirements

- Correct critical sections have the following requirements
 - mutual exclusion
 - at most one thread is in the critical section
 - progress
 - if thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
 - bounded waiting (no starvation)
 - if thread T is waiting on the critical section, then T will eventually enter the critical section
 - assumes threads eventually leave critical sections
 - performance
 - the overhead of entering and exiting the critical section is small with respect to the work being done within it

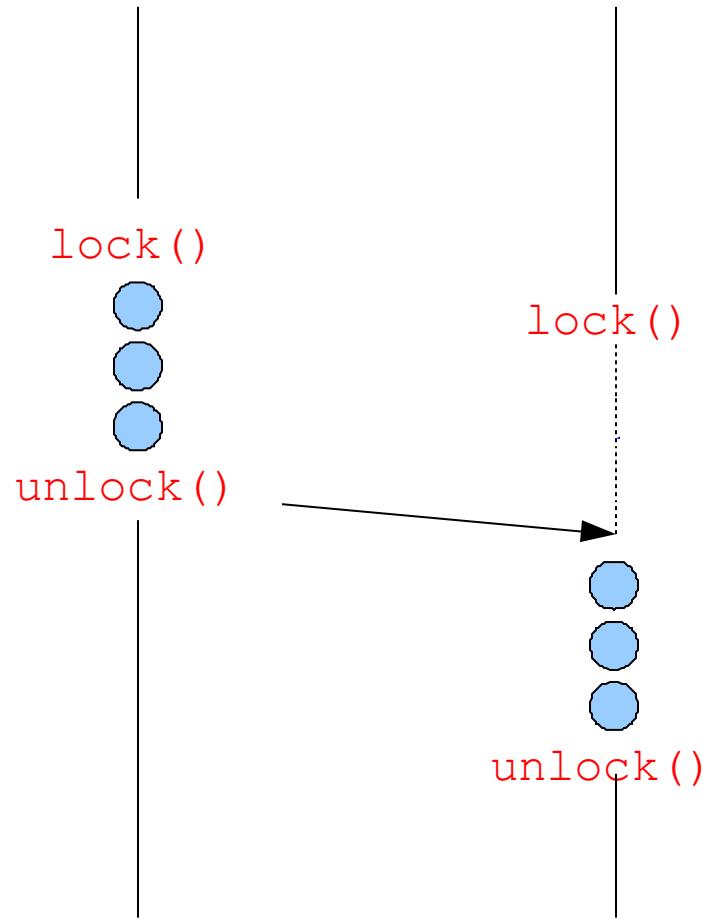
Mechanisms for building critical sections

- Spinlocks
 - primitive, minimal semantics; used to build others
- Semaphores (and non-spinning locks)
 - basic, easy to get the hang of, somewhat hard to program with
- Monitors
 - higher level, requires language support, implicit operations
 - easier to program with; Java “`synchronized()`” as an example
- Messages
 - simple model of communication and synchronization based on (atomic) transfer of data across a channel
 - direct application to distributed systems

Locks

- A lock is a memory object with two operations:
 - `acquire()`: obtain the right to enter the critical section
 - `release()`: give up the right to be in the critical section
- `acquire()` prevents progress of the thread until the lock can be acquired
- Note: terminology varies: acquire/release, lock/unlock

Locks: Example



Acquire/Release

- Threads pair up calls to `acquire()` and `release()`
 - between `acquire()` and `release()`, the thread **holds** the lock
 - `acquire()` does not return until the caller “owns” (holds) the lock
 - at most one thread can hold a lock at a time
- What happens if the calls aren’t paired
 - I acquire, but neglect to release?
- What happens if the two threads acquire different locks
 - I think that access to a particular shared data structure is mediated by lock A, and you think it’s mediated by lock B?
- What is the right granularity of locking?

Using locks

```
int withdraw(account, amount) {  
    acquire(lock);  
    balance = get_balance(account);  
    balance -= amount;  
    put_balance(account, balance);  
    release(lock);  
    spit out cash;  
}
```

} critical section

```
acquire(lock)  
balance = get_balance(account);  
balance -= amount;
```

```
acquire(lock)
```

```
put_balance(account, balance);  
release(lock);
```

```
balance = get_balance(account);  
balance -= amount;  
put_balance(account, balance);  
release(lock);  
spit out cash;
```

```
spit out cash;
```

- What happens when green tries to acquire the lock?

Spinlocks

- How do we implement spinlocks? Here's one attempt:

```
struct lock_t {  
    int held = 0;  
}  
  
void acquire(lock) {  
    while (lock->held);  
    lock->held = 1;  
}  
  
void release(lock) {  
    lock->held = 0;  
}
```

the caller "busy-waits",
or spins, for lock to be
released ⇒ hence spinlock

- Race condition in acquire

Implementing spinlocks

- Problem is that implementation of spinlocks has critical sections, too!
 - the acquire/release must be **atomic**
 - atomic == executes as though it could not be interrupted
 - code that executes “all or nothing”
 - Compiler can hoist code that is invariant
- Need help from the hardware
 - atomic instructions
 - test-and-set, compare-and-swap, ...

Spinlocks: Hardware Test-and-Set

- CPU provides the following as **one atomic instruction**:

```
bool test_and_set(bool *flag) {  
    bool old = *flag;  
    *flag = True;  
    return old;  
}
```

- Remember, this is a single atomic instruction ...

Implementing spinlocks using Test-and-Set

- So, to fix our broken spinlocks:

```
struct lock {  
    int held = 0;  
}  
void acquire(lock) {  
    while(test_and_set(&lock->held));  
}  
void release(lock) {  
    lock->held = 0;  
}
```

- mutual exclusion? (at most one thread in the critical section)
- progress? (T outside cannot prevent S from entering)
- bounded waiting? (waiting T will eventually enter)
- performance? (low overhead (modulo the spinning part ...))

Reminder of use ...

```
int withdraw(account, amount) {  
    acquire(lock);  
    balance = get_balance(account);  
    balance -= amount;  
    put_balance(account, balance);  
    release(lock);  
    spit out cash;  
}
```

} critical section

```
acquire(lock)  
balance = get_balance(account);  
balance -= amount;
```

```
acquire(lock)
```

```
put_balance(account, balance);  
release(lock);
```

```
balance = get_balance(account);  
balance -= amount;  
put_balance(account, balance);  
release(lock);  
spit out cash;
```

```
spit out cash;
```

- How does a thread blocked on an “acquire” (that is, stuck in a test-and-set loop) yield the CPU?
 - calls `yield()` (*spin-then-block*)
 - there’s an involuntary context switch (e.g., timer interrupt)

Problems with spinlocks

- Spinlocks work, but are wasteful!
 - if a thread is spinning on a lock, the thread holding the lock cannot make progress
 - You'll spin for a scheduling quantum
 - (`pthread_spin_t`)
- Only want spinlocks as primitives to build higher-level synchronization constructs
 - Ok as ensure acquiring only happens for a short time
- We'll see later how to build blocking locks
 - But there is overhead – can be cheaper to spin

Summary

- Synchronization introduces temporal ordering
- Synchronization can eliminate races
- Synchronization can be provided by locks, semaphores, monitors, messages ...
- Spinlocks are the lowest-level mechanism
 - primitive in terms of semantics – error-prone
 - implemented by spin-waiting (crude) or by disabling interrupts (also crude, and can only be done in the kernel)

Operating Systems

Semaphores, Condition Variables,
and Monitors

Lecture 6
Michael O'Boyle

Semaphore

- More sophisticated Synchronization mechanism
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations
 - **wait()** and **signal()**
 - Originally called **P()** and **V()**
- Definition

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

- Definition

```
signal(S) {  
    S++;  
}
```

Do these operations *atomically*

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
 - Same as a **lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2
Create a semaphore “**synch**” initialized to 0

P1 :

```
S1;  
    signal(synch);
```

P2 :

```
wait(synch);  
S2;
```

- Can implement a counting semaphore S as a binary semaphore

Implementation with no Busy waiting

Each semaphore has an associated queue of threads

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this thread to S->list;  
        block();  
    }  
}  
  
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a thread T from S->list;  
        wakeup(T);  
    }  
}
```

Binary semaphore usage

- From the programmer's perspective, P and V on a binary semaphore are just like Acquire and Release on a lock

P(sem)

:

:

do whatever stuff requires mutual exclusion; could conceivably
be a lot of code

:

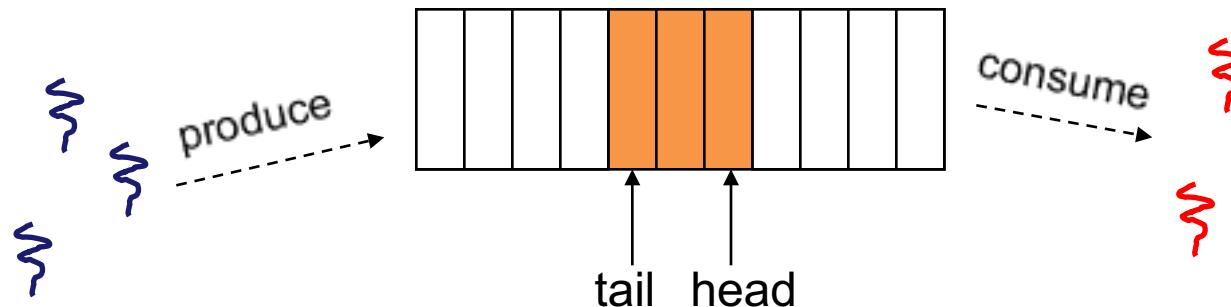
:

V(sem)

- same lack of programming language support for correct usage
- Important differences in the underlying implementation, however
- No busy waiting

Example: Bounded buffer problem

- AKA “producer/consumer” problem
 - there is a circular buffer in memory with N entries (slots)
 - producer threads insert entries into it (one at a time)
 - consumer threads remove entries from it (one at a time)
- Threads are concurrent
 - so, we must use synchronization constructs to control access to shared variables describing buffer state



Bounded buffer using semaphores (both binary and counting)

```
var mutex: semaphore = 1      ; mutual exclusion to shared data
empty: semaphore = n    ; count of empty slots (all empty to start)
full: semaphore = 0      ; count of full slots (none full to start)
```

producer:

```
P(empty) ; block if no slots available
P(mutex) ; get access to pointers
<add item to slot, adjust pointers>
V(mutex) ; done with pointers
V(full) ; note one more full slot
```

consumer:

```
P(full) ; wait until there's a full slot
P(mutex) ; get access to pointers
<remove item from slot, adjust pointers>
V(mutex) ; done with pointers
V(empty) ; note there's an empty slot
<use the item>
```

Example: Readers/Writers

- Description:
 - A single object is shared among several threads/processes
 - Sometimes a thread just reads the object
 - Sometimes a thread updates (writes) the object
 - **We can allow multiple readers at a time**
 - **Do not change state – no race condition**
 - **We can only allow one writer at a time**
 - Change state- race condition



Readers/Writers using semaphores

```
var mutex: semaphore = 1      ; controls access to readcount  
wrt: semaphore = 1      ; control entry for a writer or first reader  
readcount: integer = 0      ; number of active readers
```

writer:

```
P(wrt)          ; any writers or readers?  
    <perform write operation>  
V(wrt)          ; allow others
```

reader:

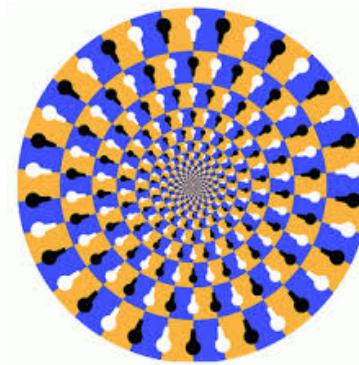
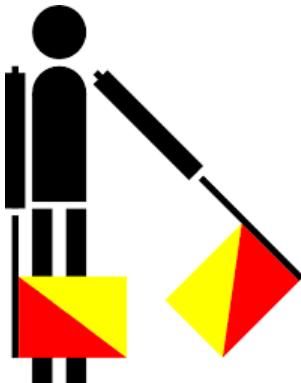
```
P(mutex)          ; ensure exclusion  
    readcount++      ; one more reader  
    if readcount == 1 then P(wrt)    ; if we're the first, synch with writers  
V(mutex)  
    <perform read operation>  
P(mutex)          ; ensure exclusion  
    readcount--      ; one fewer reader  
    if readcount == 0 then V(wrt)    ; no more readers, allow a writer  
V(mutex)
```

Readers/Writers notes

- Notes:
 - the first reader blocks on P(wrt) if there is a writer
 - any other readers will then block on P(mutex)
 - if a waiting writer exists, the last reader to exit signals the waiting writer
 - A new reader cannot get in while a writer is waiting
 - When writer exits, if there is both a reader and writer waiting, which one goes next?

Semaphores vs. Spinlocks

- Threads that are blocked at the level of program logic (that is, by the semaphore P operation) are placed on queues, rather than busy-waiting
- Busy-waiting may be used for the “real” mutual exclusion required to implement P and V
 - but these are very short critical sections – totally independent of program logic
 - and they are not implemented by the application programmer



Abstract implementation

- $P/\text{wait}(\text{sem})$
 - acquire “real” mutual exclusion
 - if sem is “available” (>0), decrement sem ; release “real” mutual exclusion; let thread continue
 - otherwise, place thread on associated queue; release “real” mutual exclusion; run some other thread
- $V/\text{signal}(\text{sem})$
 - acquire “real” mutual exclusion
 - if thread(s) are waiting on the associated queue, unblock one (place it on the ready queue)
 - if no threads are on the queue, sem is incremented
 - » the signal is “remembered” for next time $P(\text{sem})$ is called
 - release “real” mutual exclusion
 - the “V-ing” thread continues execution

Problems with semaphores, locks

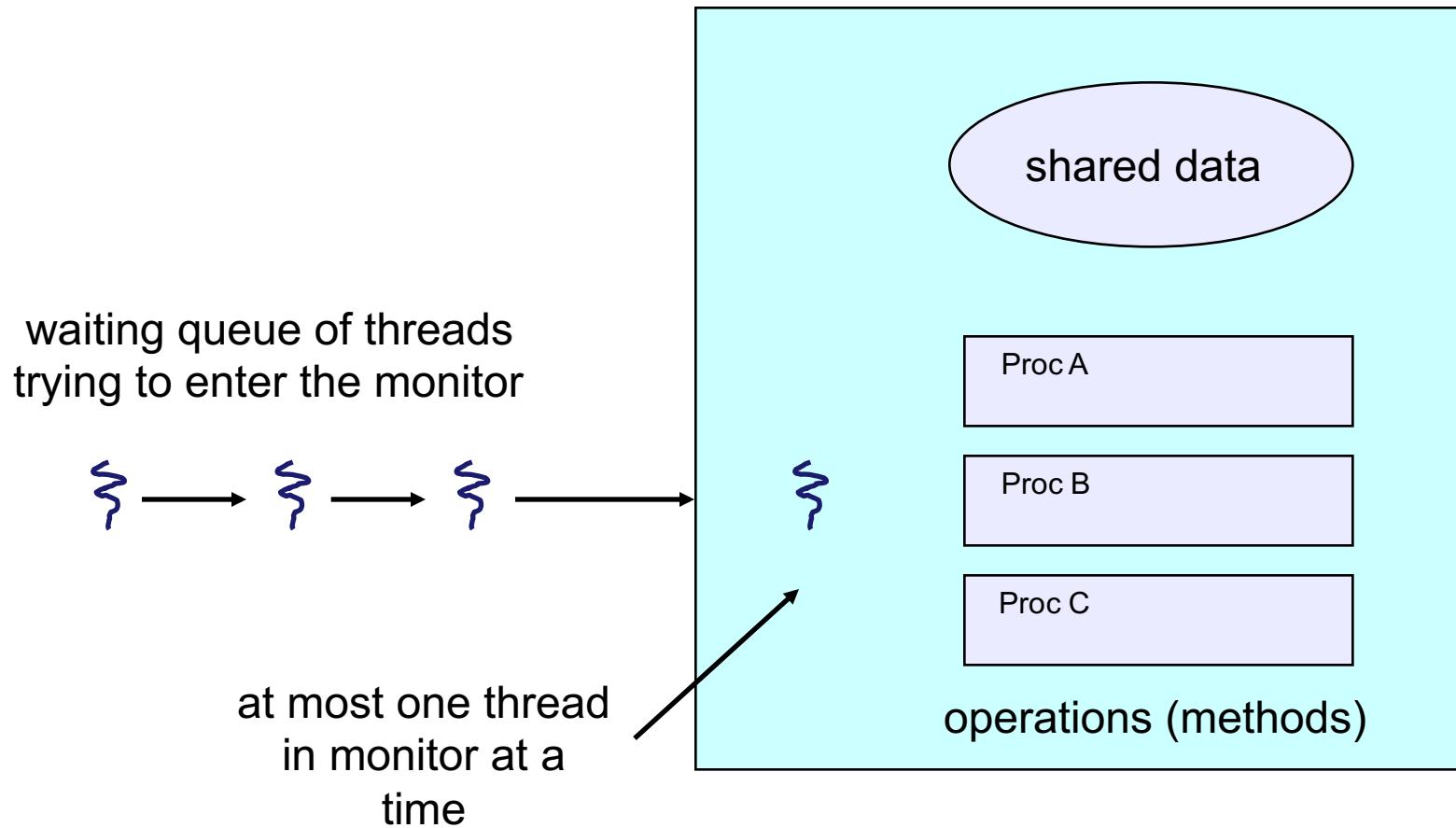
- They can be used to solve any of the traditional synchronization problems, but it's easy to make mistakes
 - they are essentially shared global variables
 - can be accessed from anywhere (bad software engineering)
 - there is no connection between the synchronization variable and the data being controlled by it
 - No control over their use, no guarantee of proper usage
 - Semaphores: will there ever be a V()?
 - Locks: did you lock when necessary? Unlock at the right time? At all?
- Thus, they are prone to bugs
 - We can reduce the chance of bugs by “stylizing” the use of synchronization
 - Language help is useful for this



One More Approach: Monitors

- A programming language construct supports controlled shared data access
 - synchronization code is added by the compiler
- A class in which every method automatically acquires a lock on entry, and releases it on exit – it combines:
 - **shared data** structures (object)
 - **procedures** that operate on the shared data (object methods)
 - **synchronization** between concurrent threads that invoke those procedures
- Data can only be accessed from within the
 - protects the data from unstructured access
 - Prevents ambiguity about what the synchronization variable protects
- Addresses the key usability issues that arise with semaphores

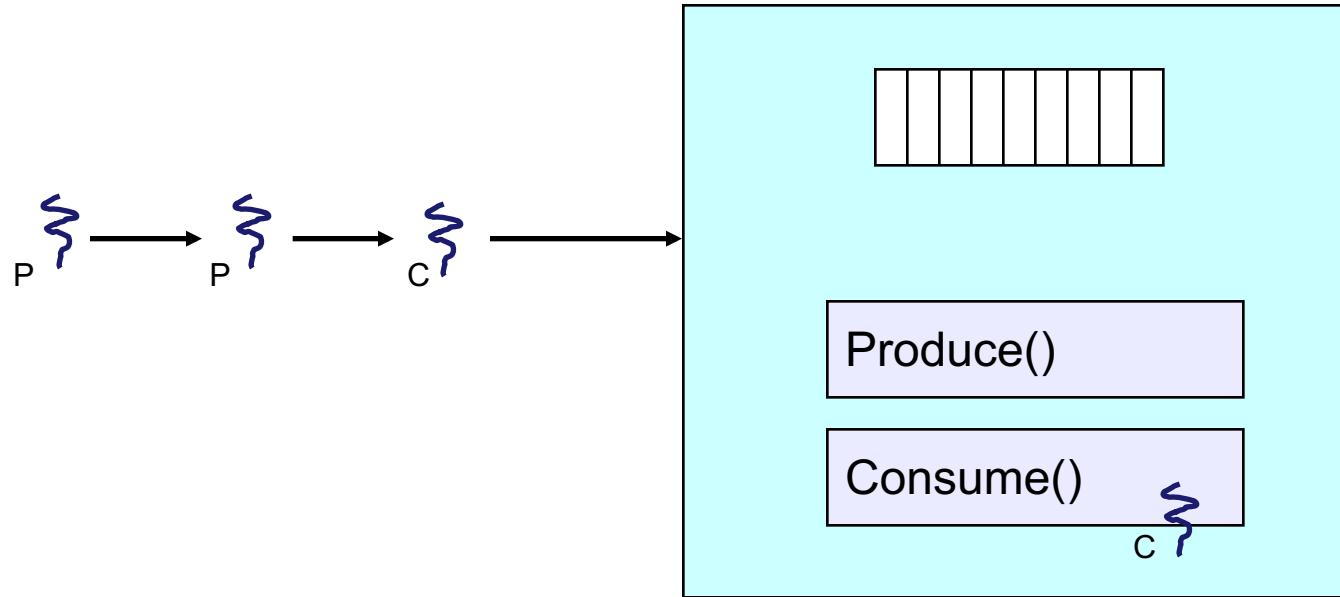
A monitor



Monitor facilities

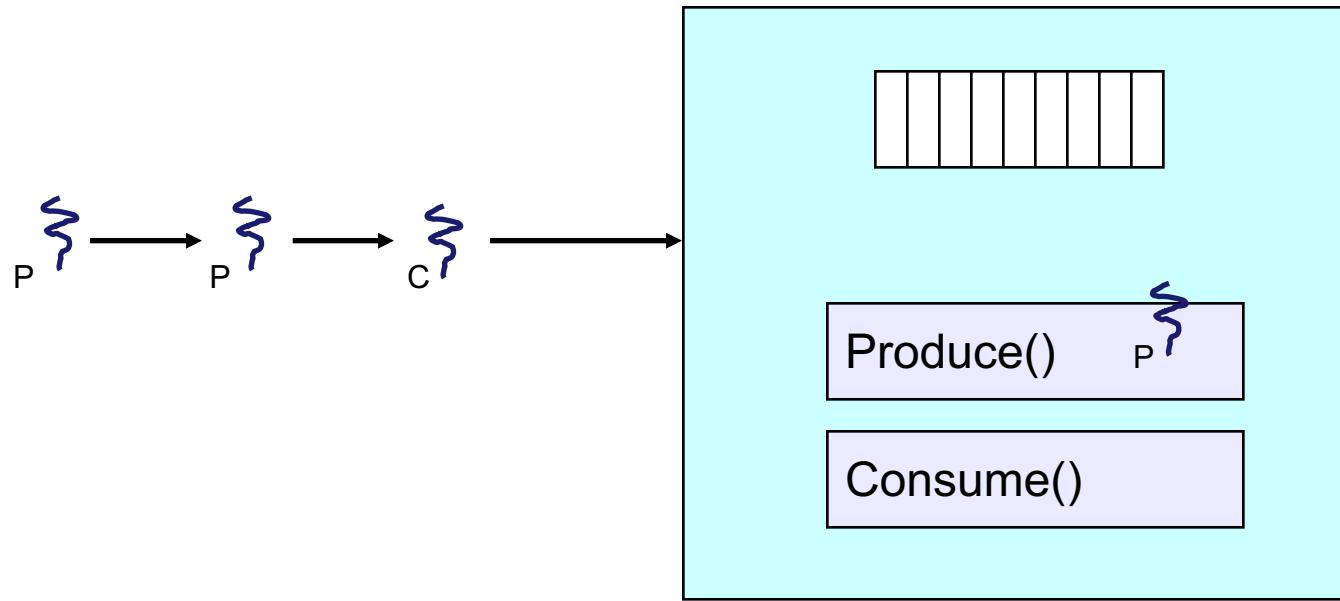
- “Automatic” mutual exclusion
 - only one thread can be executing inside at any time
 - thus, synchronization is implicitly associated with the monitor – it “comes for free”
 - if a second thread tries to execute a monitor procedure, it blocks until the first has left the monitor
 - more restrictive than semaphores
 - but easier to use (most of the time)
- But, there’s a problem...

Problem: Bounded Buffer Scenario



- Buffer is empty
- Now what?

Problem: Bounded Buffer Scenario



- Buffer is full
- Now what?

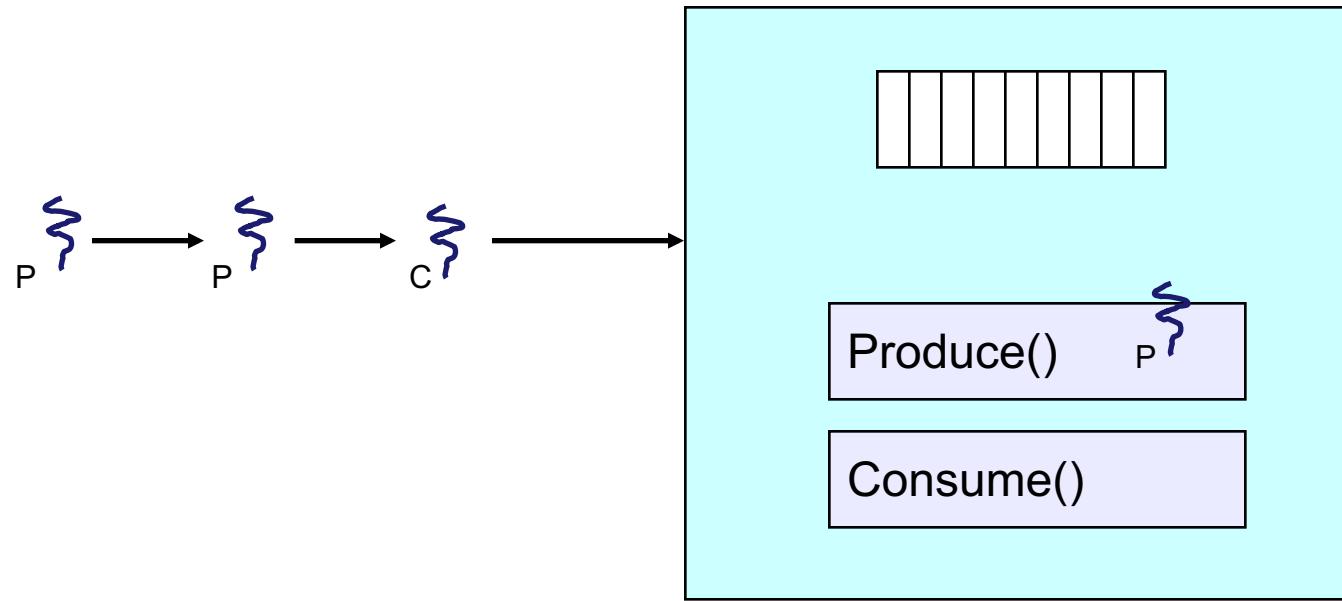
Solution?

- Monitors require condition variables
- Operations on condition variables
 - **wait(c)**
 - release monitor lock, so somebody else can get in
 - wait for somebody else to signal condition
 - thus, condition variables have associated wait queues
 - **signal(c)**
 - wake up at most one waiting thread
 - “Hoare” monitor: wakeup immediately, signaller steps outside
 - if no waiting threads, signal is lost
 - this is different than semaphores: no history!
 - **broadcast(c)**
 - wake up all waiting threads

Bounded buffer using (Hoare) monitors

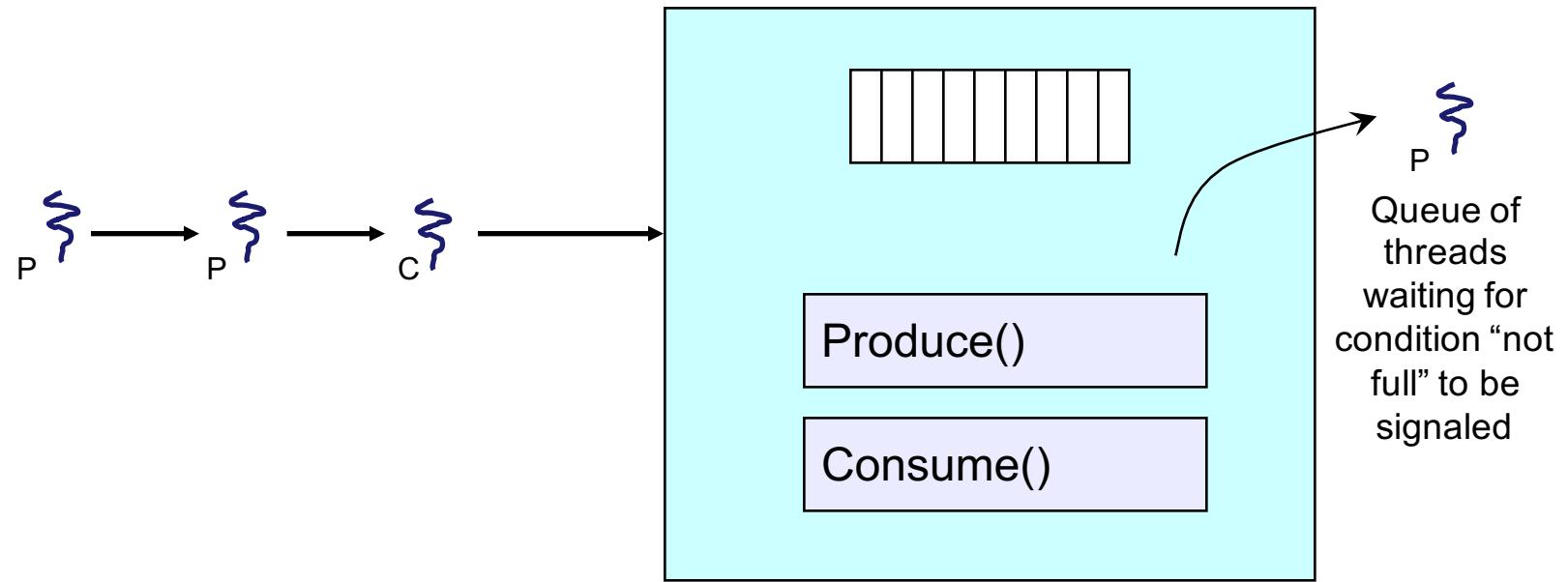
```
Monitor bounded_buffer {  
    buffer resources[N];  
    condition not_full, not_empty;  
  
produce(resource x) {  
    if (array “resources” is full, determined maybe by a count)  
        wait(not_full);  
    insert “x” in array “resources”  
    signal(not_empty);  
}  
  
consume(resource *x) {  
    if (array “resources” is empty, determined maybe by a count)  
        wait(not_empty);  
    *x = get resource from array “resources”  
    signal(not_full);  
}  
}
```

Problem: Bounded Buffer Scenario



- Buffer is full
- Now what?

Bounded Buffer Scenario with CV's



- Buffer is full
- Now what?

Runtime system calls for (Hoare) monitors

- EnterMonitor(m) {guarantee mutual exclusion}
- ExitMonitor(m) {hit the road, letting someone else run}
- Wait(c) {step out until condition satisfied}
- Signal(c) {if someone's waiting, step out and let him run}
- EnterMonitor and ExitMonitor are inserted automatically by the compiler.
- This guarantees mutual exclusion for code inside of the monitor.

Bounded buffer using (Hoare) monitors

```
Monitor bounded_buffer {  
    buffer resources[N];  
    condition not_full, not_empty;  
  
    procedure add_entry(resource x) { ..... EnterMonitor(m)  
        if (array "resources" is full, determined maybe by a count)  
            wait(not_full);  
        insert "x" in array "resources"  
        signal(not_empty); ..... ExitMonitor(m)  
    }  
    procedure get_entry(resource *x) { ..... EnterMonitor(m)  
        if (array "resources" is empty, determined maybe by a count)  
            wait(not_empty);  
        *x = get resource from array "resources"  
        signal(not_full); ..... ExitMonitor(m)  
    }  
}
```

Monitor Summary

- Language supports monitors
- Compiler understands them
 - Compiler inserts calls to runtime routines for
 - monitor entry
 - monitor exit
 - Programmer inserts calls to runtime routines for
 - signal
 - wait
 - Language/object encapsulation ensures correctness
 - Sometimes! With conditions, you *still* need to think about synchronization
- Runtime system implements these routines
 - moves threads on and off queues
 - *ensures mutual exclusion!*

Operating Systems

Deadlock

Lecture 7
Michael O'Boyle

Definition

- A thread is deadlocked when it's waiting for an event that can never occur
 - I'm waiting for you to clear the intersection, so I can proceed
 - but you can't move until he moves, and he can't move until she moves, and she can't move until I move
- Thread A is in critical section 1,
 - waiting for access to critical section 2;
- Thread B is in critical section 2,
 - waiting for access to critical section 1

Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);
    pthread_exit(0);
}

/* thread two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /** * Do some work */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);
    pthread_exit(0);
}
```

Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount)
{
    mutex lock1, lock2;
    lock1 = get_lock(from);
    lock2 = get_lock(to);
    acquire(lock1);
    acquire(lock2);
    withdraw(from, amount);
    deposit(to, amount);
    release(lock2);
    release(lock1);
}
```

Transactions 1 and 2 execute concurrently.

Transaction 1 transfers \$25 from account A to account B, and

Transaction 2 transfers \$50 from account B to account A

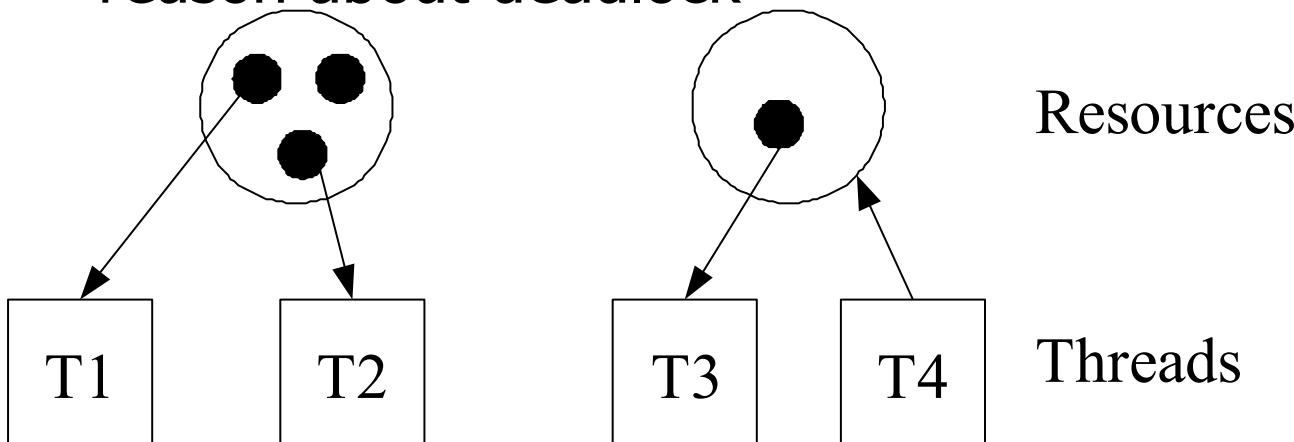
Four conditions must exist for deadlock to be possible

1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

We'll see that deadlocks can be addressed by attacking any of these four conditions.

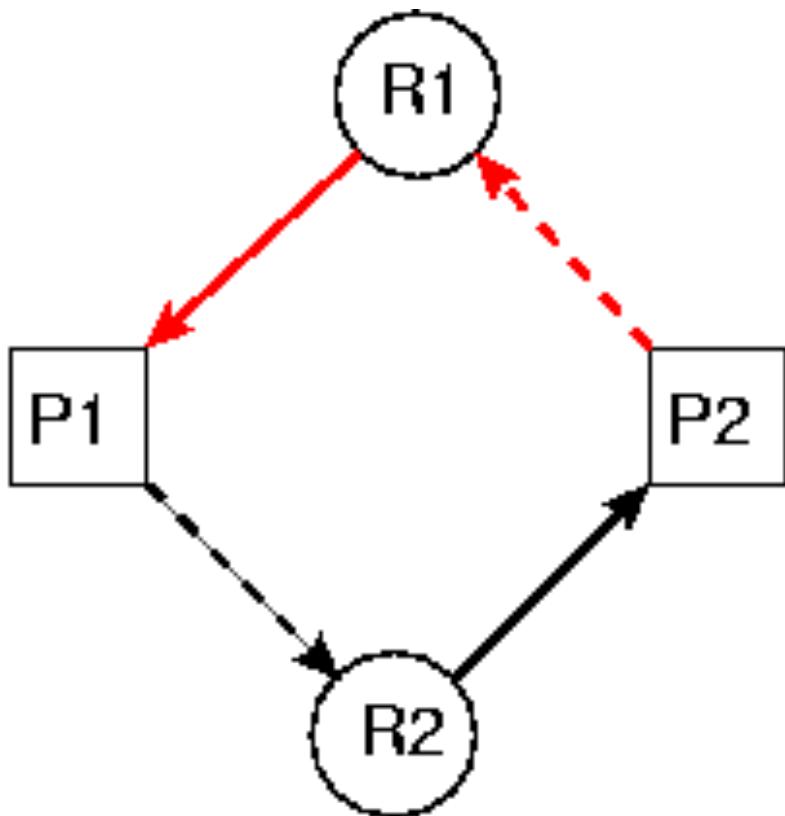
Resource Graphs

- Resource graphs are a way to visualize the (deadlock-related) state of the threads, and to reason about deadlock



- 1 or more identical units of a resource are available
- A thread may hold resources (arrows to threads)
- A thread may request resources (arrows from threads)

Deadlock



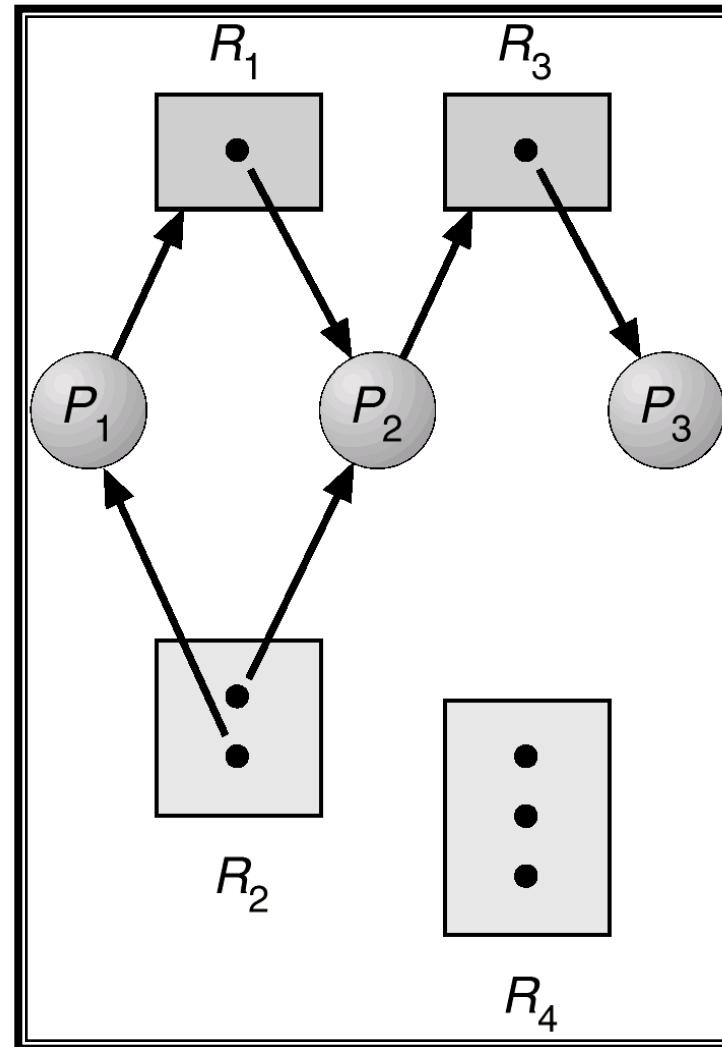
- R1 is held by
- - → is waiting for R1
- R2 is held by
- - → is waiting for R2

- A deadlock exists if there is an *irreducible cycle* in the resource graph (such as the one above)

Graph reduction

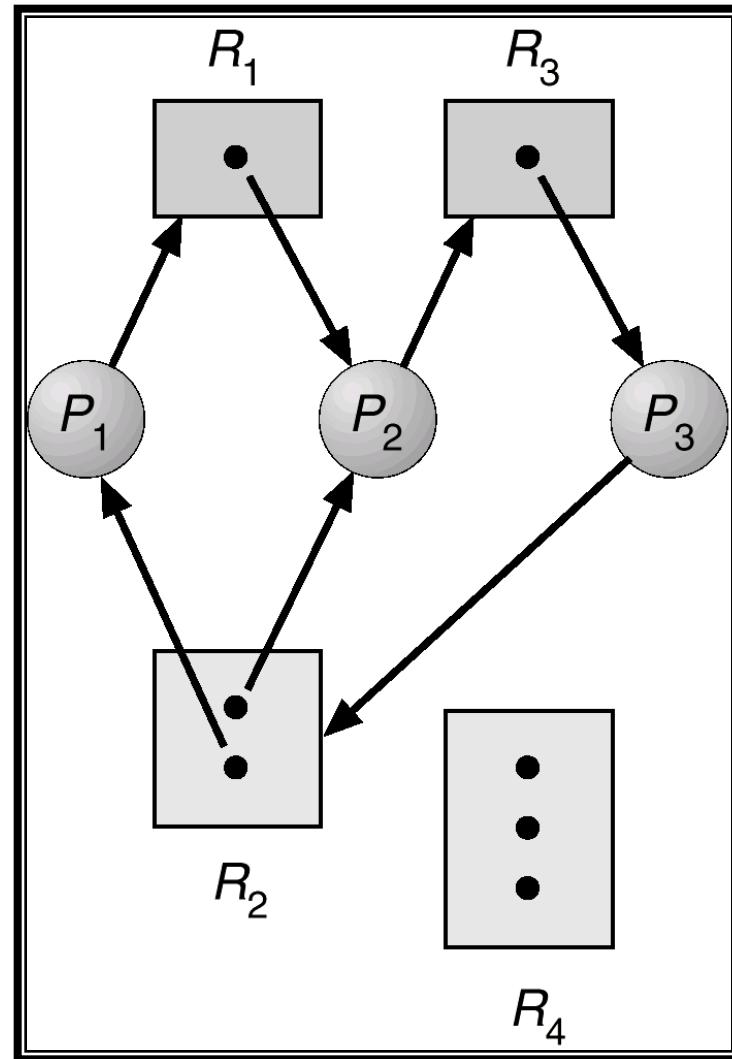
- A graph can be *reduced* by a thread if all of that thread's requests can be granted
 - in this case, the thread eventually will terminate – all resources are freed – all arcs (allocations) to/from it in the graph are deleted
- Miscellaneous theorems (Holt, Havender):
 - There are no deadlocked threads iff the graph is completely reducible
 - The order of reductions is irrelevant

Resource allocation graph with no cycle

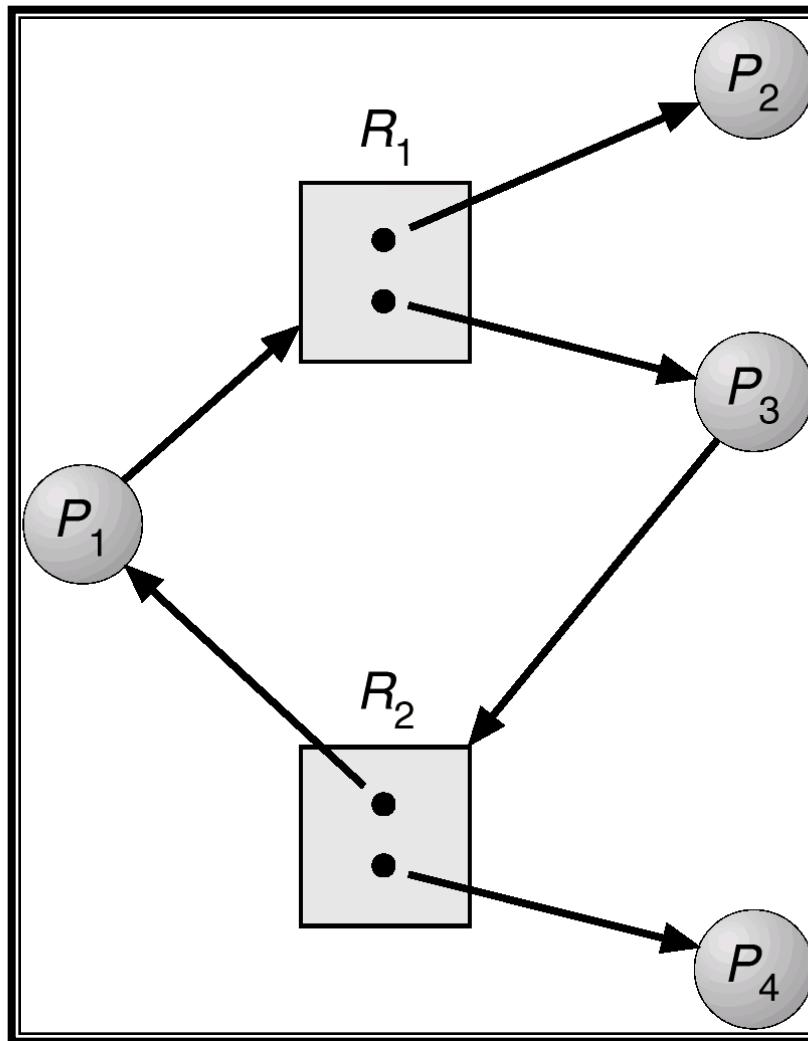


What would cause a deadlock?

Resource allocation graph with a deadlock



Resource allocation graph with a cycle but no deadlock



Handling Deadlock

- Eliminate one of the four required conditions
 - Mutual Exclusion
 - Hold and Wait
 - No Preemption
 - Circular Wait
- Broadly classified as:
 - Prevention, or
 - Avoidance, or
 - Detection (and recovery)

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
 - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No (resource) Preemption –**
 - If a process holding some resources requests another unavailable resource all resources currently held are released
 - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait**
 - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Avoidance

Less severe restrictions on program behavior

- Eliminating circular wait
 - each thread states its maximum claim for every resource type
 - system runs the Banker's Algorithm at each allocation request
 - Banker \Rightarrow highly conservative
- More on this shortly

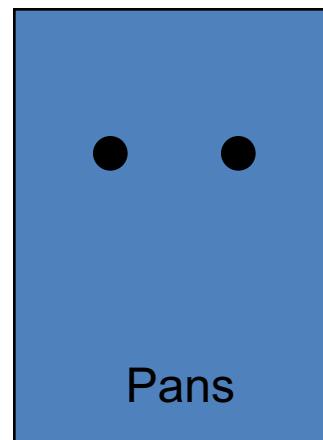
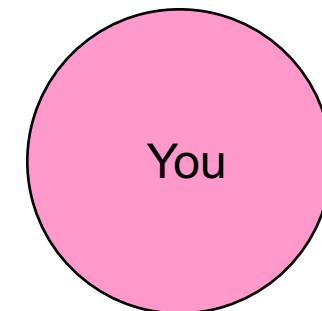
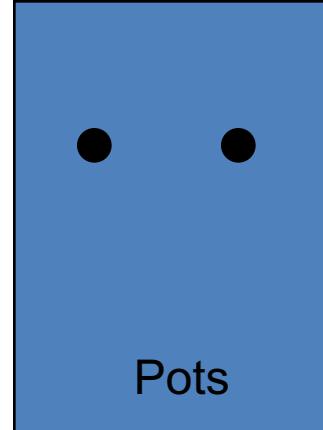
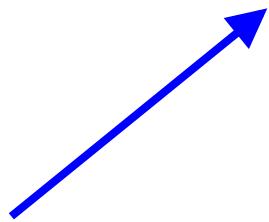
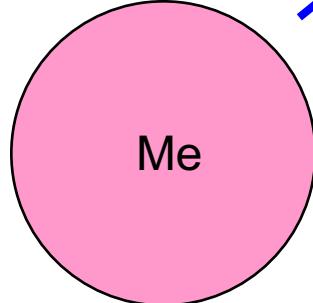
Detect and recover

- Every once in a while, check to see if there's a deadlock
 - how?
- If so, eliminate it
 - how?

Avoidance: Banker's Algorithm example

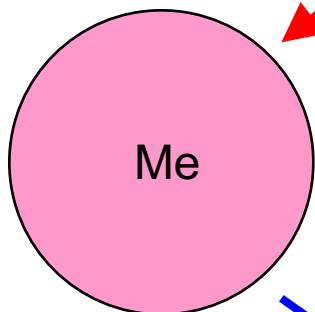
- Background
 - The set of controlled resources is known to the system
 - The number of units of each resource is known to the system
 - Each application must declare its maximum possible requirement of each resource type
- Then, the system can do the following:
 - When a request is made
 - pretend you granted it
 - pretend all other legal requests were made
 - can the graph be reduced?
 - if so, allocate the requested resource
 - if not, block the thread until some thread releases resources, and then try pretending again

1. I request a pot

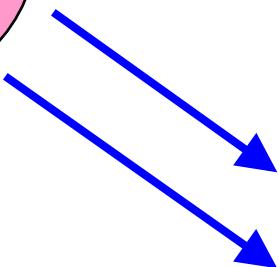
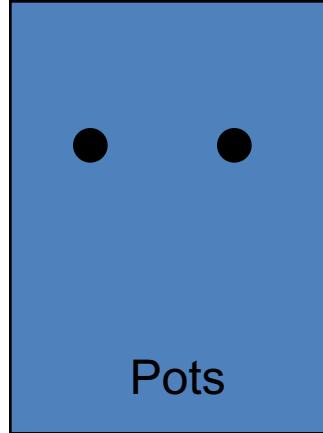
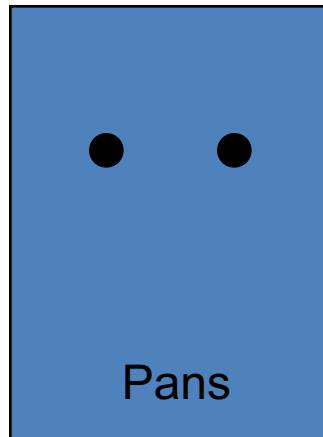
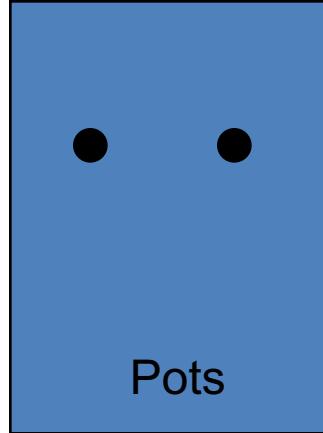


Max:
1 pot
2 pans

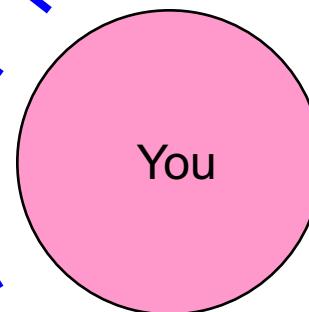
Max:
2 pots
1 pan



Max:
1 pot
2 pans



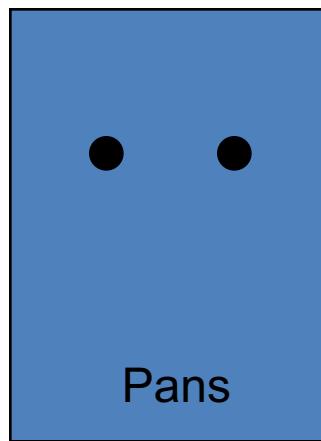
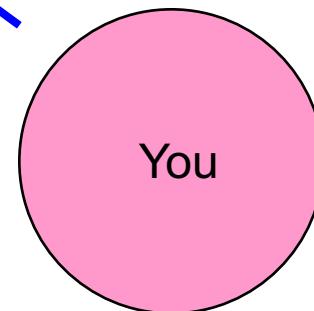
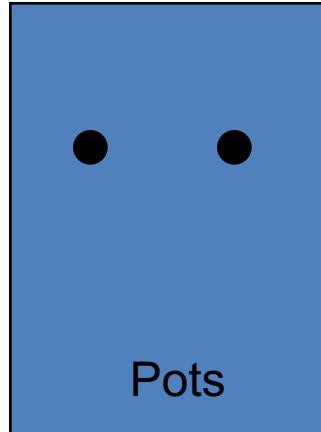
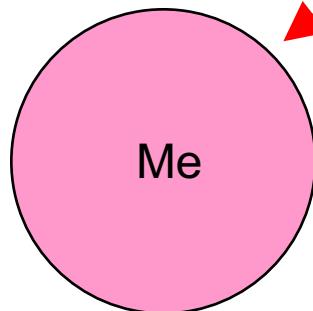
Suppose we allocate, and then everyone requests their max? It's OK; there is a way for me to complete, and then you can complete



Max:
2 pots
1 pan

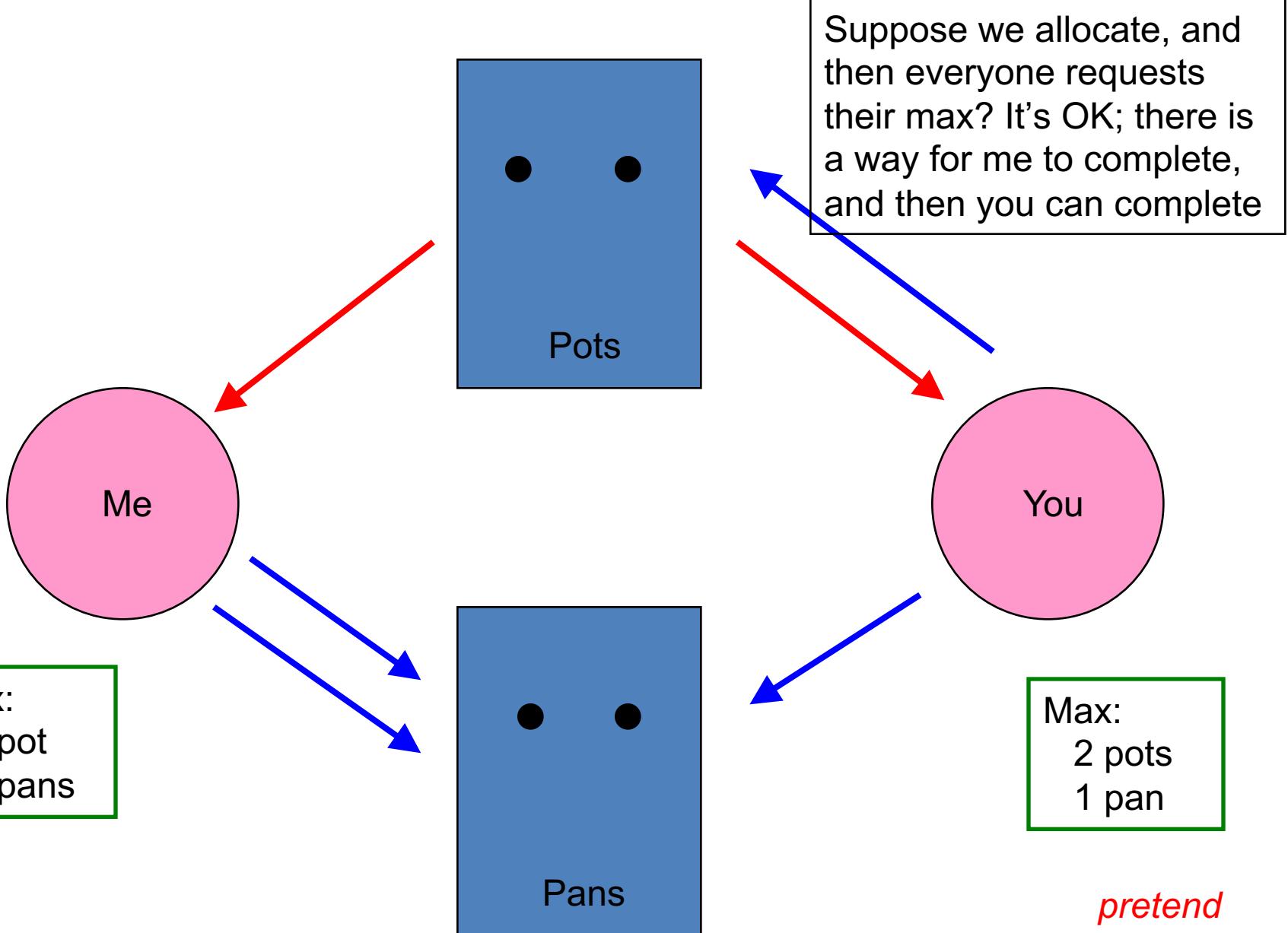
pretend

2. You request a pot

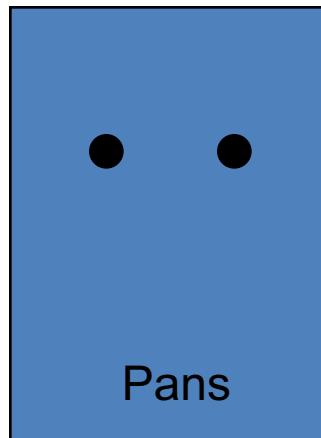
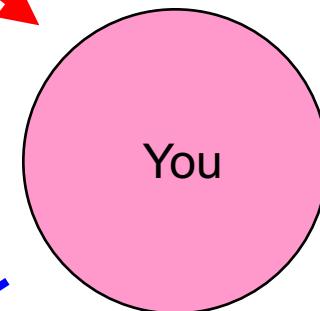
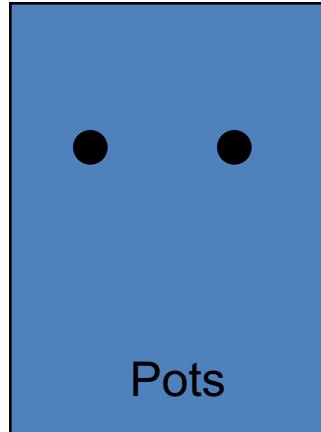
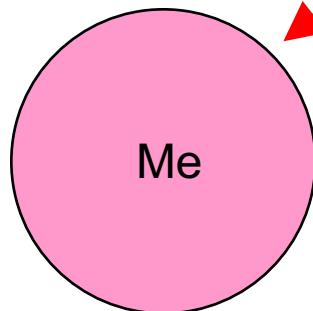


Max:
1 pot
2 pans

Max:
2 pots
1 pan

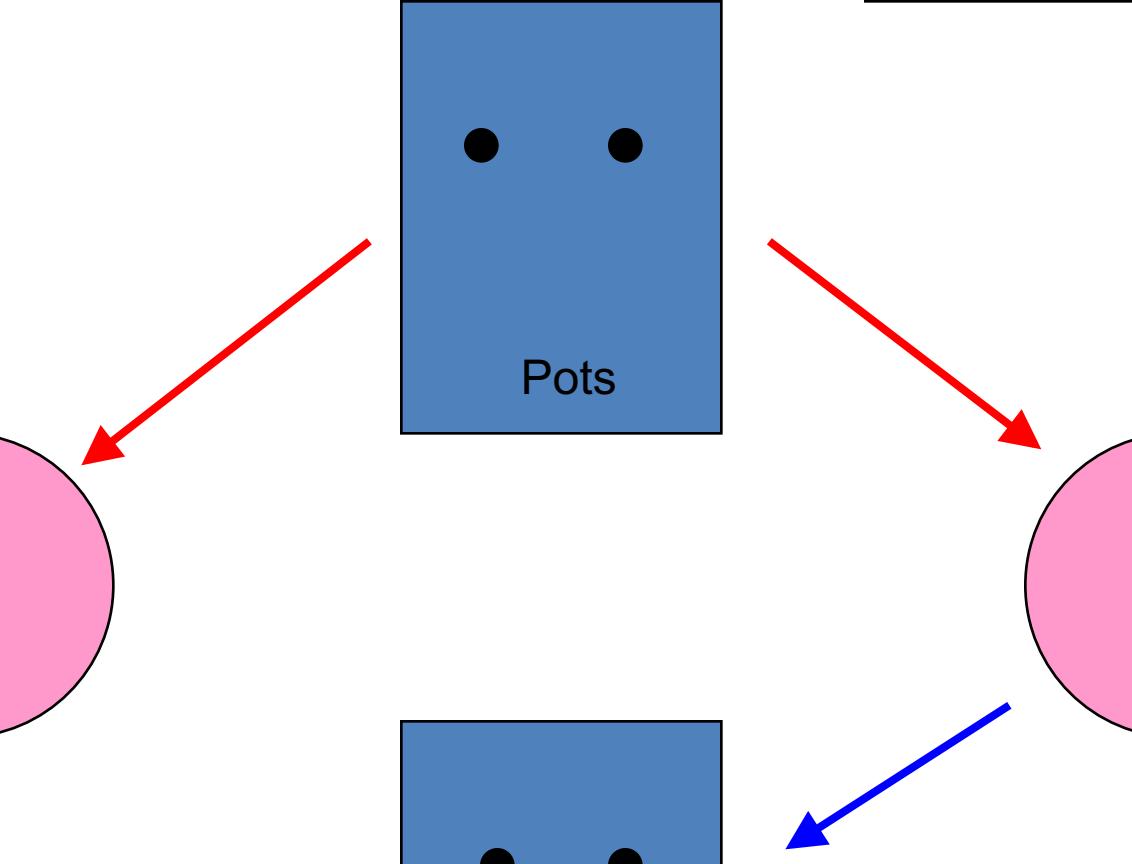


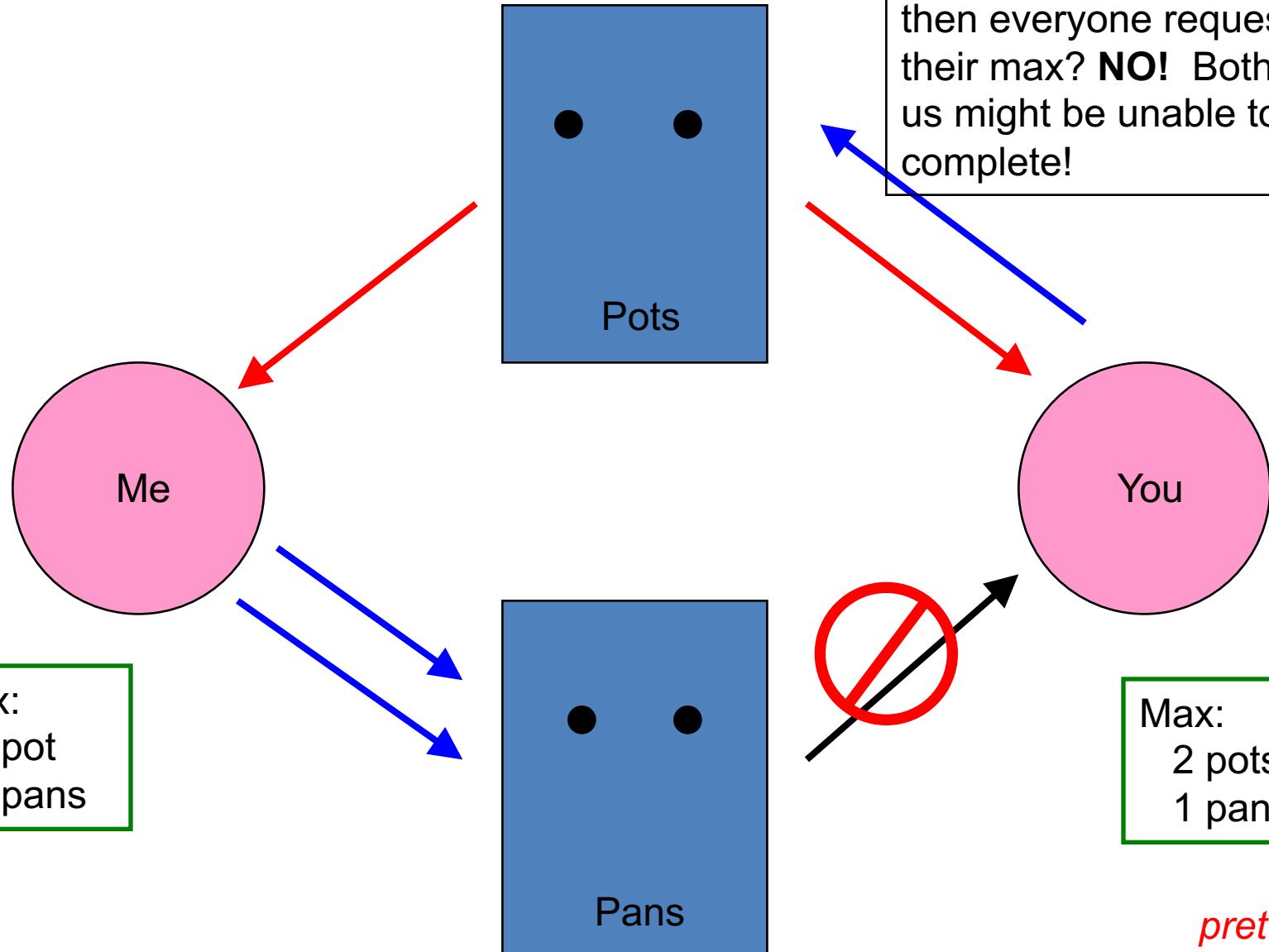
3a. You request a pan



Max:
1 pot
2 pans

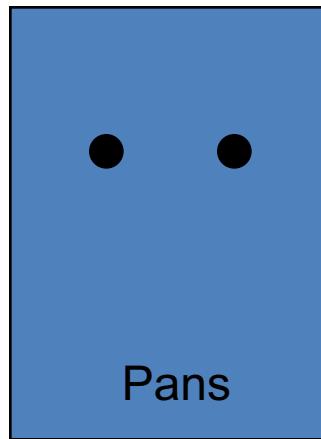
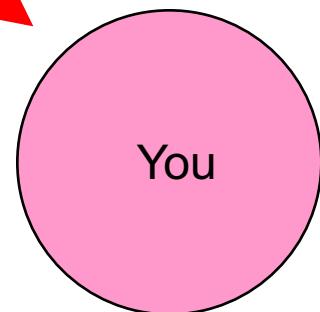
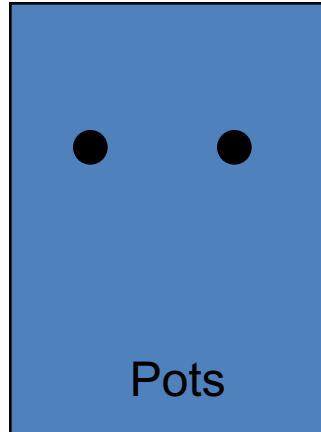
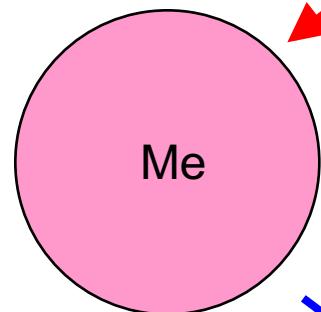
Max:
2 pots
1 pan





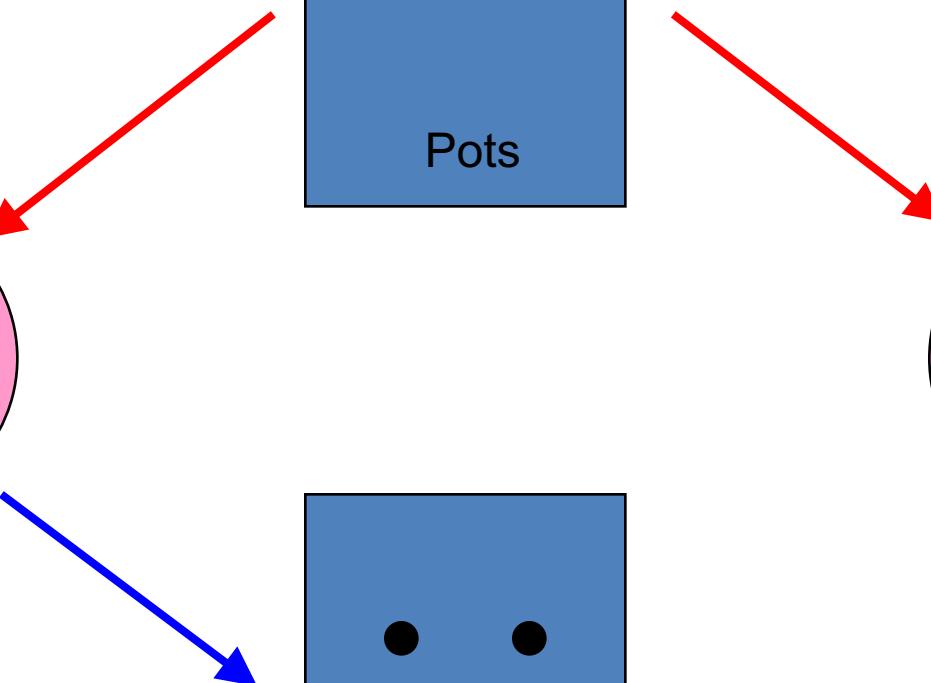
pretend

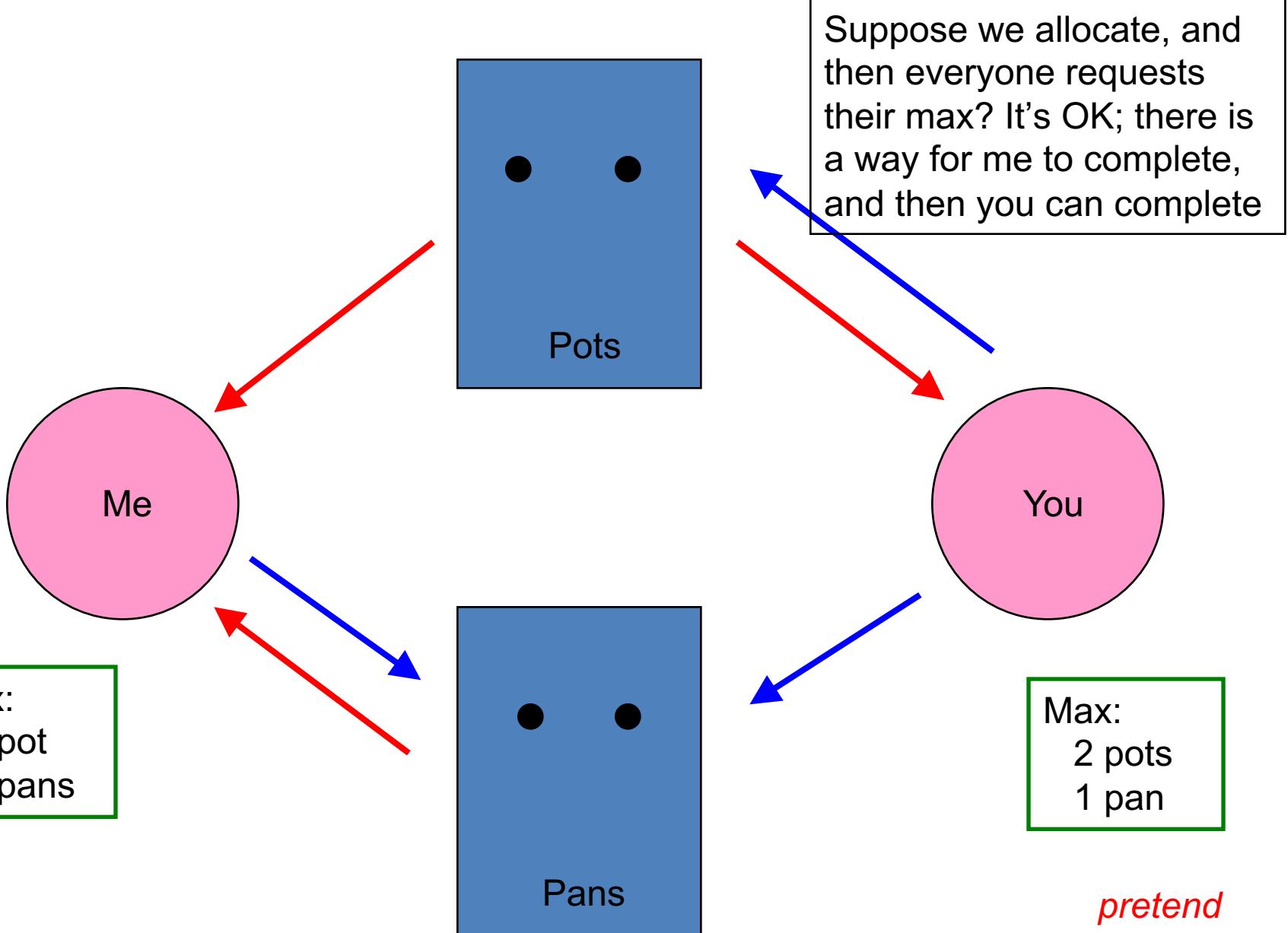
3b. I request a pan



Max:
1 pot
2 pans

Max:
2 pots
1 pan

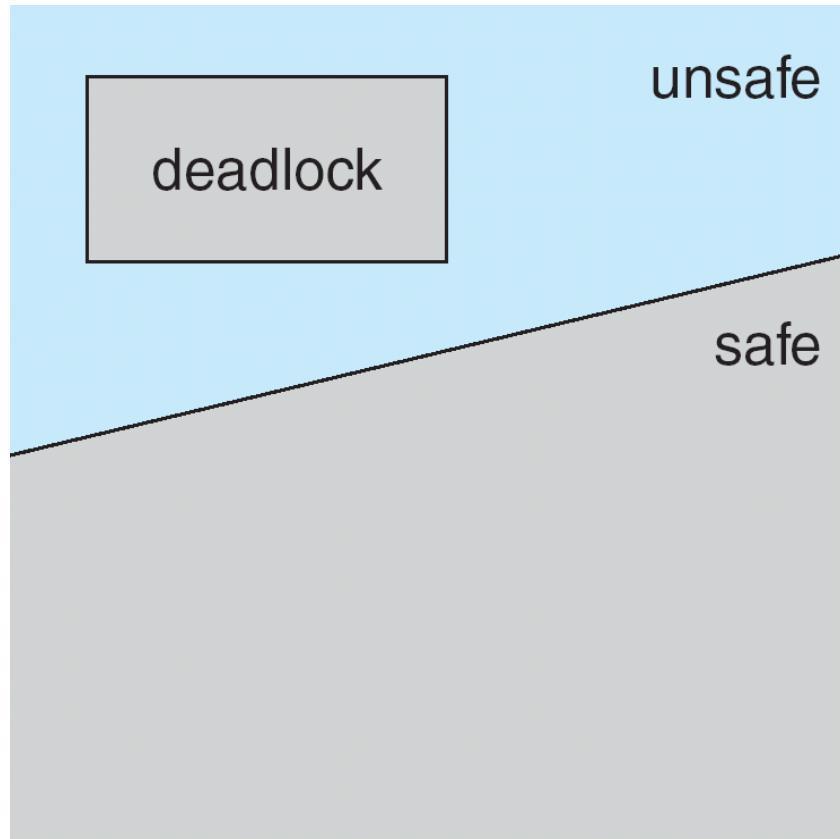




Safe State

- When requesting an available resource decide if allocation leaves the system in a safe state
- In **safe state** if there exists a sequence $\langle P_1, P_2, \dots, P_n \rangle$ of ALL the processes in the systems
 - such that for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$
- That is:
 - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
 - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
 - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Safe, Unsafe, Deadlock State



Data Structures for the Banker' s Algorithm

Let n = number of processes, and m = number of resources types.

- **Available:** Vector of length m . If Available [j] = k , there are k instances of resource type R_j available
- **Max:** $n \times m$ matrix. If Max [i,j] = k , then process P_i may request at most k instances of resource type R_j
- **Allocation:** $n \times m$ matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- **Need:** $n \times m$ matrix. If Need[i,j] = k , then P_i may need k more instances of R_j to complete its task

$$\text{Need } [i,j] = \text{Max}[i,j] - \text{Allocation } [i,j]$$

Safety Algorithm

1. Let **Work** and **Finish** be vectors of length m and n , respectively. Initialize:

Work = Available

Finish [i] = false for $i = 0, 1, \dots, n-1$

2. Find an i such that both:

(a) **Finish [i] = false**

(b) **Need_i ≤ Work**

If no such i exists, go to step 4

3. **Work = Work + Allocation_i**,

Finish[i] = true

go to step 2

4. If **Finish [i] == true** for all i , then the system is in a safe state

Resource-Request Algorithm for Process P_i

Request_i = request vector for process P_i . If $\text{Request}_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $\text{Request}_i \leq \text{Need}_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $\text{Request}_i \leq \text{Available}$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$\text{Available} = \text{Available} - \text{Request}_i;$$

$$\text{Allocation}_i = \text{Allocation}_i + \text{Request}_i;$$

$$\text{Need}_i = \text{Need}_i - \text{Request}_i;$$

- If safe \Rightarrow the resources are allocated to P_i ,
- If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored

Example of Banker' s Algorithm

- 5 processes P_0 through P_4 ;
3 resource types:
 A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time T_0 :

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Example (Cont.)

- The content of the matrix **Need** is defined to be **Max – Allocation**

	<u>Need</u>		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

- The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request \leq Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

- Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

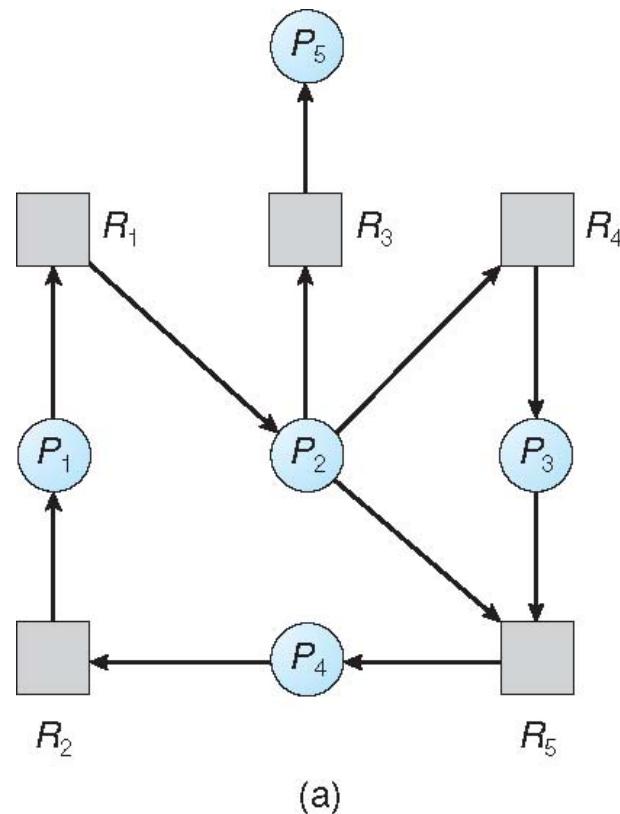
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

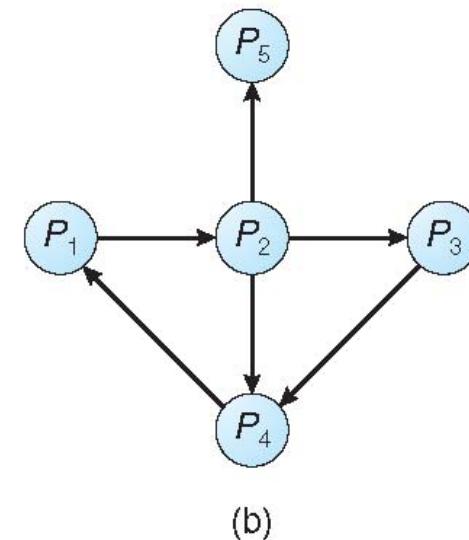
Single Instance of Each Resource Type

- Maintain **wait-for** graph
 - Nodes are processes
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph.
 - If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph
 - requires an order of n^2 operations,
 - where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
 - How often a deadlock is likely to occur?
 - How many processes will need to be rolled back?
 - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily,
 - there may be many cycles in the resource graph
 - we would not be able to tell which deadlocked processes “caused” the deadlock.

Recovery from Deadlock:

- Process Termination
 - Abort all deadlocked processes
 - Abort one process at a time until the deadlock cycle is eliminated
 - In which order should we choose to abort?
- Resource Preemption
 - **Selecting a victim** – minimize cost
 - **Rollback** – return to some safe state, restart process for that state
 - **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

Summary

- Deadlock is bad!
- We can deal with it either statically (prevention) or dynamically (avoidance and/or detection)
- In practice, you'll encounter lock ordering, periodic deadlock detection/correction, and minefields

Operating Systems

Scheduling

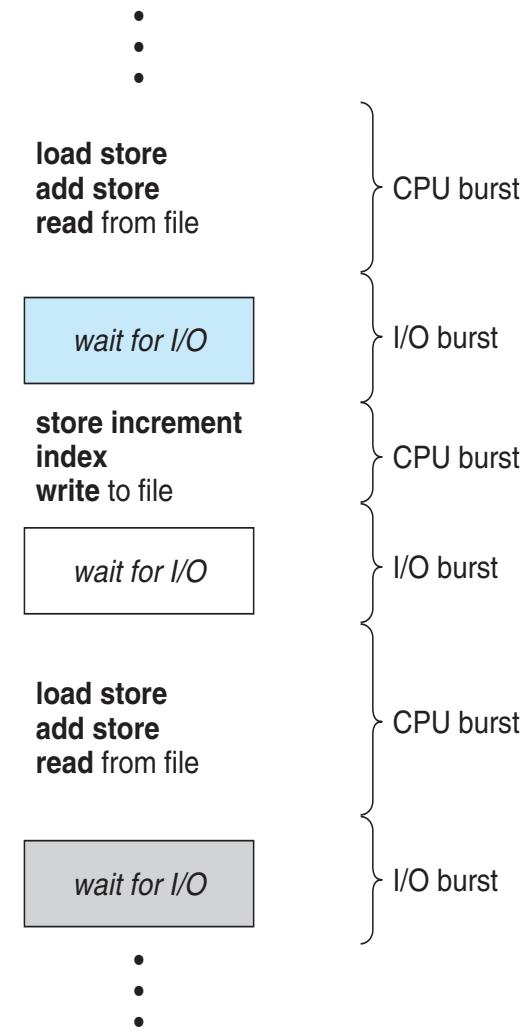
Lecture 8
Michael O'Boyle

Scheduling

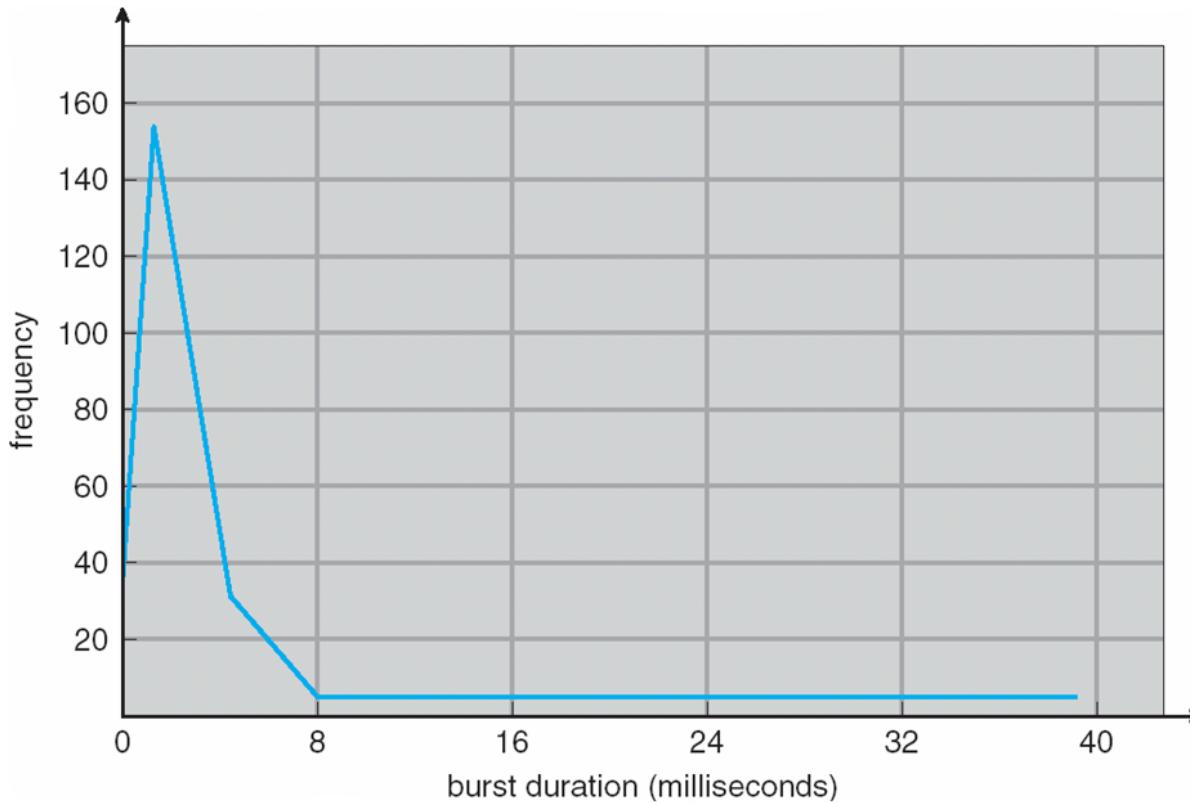
- We have talked about **context switching**
 - an interrupt occurs (device completion, timer interrupt)
 - a thread causes a trap or exception
 - may need to choose a different thread/process to run
- Glossed over which process or thread to run next
 - “some thread from the ready queue”
- This decision is called **scheduling**
 - scheduling is a **policy**
 - context switching is a **mechanism**

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



Histogram of CPU-burst Times



Exploit this : let another job use CPU

Classes of Schedulers

- Batch
 - Throughput / utilization oriented
 - Example: audit inter-bank funds transfers each night, Pixar rendering, Hadoop/MapReduce jobs
- Interactive
 - Response time oriented
- Real time
 - Deadline driven
 - Example: embedded systems (cars, airplanes, etc.)
- Parallel
 - Speedup-driven
 - Example: “space-shared” use of a 1000-processor machine for large simulations

We'll be talking primarily about interactive schedulers

Multiple levels of scheduling decisions

- Long term
 - Should a new “job” be “initiated,” or should it be held?
 - typical of batch systems
- Medium term
 - Should a running program be temporarily marked as non-runnable (e.g., swapped out)?
- Short term
 - Which thread should be given the CPU next? For how long?
 - Which I/O operation should be sent to the disk next?
 - On a multiprocessor:
 - should we attempt to coordinate the running of threads from the same address space in some way?
 - should we worry about cache state (processor affinity)?

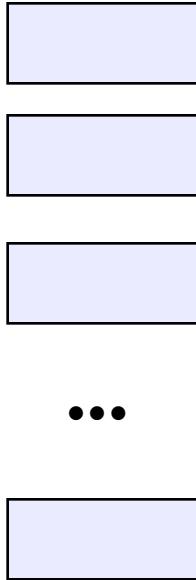
Scheduling Goals I: Performance

- Many possible metrics / performance goals (which sometimes conflict)
 - maximize **CPU utilization**
 - maximize **throughput** (requests completed / s)
 - minimize **average response time** (average time from submission of request to completion of response)
 - minimize **average waiting time** (average time from submission of request to start of execution)
 - minimize **energy** (joules per instruction) subject to some constraint (e.g., frames/second)

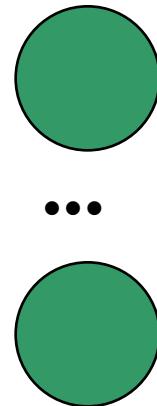
Scheduling Goals II: Fairness

- No single, compelling definition of “fair”
 - How to measure fairness?
 - Equal CPU consumption? (over what time scale?)
 - Fair per-user? per-process? per-thread?
 - What if one process is CPU bound and one is I/O bound?
- Sometimes the goal is to be unfair:
 - Explicitly favor some particular class of requests (priority system), but...
 - avoid starvation (be sure everyone gets at least some service)

The basic situation



Schedulable units



Resources

Scheduling:

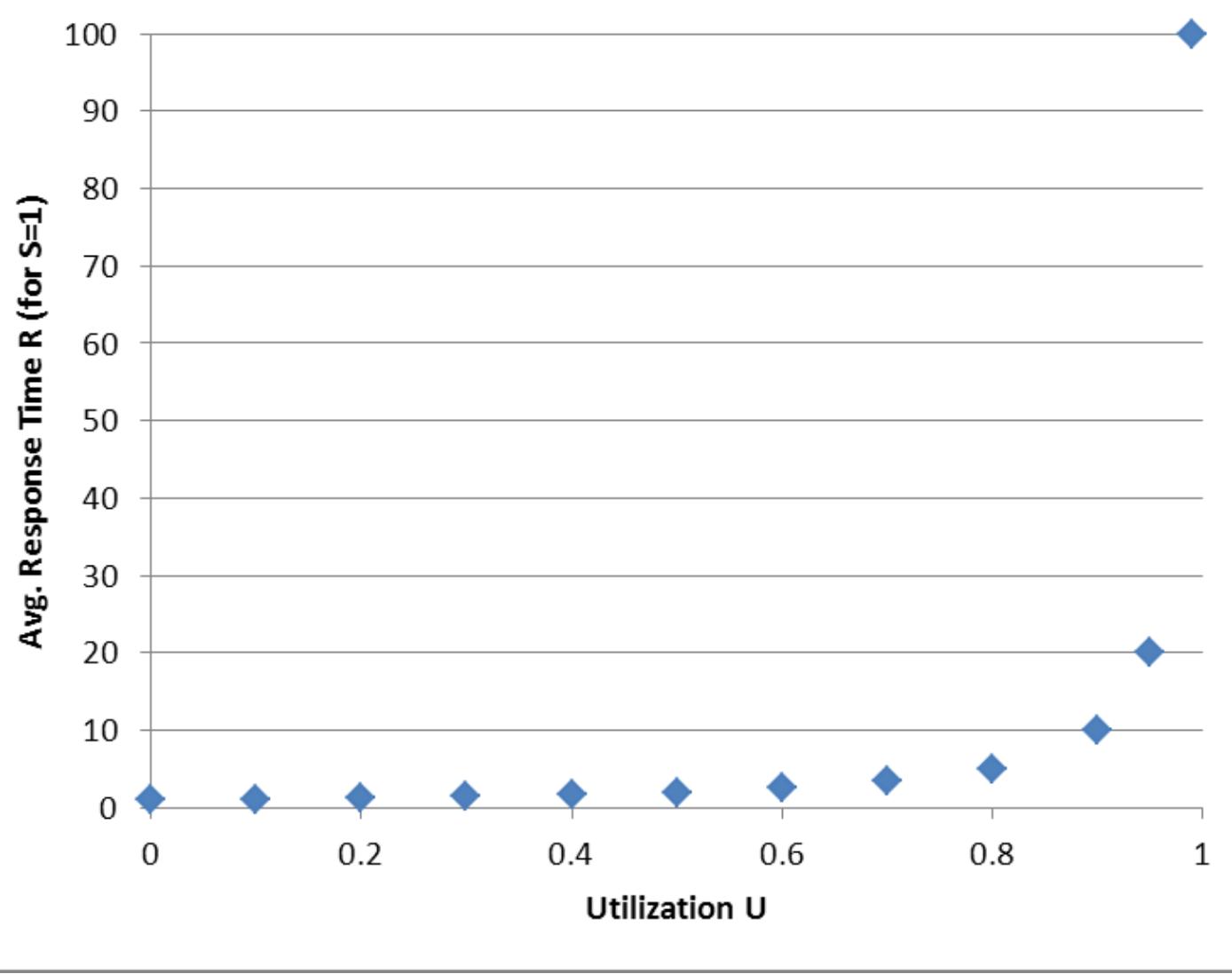
- Who to assign each resource to
- When to re-evaluate your decisions

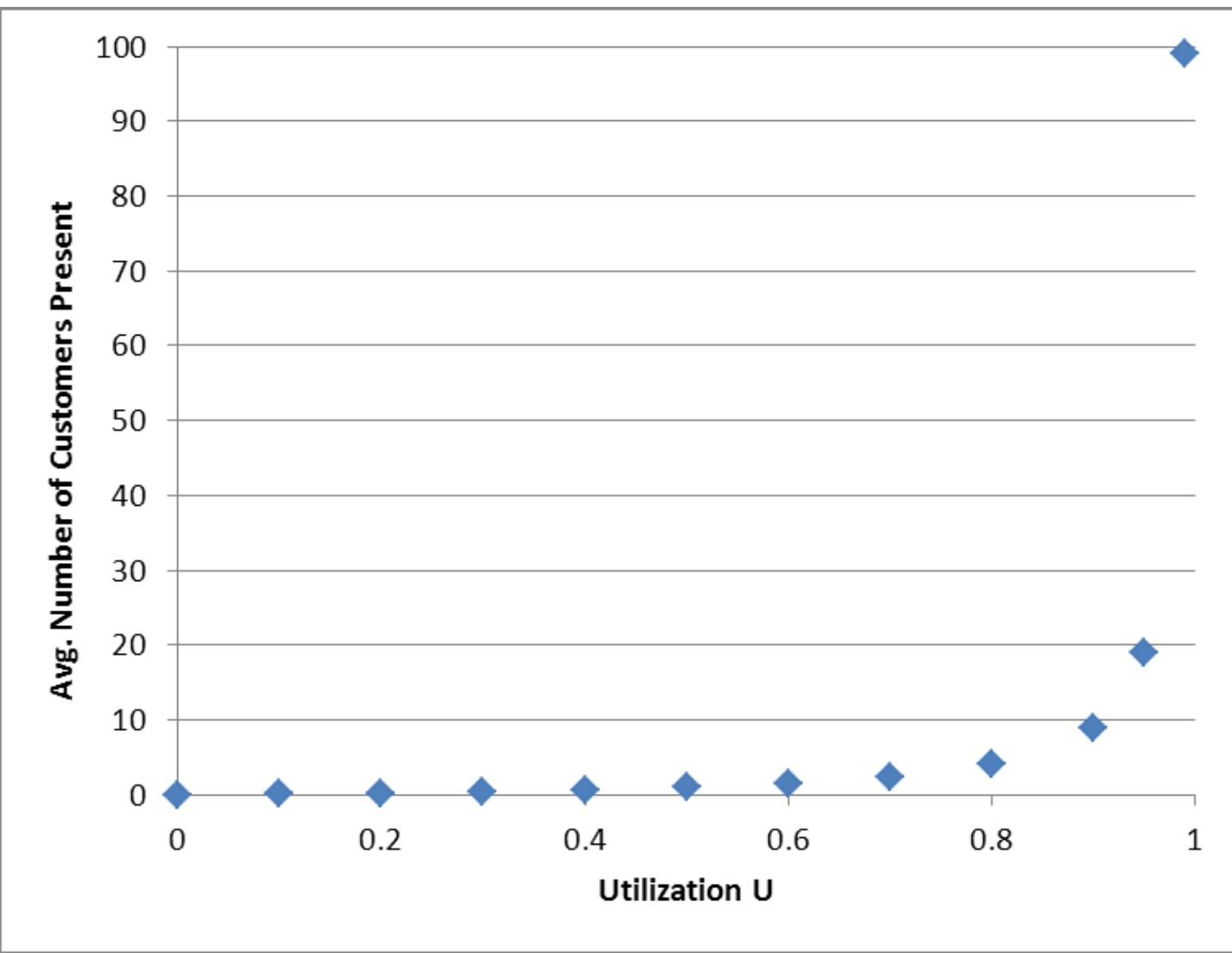
When to assign?

- Pre-emptive vs. non-preemptive schedulers
 - Non-preemptive
 - once you give somebody the green light, they've got it until they relinquish it
 - an I/O operation
 - allocation of memory in a system without swapping
 - Preemptive
 - you can re-visit a decision
 - setting the timer allows you to preempt the CPU from a thread even if it doesn't relinquish it voluntarily
 - Re-assignment always involves some overhead
 - Overhead doesn't contribute to the goal of any scheduler
- We'll assume "work conserving" policies
 - Never leave a resource idle when someone wants it
 - Why even mention this? When might it be useful to do something else?

Laws and Properties

- The Utilization Law: $U = X * S$
 - U is utilization,
 - X is throughput (requests per second)
 - S is average service time
 - This means that utilization is constant, independent of the schedule, so long as the workload can be processed
- Little's Law: $N = X * R$
 - Where N is average number in system, X is throughput, and R is average response time (average time in system)
 - This means that better average response time implies fewer in system, and vice versa
- Response Time R at a single server under FCFS scheduling:
 - $R = S / (1-U)$ and
 - $N = U / (1-U)$





Algorithm #1: FCFS/FIFO

- First-come first-served / First-in first-out (**FCFS/FIFO**)
 - schedule in the order that they arrive
 - “real-world” scheduling of people in (single) lines
 - supermarkets
 - jobs treated equally, no starvation
 - In what sense is this “fair”?
- Sounds perfect!
 - in the real world, does FCFS/FIFO work well?

First- Come, First-Served (FCFS) Scheduling

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

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



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- **Convoy effect** - short process behind long process
 - Consider one CPU-bound and many I/O-bound processes

FCFS/FIFO drawbacks

- Average response time can be poor: small requests wait behind big ones
- May lead to poor utilization of other resources
 - if you send me on my way, I can go keep another resource busy
 - FCFS may result in poor overlap of CPU and I/O activity
 - E.g., a CPU-intensive job prevents an I/O-intensive job from a small bit of computation, preventing it from going back and keeping the I/O subsystem busy
- The more copies of the resource there are to be scheduled
 - the less dramatic the impact of occasional very large jobs (so long as there is a single waiting line)
 - E.g., many cores vs. one core

Algorithm #2: Shortest-Job-First (SJF) Scheduling

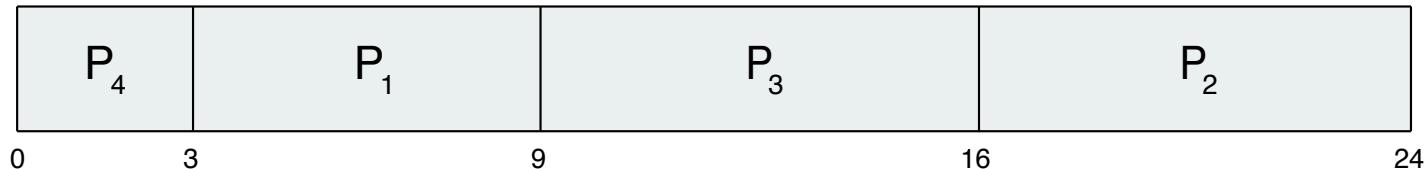
- Associate with each process the length of its next CPU burst
 - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
 - The difficulty is knowing the length of the next CPU request
 - Could ask the user

Example of SJF

<u>Process</u>	<u>Burst Time</u>
P_1	6
P_2	8
P_3	7
P_4	3

Algorithm #2:

- SJF scheduling chart

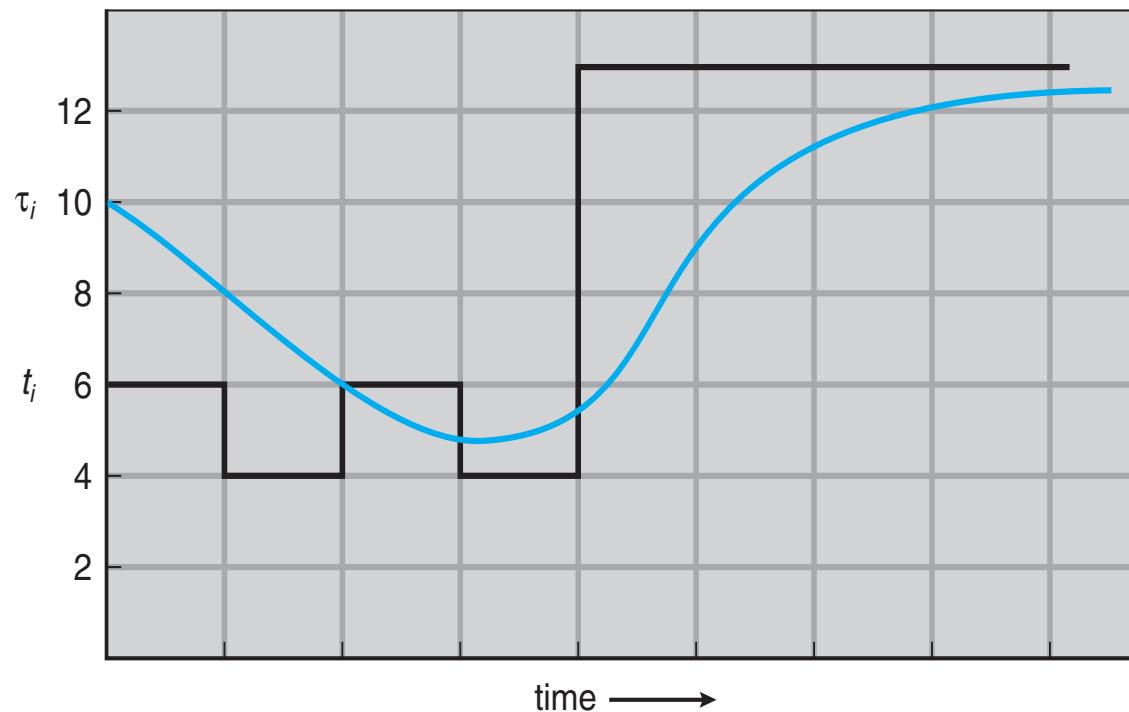


- Average waiting time = $(3 + 16 + 9 + 0) / 4 = 7$

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
 - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
 1. t_n = actual length of n^{th} CPU burst
 2. τ_{n+1} = predicted value for the next CPU burst
 3. $\alpha, 0 \leq \alpha \leq 1$
 4. Define : $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$.
- Commonly, α set to $\frac{1}{2}$
- Preemptive version called **shortest-remaining-time-first**

Prediction of the Length of the Next CPU Burst



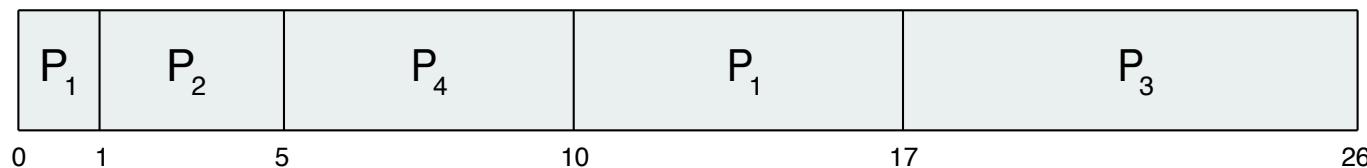
CPU burst (t_i)	6	4	6	4	13	13	13	...	
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Preemptive SJF Gantt Chart*



- Average waiting time = $[(10-1)+(1-1)+(17-2)+5-3]/4 = 26/4 = 6.5$ msec

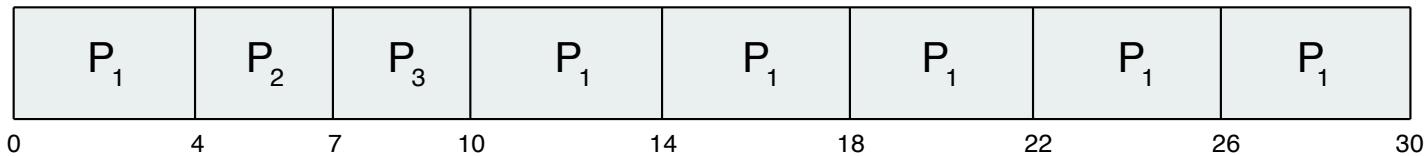
Algorithm #3: Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds.
 - After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q ,
 - then each process gets $1/n$ of the CPU time in chunks of at most q time units at once.
 - No process waits more than $(n-1)q$ time units.
- Timer interrupts every quantum to schedule next process
- Performance
 - q large \Rightarrow FIFO
 - q small \Rightarrow q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

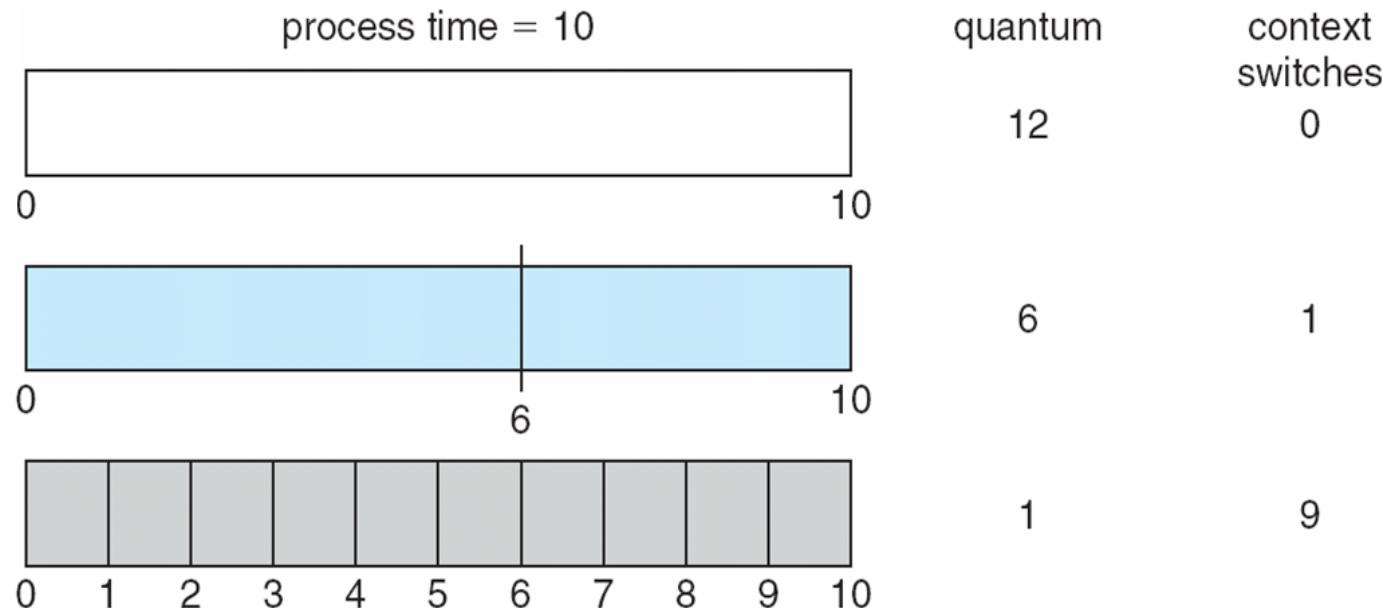
<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

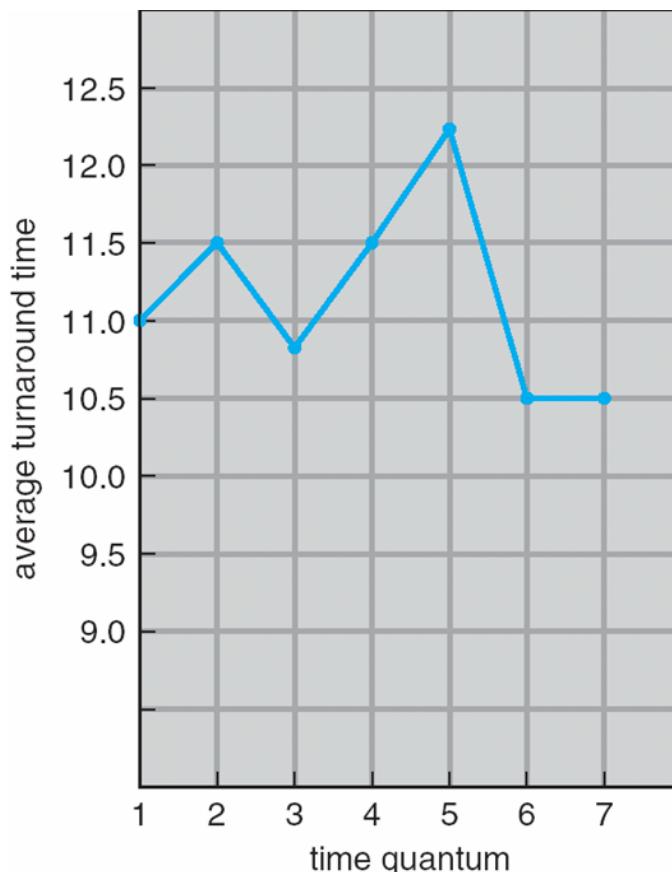


- Typically, higher average turnaround than SJF,
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts
should be shorter than q

RR drawbacks

- What if all jobs are exactly the same length?
 - What would the pessimal schedule be (with average response time as the measure)?
- What do you set the quantum to be?
 - no value is “correct”
 - if small, then context switch often, incurring high overhead
 - if large, then response time degrades
- Treats all jobs equally
 - What about CPU vs I/O bound?

Algorithm #4: Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority)
 - Preemptive
 - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem = **Starvation** – low priority processes may never execute
- Solution = **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



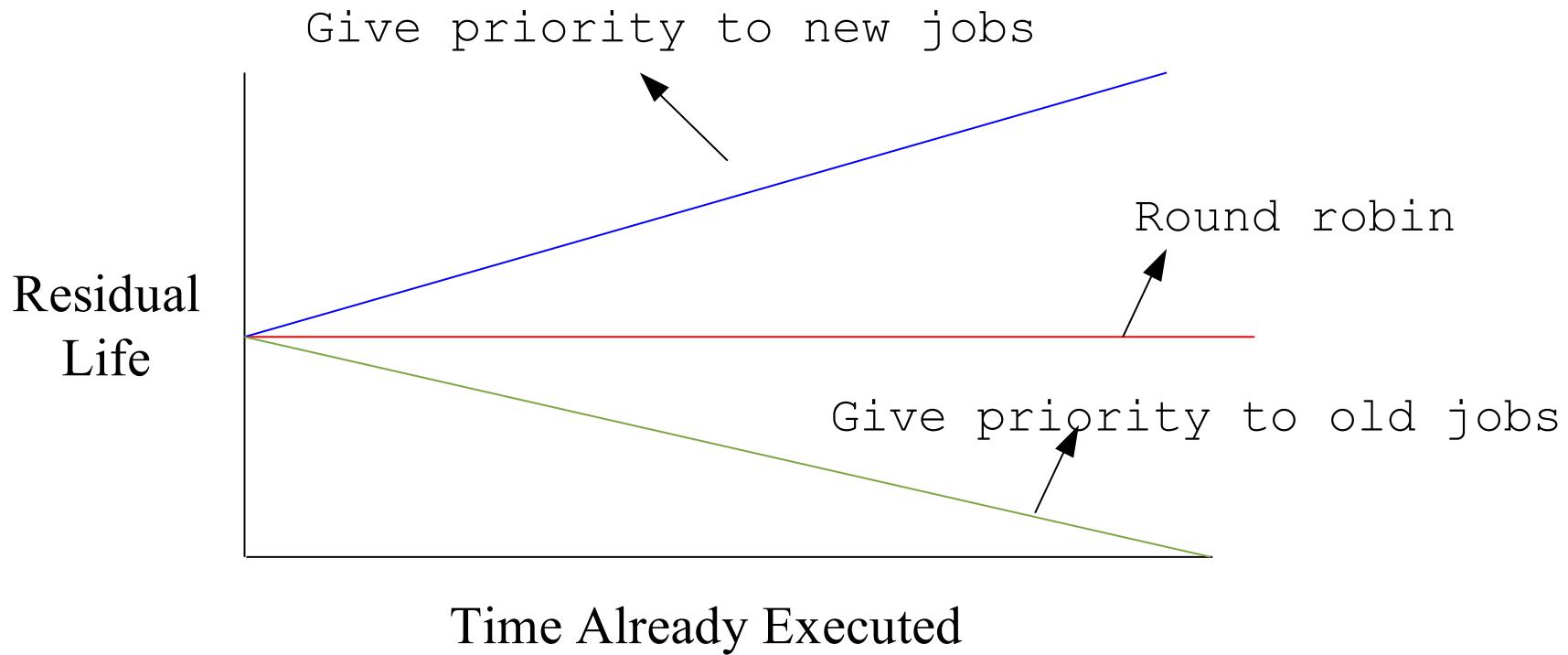
- Average waiting time = 8.2 msec
- Error in Gantt: P2, P5, P1, P3, P4

Program behavior and scheduling

- An analogy:
 - Say you're at the airport waiting for a flight
 - There are two identical ATMs:
 - ATM 1 has 3 people in line
 - ATM 2 has 6 people in line
 - You get into the line for ATM 1
 - ATM 2's line shrinks to 4 people
 - Why might you now switch lines, preferring 5th in line for ATM 2 over 4th in line for ATM 1?

Residual Life

- Given that a job has already executed for X seconds, how much longer will it execute, on average, before completing?



Multi-level Feedback Queues (MLFQ)

- It's been observed that workloads tend to have increasing residual life – “if you don’t finish quickly, you’re probably a lifer”
- This is exploited in practice by using a policy that discriminates against the old
- **MLFQ:**
 - there is a hierarchy of queues
 - there is a priority ordering among the queues
 - new requests enter the highest priority queue
 - each queue is scheduled RR
 - requests move between queues based on execution history

UNIX scheduling

- Canonical scheduler is pretty much MLFQ
 - 3-4 classes spanning ~170 priority levels
 - timesharing: lowest 60 priorities
 - system: middle 40 priorities
 - real-time: highest 60 priorities
 - priority scheduling across queues, RR within
 - process with highest priority always run first
 - processes with same priority scheduled RR
 - processes dynamically change priority
 - increases over time if process blocks before end of quantum
 - decreases if process uses entire quantum
- Goals:
 - reward interactive behavior over CPU hogs
 - interactive jobs typically have short bursts of CPU

Summary

- Scheduling takes place at many levels
- It can make a huge difference in performance
 - this difference increases with the variability in service requirements
- Multiple goals, sometimes conflicting
- There are many “pure” algorithms, most with some drawbacks in practice – FCFS, SPT, RR, Priority
- Real systems use hybrids that exploit observed program behavior
- Scheduling is important
 - Look at multicore/GPU systems in later research lecture

Operating Systems

Memory Management

Lecture 9
Michael O'Boyle

Memory Management

- Background
- Logical/Virtual Address Space vs Physical Address Space
- Swapping
- Contiguous Memory Allocation
- Segmentation

Goals and Tools of memory management

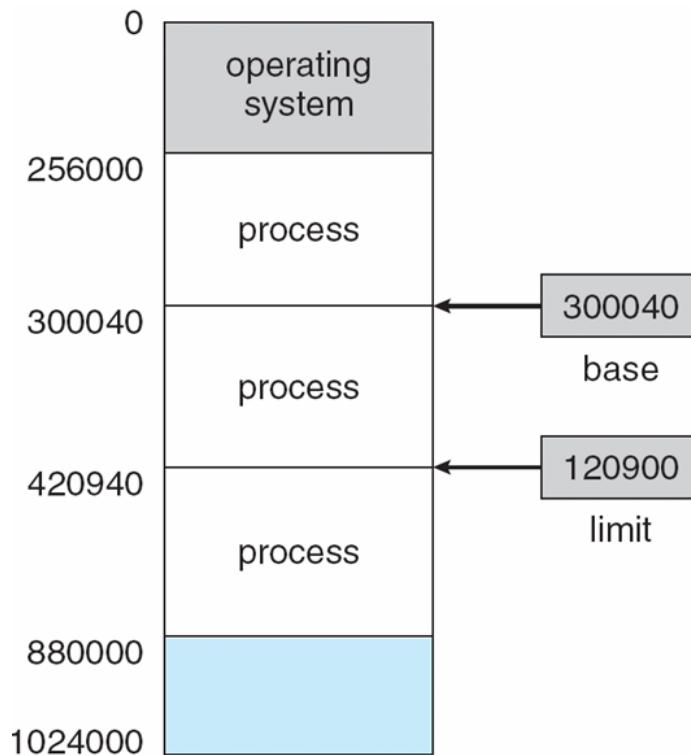
- Allocate memory resources among competing processes,
 - maximizing memory utilization and system throughput
- Provide isolation between processes
 - Addressability and protection: orthogonal
- Convenient abstraction for programming
 - and compilers, etc.
- Tools
 - Base and limit registers
 - Swapping
 - Segmentation
 - Paging, page tables and TLB (Next time)
 - Virtual memory: (Next next time)

Background

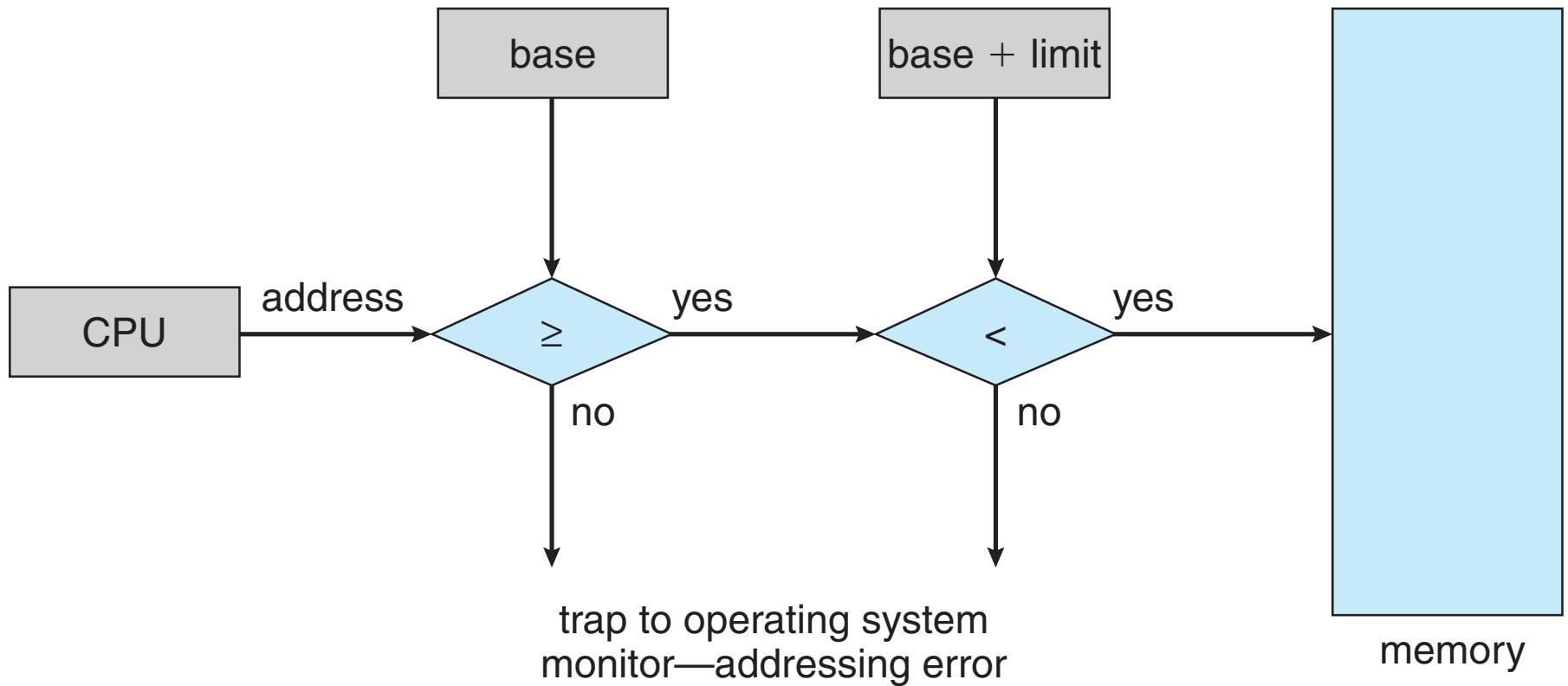
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Base and Limit Registers

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



Hardware Address Protection



Virtual addresses for multiprogramming

- To make it easier to manage memory of multiple processes, make processes **use logical or virtual addresses**
 - Logical/virtual addresses are independent of location in physical memory data lives
 - OS determines location in physical memory
- Instructions issued by CPU reference logical/virtual addresses
 - e.g., pointers, arguments to load/store instructions, PC ...
- Logical/virtual addresses are translated by hardware into physical addresses (with some setup from OS)

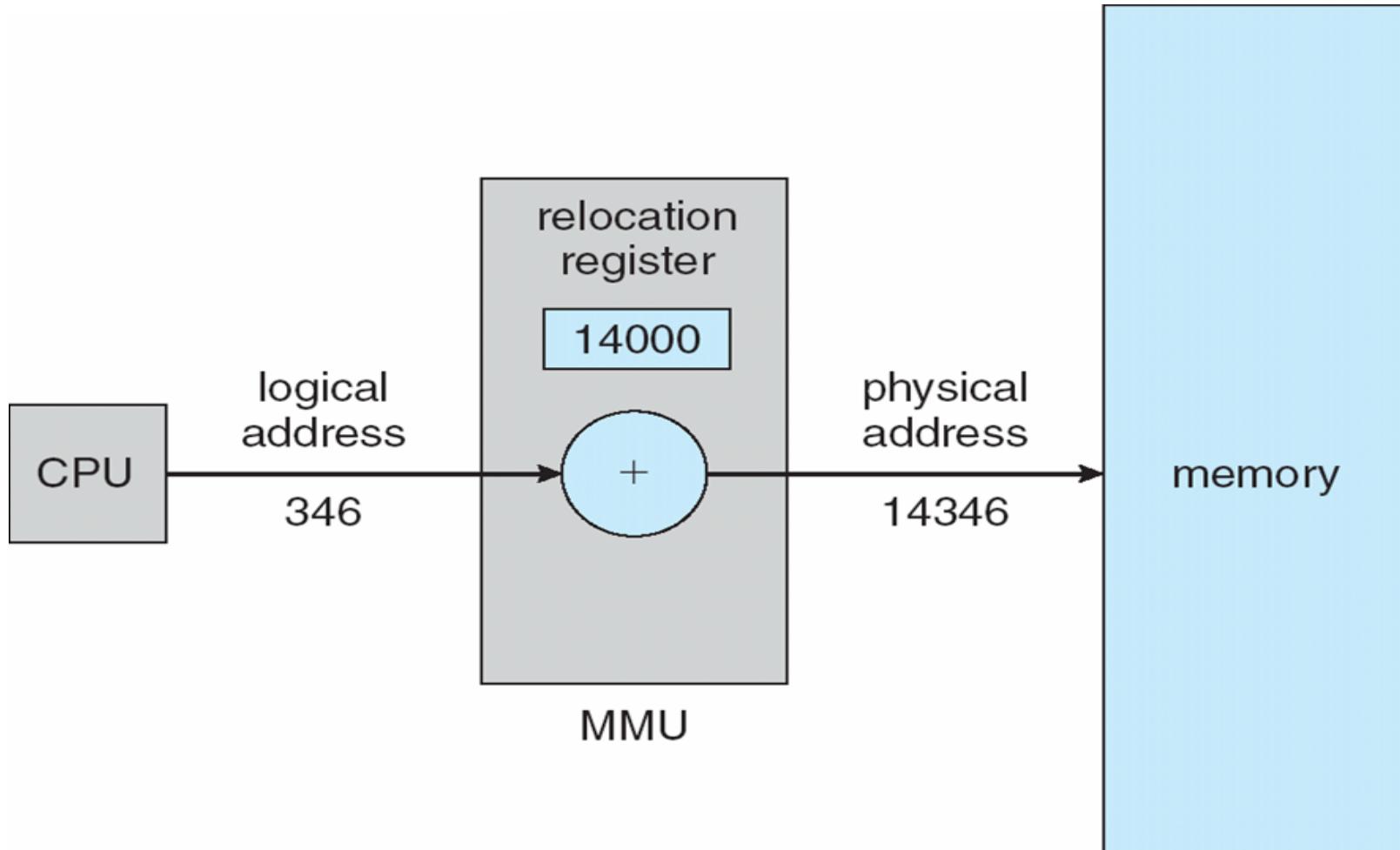
Logical/Virtual Address Space

- The set of logical/virtual addresses a process can reference is its **address space**
 - many different possible mechanisms for translating logical/virtual addresses to physical addresses
- Program issues addresses in a logical/virtual address space
 - must be **translated** to physical address space
 - Think of the program as having a contiguous logical/virtual address space that starts at 0,
 - and a contiguous physical address space that starts somewhere else
- **Logical/virtual address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

- Hardware device
 - at run time maps virtual to physical address
- Many methods possible
- Simple scheme: value in the relocation register is added to every address generated by a user process at the time it is sent to memory
 - Base register now called **relocation register**
 - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
 - Execution-time binding occurs when reference is made to location in memory
 - Logical address bound to physical addresses

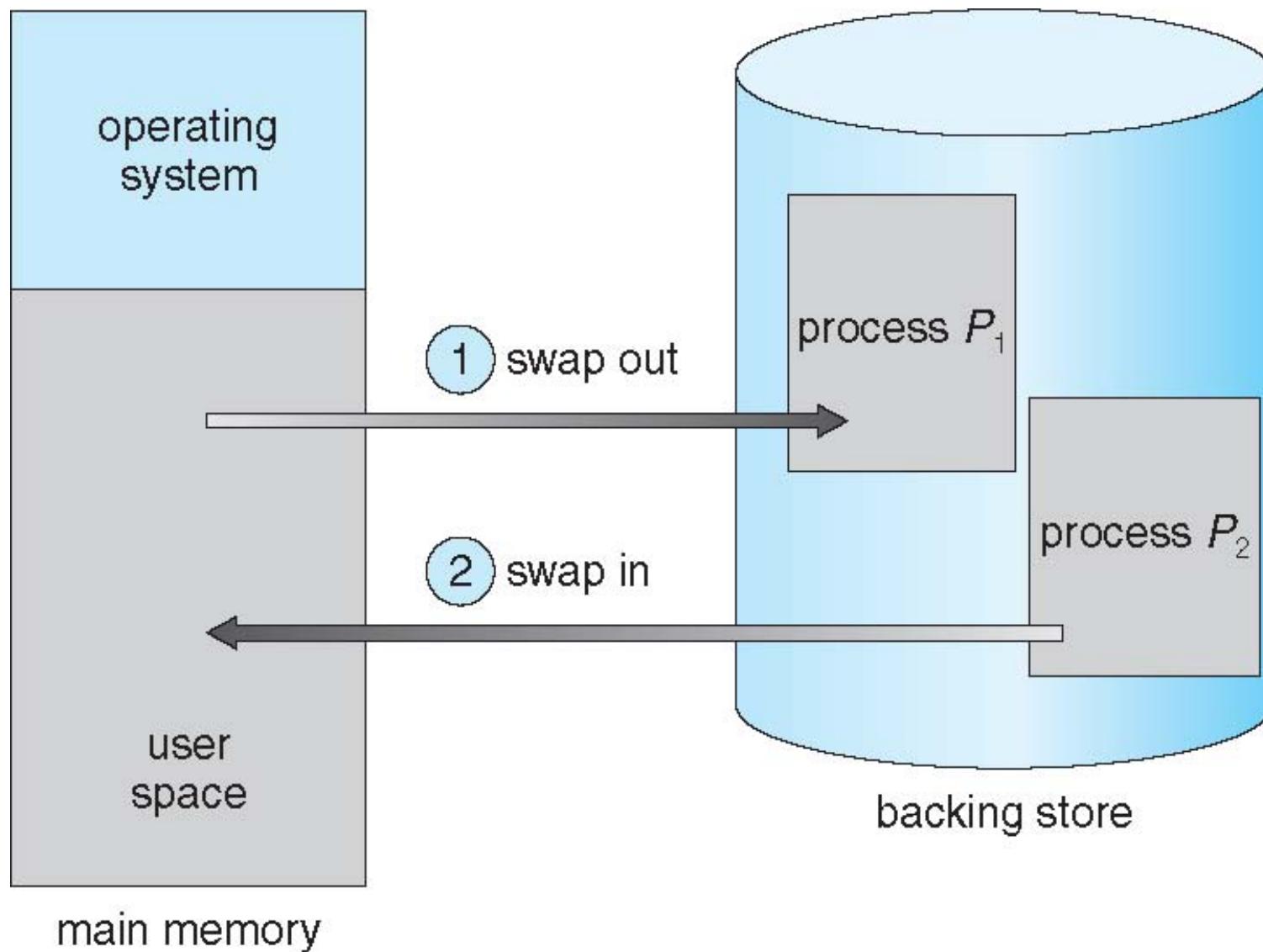
MMU as a relocation register



Swapping

- What if not enough memory to hold all processes?
- A process can be **swapped** temporarily
 - out of memory to a backing store,
 - brought back into memory for continued execution
 - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk
 - large enough to accommodate copies of all memory images for all users;
 - must provide direct access to these memory images
- **Roll out, roll in** – swapping variant
 - used for priority-based scheduling algorithms;
 - lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time;
 - total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue**
 - ready-to-run processes which have memory images on disk

Schematic View of Swapping

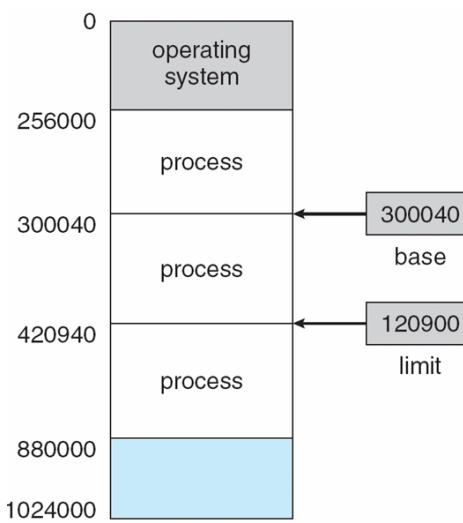


Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory,
 - need to swap out a process and swap in target process
- Context switch time can then be very high
- Can reduce cost
 - reduce size of – by knowing how much memory really being used
 - inform OS of memory use via `request_memory()` and `release_memory()`
- Other constraints as well on swapping
 - Pending I/O – can't swap out as I/O would occur to wrong process
- Or always transfer I/O to kernel space, then to I/O device
 - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
 - But modified version common
 - Swap only when free memory extremely low

Contiguous Allocation

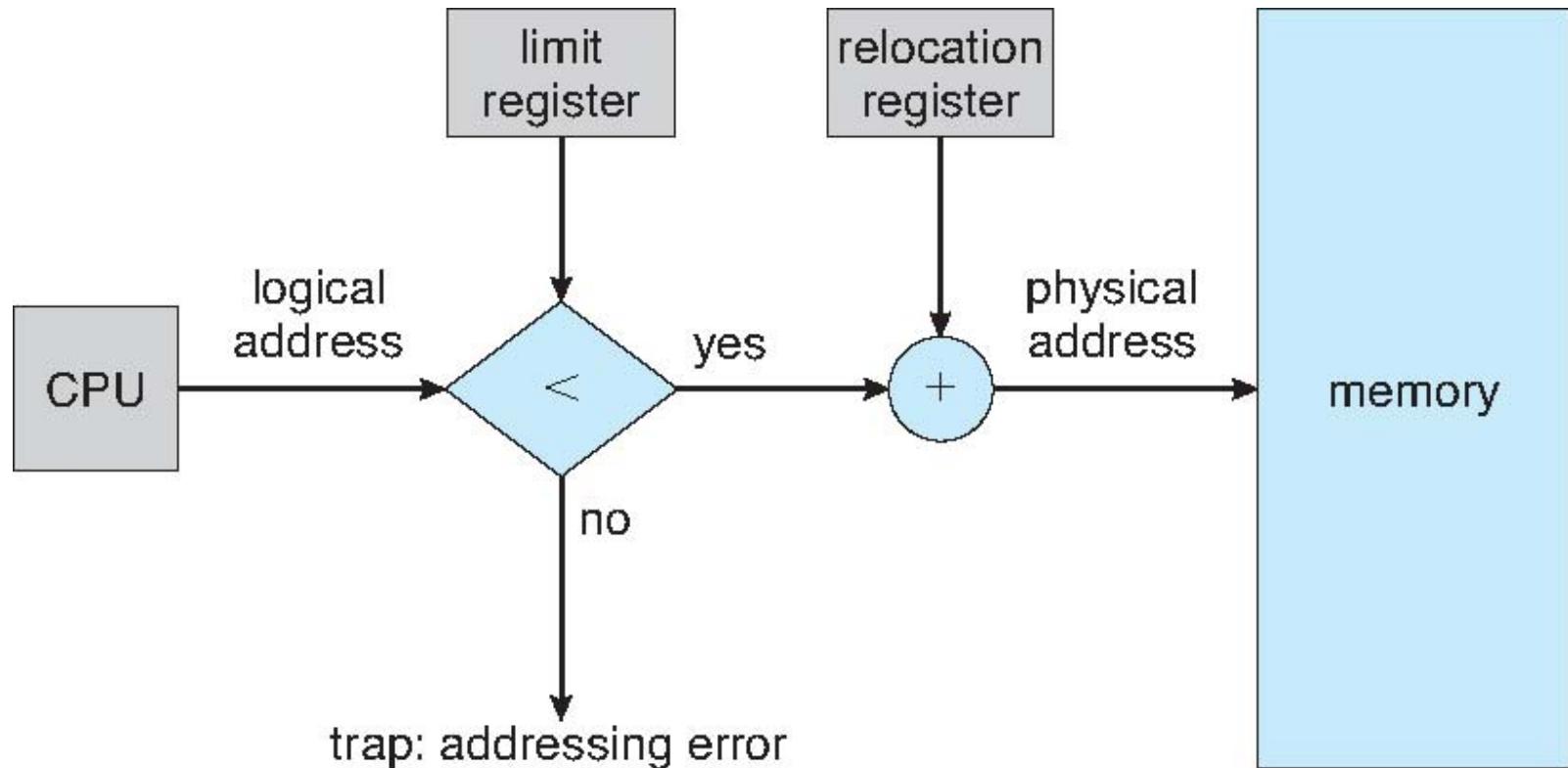
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
 - Resident operating system, usually held in low memory with interrupt vector
 - User processes then held in high memory
 - Each process contained in single contiguous section of memory



Contiguous Allocation

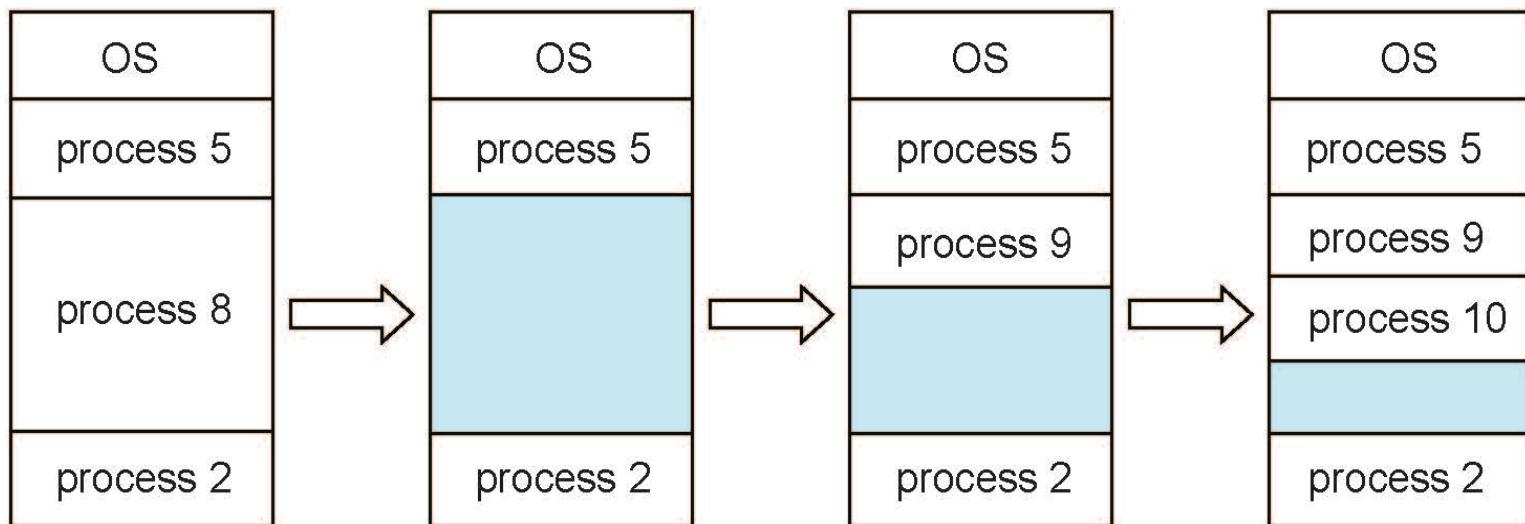
- Relocation registers
 - used to protect user processes from each other, and from changing operating-system code and data
 - Base register contains value of smallest physical address
 - Limit register contains range of logical addresses – each logical address must be less than the limit register
- MMU maps logical address *dynamically*
 - Can then allow actions such as kernel code being **transient** and kernel changing size

Hardware Support for Relocation and Limit Registers



Multiple-partition allocation

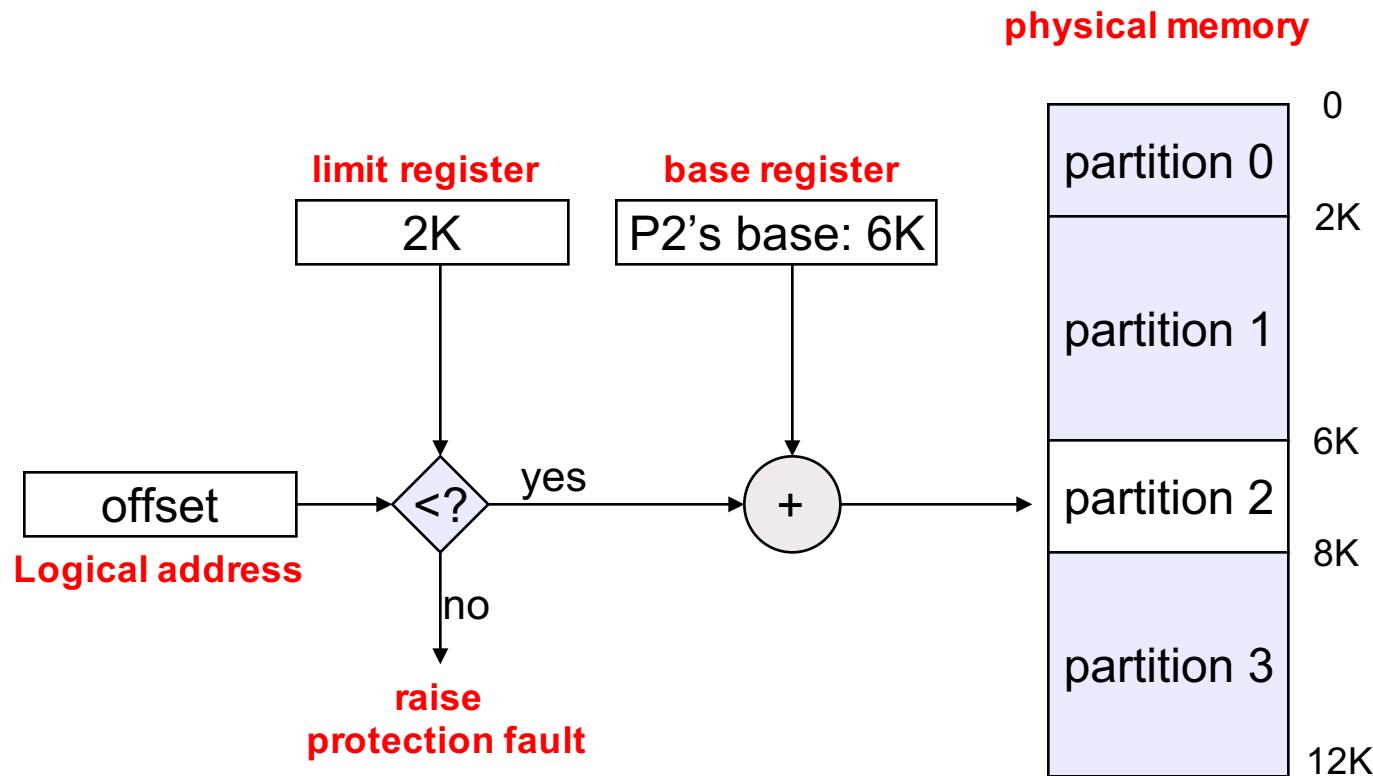
- Multiple-partition allocation
 - Degree of multiprogramming limited by number of partitions
 - Exam 2 approaches
 - Fixed partition
 - Variable partition



Old technique #1: Fixed partitions

- Physical memory is broken up into fixed partitions
 - partitions may have different sizes, but partitioning never changes
 - hardware requirement: **base/relocation register, limit register**
 - physical address = logical address + base register
 - base register loaded by OS when it switches to a process
- Advantages
 - Simple
- Problems
 - **internal fragmentation**: the available partition is larger than what was requested

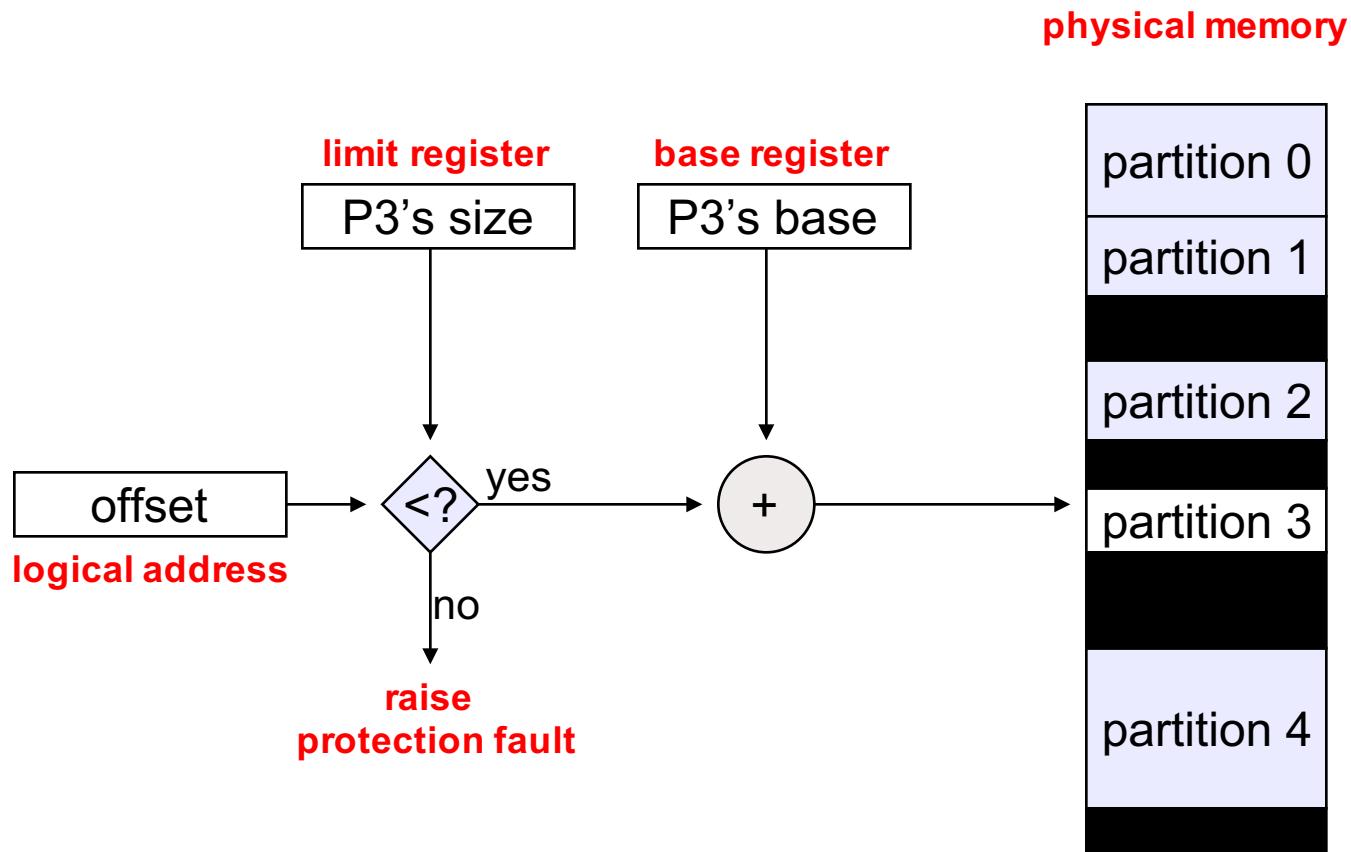
Mechanics of fixed partitions



Old technique #2: Variable partitions

- Obvious next step: physical memory is broken up into partitions dynamically – partitions are tailored to programs
 - hardware requirements: **base register, limit register**
 - physical address = logical address + base register
- Advantages
 - no internal fragmentation
 - simply allocate partition size to be just big enough for process (assuming we know what that is!)
- Problems
 - **external fragmentation**
 - as we load and unload jobs, holes are left scattered throughout physical memory

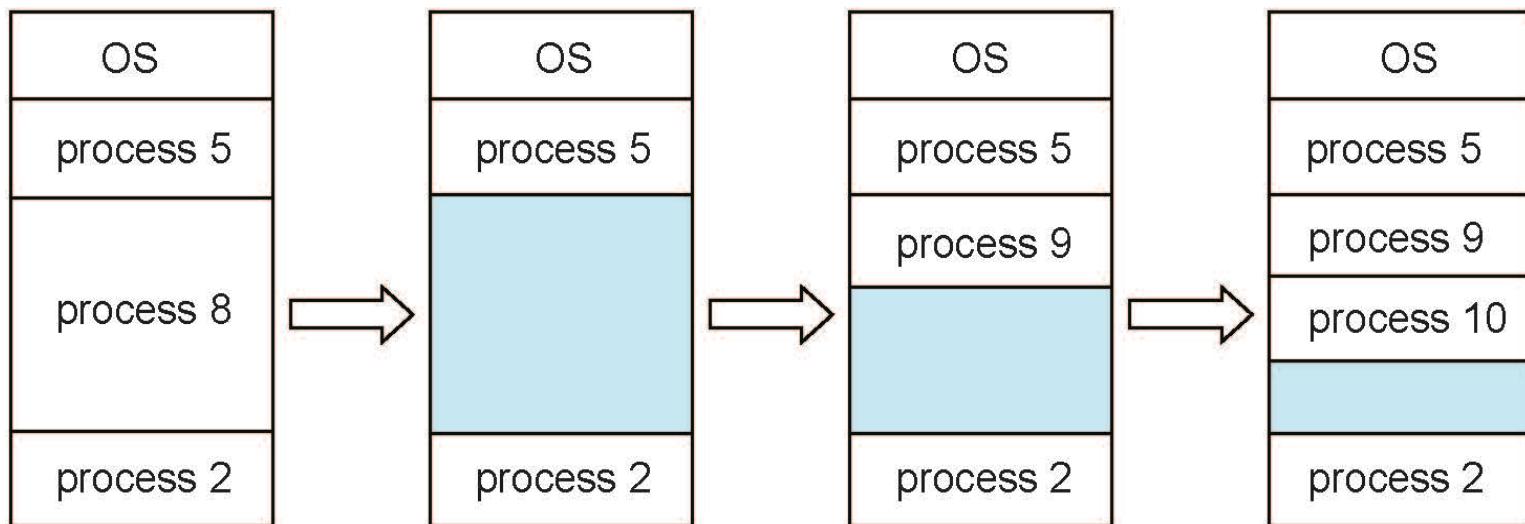
Mechanics of variable partitions



Multiple-partition allocation

- Multiple-partition allocation

- **Variable-partition** sizes for efficiency (sized to a given process' needs)
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
 - a) allocated partitions
 - b) free partitions (hole)



Dynamic Storage-Allocation Problem

How to satisfy a request of size n from a list of free holes?

- **First-fit:** Allocate the *first* hole that is big enough
- **Best-fit:** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
 - Produces the smallest leftover hole
- **Worst-fit:** Allocate the *largest* hole; must also search entire list
 - Produces the largest leftover hole

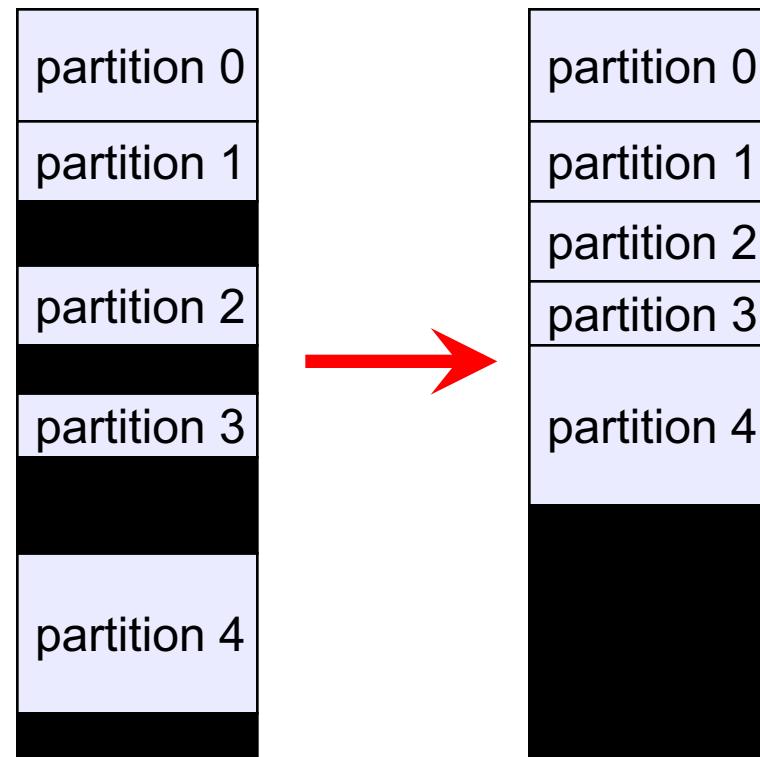
First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory;
- First fit analysis reveals that given N blocks allocated, $0.5 N$ blocks lost to fragmentation
 - 1/3 may be unusable -> **50-percent rule**

Dealing with fragmentation

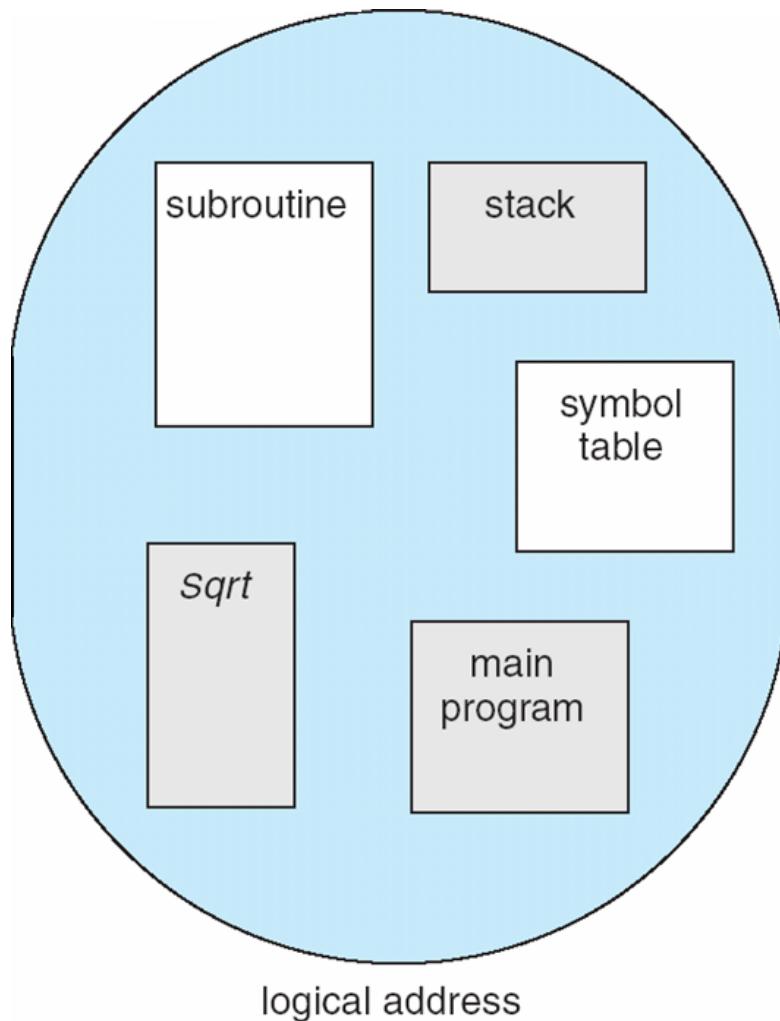
- Compact memory by copying
 - Swap a program out
 - Re-load it, adjacent to another
 - Adjust its base register
 - Compaction is possible *only if relocation is dynamic*
 - I/O problem
 - Latch job in memory while it is involved in I/O
 - Do I/O only into OS buffers



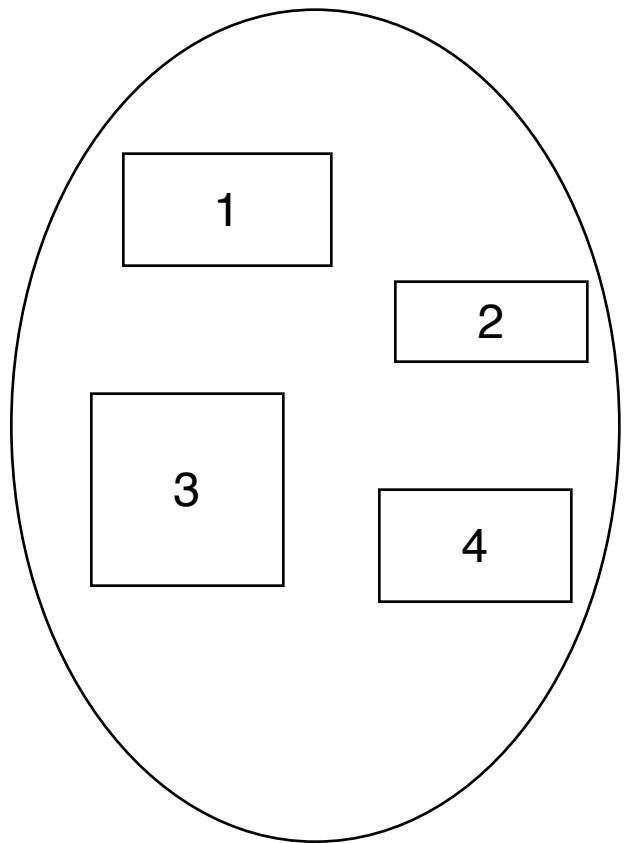
Segmentation

- Dealing with fragmentation
 - Why not remove need for continuous addresses?
- Segmentation
 - partition an address space into *logical* units
 - stack, code, heap, subroutines, ...
 - a virtual address is <segment #, offset>
- Facilitates sharing and reuse
 - a segment is a natural unit of sharing – a subroutine or function
- A natural extension of variable-sized partitions
 - variable-sized partition = 1 segment/process
 - segmentation = many segments/process

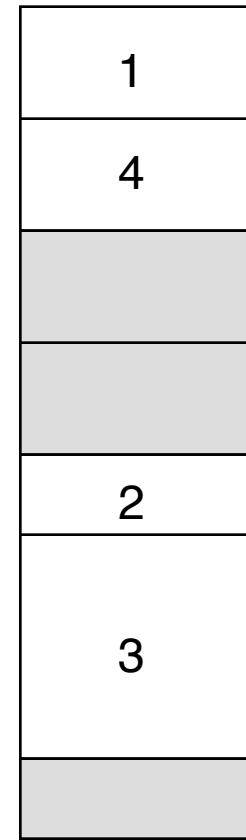
User's View of a Program



Logical View of Segmentation



user space

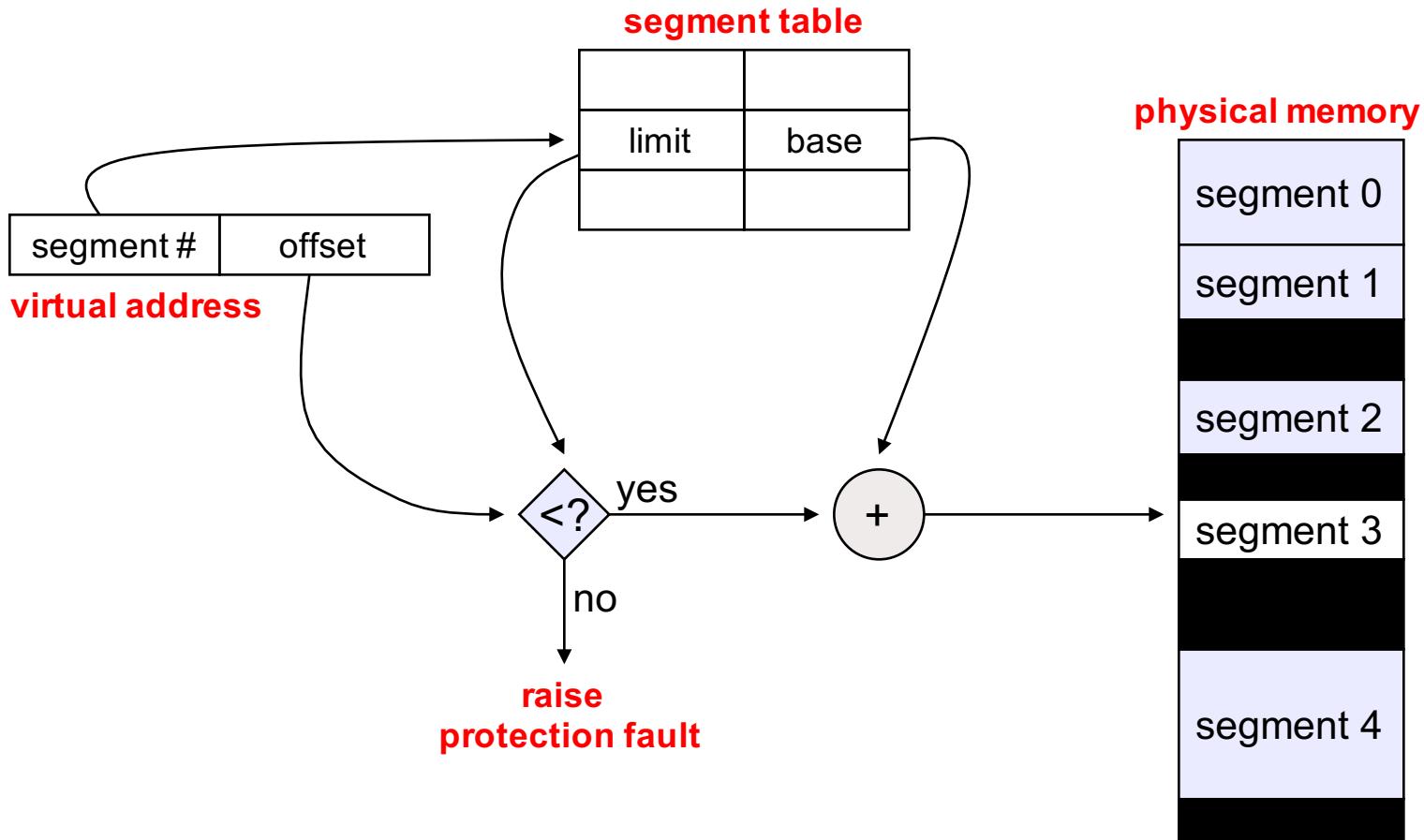


physical memory space

Hardware support

- Segment table
 - multiple base/limit pairs, **one per segment**
 - segments named by segment #, used as index into table
 - a logical/virtual address is **<segment #, offset>**
 - offset of virtual address added to base address of segment to yield physical address

Segment lookups



Pros and cons

- Logical and it facilitates sharing and reuse
- Allows non-contiguous physical addresses
 - Helps exploits varying sized holes
- But it has the complexity of a variable partition system
 - except that linking is simpler, and the “chunks” that must be allocated are smaller than a “typical” linear address space
- Segmentation rarely used alone
 - Paging is the basis for modern memory management
 - Covered in next lecture

Summary

- Logical/Virtual Address Space vs Physical Address Space
- Swapping
- Contiguous Memory Allocation
- Fragmentation
- Segmentation
- Paging
 - A better solution
 - Next lecture

Operating Systems

Paging

Lecture 10
Michael O'Boyle

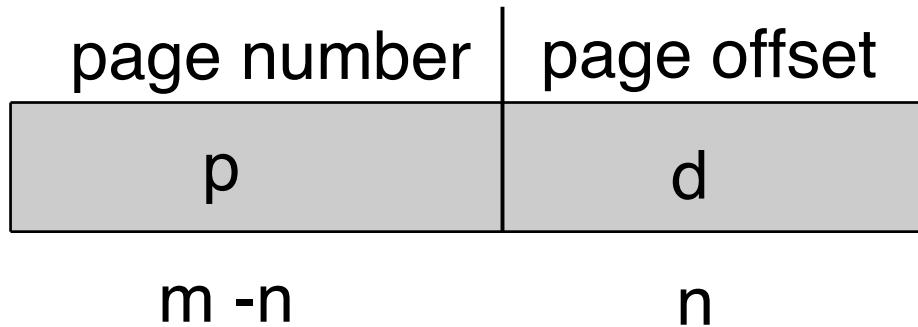
Overview

- Paging
- Page Tables
- TLB
- Shared Pages
- Hierarchical Pages
- Hashed Pages
- Inverted Pages
- Uses

Address Translation Scheme

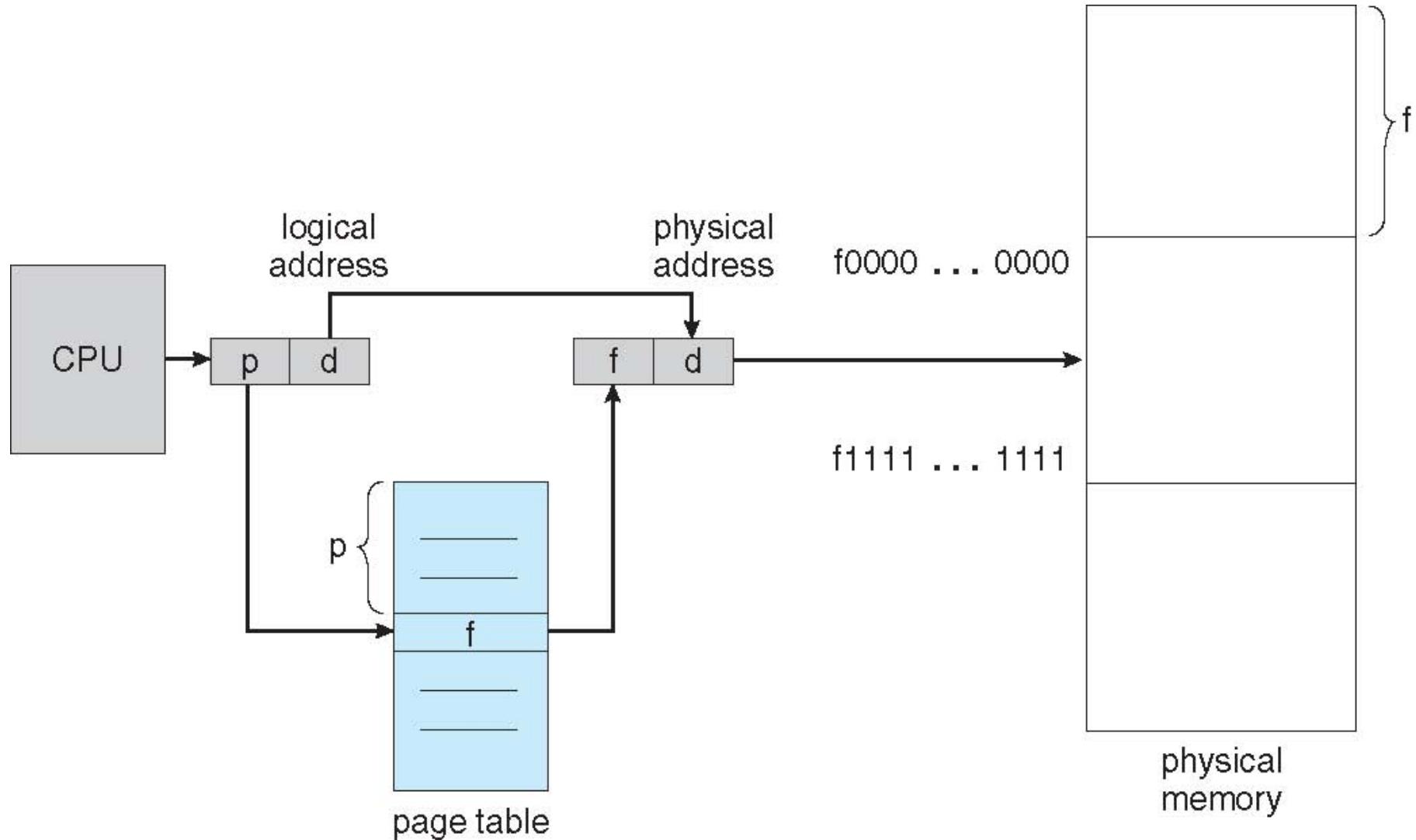
- Address generated by CPU is divided into:

- **Page number (p)** – used as an index into a **page table** which contains base address of each page in physical memory
- **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

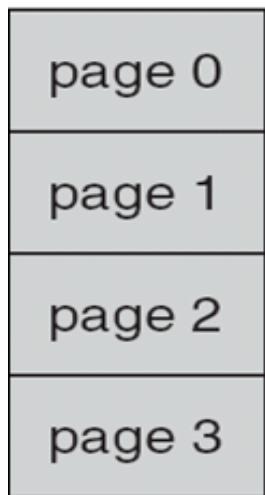


- For given logical address space 2^m and page size 2^n

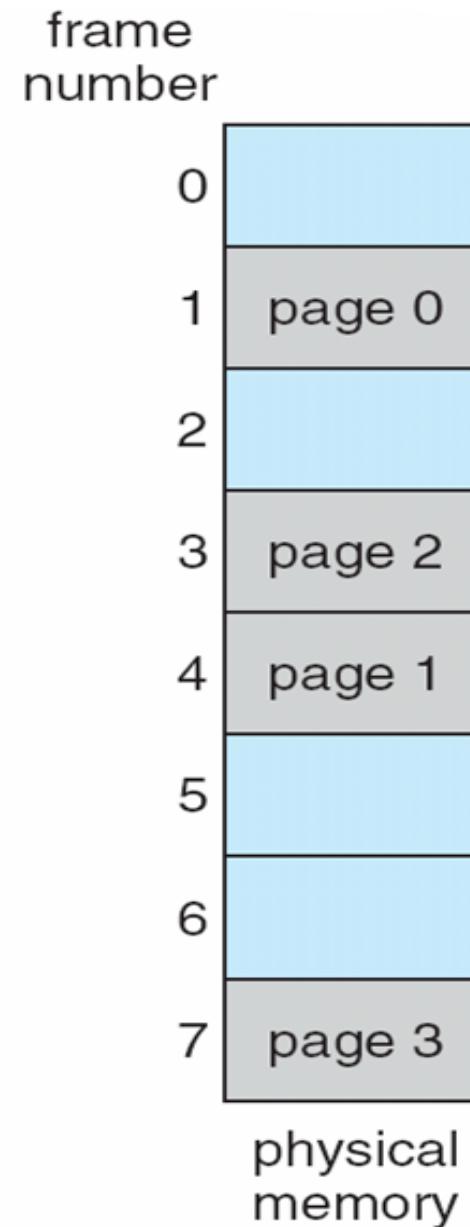
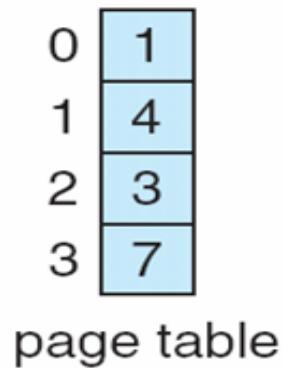
Paging Hardware



Paging Model of Logical and Physical Memory



logical
memory



Paging Example

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

0	5
1	6
2	1
3	2

page table

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory

$n=2$ and $m=4$ 32-byte memory and 4-byte pages

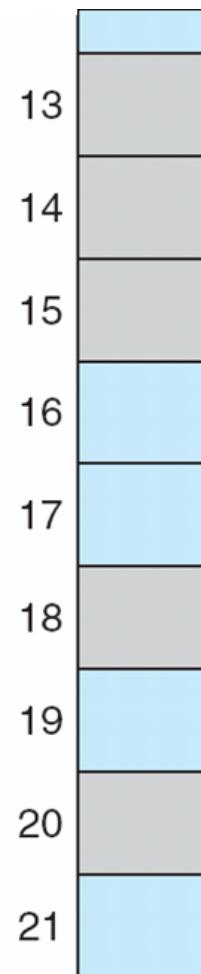
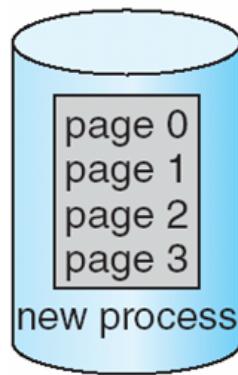
Paging (Cont.)

- Calculating internal fragmentation
 - Page size = 2,048 bytes
 - Process size = 72,766 bytes
 - 35 pages + 1,086 bytes
 - Internal fragmentation of $2,048 - 1,086 = 962$ bytes
 - Worst case fragmentation = 1 frame – 1 byte
 - On average fragmentation = 1 / 2 frame size
 - So small frame sizes desirable?
 - But each page table entry takes memory to track
 - Page sizes growing over time
 - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- By implementation process can only access its own memory

Free Frames

free-frame list

14
13
18
20
15

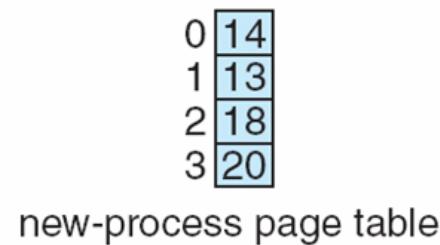
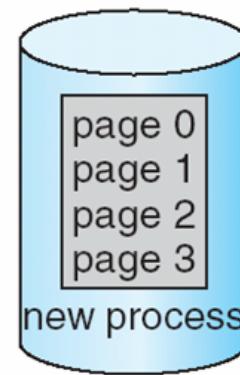


(a)

Before allocation

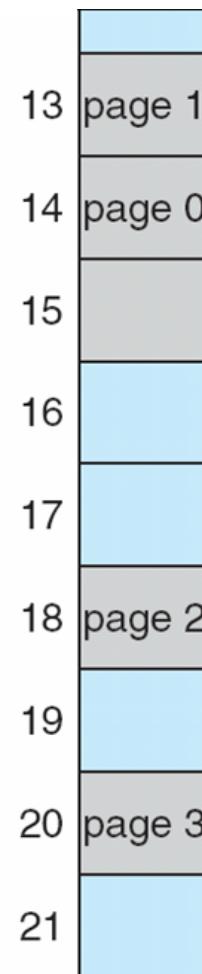
free-frame list

15



(b)

After allocation



Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
 - One for the page table and one for the data / instruction
- The two memory access problem can be solved
 - by the use of a special fast-lookup hardware cache
 - called **associative memory** or **translation look-aside buffers (TLBs)**

Implementation of Page Table

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry –
 - uniquely identifies each process
 - provide address-space protection for that process
 - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
 - Replacement policies must be considered
 - Some entries can be **wired down** for permanent fast access

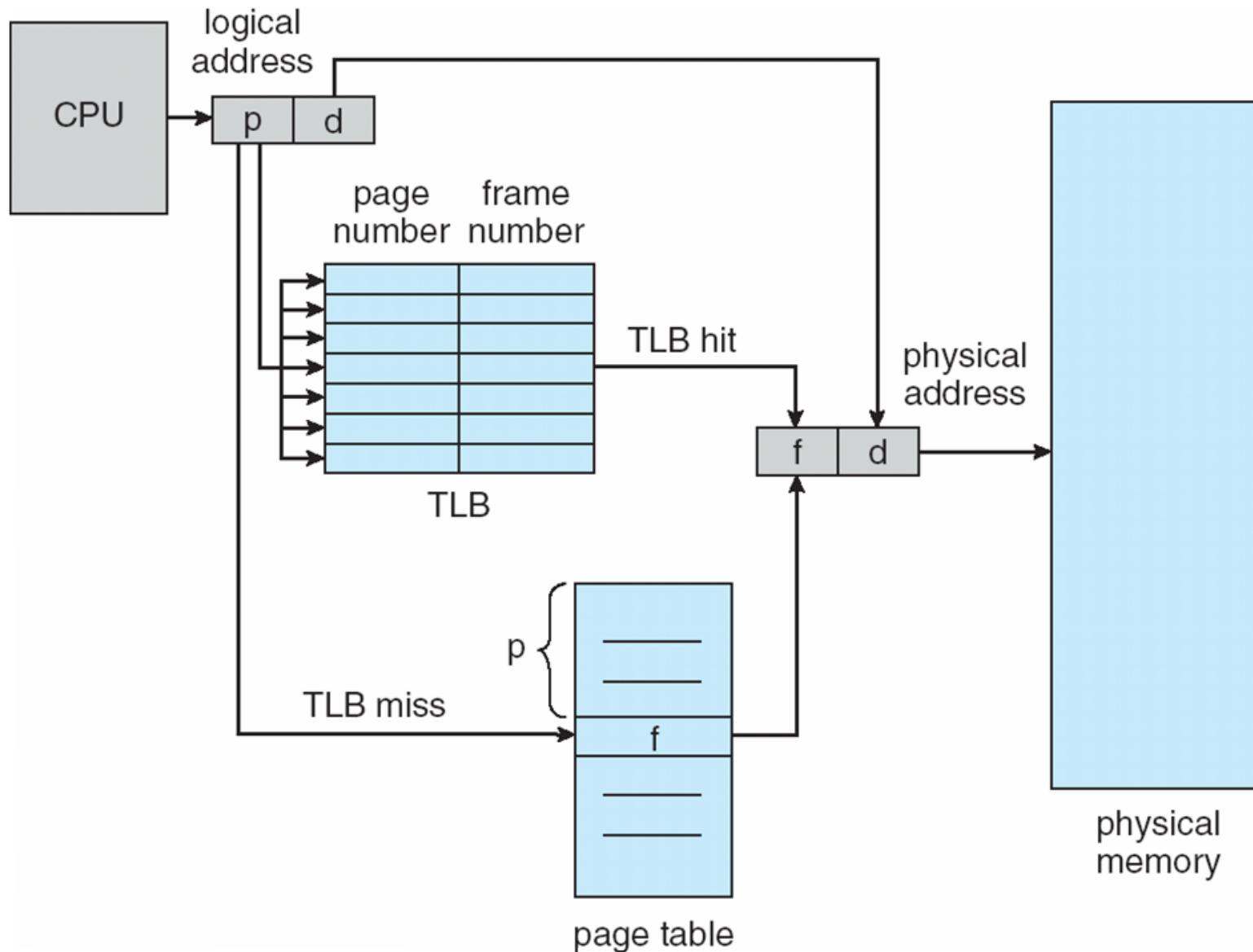
Associative Memory

- Associative memory – parallel search

Page #	Frame #

- Address translation (p, d)
 - If p is in associative register, get frame # out
 - Otherwise get frame # from page table in memory

Paging Hardware With TLB



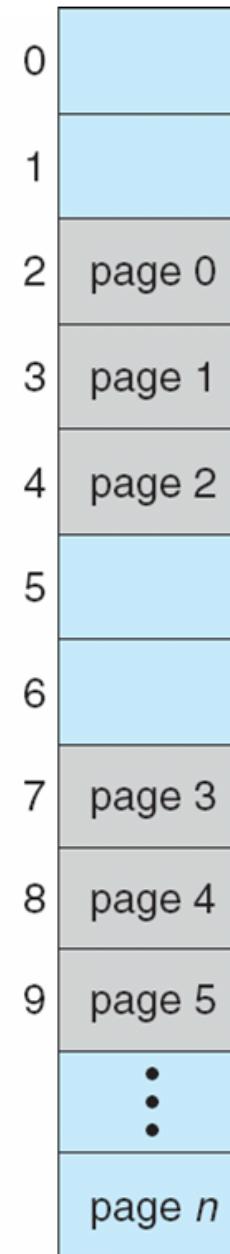
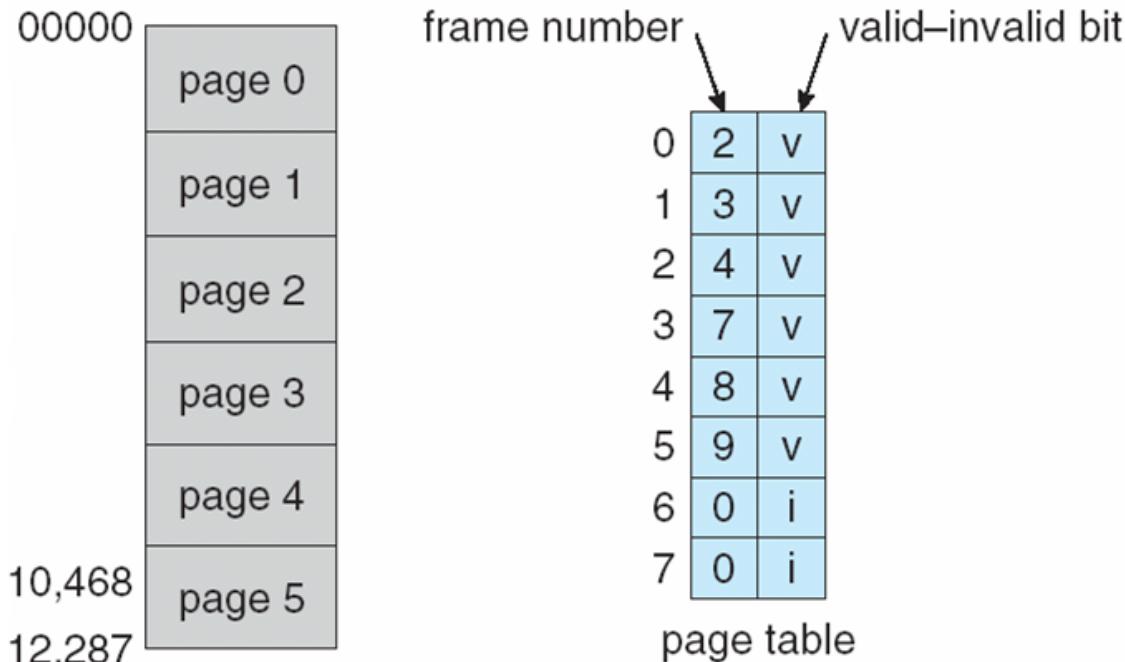
Effective Access Time

- Associative Lookup
 - Extremely fast
- Hit ratio = α
 - Hit ratio – percentage of times that a page number is found in the associative memory ;
 - Consider $\alpha = 80\%$, 100ns for memory access
- Consider $\alpha = 80\%$, 100ns for memory access
 - $EAT = 0.80 \times 100 + 0.20 \times 200 = 120\text{ns}$
- Consider hit ratio $\alpha = 99$, 100ns for memory access
 - $EAT = 0.99 \times 100 + 0.01 \times 200 = 101\text{ns}$

Memory Protection

- Memory protection implemented
 - by associating protection bit with each frame
 - to indicate if read-only or read-write access is allowed
 - Can also add more bits to indicate page execute-only, and so on
- **Valid-invalid** bit attached to each entry in the page table:
 - “valid” indicates that the associated page
 - is in the process’ logical address space, and is thus a legal page
 - “invalid” indicates that the page I
 - is not in the process’ logical address space
 - Or use **page-table length register (PTLR)**
 - Page Table Entries (PTEs) can contain more information
- Any violations result in a trap to the kernel

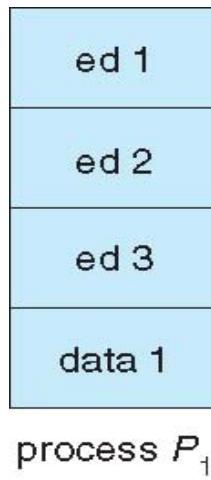
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

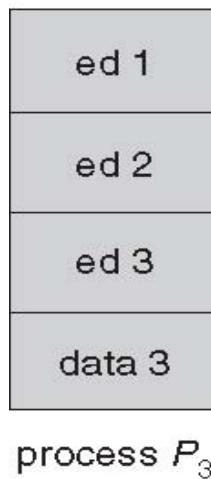
- **Shared code**
 - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
 - Similar to multiple threads sharing the same process space
 - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Private code and data**
 - Each process keeps a separate copy of the code and data
 - The pages for the private code and data can appear anywhere in the logical address space

Shared Pages Example



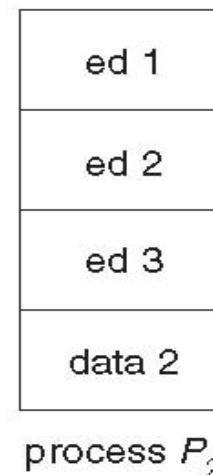
3
4
6
1

page table
for P_1



3
4
6
2

page table
for P_3



3
4
6
7

page table
for P_2

0
1
2
3
4
5
6
7
8
9
10
11

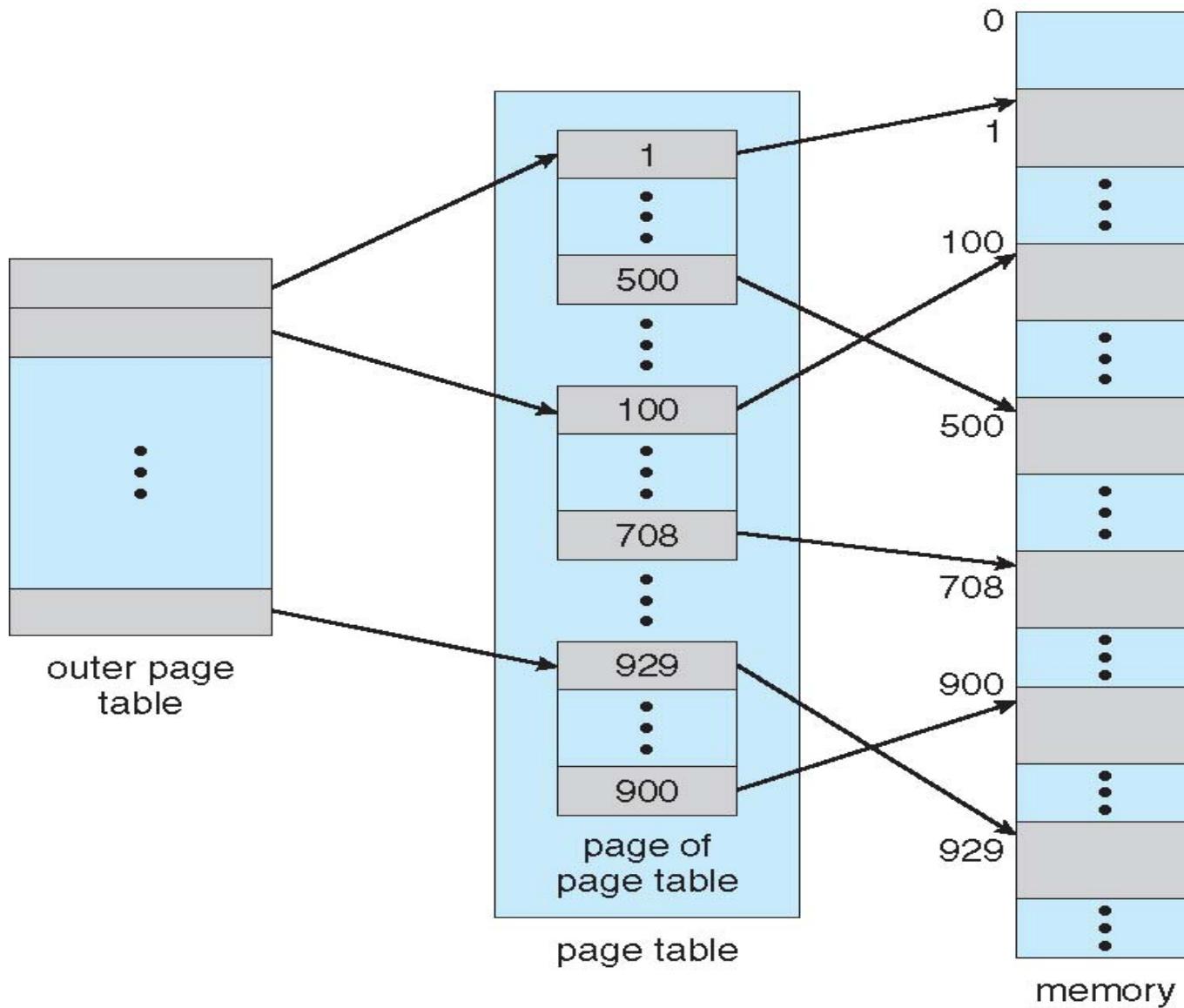
Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
 - Consider a 32-bit logical address space as on modern computers
 - Page size of 4 KB (2^{12})
 - Page table would have 1 million entries ($2^{32} / 2^{12}$)
 - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
 - That amount of memory used to cost a lot
 - Don't want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

Two-Level Page-Table Scheme

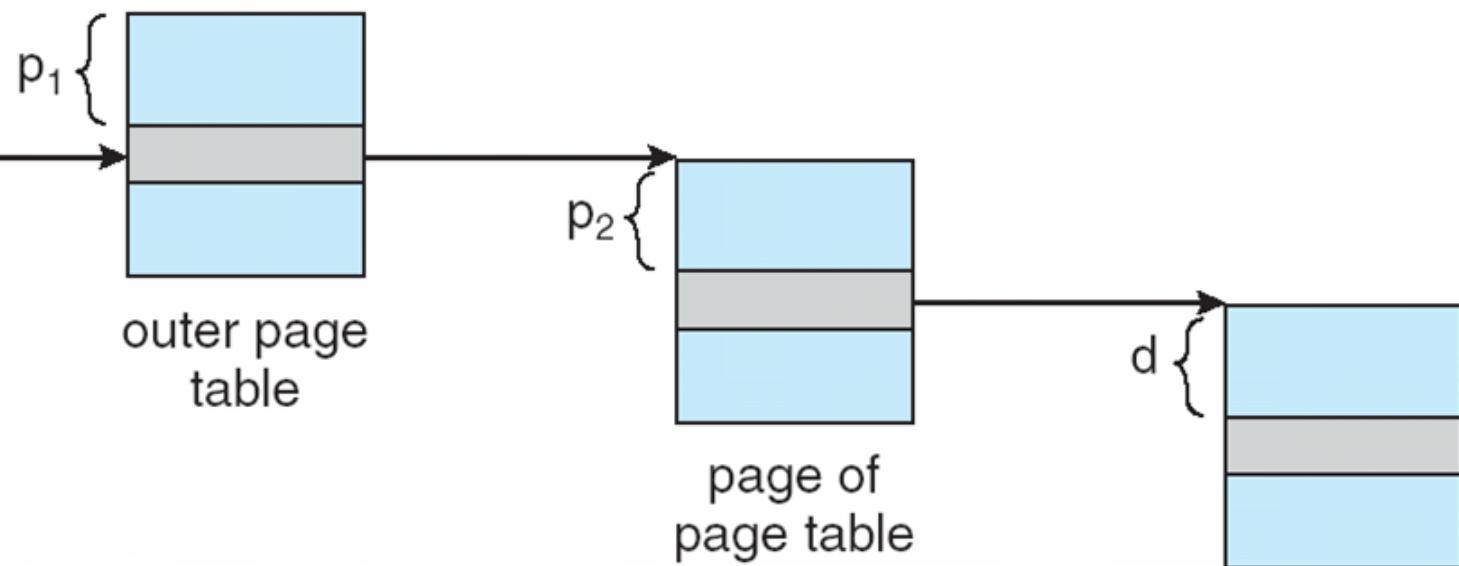
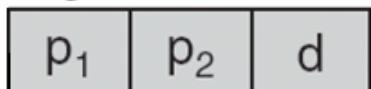


Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
 - a page number consisting of 22 bits
 - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
 - a 12-bit page number
 - a 10-bit page offset
- Thus, a logical address is as follows:
- | page number | | page offset |
|-------------|-------|-------------|
| p_1 | p_2 | d |
| 12 | 10 | 10 |
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

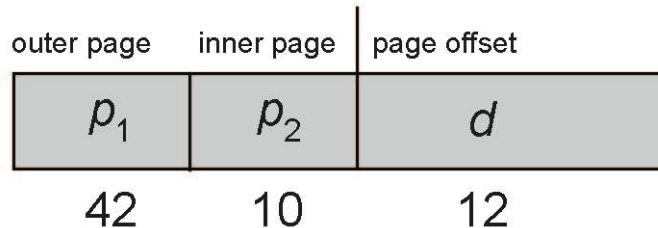
Address-Translation Scheme

logical address



64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
 - Then page table has 2^{52} entries
 - If two level scheme, inner page tables could be 2^{10} 4-byte entries
 - Address would look like



- Outer page table has 2^{42} entries or 2^{44} bytes
- One solution is to add a 2nd outer page table
- But in the following example the 2nd outer page table is still 2^{34} bytes in size
 - ▶ And possibly 4 memory access to get to one physical memory location

Three-level Paging Scheme

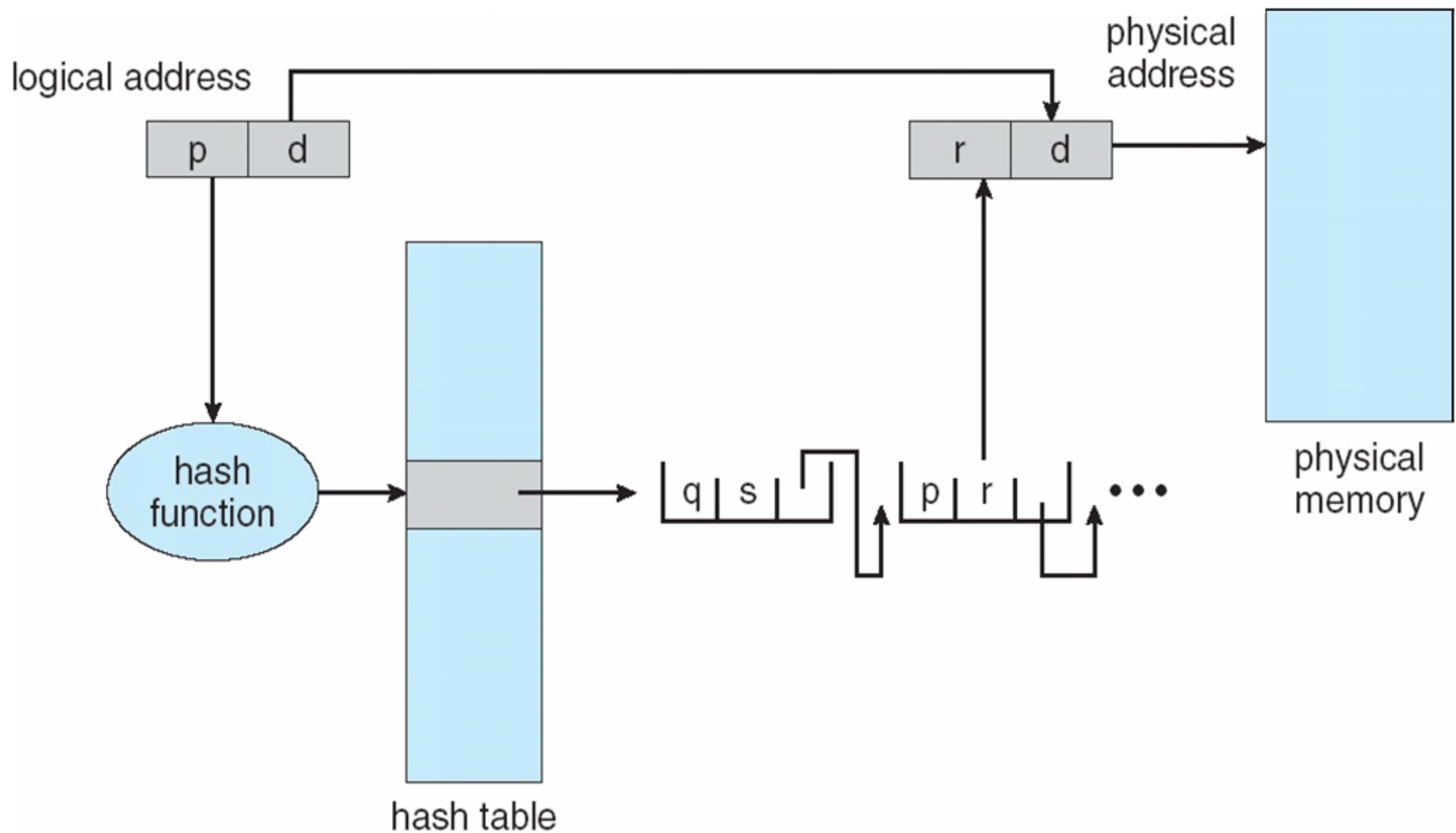
outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Hashed Page Tables

- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
 - This page table contains a chain of elements hashing to the same location
- Each element contains
 - (1) the virtual page number
 - (2) the value of the mapped page frame
 - (3) a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
 - If a match is found, the corresponding physical frame is extracted
- Variation for 64-bit addresses is **clustered page tables**
 - Similar to hashed but each entry refers to several pages (such as 16) rather than 1
 - Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

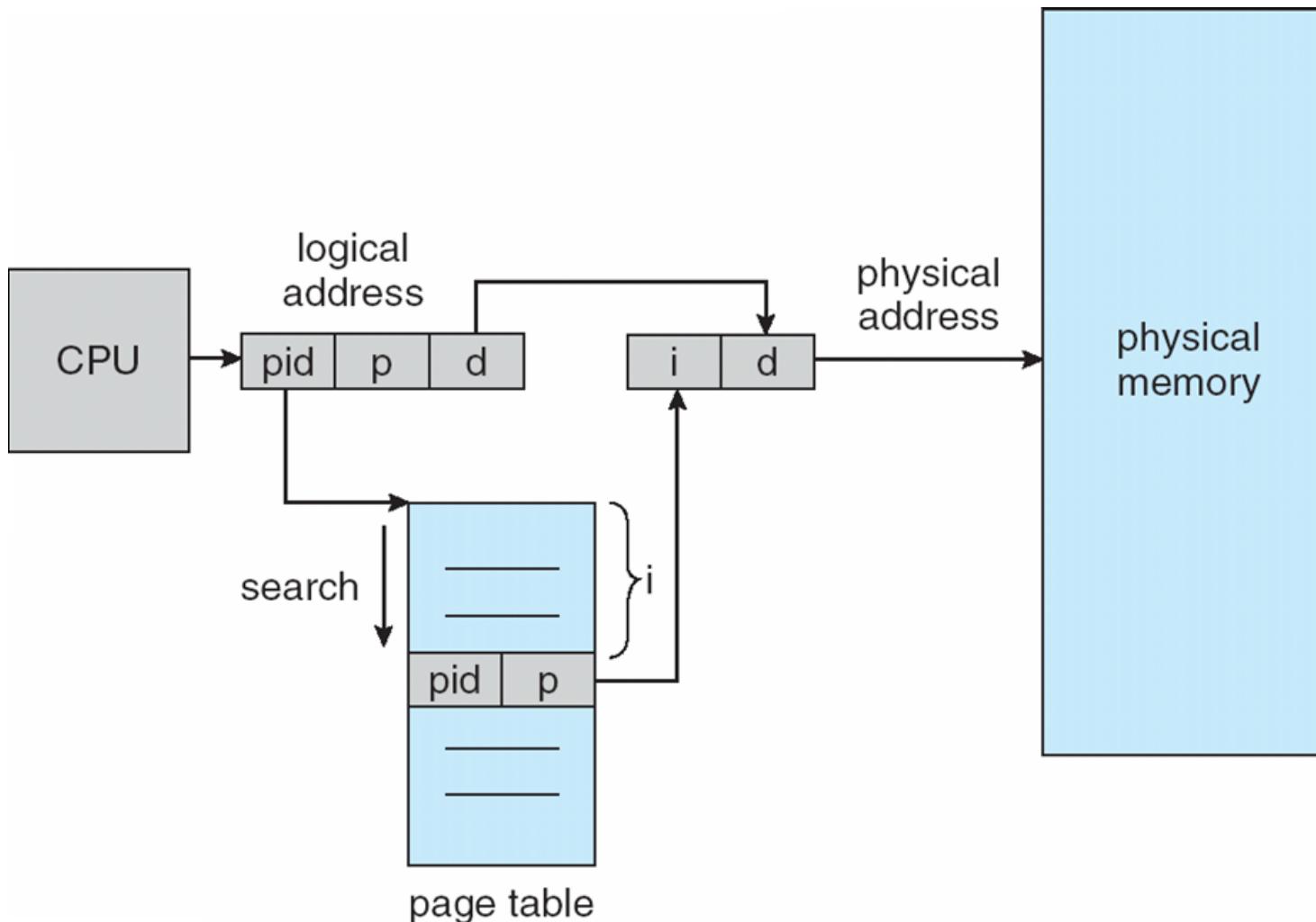
Hashed Page Table



Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages,
 - track all physical pages
- One entry for each real page of memory
- Entry consists of
 - the virtual address of the page stored in that real memory location,
 - information about the process that owns that page
- Decreases memory needed to store each page table
 - but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one/few page-table entries
 - TLB can accelerate access
- But how to implement shared memory?
 - One mapping of a virtual address to the shared physical address

Inverted Page Table Architecture



Functionality enhanced by page tables

- Code (instructions) is read-only
 - A bad pointer can't change the program code
- Dereferencing a null pointer is an error caught by hardware
 - Don't use the first page of the virtual address space – mark it as invalid – so references to address 0 cause an interrupt
- Inter-process memory protection
 - My address XYZ is different than your address XYZ
- Shared libraries
 - All running C programs use libc
 - Have only one (partial) copy in physical memory, not one per process
 - All page table entries mapping libc point to the same set of physical frames
 - DLL's in Windows

More functionality

- Generalizing the use of “shared memory”
 - Regions of two separate processes’s address spaces map to the same physical frames
 - Faster inter-process communication
 - Just read/write from/to shared memory
 - Don’t have to make a syscall
 - Will have separate Page Table Entries (PTEs) per process, so can give different processes different access rights
 - E.g., one reader, one writer
- Copy-on-write (CoW), e.g., on fork()
 - Instead of copying all pages, create shared mappings of parent pages in child address space
 - Make shared mappings read-only for both processes
 - When either process writes, fault occurs, OS “splits” the page

Less familiar uses

- Memory-mapped files
 - instead of using open, read, write, close
 - “map” a file into a region of the virtual address space
 - e.g., into region with base ‘X’
 - accessing virtual address ‘X+N’ refers to offset ‘N’ in file
 - initially, all pages in mapped region marked as invalid
 - OS reads a page from file whenever invalid page accessed
 - OS writes a page to file when evicted from physical memory
 - only necessary if page is dirty

More unusual use

- Use “soft faults”
 - faults on pages that are actually in memory,
 - but whose PTE entries have artificially been marked as invalid
- That idea can be used whenever it would be useful to trap on a reference to some data item
- Example: debugger watchpoints
- Limited by the fact that the granularity of detection is the page

Summary

- Paging
- Page Tables
- TLB
- Shared Pages
- Hierarchical Pages
- Hashed Pages
- Inverted Pages
- Uses
- Next time: Virtual Memory

Operating Systems

Virtual Memory

Lecture 11
Michael O'Boyle

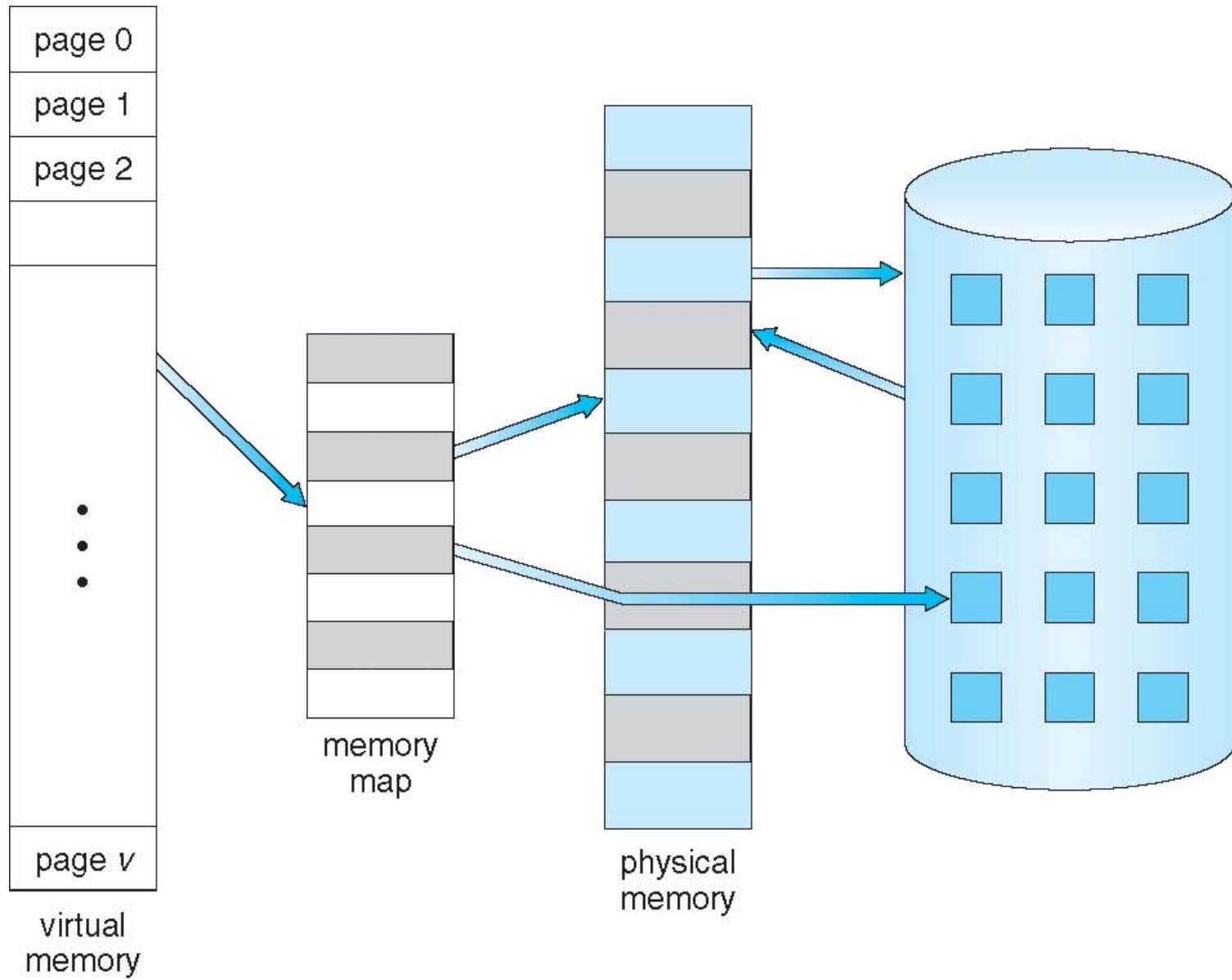
Paged virtual memory

Allows a larger logical address space than physical memory

All pages of address space do not need to be in memory

- the full (used) address space on disk in page-sized blocks
- main memory used as a (page) cache
- Needed page transferred to a free page frame
 - if no free page frames, evict a page
 - evicted pages go to disk only if **dirty**
 - Transparent to the application, except for performance
 - managed by hardware and OS
- Traditionally called **paged virtual memory**

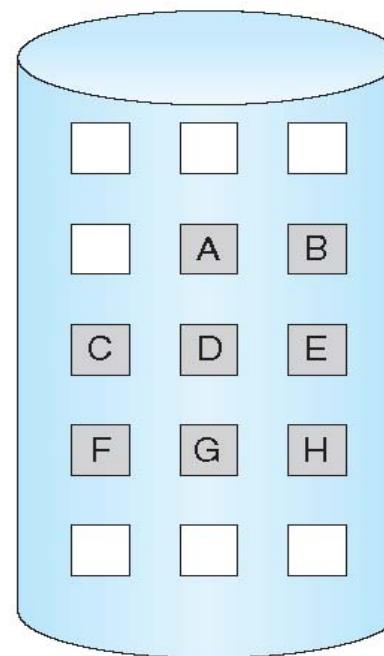
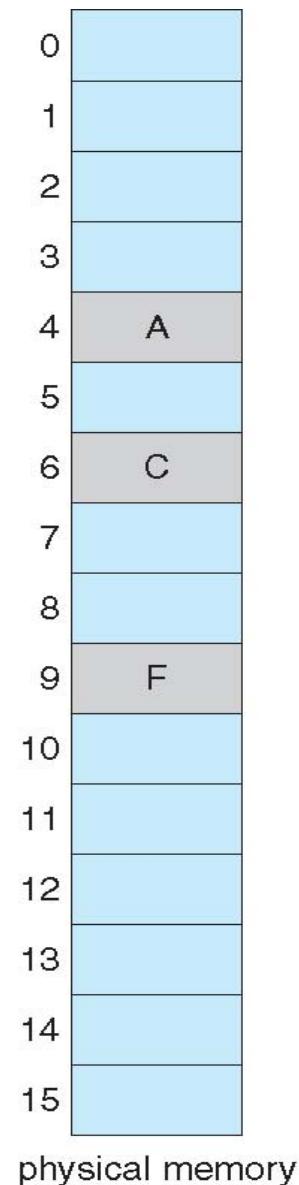
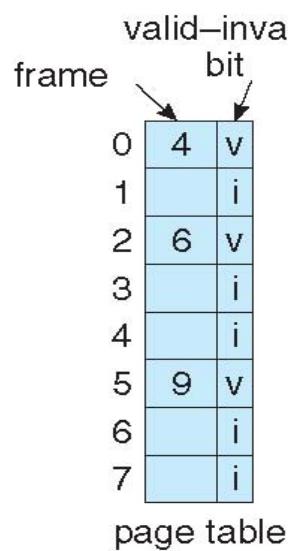
Virtual Memory That is Larger Than Physical Memory



Page Table When Some Pages Are Not in Main Memory

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

logical memory



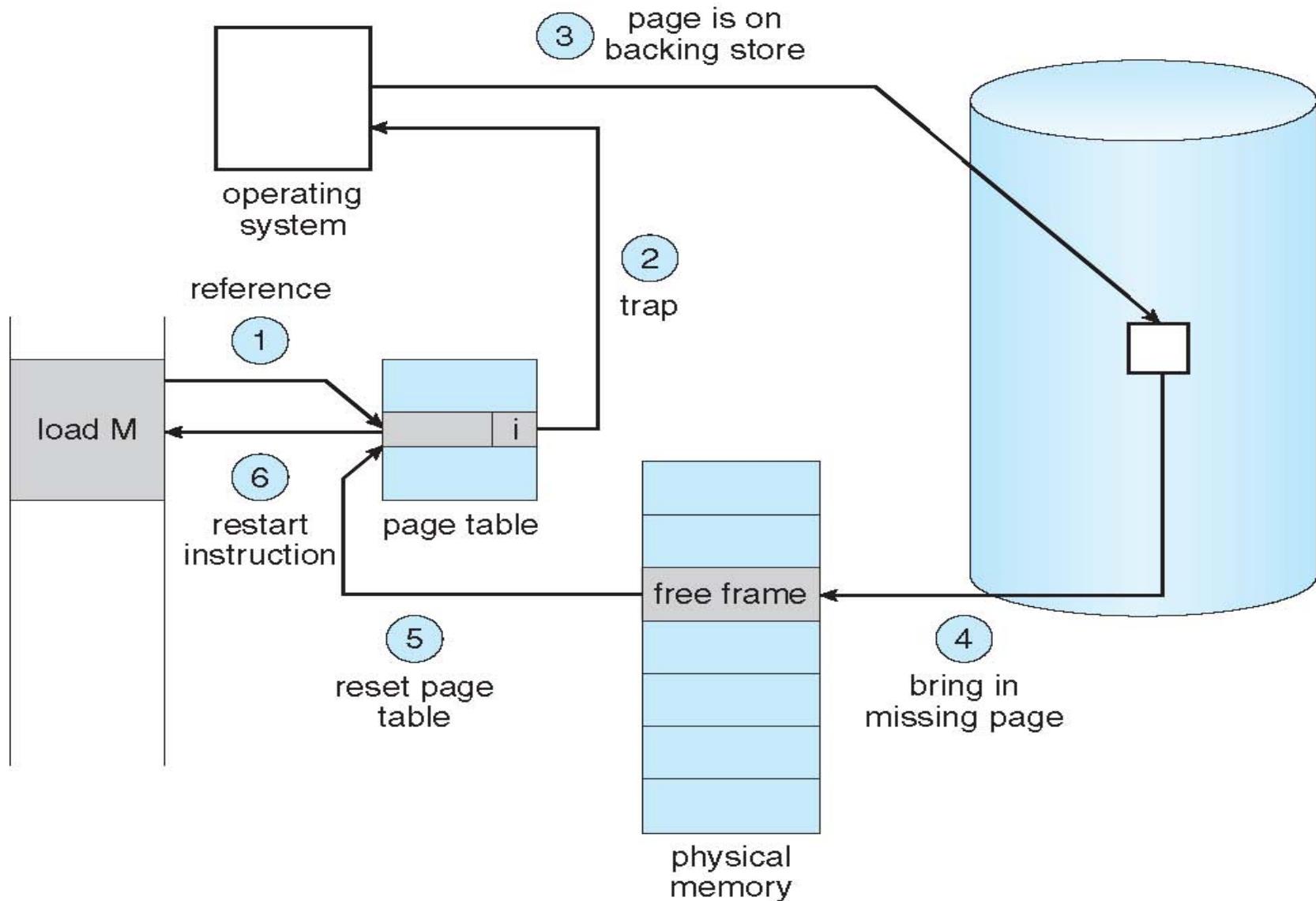
Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

page fault

1. Operating system looks at another table to decide:
 - Invalid reference \Rightarrow abort
 - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
Set validation bit = **v**
5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault



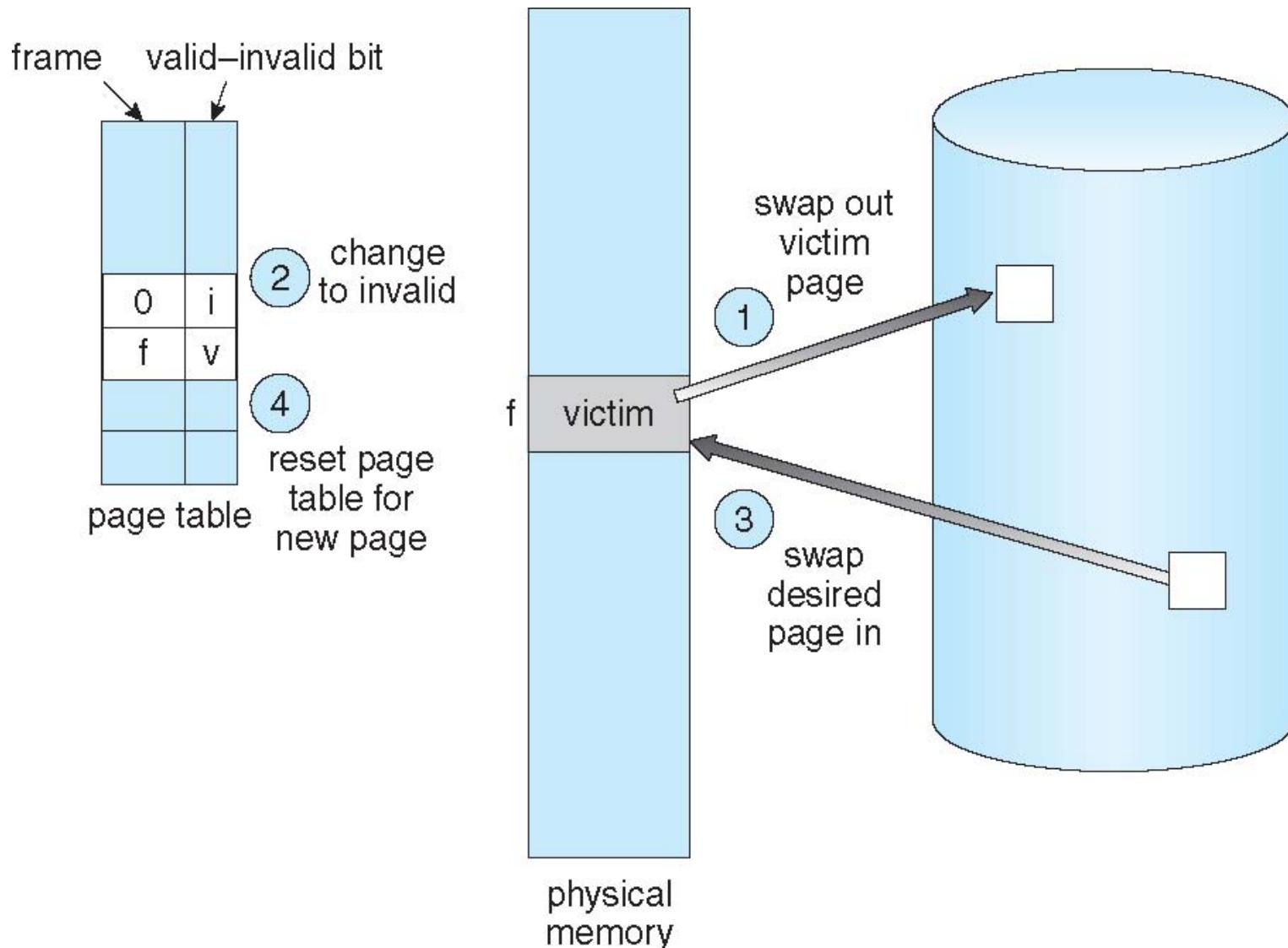
Demand paging

- Pages only brought into memory when referenced
 - Only code/data that is needed by a process needs to be loaded
 - What's needed changes over time
 - Hence, it's called **demand paging**
- Few systems try to anticipate future needs
- But sometimes cluster pages
 - OS keeps track of pages that should come and go together
 - bring in all when one is referenced
 - interface may allow programmer or compiler to identify clusters

Page replacement

- When you read in a page, where does it go?
 - if there are free page frames, grab one
 - if not, must evict something else
 - this is called **page replacement**
- Page replacement algorithms
 - try to pick a page that won't be needed in the near future
 - try to pick a page that hasn't been modified (thus saving the disk write)
- OS tries to keep a pool of free pages around
 - so that allocations don't inevitably cause evictions
- OS tries to keep some “clean” pages around
 - so that even if you have to evict a page, you won't have to write it

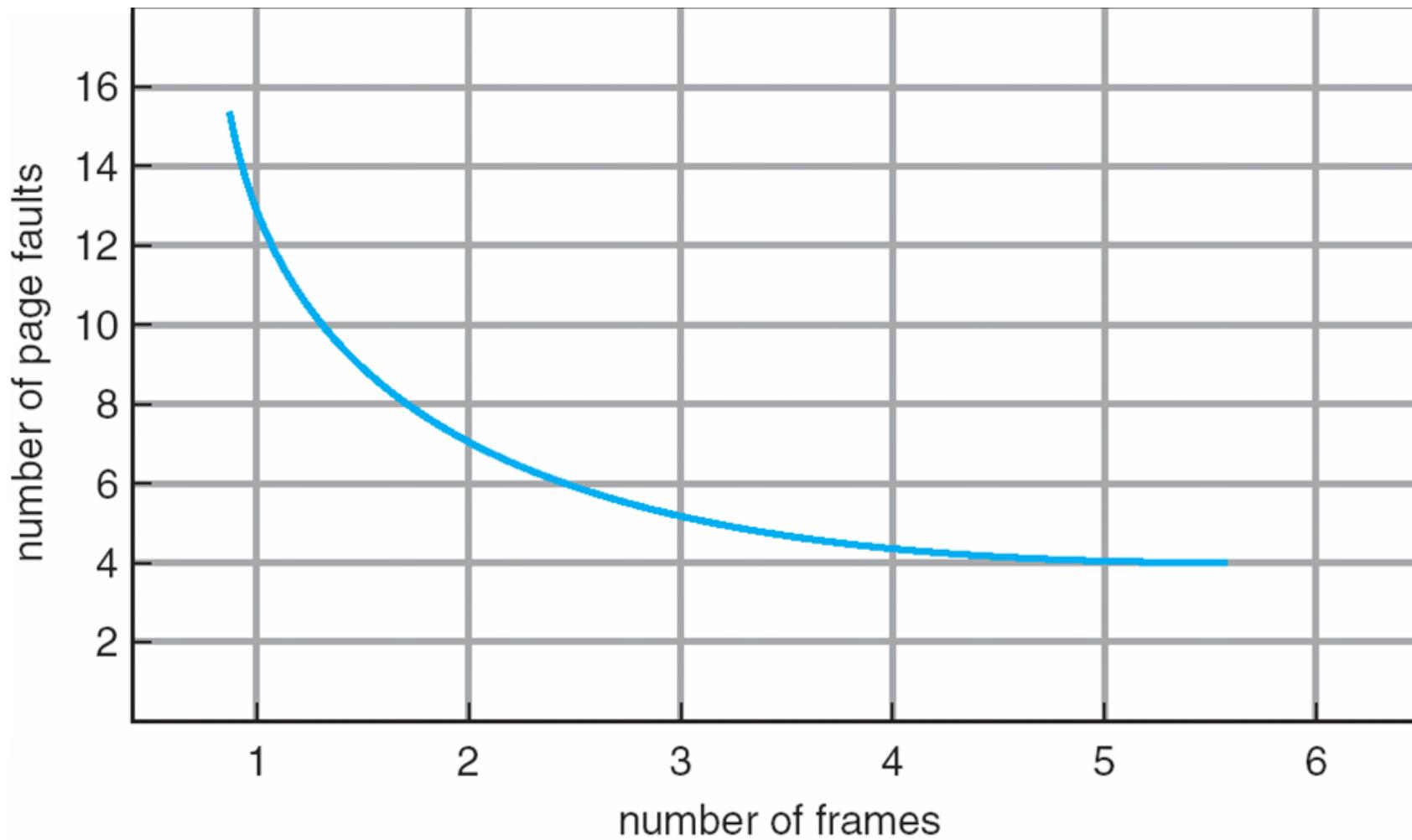
Page Replacement



Evicting the best page

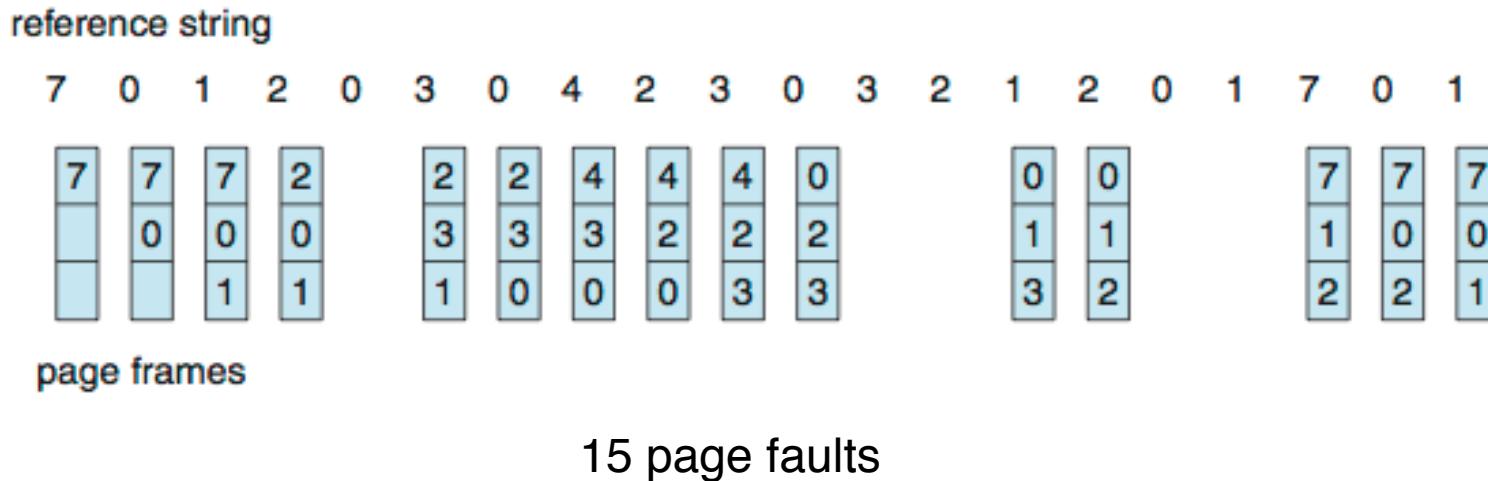
- The goal of the page replacement algorithm:
 - reduce fault rate by selecting best victim page to remove
 - the best page to evict is one that will never be touched again
 - Belady's proof:
 - evicting the page that won't be used for the longest period of time minimizes page fault rate
- Examine **page replacement algorithms**
 - assume that a process pages against itself
 - using a fixed number of page frames
- Number of frames available impacts page fault rate
 - Note Belady's anomaly

Graph of Page Faults Versus The Number of Frames



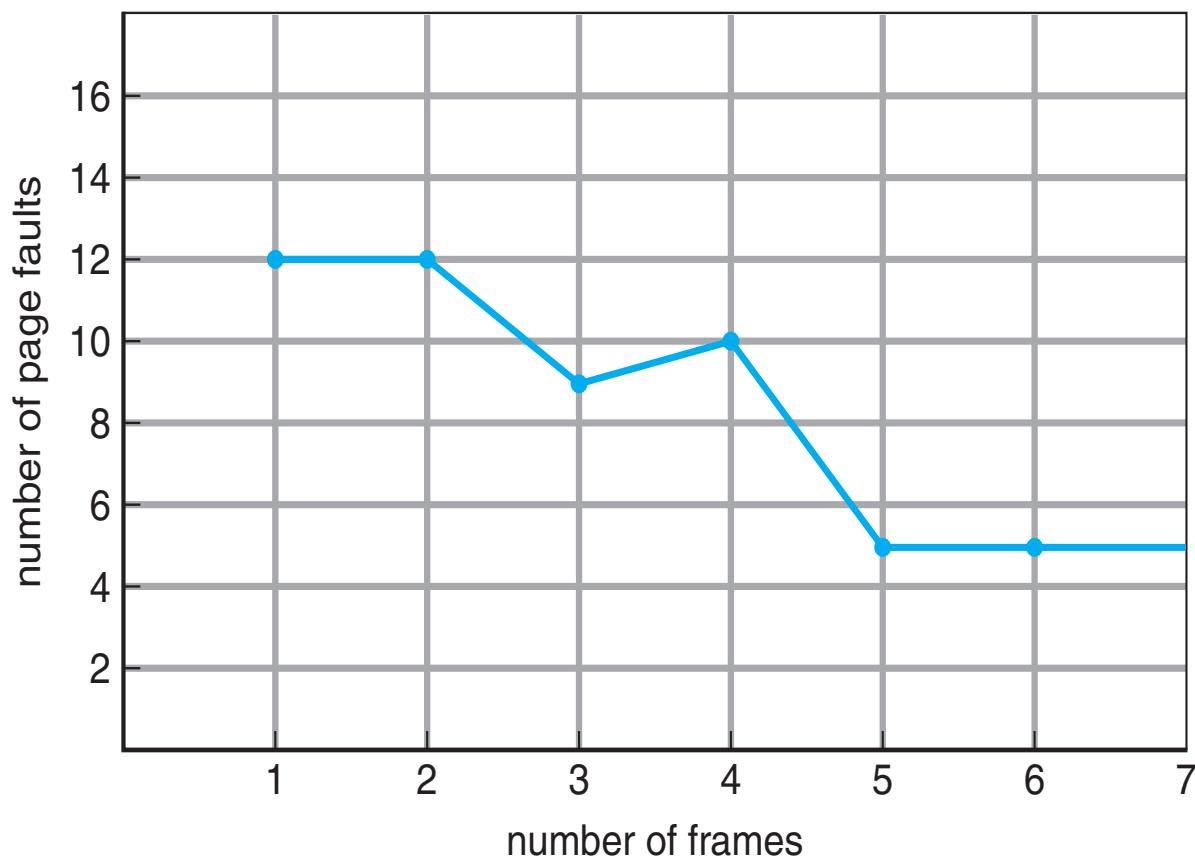
First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
 - Adding more frames can cause more page faults!
 - **Belady's Anomaly**

FIFO Illustrating Belady' s Anomaly

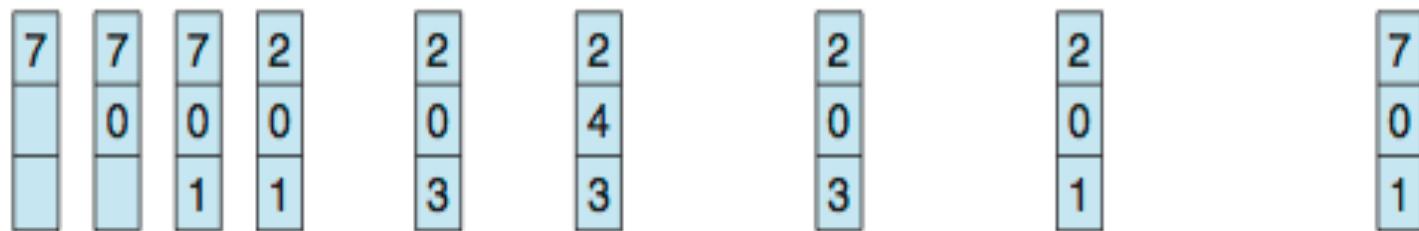


Belady's Optimal Algorithm

- Replace page that will not be used for longest period of time
 - 9 is optimal for the example
- How do you know this?
 - Can't read the future
- Used for measuring how well your algorithm performs

reference string

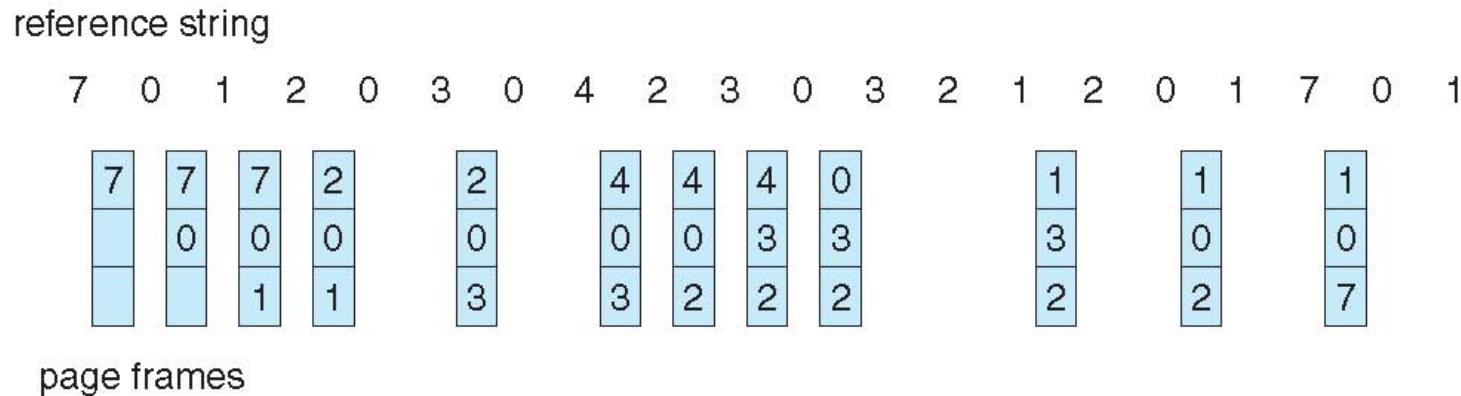
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



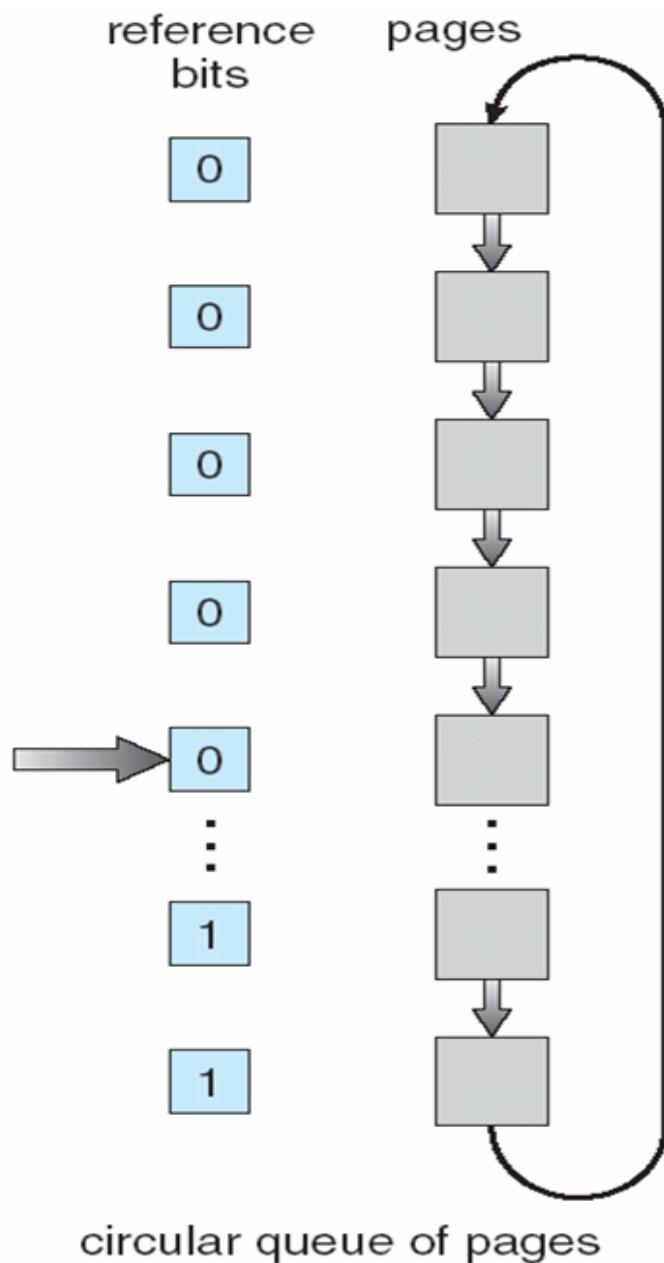
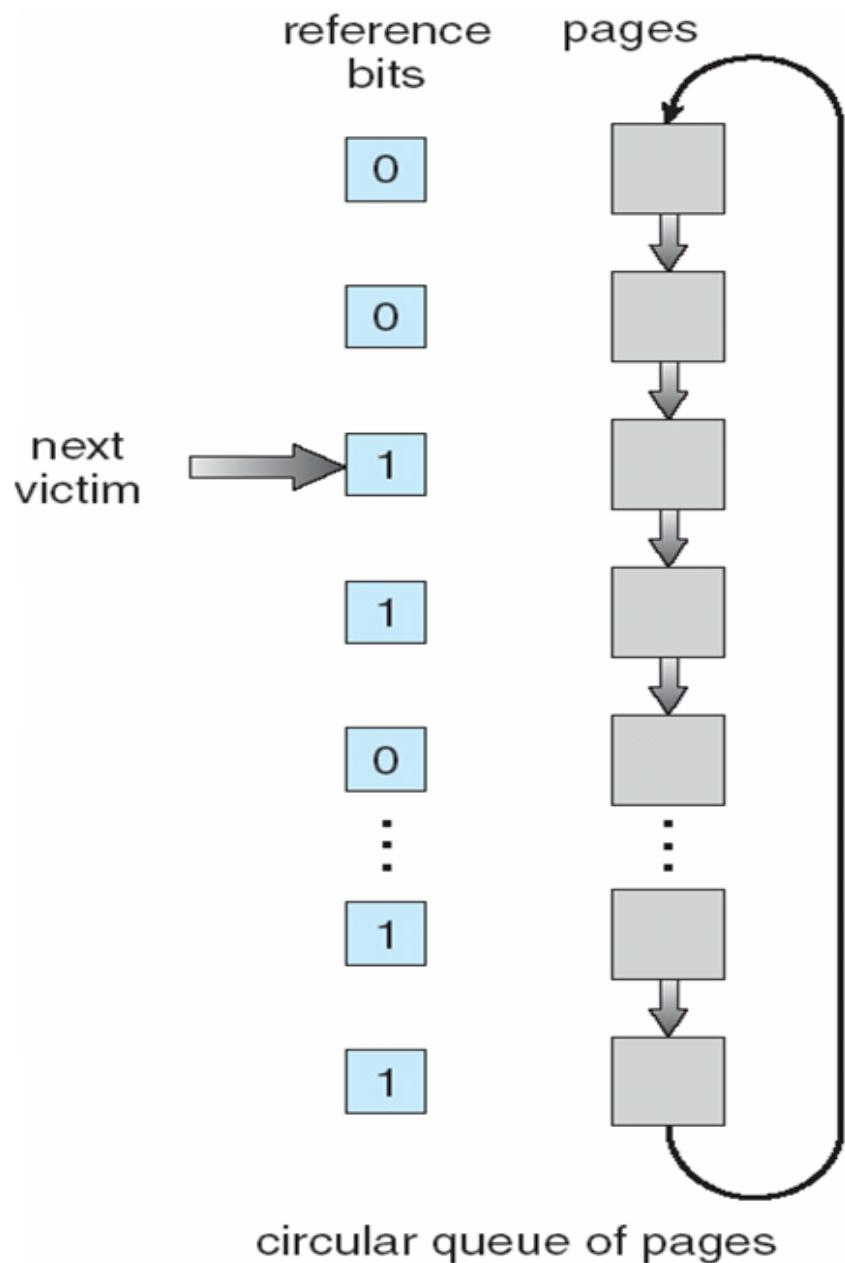
- 12 faults – better than FIFO but worse than Belady's/ OPT
- Generally good algorithm and frequently used
- But how to implement?

Approximating LRU

- Many approximations, all use the PTE's referenced bit
 - keep a counter for each page
 - at some regular interval, for each page, do:
 - if ref bit = 0, increment the counter (hasn't been used)
 - if ref bit = 1, zero the counter (has been used)
 - regardless, zero ref bit
 - the counter will contain the # of intervals since the last reference to the page
 - page with largest counter is least recently used
- Some architectures don't have PTE reference bits
 - can simulate reference bit using the valid bit to induce faults

Second-chance Clock

- Not Recently Used (NRU) or Second Chance
 - replace page that is “old enough”
 - logically, arrange all physical page frames in a big circle (clock)
 - just a circular linked list
- A “clock hand” is used to select a good LRU candidate
 - sweep through the pages in circular order like a clock
- If ref bit is off, it hasn’t been used recently, we have a victim
- If the ref bit is on, turn it off and go to next page
 - arm moves quickly when pages are needed



(a)

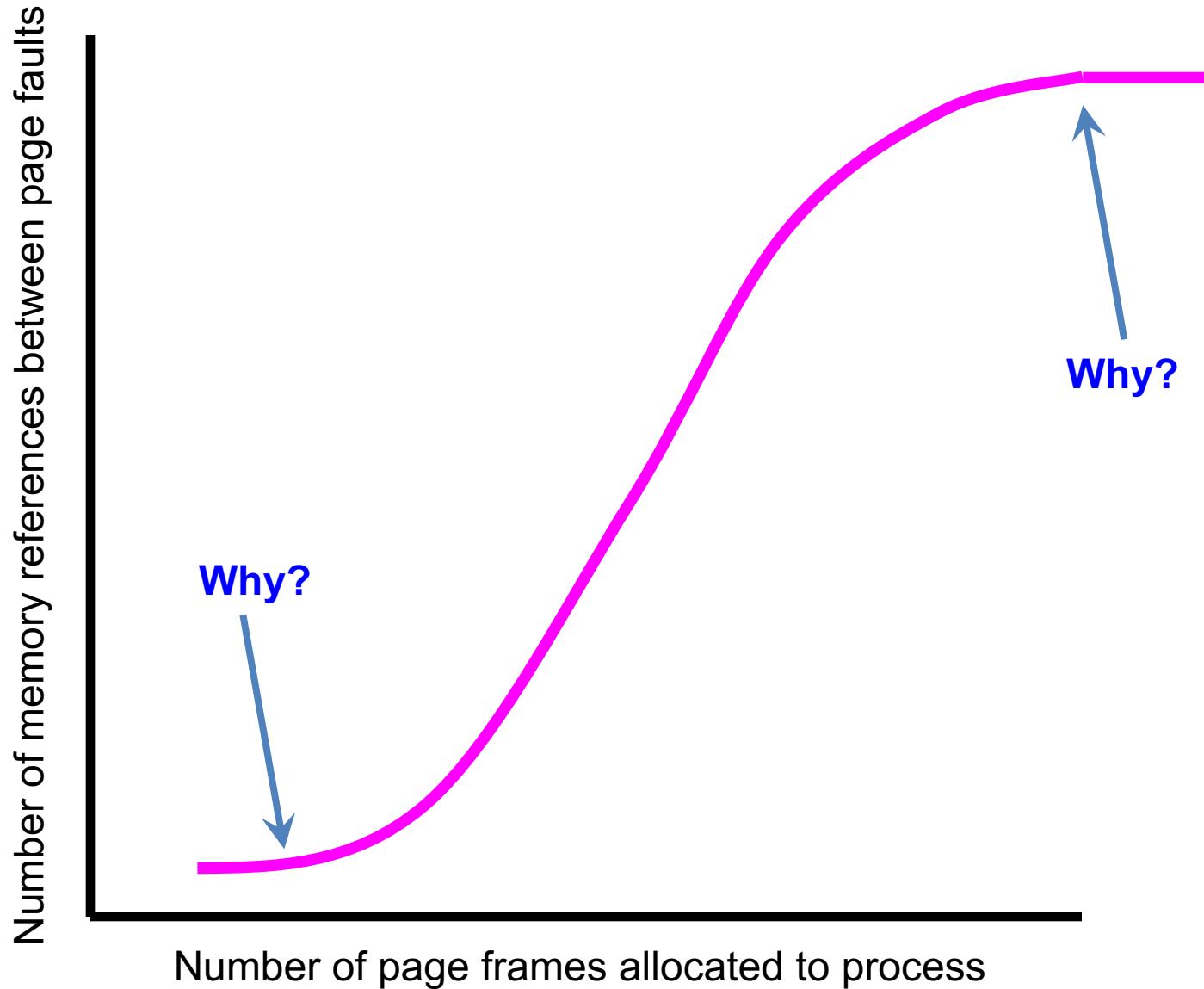
(b)

Allocation of frames among processes

- FIFO and LRU Clock each can be implemented as either **local** or **global** replacement algorithms
 - local
 - each process is given a limit of pages it can use
 - it “pages against itself” (evicts its own pages)
 - global
 - the “victim” is chosen from among all page frames, regardless of owner
 - processes’ page frame allocation can vary dynamically
- Issues with local replacement?
 - poor utilization of free page frames, long access time
- Issues with global replacement?
 - Linux uses global replacement: global thrashing

The *working set model* of program behavior

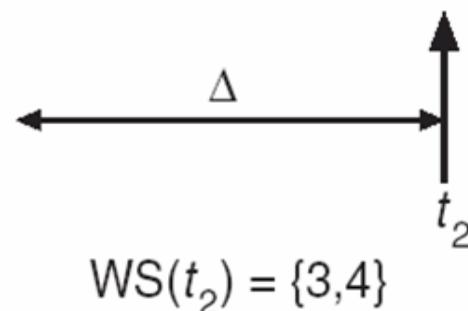
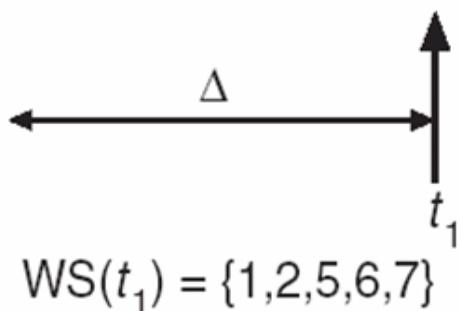
- **Working set** of a process is used to model the dynamic locality of its memory usage
 - working set = set of pages process currently “needs”
 - formally defined by Peter Denning in the 1960’s
- Definition:
 - $WS(t,w) = \{ \text{pages } P \text{ such that } P \text{ was referenced in the time interval } (t, t-w) \}$
 - t : time
 - w : working set *window* (measured in page refs)
 - a page in WS only if it was referenced in the last w references
- Working set varies over the life of the program
 - so does the **working set size**



Example: Working set

page reference table

... 2 6 1 5 7 7 7 5 1 6 2 3 4 1 2 3 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



Working set size

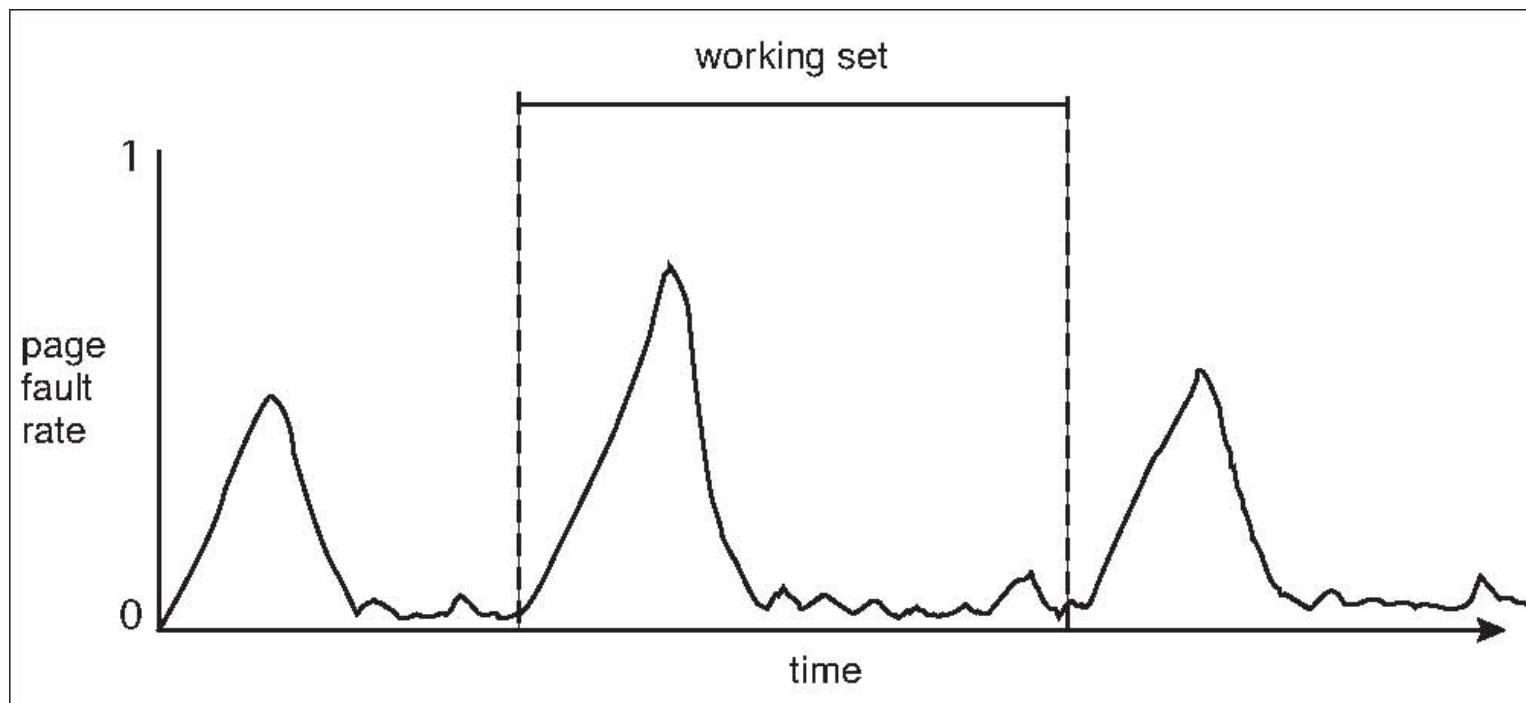
- The working set size, $|WS(t,w)|$,
 - Changes with program locality
- During periods of poor locality,
 - more pages are referenced
- Within that period of time,
 - the working set size is larger
- Intuitively, the working set must be in memory,
 - otherwise you'll experience heavy faulting
 - **thrashing**

Hypothetical Working Set algorithm

- Estimate $|WS(0,w)|$ for a process
 - Allow that process to start only if you can allocate it that many page frames
- Use a local replacement algorithm (LRU Clock?)
 - make sure that the working set are occupying the process's frames
- Track each process's working set size,
 - and re-allocate page frames among processes dynamically
- Problem
 - keep track of working set size.
- Use reference bit with a fixed-interval timer interrupt

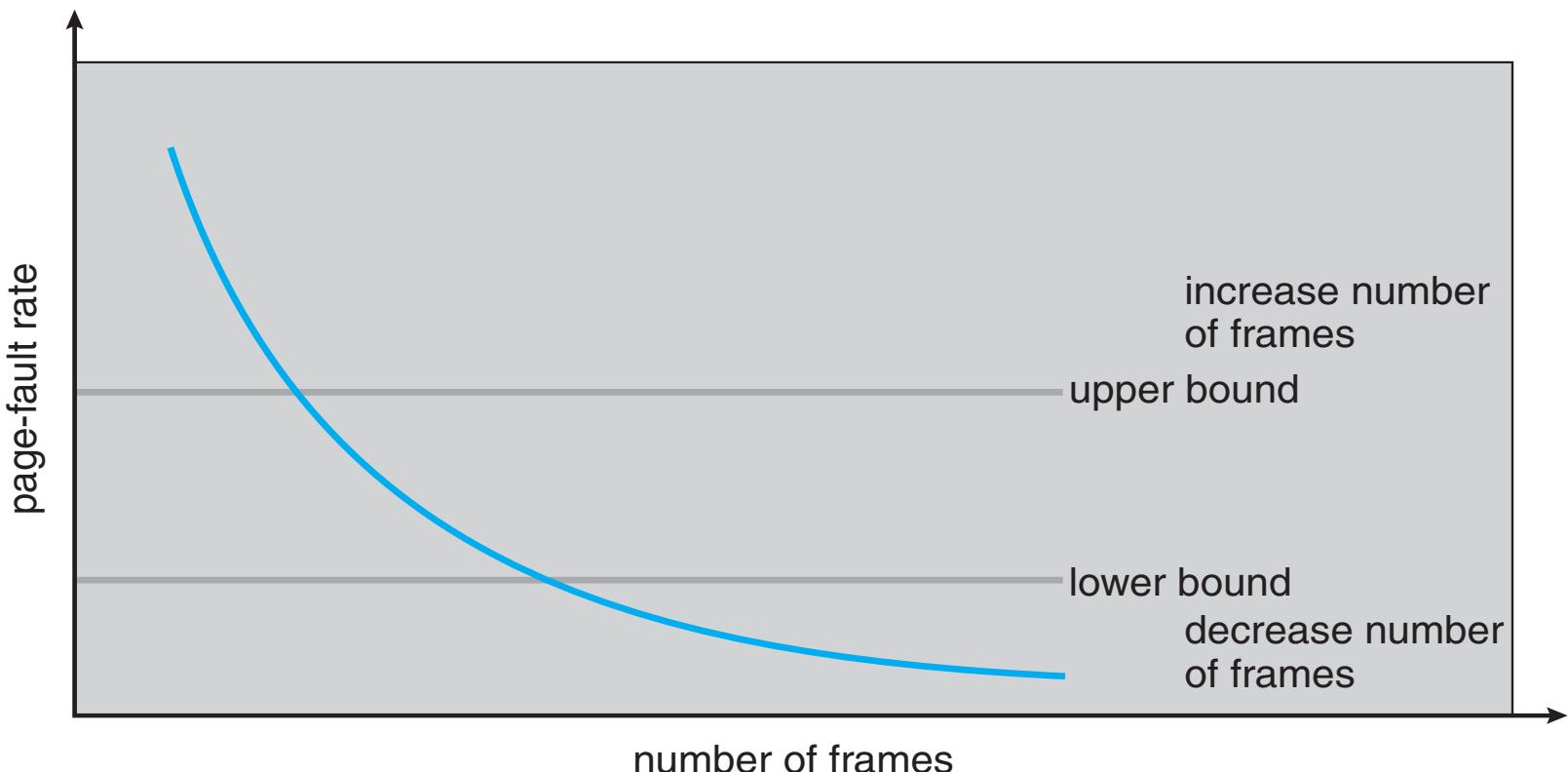
Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



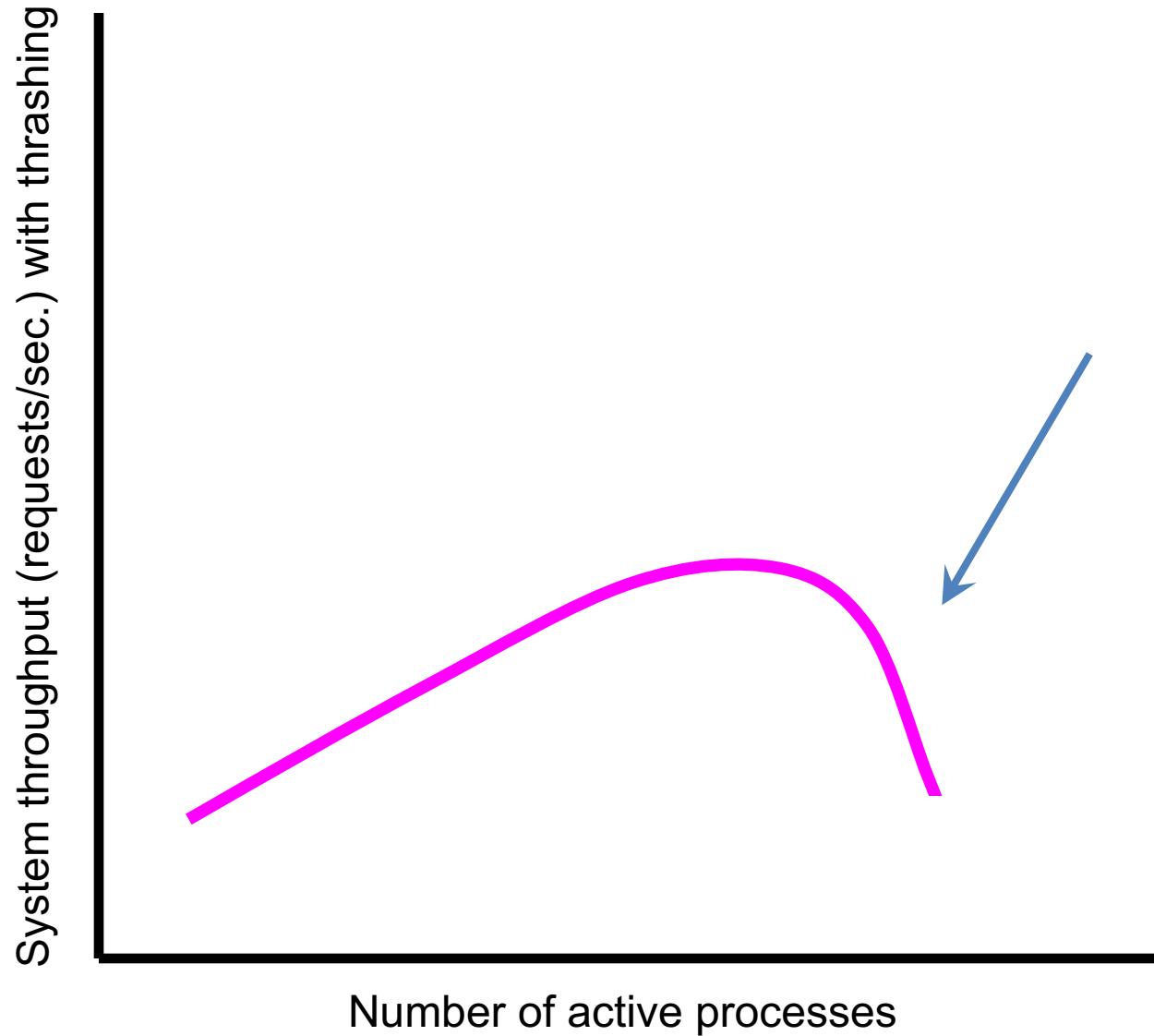
Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame



Thrashing

- Thrashing
 - when the system spends most of its time servicing page faults, little time doing useful work
- Could be that there is enough memory
 - but a poor replacement algorithm - incompatible with program behavior
- Could be that memory is over-committed
 - OS sees CPU poorly utilized and adds more processes
 - too many active processes
 - Makes problem worse



Summary

- Virtual memory
- Page faults
- Demand paging
 - don't try to anticipate
- Page replacement
 - Belady, LRU, Clock,
 - local, global
- Locality
 - temporal, spatial
- Working set
- Thrashing

Operating Systems

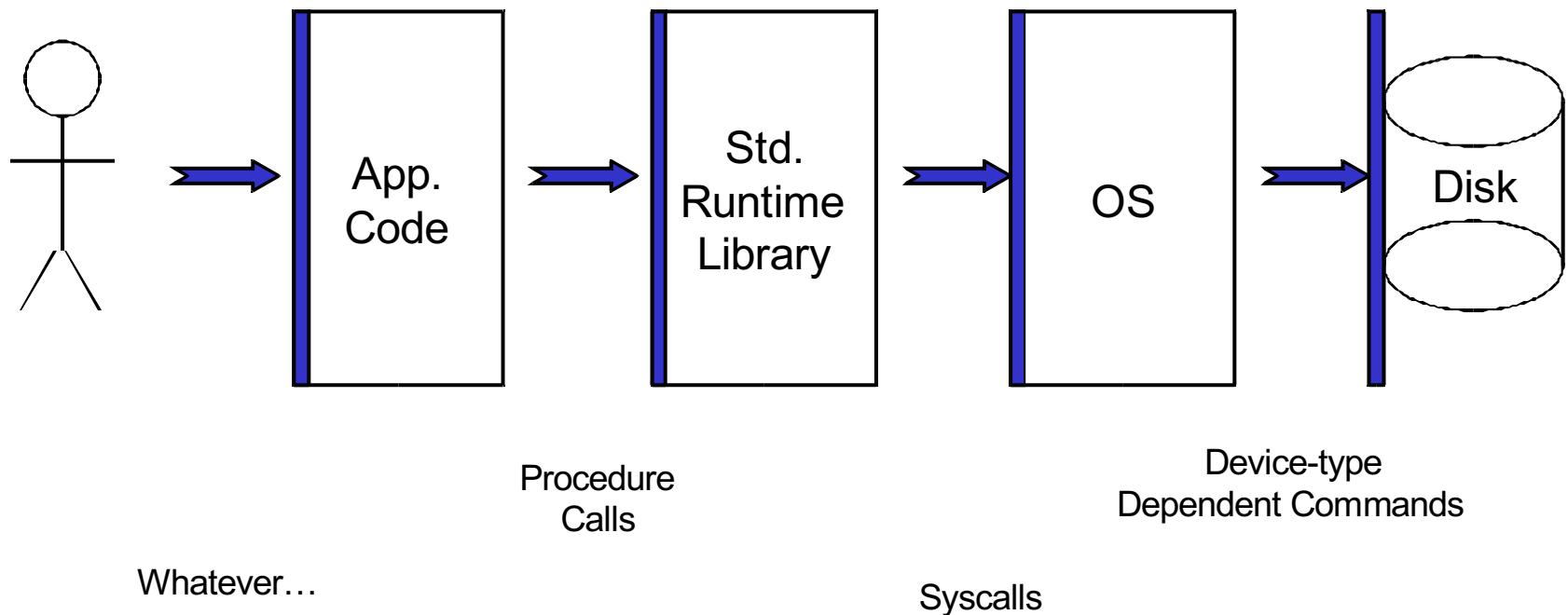
File Systems

Lecture 13
Michael O'Boyle

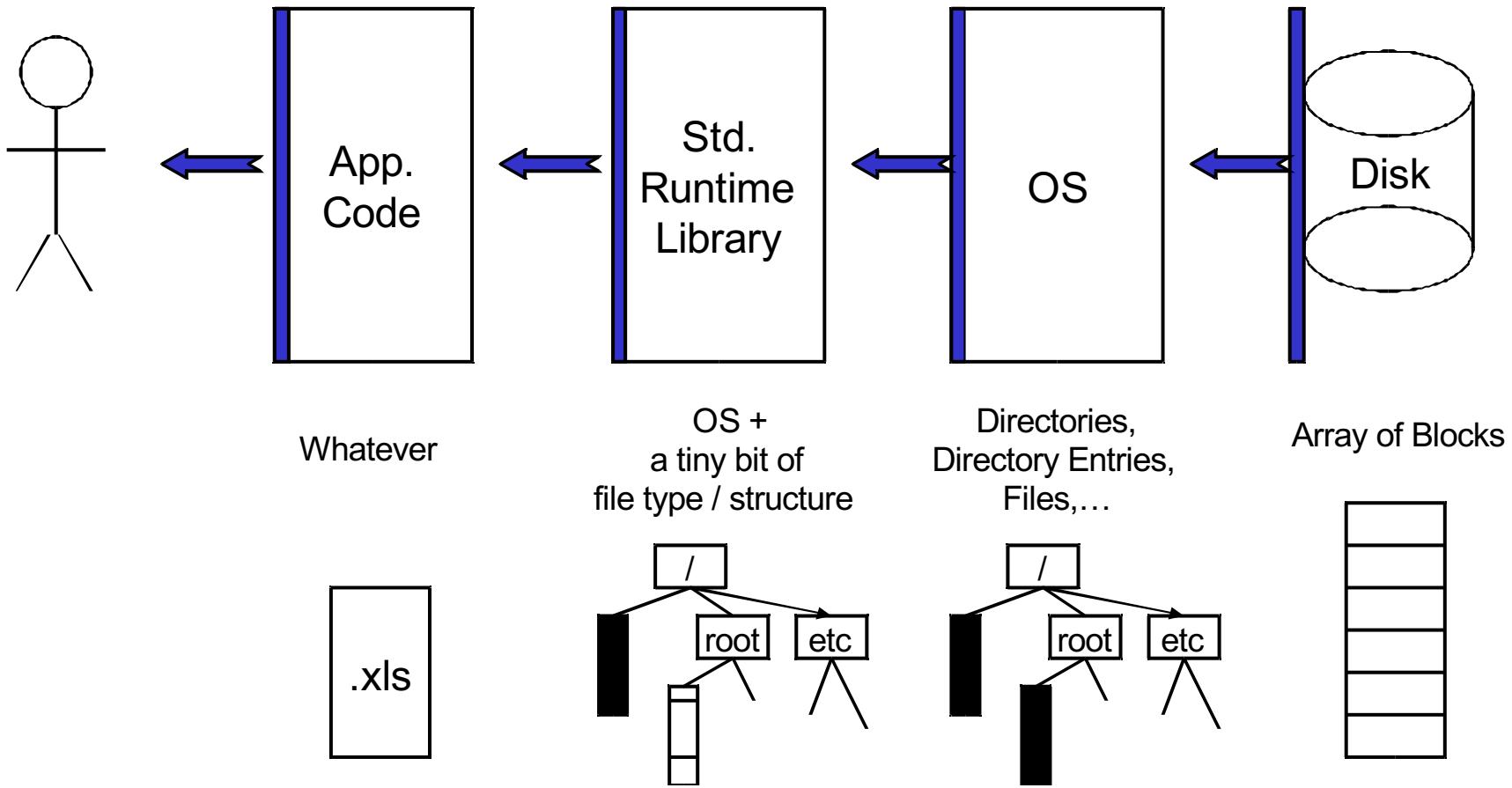
Overview

- Roles
- Files
- Directories
- File Protection
- Unix
 - inodes

Interface Layers

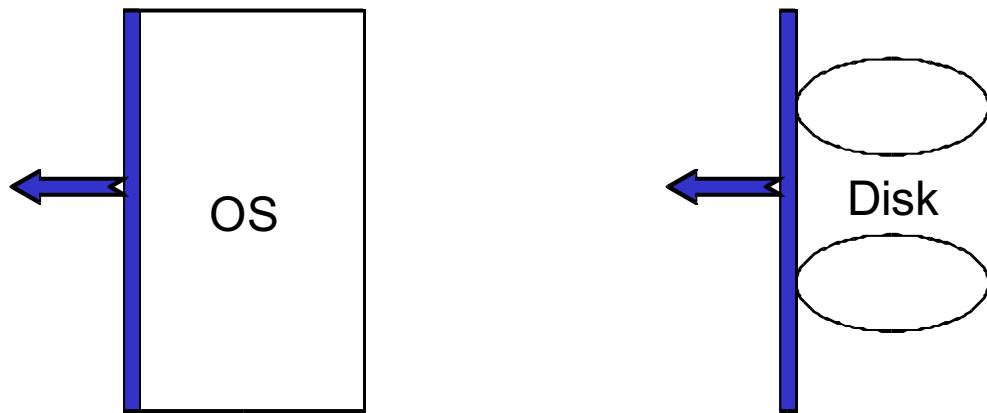


Exported Abstractions



Primary Roles of the OS (file system)

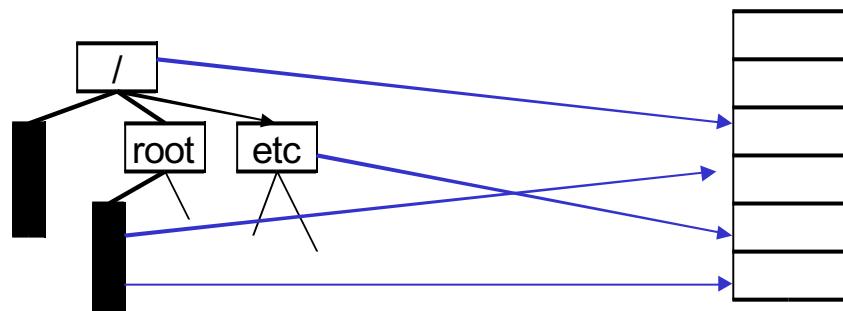
1. Hide hardware specific interface
2. Allocate disk blocks
3. Check permissions
4. Understand directory file structure
5. Maintain *metadata*
6. Performance
7. Flexibility



Why does the OS define directories?

Why not leave that to the library/application layer?

(Why would you want to leave it to the app/library?)



File systems

- The concept of a file system is simple
- The implementation of the abstraction for secondary storage
 - abstraction = files
- Logical organization of files into directories
 - the directory hierarchy
- Sharing of data between
 - processes, people and machines
- Access control, consistency, ...

Files

- A collection of data with some properties
 - contents, size, owner, last read/write time, protection ...
- Files types
 - understood by file system
 - device, directory, symbolic link
 - understood by other parts of OS or by runtime libraries
 - executable, dll, source code, object code, text file, ...
- Type can be encoded in the file's name or contents
 - windows encodes type in name
 - .com, .exe, .bat, .dll, .jpg, .mov, .mp3, ...
- Old Mac OS stored the name of the creating program along with the file
- Unix - initial characters (e.g., #)

Basic operations

Unix

- `create(name)`
- `open(name, mode)`
- `read(fd, buf, len)`
- `write(fd, buf, len)`
- `sync(fd)`
- `seek(fd, pos)`
- `close(fd)`
- `unlink(name)`
- `rename(old, new)`

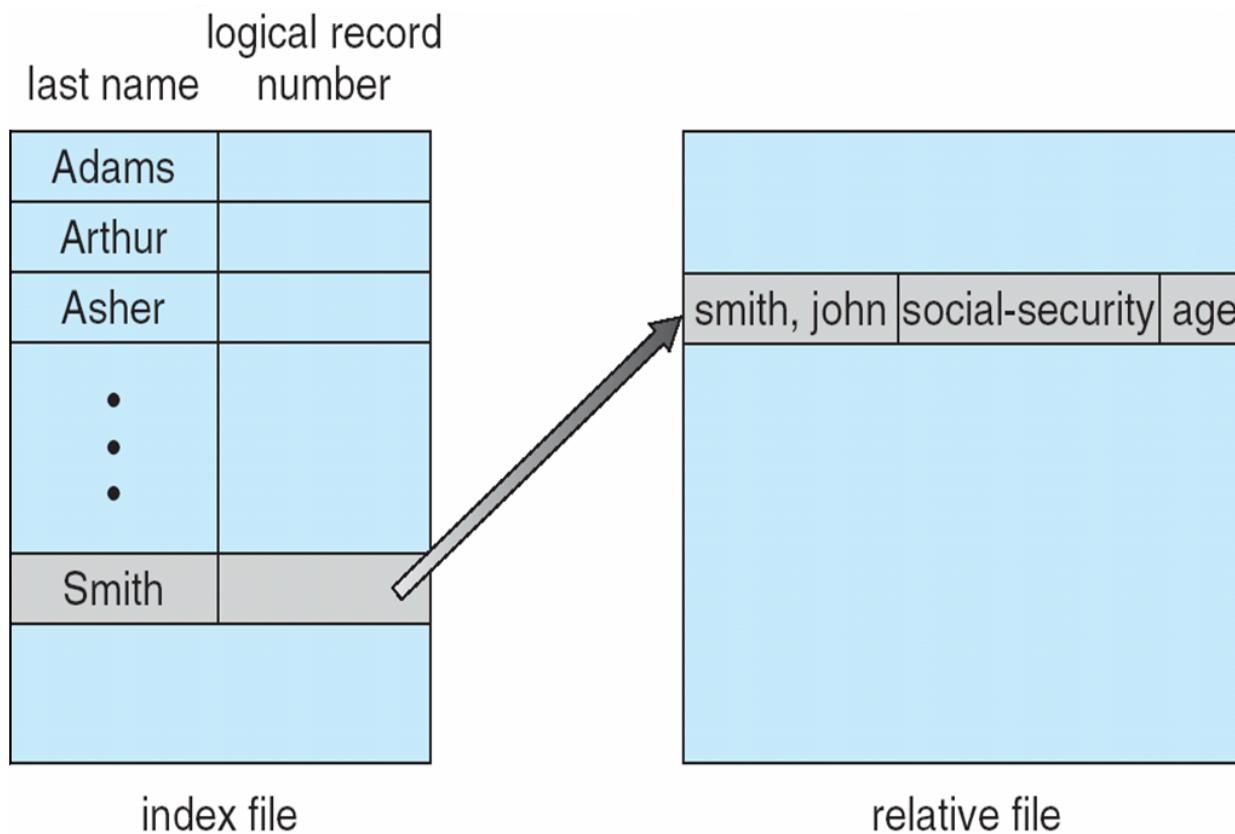
Windows

- `CreateFile(name, CREATE)`
- `CreateFile(name, OPEN)`
- `ReadFile(handle, ...)`
- `WriteFile(handle, ...)`
- `FlushFileBuffers(handle, ...)`
- `SetFilePointer(handle, ...)`
- `CloseHandle(handle, ...)`
- `DeleteFile(name)`
- `CopyFile(name)`
- `MoveFile(name)`

File access methods

- File systems provide different **access methods**
 - sequential access
 - read bytes one at a time, in order
 - direct access
 - random access given a block/byte #
 - record access
 - file is array of fixed- or variable-sized records
 - indexed access
 - One file contains an index to a record in another file
 - apps can find a file based on value in that record (similar to DB)
- Why do we care about distinguishing sequential from direct access?
 - what might the FS do differently in these cases?

Example of Index and Relative Files



Directories

- Directories provide:
 - a way for users to organize their files
 - a convenient file name space for both users and FS's
- Most file systems support multi-level directories
 - naming hierarchies (/, /usr, /usr/local, /usr/local/bin, ...)
- Most file systems support the notion of current directory
- Absolute names: fully-qualified starting from root of FS

```
bash$ cd /usr/local
```
- Relative names: specified with respect to current directory

```
bash$ cd /usr/local    (absolute)
bash$ cd bin            (relative, equivalent to cd /usr/local/bin)
```

Directory internals

- A directory is typically just a file that happens to contain special metadata
- Directory
 - list of (name of file, file attributes)
- Attributes include such things as:
 - size, protection, location on disk, creation time, access time, ...
- The directory list is usually unordered (effectively random)
 - when you type “ls”, the “ls” command sorts the results for you

Path name translation

- You want to open “/one/two/three”

```
fd = open("/one/two/three", O_RDWR);
```

- Inside the file system

- open directory “/” (well known, can always find)
 - search the directory for “one”, get location of “one”
 - open directory “one”, search for “two”, get location of “two”
 - open directory “two”, search for “three”, get loc. of “three”
 - open file “three”
 - (of course, permissions are checked at each step)

- FS spends much time walking down directory paths
 - this is why open is separate from read/write (session state)
 - OS will cache prefix lookups to enhance performance
 - /a/b, /a/bb, /a/bbb all share the “/a” prefix

File protection

- FS implements a protection system
 - to control who can access a file (user)
 - to control how they can access it (e.g., read, write, or exec)
- Often generalised:
 - generalize files to **objects** (the “what”)
 - generalize users to **principals** (the “who”, user or program)
 - generalize read/write to **actions** (the “how”, or operations)
- Protection system dictates
 - whether a given action
 - performed by a given principal
 - on a given object should be allowed
- E.g., you can read or write your files, but others cannot
- E.g., you can read /group/teaching/cs3 but you cannot write to it

Model for representing protection

- Two different ways of thinking about it:
 - access control lists (ACLs)
 - for each object, keep list of principals and principals' allowed actions
 - capabilities
 - for each principal, keep list of objects and principal's allowed actions
- Both can be represented with the following matrix:

		objects		
	/etc/passwd	/home/gribble	/home/guest	
principals	rw	rw	rw	
gribble	r	rw	r	
guest			r	capability

ACL

ACLs vs. Capabilities

- Capabilities are easy to transfer
 - they are like keys: can hand them off
 - they make sharing easy
- ACLs are easier to manage
 - object-centric, easy to grant and revoke
 - to revoke capability, need to keep track of principals that have it
 - hard to do, given that principals can hand off capabilities
- ACLs grow large when object is heavily shared
 - can simplify by using “groups”
 - put users in groups, put groups in ACLs
- Additional benefit
 - change group membership, affects ALL objects that have this group in its ACL

The original Unix file system

- Dennis Ritchie and Ken Thompson, Bell Labs, 1969
- “UNIX rose from the ashes of a multi-organizational effort in the early 1960s to develop a dependable timesharing operating system” – Multics
- Designed for a “workgroup” sharing a single system
- Did its job well
 - Although it has been stretched in many directions

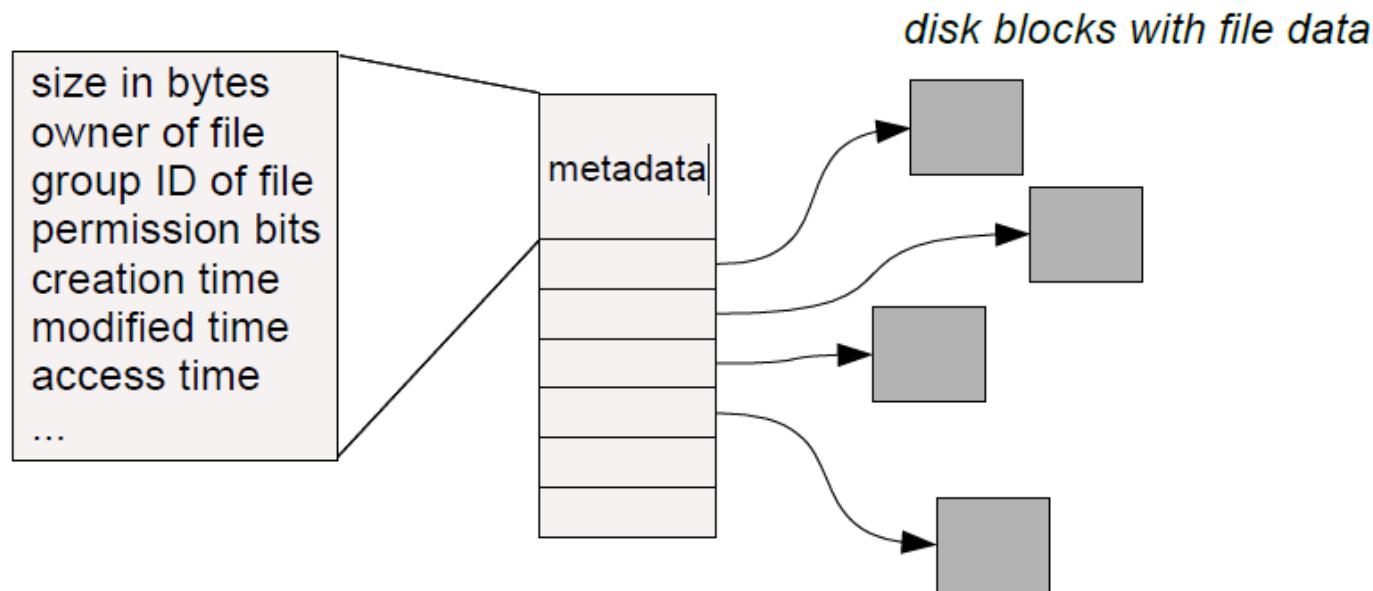


(Old) Unix disks are divided into five parts ...

- Boot block
 - can boot the system by loading from this block
- Superblock
 - specifies boundaries of next 3 areas, and contains head of freelists of inodes and file blocks
- inode area
 - contains descriptors (inodes) for each file on the disk; all inodes are the same size; head of freelist is in the superblock
- File contents area
 - fixed-size blocks; head of freelist is in the superblock
- Swap area
 - holds processes that have been swapped out of memory

Basic file system structures

- Every file and directory is represented by an inode
 - Stands for “index node”
- Contains two kinds of information:
 - 1) Metadata describing the file's owner, access rights, etc.
 - 2) Location of the file's blocks on disk



inode format

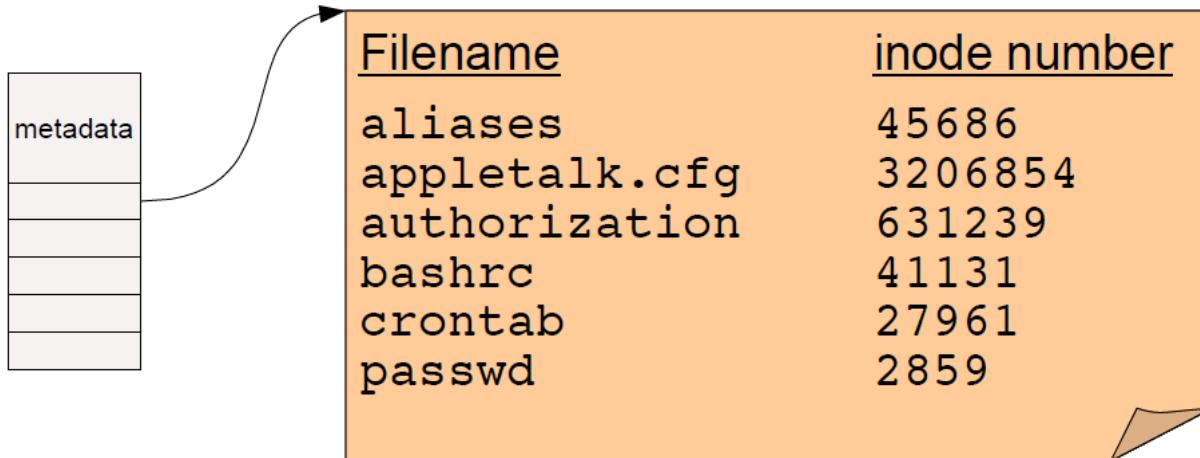
- User number
- Group number
- Protection bits
- Times (file last read, file last written, inode last written)
- File code: specifies if the inode represents a directory, an ordinary user file, or a “special file” (typically an I/O device)
- Size: length of file in bytes
- Block list: locates contents of file (in the file contents area)
- Link count: number of directories referencing this inode

The flat (inode) file system

- Each file is known by a number, which is the number of the inode
 - 1, 2, 3, etc.!
- Files are created empty, and grow when extended through writes

The tree (directory, hierarchical) file system

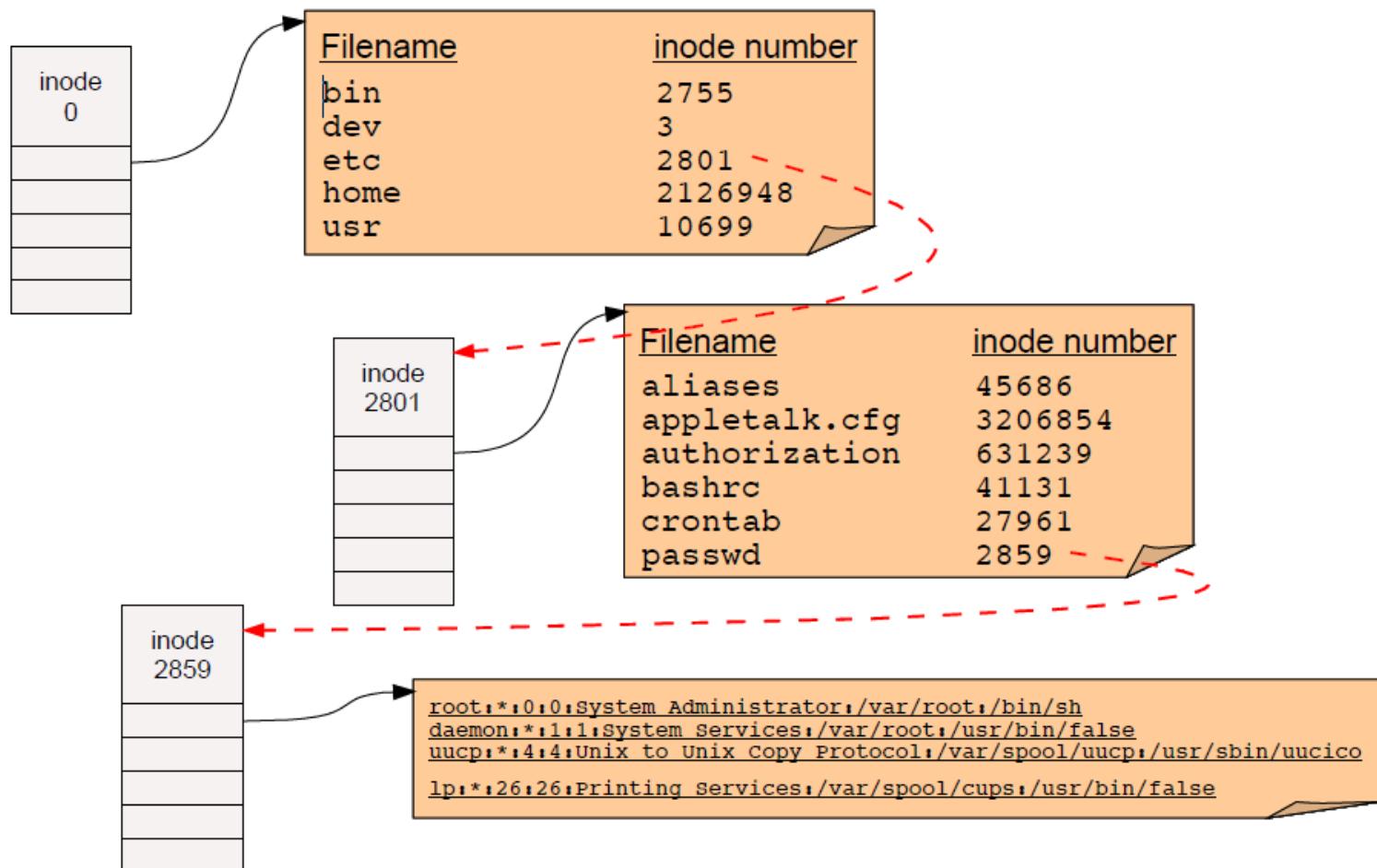
- A directory is a flat file of fixed-size entries
- Each entry consists of an inode number and a file name



- These are the contents of the directory “file data” itself – NOT the directory's inode
- Filenames (in UNIX) are not stored in the inode at all
- Special inodes:
 - inode 2 is the root directory
 - Inode 1 – a hidden file containing all bad blocks

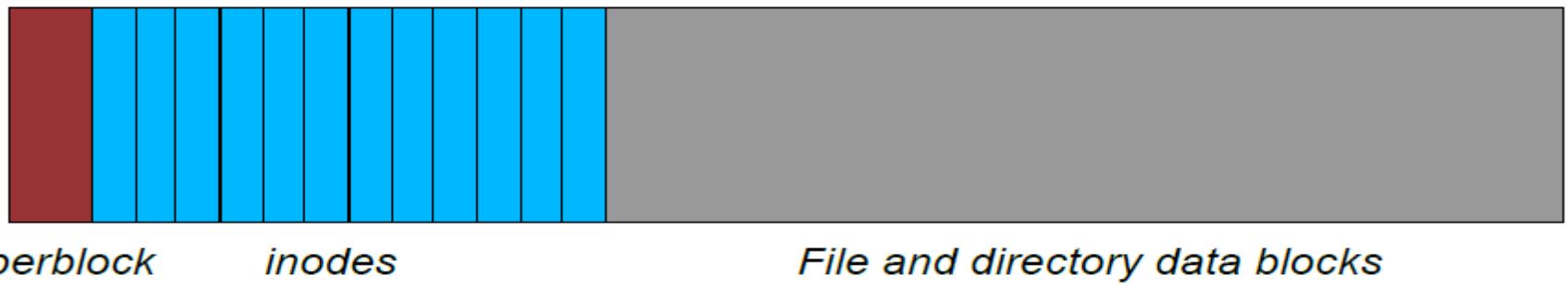
Pathname resolution

- To look up a pathname “/etc/passwd”, start at root directory and walk down chain of inodes...



Locating inodes on disk

- Directories give the **inode number** of a file.
 - How do we find the inode itself on disk?
- Basic idea: Top part of filesystem contains *all* of the inodes



- inode number is just the “index” of the inode
- Easy to compute the block address of a given inode:
 - $\text{block_addr(inode_num)} = \text{block_offset_of_first_inode} + (\text{inode_num} * \text{inode_size})$
 - This implies that a filesystem has a fixed number of potential inodes
 - This number is generally set when the filesystem is created
 - The superblock stores important metadata on filesystem layout, list of free blocks, etc.

Directory issues

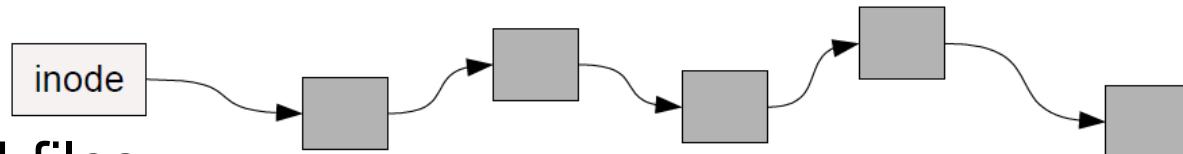
- Directories map filenames to inode numbers.
- We can create multiple pointers to the same inode in different directories
 - Or even the same directory with different filenames
- In UNIX this is called a “hard link” and can be done using “ln”

```
bash$ ls -i /home/foo  
287663 /home/foo      (This is the inode number of “foo”)  
bash$ ln /home/foo /tmp/foo  
bash$ ls -i /home/foo /tmp/foo  
287663 /home/foo  
287663 /tmp/foo
```

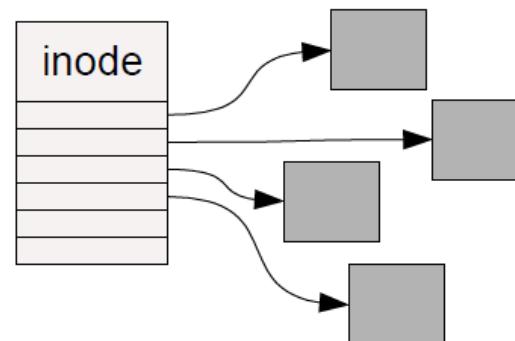
- “/home/foo” and “/tmp/foo” now refer to the same file on disk
 - Not a copy! You will always see identical data no matter which filename you use to read or write the file.

How should we organize blocks on a disk?

- Very simple policy: A file consists of linked blocks
 - inode points to the first block of the file
 - Each block points to the next block in the file (just a linked list on disk)

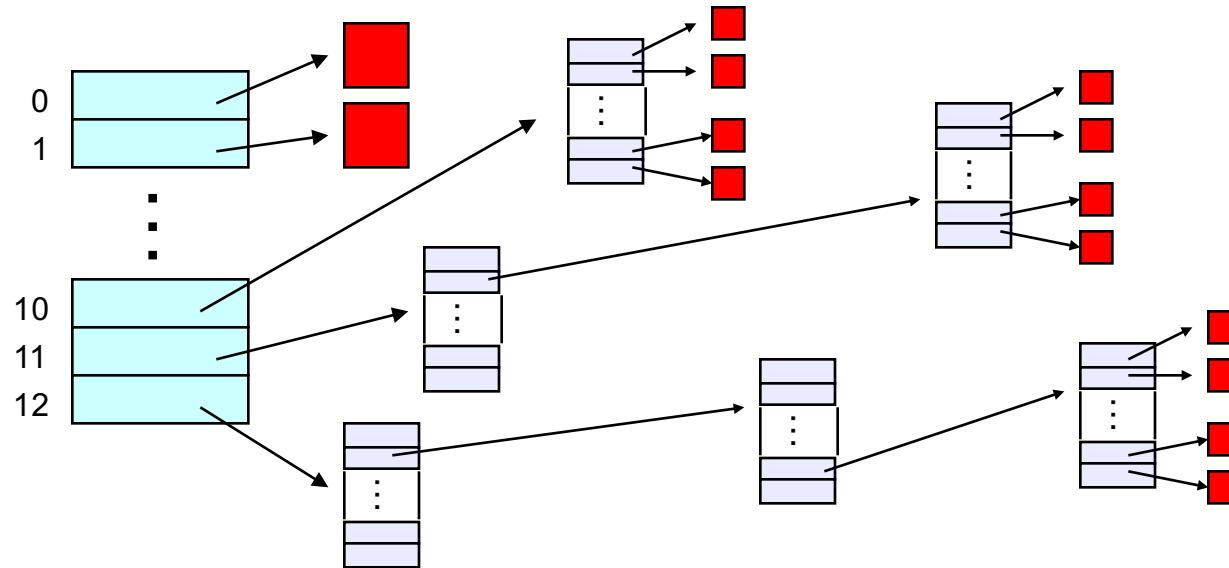


- Indexed files
 - inode contains a list of block numbers containing the file
 - Array is allocated when the file is created



The “block list” portion of the inode (Unix Version 7)

- Points to blocks in the file contents area
- Must be able to represent very small and very large files.
- Each inode contains 13 block pointers
 - first 10 are “direct pointers” (pointers to 512B blocks of file data)
 - then, single, double, and triple indirect pointers



File Size

- Only occupies $13 \times 4B$ in the inode
- Can get to $10 \times 512B = a 5120B$ file directly
 - (10 direct pointers, blocks in the file contents area are 512B)
- Can get to $128 \times 512B = an additional 65KB$ with a single indirect reference
 - (the 11th pointer in the inode gets you to a 512B block in the file contents area that contains 128 4B pointers to blocks holding file data)
- Can get to $128 \times 128 \times 512B = an additional 8MB$ with a double indirect reference
 - (the 12th pointer in the inode gets you to a 512B block in the file contents area that contains 128 4B pointers to 512B blocks in the file contents area that contain 128 4B pointers to 512B blocks holding file data)

File Size

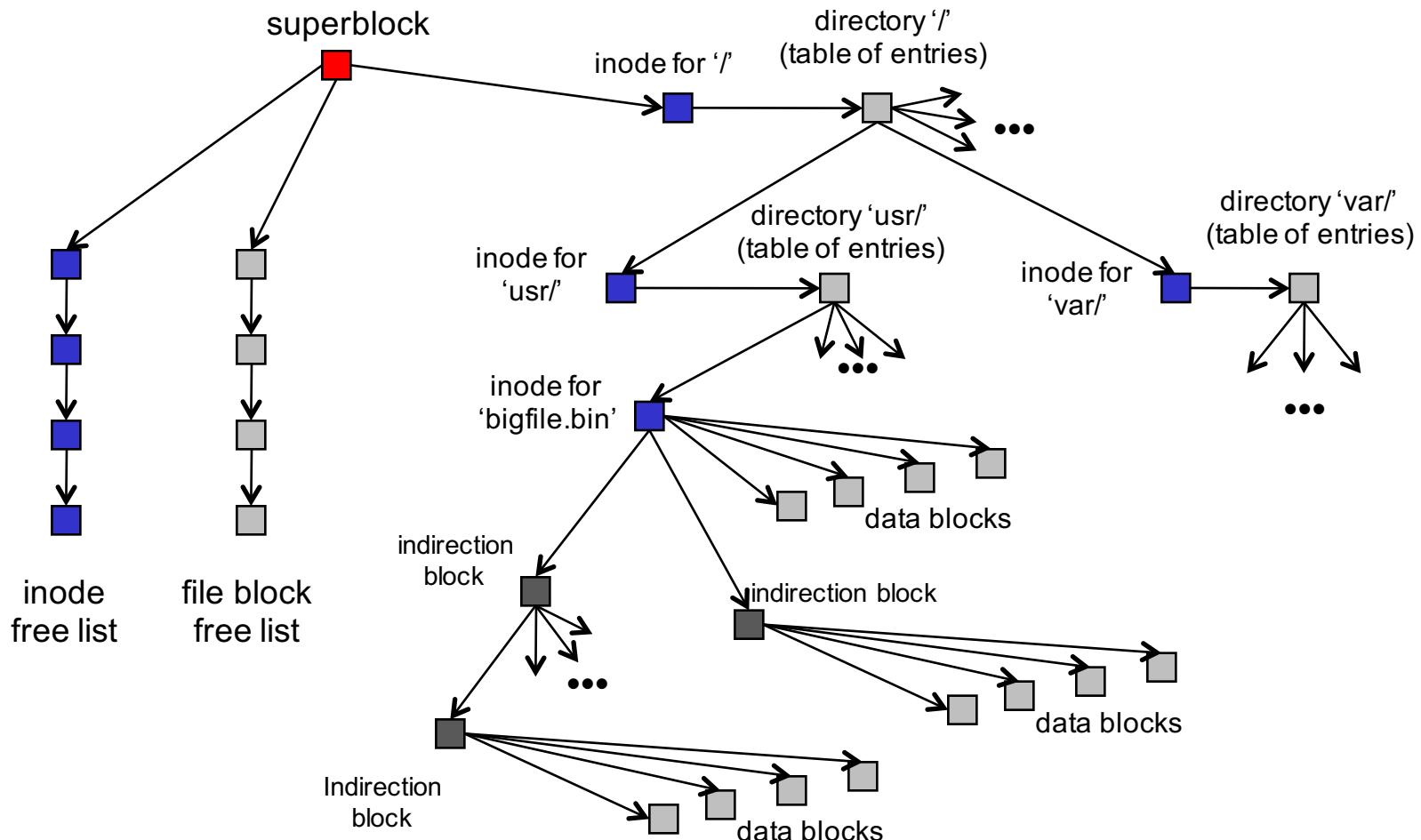
- Can get to $128 \times 128 \times 128 \times 512B =$ an additional 1GB with a triple indirect reference
 - (the 13th pointer in the inode gets you to a 512B block in the file contents area that contains 128 4B pointers to 512B blocks in the file contents area that contain 128 4B pointers to 512B blocks in the file contents area that contain 128 4B pointers to 512B blocks holding file data)
- Maximum file size is a little over 1GB +

Updated File Size

- A later version of Bell Labs Unix utilized 12 direct pointers rather than 10
- Berkeley Unix went to 1KB block sizes
 - What's the effect on the maximum file size?
 - $256 \times 256 \times 256 \times 1K = 17 \text{ GB} + \text{a smidge}$
- Subsequently went to 4KB blocks
 - $1K \times 1K \times 1K \times 4K = 4\text{TB}$

Putting it all together

- The file system is just a huge data structure



File system layout

- One important goal of a filesystem is to lay this data structure out on disk
 - keep in mind the physical characteristics of the disk
 - seeks are expensive
 - characteristics of the workload
 - locality across files within a directory,
 - sequential access to many files
- Old UNIX's layout is very inefficient
 - constantly seeking
 - between inode area and data block area
 - as traverse filesystem, or sequentially read files
- Newer file systems are smarter
- Newer storage devices (SSDs) change the constraints

Summary

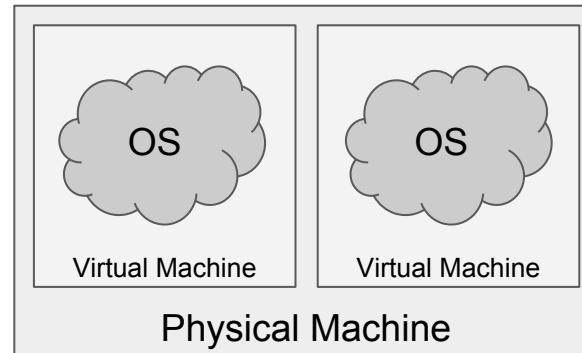
- Roles
- Files
- Directories
- File Protection
- Unix
 - Inodes
- Disks next

Virtualisation

Tom Spink

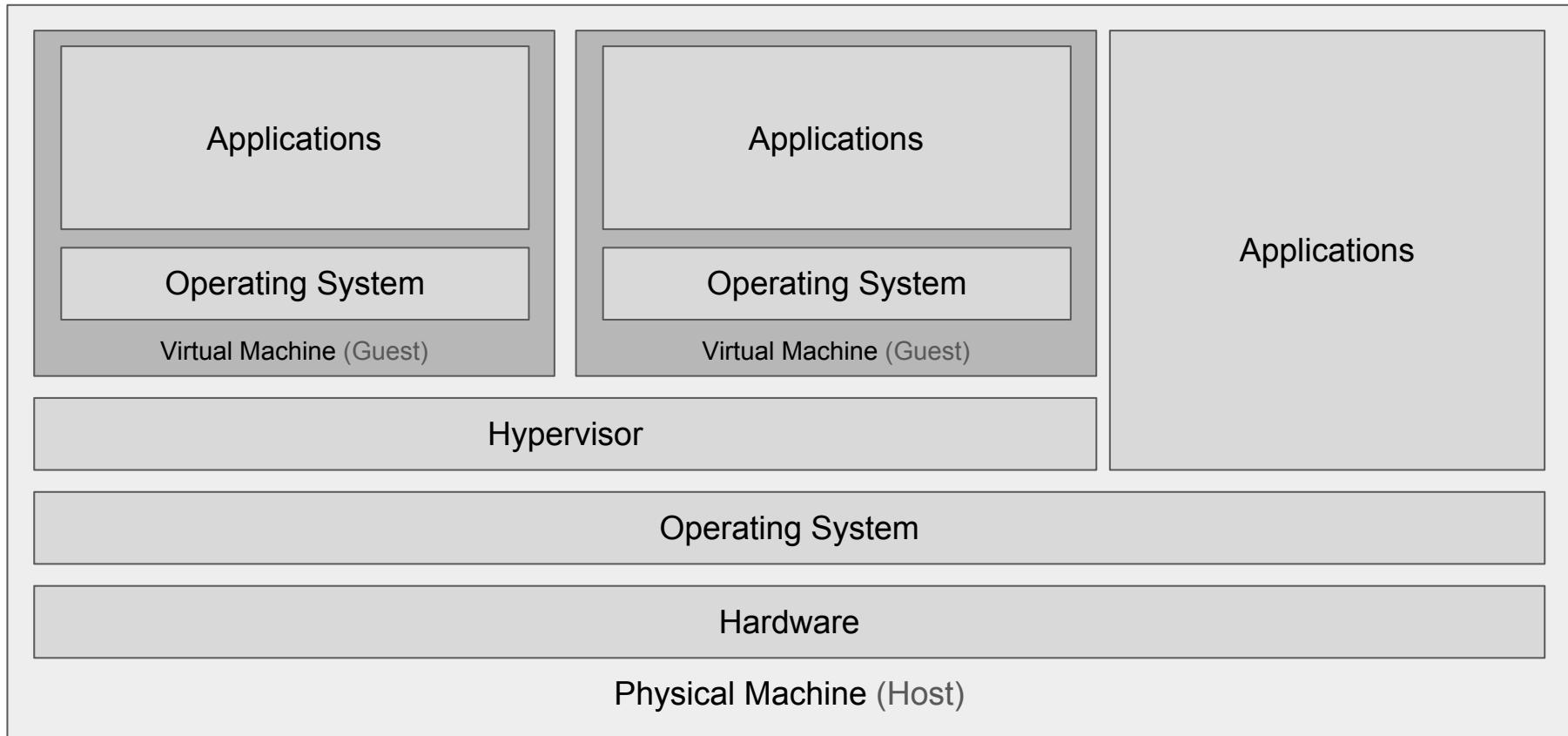
Introduction

- Virtualisation is the process of creating a **virtual** version of a **physical** object.
- In computing, **hardware virtualisation** is the process of creating a **virtual** version of real hardware.
- This virtual hardware can be used to run a complete operating system.



Terminology

- **Virtual Machine:** A virtual representation of a physical machine.
 - Not to be confused with a Java Virtual Machine or the CLR (.NET)
- **Virtual Machine Monitor** or **Hypervisor:** A software application that monitors and manages running virtual machines.
- **Host Machine:** The physical machine that a virtual machine is running on.
- **Guest Machine:** The virtual machine, running on the host machine.

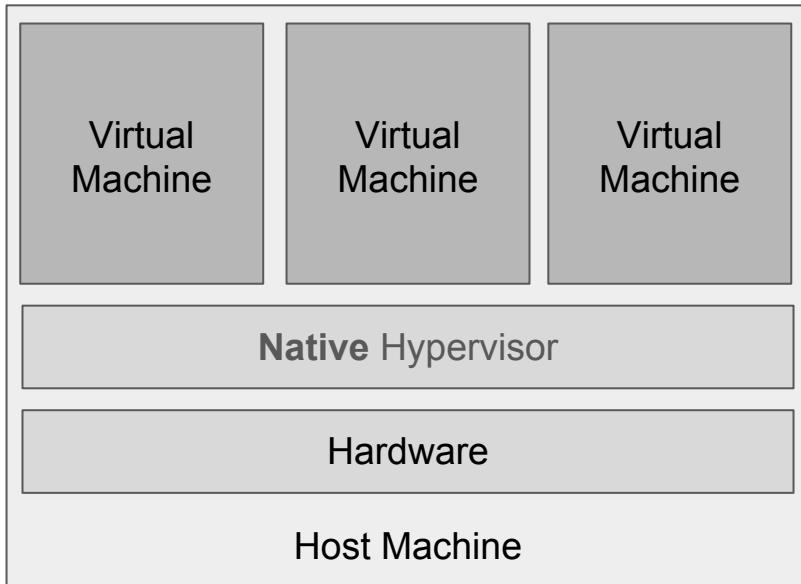


Virtual Machine Diagram

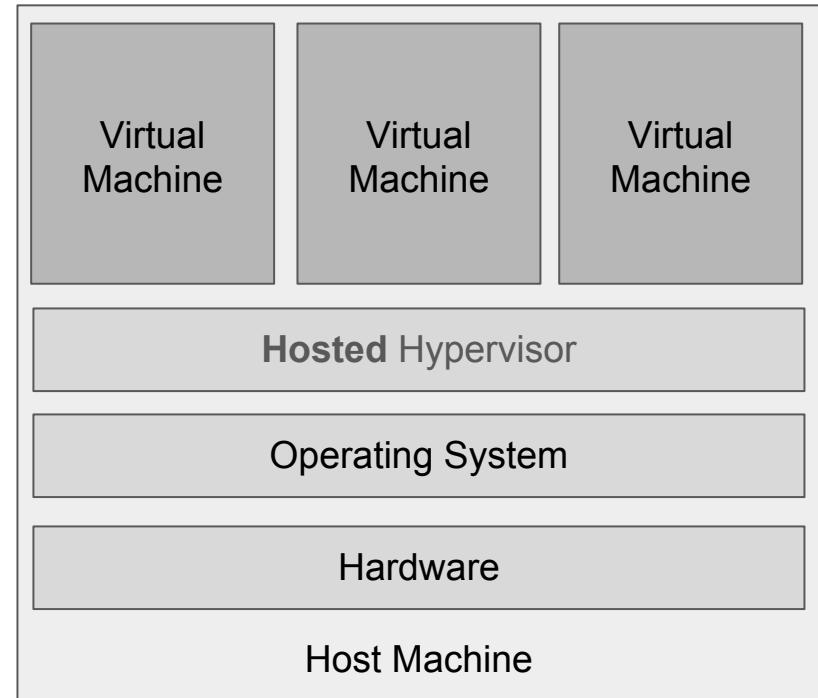
Virtual Machine Monitor (Hypervisor)

- The VMM is in charge of running the virtual machines.
- There are two main types of VMM:
 - **Type 1:** Native
 - **Type 2:** Hosted
- **Type 1: Native Hypervisors** run directly on the host machine, and share out resources (such as memory and devices) between guest machines.
 - e.g. XEN, Oracle VM Server
- **Type 2: Hosted Hypervisors** run as an application inside an operating system, and support virtual machines running as individual processes.
 - e.g. VirtualBox, Parallels Desktop, QEMU

Type 1 - Native



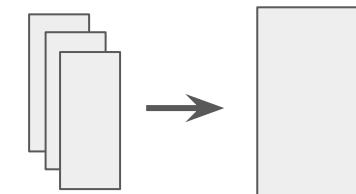
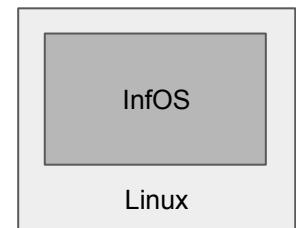
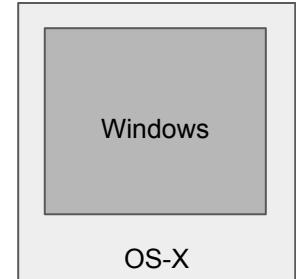
Type 2 - Hosted



Hypervisor Types

Uses of Virtualisation

- **Personal** (e.g. Parallels Desktop/VirtualBox)
 - Running multiple operating systems on one host, without the inconvenience of rebooting.
 - e.g. Running Windows inside OS X.
 - Some hypervisors support “seamless integration”.
- **Technical** (e.g. QEMU as used in the coursework)
 - Operating System/Hardware Design.
 - Kernel Debugging/Testing.
 - Prototyping new architectures/architectural features.
- **Commercial** (e.g. XEN/VMWare)
 - Data centre server consolidation.
 - High availability/Migration.



Many Servers One Big Server

Types of Virtualisation

- **Software Emulation**
 - Maximum flexibility for virtualisation, but very slow to run (high overhead).
 - Each guest instruction is emulated (can use binary translation for speed-up)
- **Containers/Namespace**
 - Isolate processes/groups of processes within a single operating system, e.g. Docker.
- **Full System or Hardware Virtualisation**
 - Isolate multiple operating systems from each other, within a single physical machine.
- **Same-architecture Virtualisation**
 - Guest Machine is the **same** architecture as the Host Machine, e.g. Intel x86 on Intel x86.
- **Cross-architecture Virtualisation**
 - Guest Machine has a **different** architecture than the Host Machine, e.g. ARM on Intel x86.
 - Must use software emulation to do this.

Popek and Goldberg Requirements for Virtualisation

Paper published in 1974 [1] that laid the foundations for hardware virtualisation, and formalised the requirements for an architecture to be “virtualisable”.

Three main properties for a virtual machine:

1. Efficiency

- The majority of guest instructions are executed directly on the host machine.

2. Resource Control

- The virtual machine monitor must remain in control of all machine resources.

3. Equivalence

- The virtual machine must behave in a way that is indistinguishable from if it was running as a physical machine.

[1] Gerald J. Popek and Robert P. Goldberg. 1974. Formal requirements for virtualizable third generation architectures. *Commun. ACM* 17, 7 (July 1974), 412-421.
DOI: <http://dx.doi.org/10.1145/361011.361073>

Efficiency

- “All innocuous instructions are executed by the hardware directly, with no intervention at all on the part of the control program.”

Normal guest machine instructions should be executed directly on the processor. System instructions need to be emulated by the VMM.

Resource Control

- “It must be impossible for that arbitrary program to affect the system resources, i.e. memory, available to it; the allocator of the control program is to be invoked upon any attempt.”

The virtual machine should not be able to affect the host machine in any adverse way. The host machine should remain in control of all physical resources, sharing them out to guest machines.

Equivalence

- “Any program K executing with a control program resident, with two possible exceptions, performs in a manner indistinguishable from the case when the control program did not exist and K had whatever freedom of access to privileged instructions that the programmer had intended.”

A formal way of saying that the operating system running on a **virtual** machine should believe it is running on a **physical** machine, i.e. the behaviour of the **virtual** machine (from the **guest OS**' point of view) is identical to that of the corresponding **physical** machine.

The two exceptions mentioned are: **temporal latency** (some instruction sequences will take longer to run) and **resource availability** (physical machine resources are shared between virtual machines).

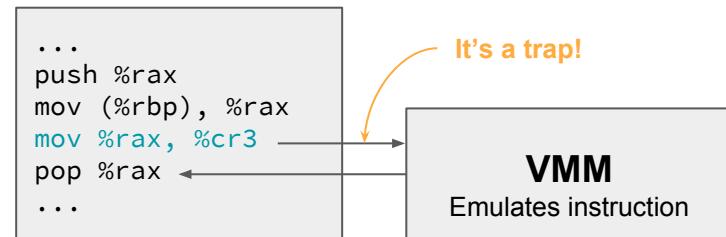
Methods of Virtualisation

- **Full Software Emulation**

- Not permitted by Popek and Goldberg because it violates the efficiency property.
 - Although, this no longer holds due to the advent of **efficient binary translation**.
- Required for **cross-architecture virtualisation**, as guest instructions cannot execute natively on the host.

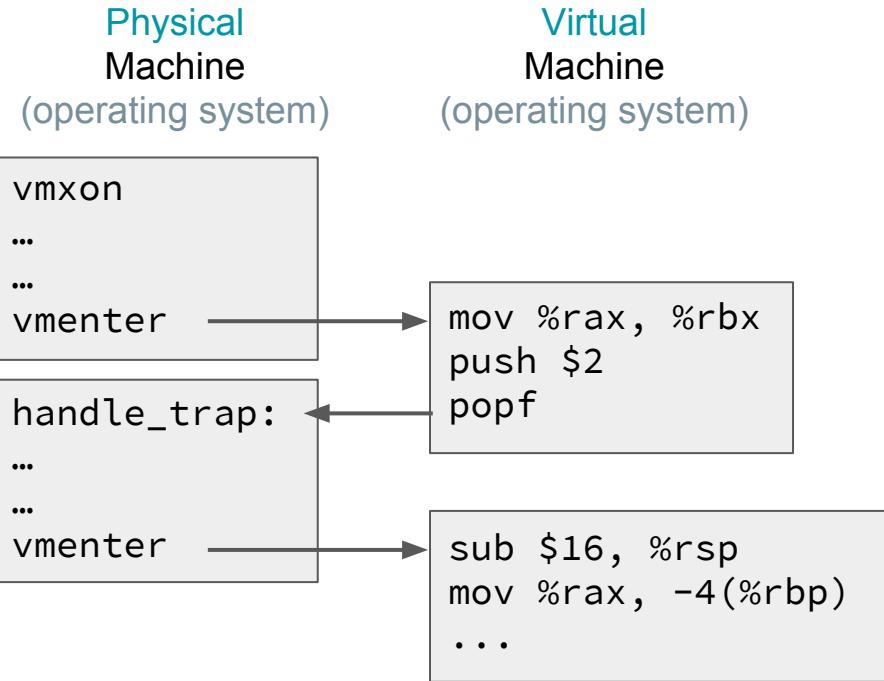
- **Trap-and-Emulate**

- The guest operating system runs “de-privileged”, all non-privileged instructions execute natively on the host.
- All privileged instructions trap to the VMM.
- VMM emulates these privileged operations.
- Guest resumes execution after emulation.

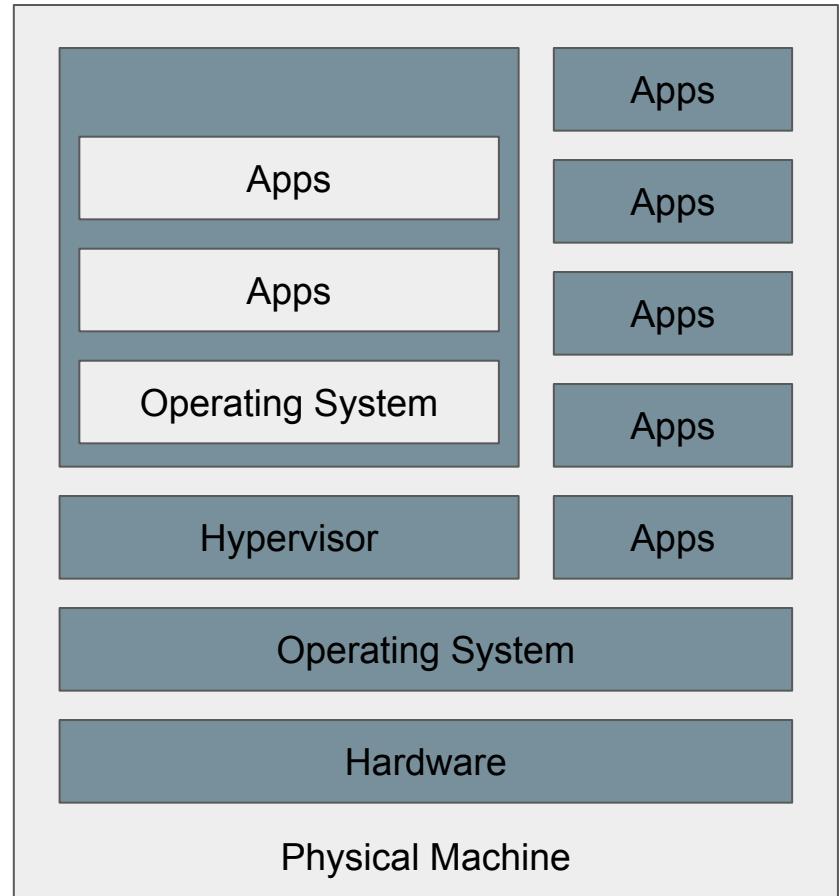


Virtualising x86

- Originally x86 was not “classically” virtualisable.
 - Some privileged instructions did not “trap”, and so could not be emulated correctly.
- Interpretation is too slow (violates efficiency)
- Code Patching leaves traces of virtualisation (violates equivalency)
- Binary Translation is better, but still incurs overhead.
- Since 2005, x86 processors now support virtualisation in hardware.
 - Intel-VT
 - AMD-V
- This enables trap-and-emulate style virtualisation.
- Unmodified operating systems can run natively on host machines.



Virtualising x86 on Modern Hardware



Hardware Acceleration for Virtualisation

- Modern processors include hardware support for running virtual machines.
 - Intel VT-X and AMD-V for x86 processors.
 - ARM Virtualization Extensions for ARM processors.
- Hardware extensions allow all guest instructions (including system instructions) to run natively on the processor.
- This works by providing an isolated view of the processor to virtual machines.
- Operating Systems can then run directly on the processor, believing they are running on **physical** hardware.
- Certain privileged operations “trap” back to the hypervisor.

Virtual Machine Access to Resources

- Virtual Machines need to be given access to resources such as:
 - Memory
 - Storage
 - Networking
 - Graphics
- It is the responsibility of the VMM to share out these resources.
- Access to physical memory is managed by the VMM.
- For an **unmodified** operating system, expecting a “real” storage device (such as a hard disk), the VMM must provide an **emulation** of that device.
- Some devices may be passed straight through to the virtual machine, e.g. dedicated **network cards**.

Paravirtualisation

- Guest operating systems are **aware** they are being **virtualised**.
- They **co-operate** with the hypervisor to enable increased memory and device performance.
- They no longer “trap-and-emulate”, but instead request privileged operations directly from the hypervisor.
- They can **co-operate** with the hypervisor so that host memory can be more efficiently distributed.
- Instead of providing an **emulated storage device**, the hypervisor can provide a paravirtualised implementation.
- Typically used in **data centres** for large-scale virtualisation.

Operating Systems

Secondary Storage

Lecture 14
Michael O'Boyle

Overview

- Disk trends
- Memory Hierarchy
- Performance
- Scheduling
- SSDs
 - Read
 - Write
 - Performance
 - Cost

Secondary storage

- Secondary storage:
 - anything outside “primary memory”
 - direct execution of instructions/ data retrieval via machine load/store
 - not permitted
- Characteristics:
 - it's large: 250-2000GB and more
 - it's cheap: \$0.05/GB for hard drives
- Persistent: data survives power loss
 - it's slow: milliseconds to access
- It *does* fail, if rarely
- Big failures
 - drive dies; Mean Time Between Failure ~3 years
 - 100K drives and MTBF is 3 years,
 - that's 1 “big failure” every 15 minutes!
- Little failures (read/write errors, one byte in 10^{13})

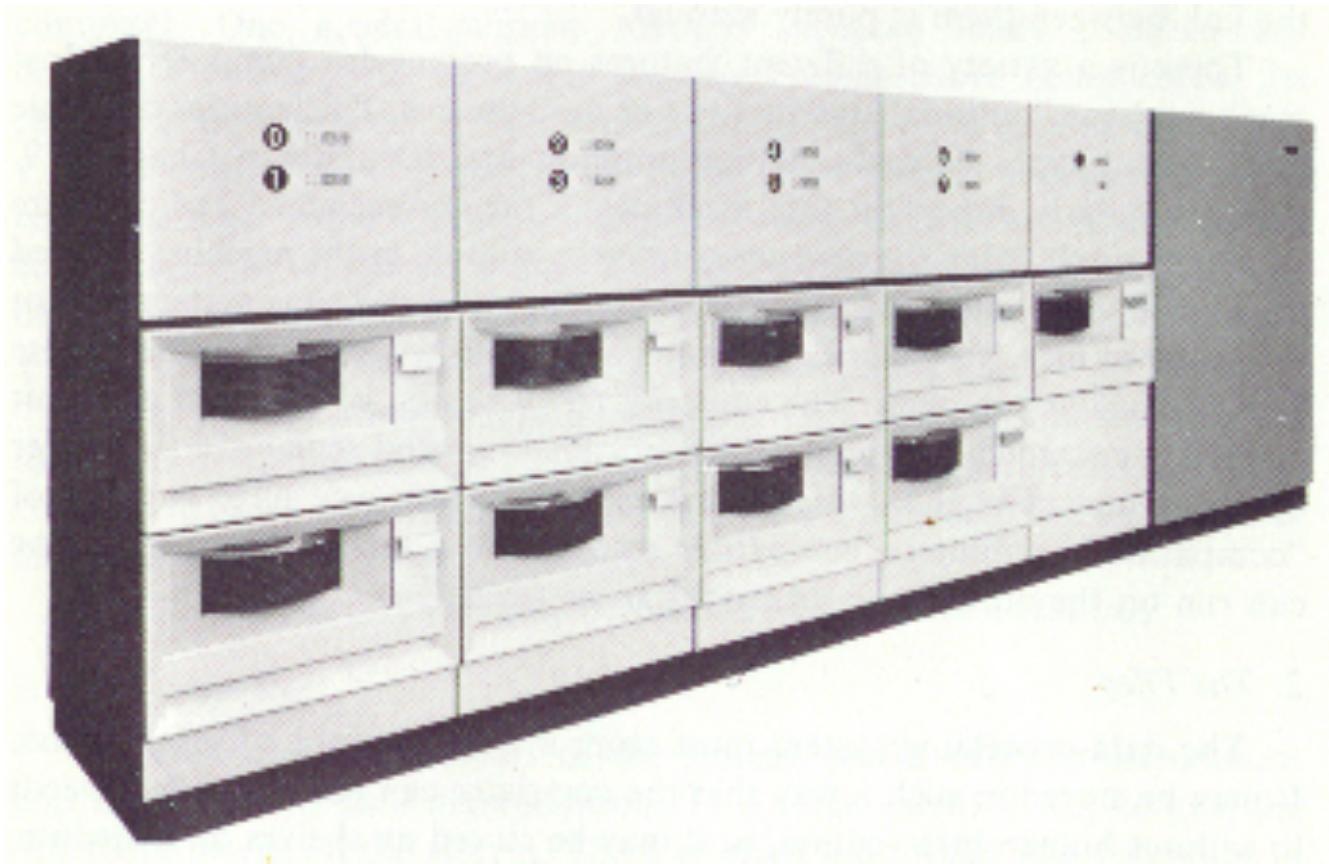
The First Commercial Disk Drive



1956
IBM RAMDAC computer
included the IBM Model
350 disk storage system

5M (7 bit) characters
50 x 24" platters
Access time = < 1 second

In the past



IBM 2314
About the size of
6 refrigerators
8 x 29MB
Required similar-
sized air cond.

.01% the capacity of this \$100
100x150x25mm item



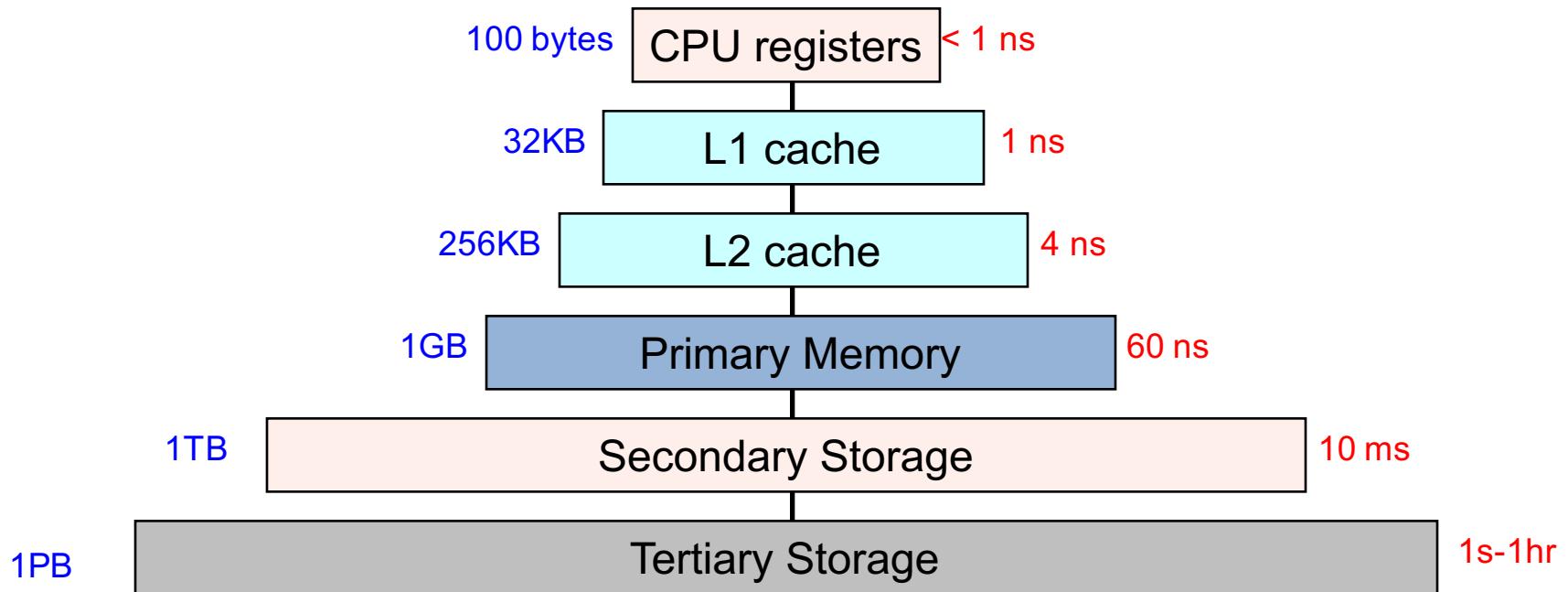
Disk trends

- Disk capacity, 1975-1989
 - doubled every 3+ years
 - 25% improvement each year
 - factor of 10 every decade
 - Still exponential, but far less rapid than processor performance
- Disk capacity, 1990-recently
 - doubling every 12 months
 - 100% improvement each year
 - factor of 1000 every decade
 - Capacity growth 10x as fast as processor performance!

Disk cost

- Only a few years ago, disks purchased by the megabyte
- Today, 1 GB (a billion bytes) costs \$0.05 from Dell
 - (except you have to buy in increments of 1000 GB)
 - => 1 TB costs \$50, 1 PB costs \$50K
- Performance analogy
 - Flying an aircraft at 600 mph 6" above the ground
 - Reading/writing a strip of postage stamps

Memory hierarchy



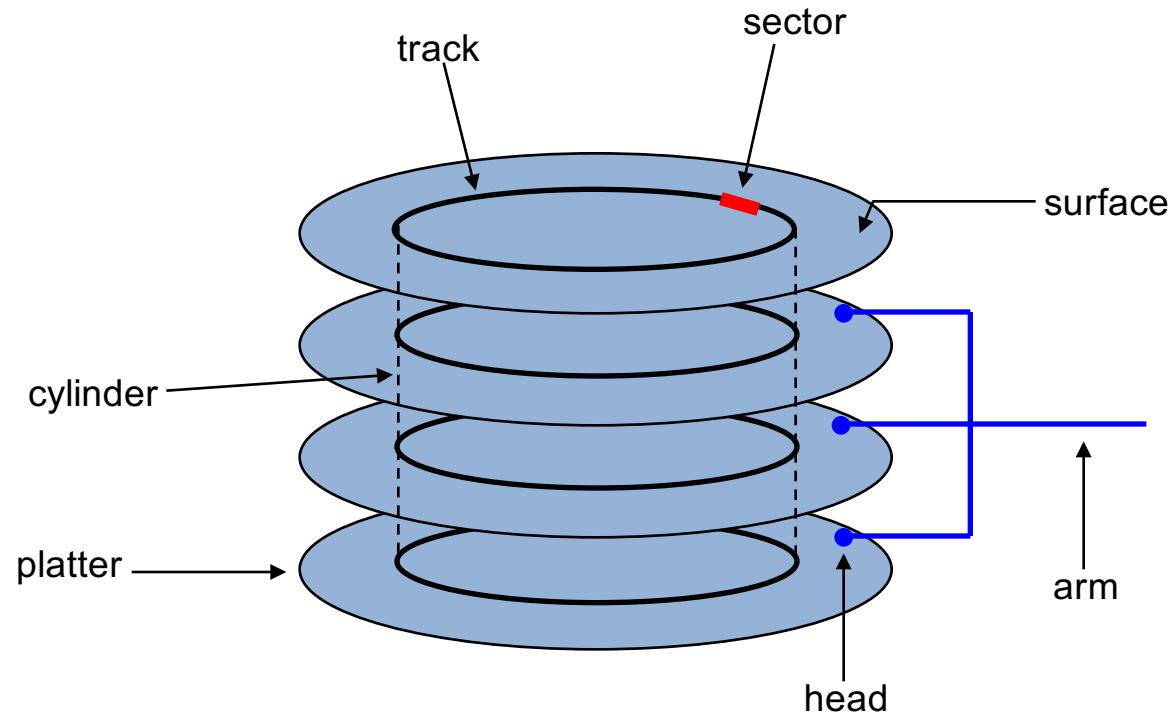
- Each level acts as a cache of lower levels

Disks and the OS

- Disks are difficult devices
 - errors, bad blocks, missed seeks, etc.
- OS abstracts this for higher-level software
 - low-level device drivers (initiate a disk read, etc.)
 - higher-level abstractions (files, databases, etc.)
 - disk hardware increasingly helps with this)
- OS provide different levels of disk access to different clients
 - physical disk block (surface, cylinder, sector)
 - disk logical block (disk block #)
 - file logical (filename, block or record or byte #)

Physical disk structure

- Disk components
 - platters
 - surfaces
 - tracks
 - sectors
 - cylinders
 - arm
 - heads



Disk Structure

- Disk drives are addressed as
 - large 1-dimensional arrays of **logical blocks**,
 - the logical block is the smallest unit of transfer
 - Low-level formatting creates **logical blocks** on physical media
- The 1-dimensional array of logical blocks
 - is mapped onto the sectors of the disk sequentially
- Sector 0 is the first sector of the first track on the outermost cylinder
 - Mapping proceeds in order through that track,
 - Then the rest of the tracks in that cylinder,
 - Then through the rest of the cylinders from outermost to innermost
- Logical to physical address should be easy
 - Except for bad sectors
 - Non-constant # of sectors per track via constant angular velocity

Disk performance

- Performance depends on a number of steps
- **Seek**: moving the disk arm to the correct cylinder
 - depends on how fast disk arm can move
 - not diminishing quickly due to physics
- **Rotation (latency)**: waiting for the sector to rotate under head
 - depends on rotation rate of disk
 - rates are slowly increasing,
- **Transfer**: transferring data from surface to disk controller,
 - then sending it back to host
 - depends on density of bytes on disk
 - increasing, relatively quickly
- When the OS uses the disk, it tries to minimize the cost of all of these steps
 - particularly seeks and rotation

Performance

- OS may increase file block size
 - in order to reduce seeking
- OS may seek to co-locate “related” items
 - in order to reduce seeking
 - blocks of the same file
 - data and metadata for a file
- Keep data or metadata in memory to reduce physical disk access
 - Waste valuable physical memory?
- If file access is sequential,
 - fetch blocks into memory before requested

Performance via disk scheduling

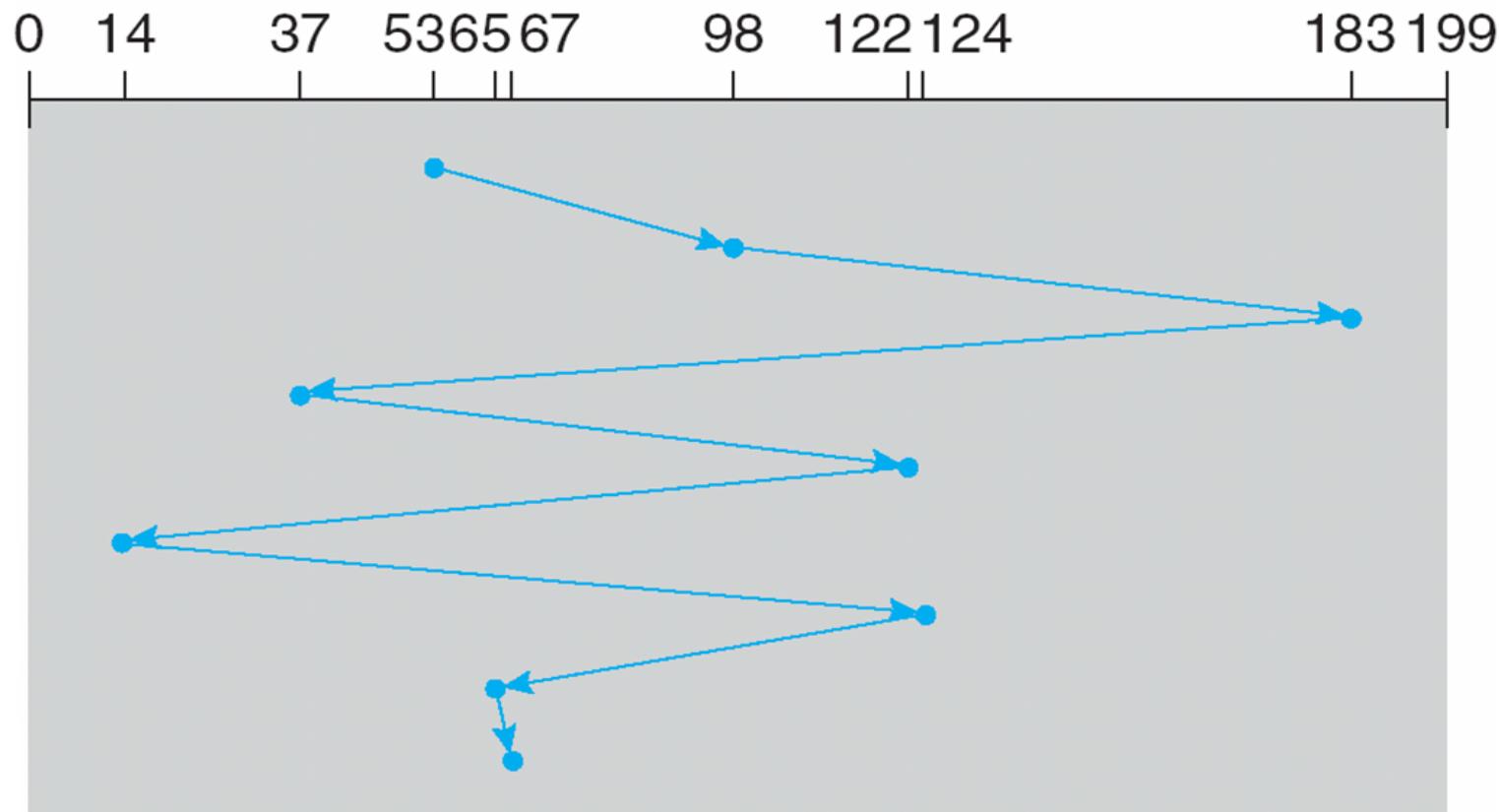
- Seek times are very expensive, so the OS attempts to schedule disk requests that are queued waiting for the disk
 - FCFS (do nothing)
 - reasonable when load is low
 - long waiting time for long request queues
 - SSTF (shortest seek time first)
 - minimize arm movement (seek time), maximize request rate
 - unfairly favors middle blocks
 - SCAN (elevator algorithm)
 - service requests in one direction until done, then reverse
 - skews wait times non-uniformly
 - C-SCAN
 - like scan, but only go in one direction (typewriter)
 - uniform wait times
 - C-LOOK
 - Similar to C-SCAN
 - The arm goes only as far as the final request in each direction

FCFS

Illustration shows total head movement of *640 cylinders*

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53

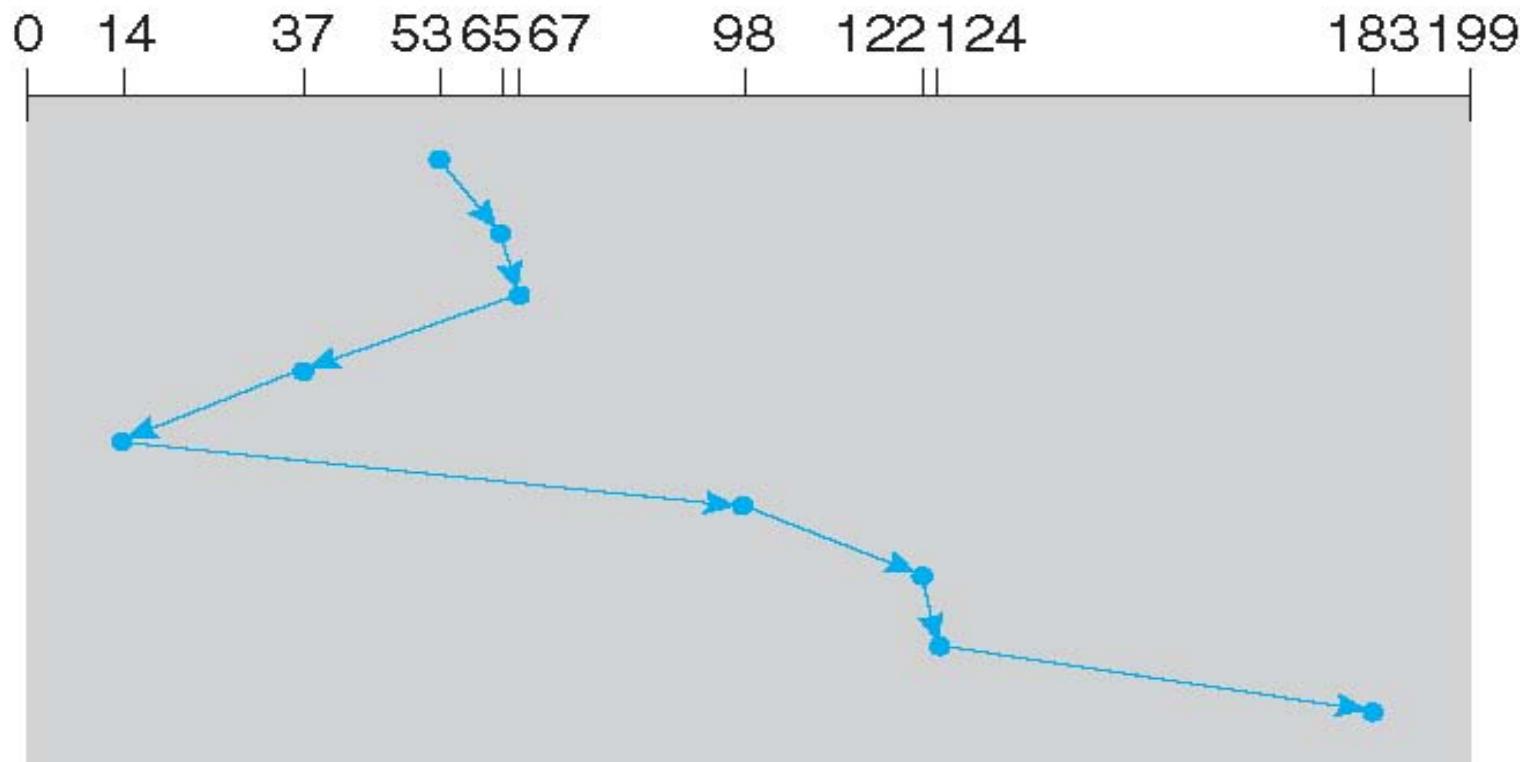


SSTF

Illustration shows total head movement of 236 cylinders
-may cause starvation

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



SCAN

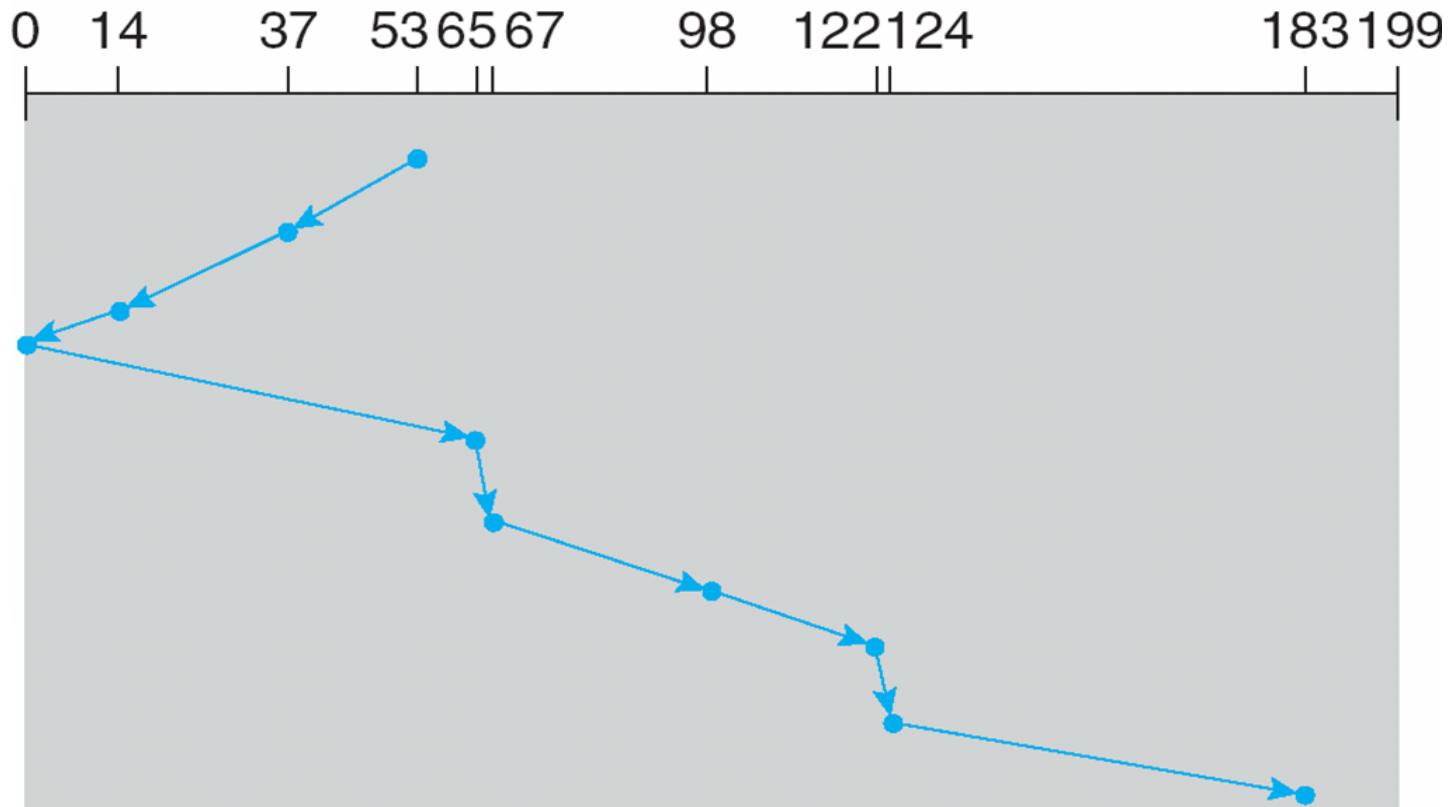
- The disk arm starts at one end of the disk,
 - and moves toward the other end,
 - servicing requests until it gets to the other end of the disk,
 - where the head movement is reversed and servicing continues.
- **SCAN algorithm** Sometimes called the **elevator algorithm**
- But note
 - that if requests are uniformly dense,
 - largest density at other end of disk
 - and those wait the longest

SCAN

Illustration shows total head movement

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



C-SCAN

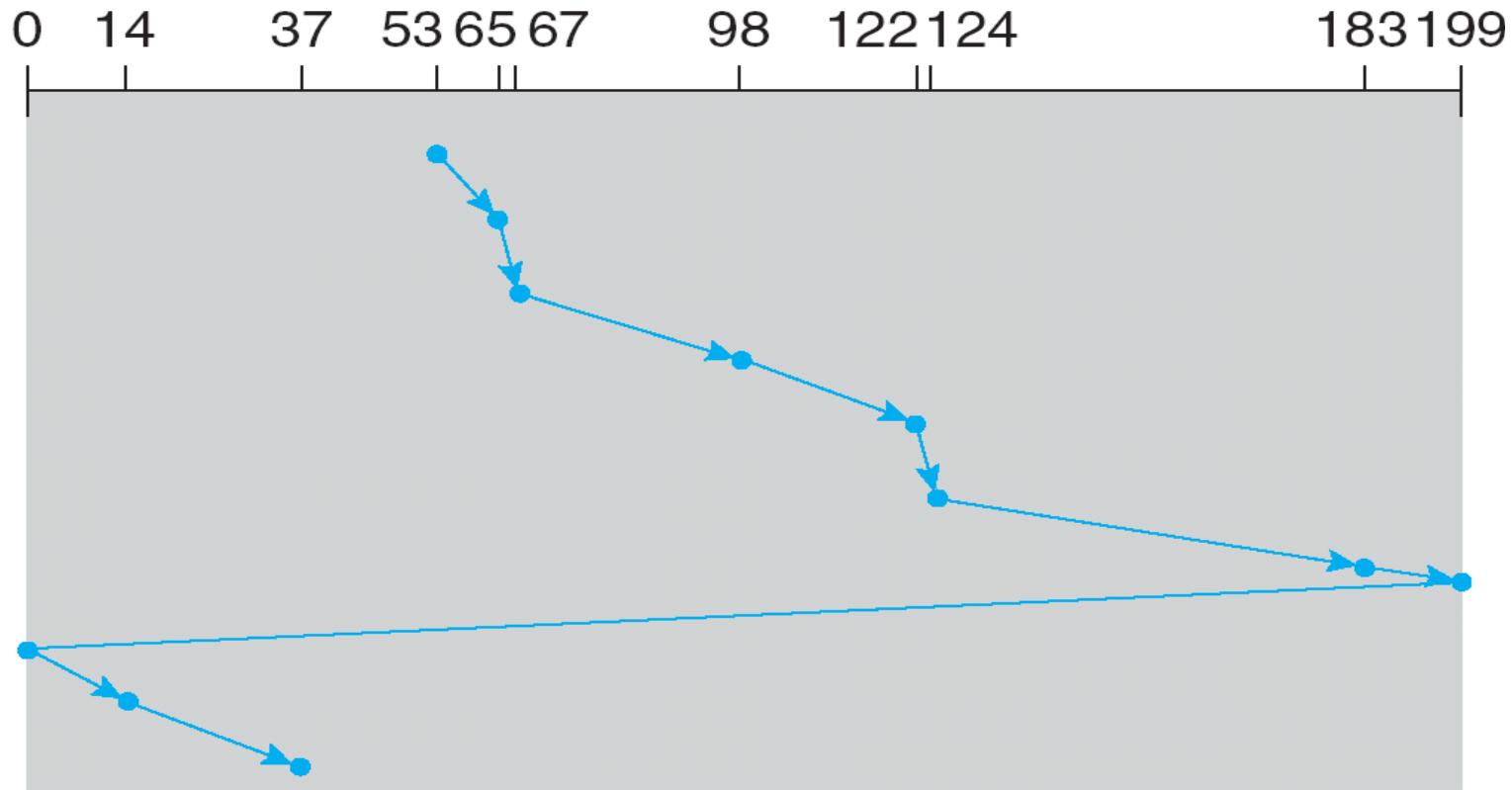
- Provides a more uniform wait time than SCAN
- The head moves from one end of the disk to the other, servicing requests as it goes
 - When it reaches the other end, however,
 - it immediately returns to the beginning of the disk
 - without servicing any requests on the return trip
- Treats the cylinders as a circular list
 - that wraps around from the last cylinder to the first one

C-SCAN

Illustration shows total head movement of *382 cylinders*

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



C-LOOK

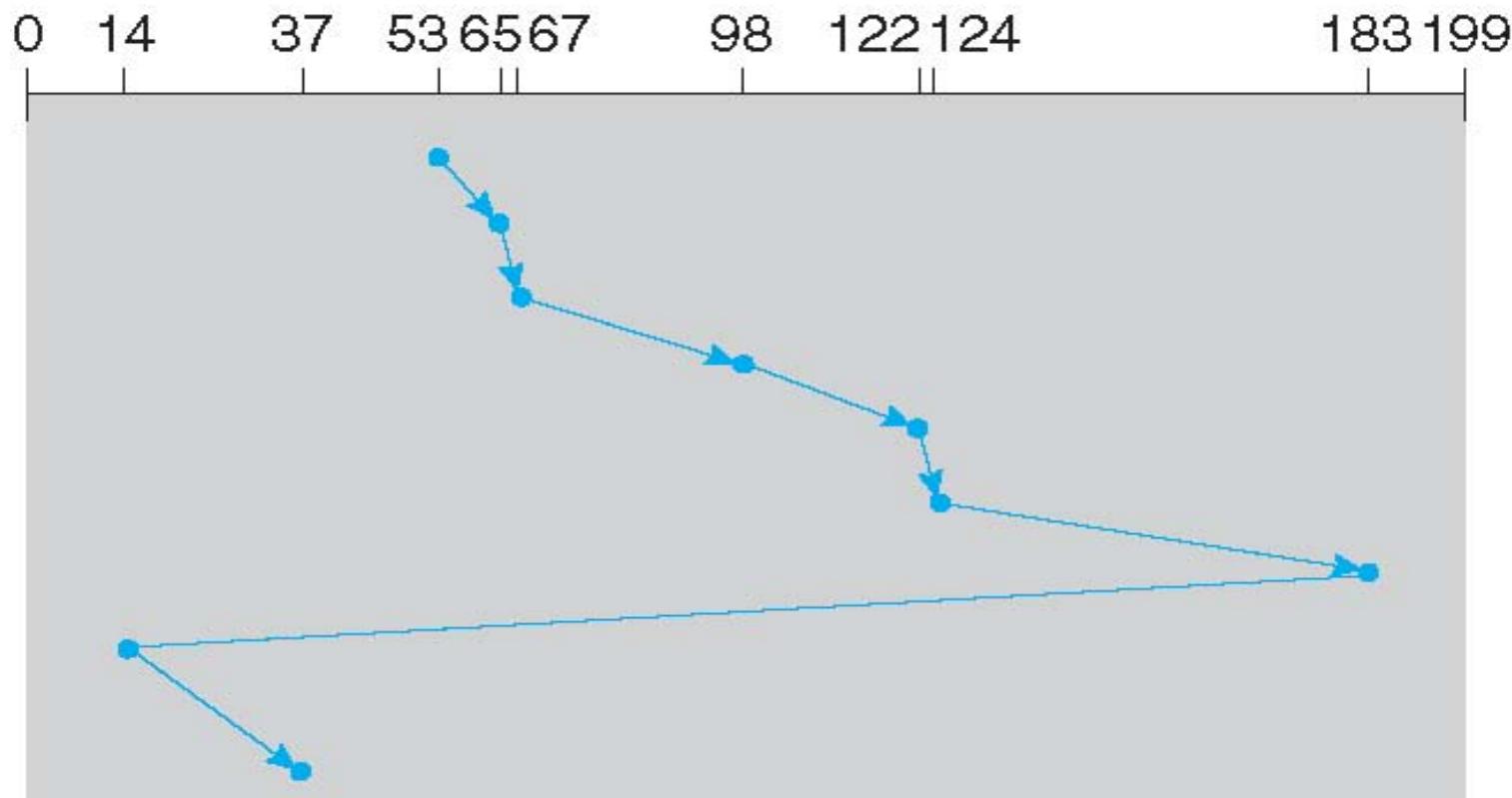
- LOOK a version of SCAN, C-LOOK a version of C-SCAN
- Arm only goes as far
 - as the last request in each direction,
 - then reverses direction immediately,
 - without first going all the way to the end of the disk

C-LOOK

Illustration shows total head movement of *322 cylinders*

queue = 98, 183, 37, 122, 14, 124, 65, 67

head starts at 53



Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk
 - Less starvation
- Performance depends on the number and types of requests
- Requests for disk service can be influenced by the file-allocation method
 - And metadata layout
- The disk-scheduling algorithm should be
 - written as a separate module of the operating system,
 - allowing it to be replaced with a different algorithm if necessary
- Either SSTF or LOOK is a reasonable choice for the default algorithm
- What about rotational latency?
 - Difficult for OS to calculate

Interacting with disks

- Previously
 - OS would specify cylinder #, sector #, surface #, transfer size
 - i.e., OS needs to know all of the disk parameters
- Modern disks more complex
 - not all sectors are the same size, sectors are remapped, ...
- Disk provides a higher-level interface, e.g., SCSI
 - exports data as a logical array of blocks [0 ... N]
 - maps **logical blocks** to cylinder/surface/sector
- OS only names logical block #,
 - disk maps this to cylinder/surface/sector
 - on-board cache
 - as a result, physical parameters are hidden from OS

Seagate Barracuda 9cm disk drive

- 1Terabyte of storage (1000 GB)
- \$100
- 4 platters, 8 disk heads
- 63 sectors (512 bytes) per track
- 16,383 cylinders (tracks)
- 164 Gbits / inch-squared (!)
- 7200 RPM
- 300 MB/second transfer
- 9 ms avg. seek, 4.5 ms avg. rotational latency
- 1 ms track-to-track seek
- 32 MB cache



Solid state drives: ongoing disruption

- Hard drives are based on spinning magnetic platters
 - *mechanics* of drives determine performance characteristics
 - sector addressable, not byte addressable
 - capacity improving exponentially
 - sequential bandwidth improving reasonably
 - random access latency improving very slowly
- Cost dictated by
 - massive economies of scale,
 - and many decades of commercial development and optimization

SSD

- Solid state drives are based on NAND flash memory
 - no moving parts; performance characteristics driven by electronics and physics – more like RAM than spinning disk
 - relative technological newcomer, so costs are still quite high in comparison to hard drives, but dropping fast



SSD performance: reads

- Reads
 - unit of read is a *page*, typically 4KB large
- Today's SSD can typically handle
 - 10,000 – 100,000 reads/s
- 0.01 – 0.1 ms read latency
 - 50-1000x better than disk seeks
- 40-400 MB/s read throughput
 - 1-3x better than disk seq. throughput

SSD performance: writes

- Writes
 - flash media must be erased before it can be written to
 - unit of erase is a block, typically 64-256 pages long
 - usually takes 1-2ms to erase a block
 - blocks can only be erased a certain number of times before they become unusable – typically 10,000 – 1,000,000 times
 - unit of write is a page
 - writing a page can be 2-10x slower than reading a page
- Writing to an SSD is complicated
 - random write to existing block: read block, erase block, write back modified block
 - leads to hard-drive like performance (300 random writes / s)
 - sequential writes to erased blocks: fast!
 - SSD-read like performance (100-200 MB/s)

SSDs: dealing with erases, writes

- Lots of higher-level strategies can help hide the warts of an SSD
- Many of these work by
 - exposing logical pages, not physical pages
- Wear-levelling:
 - when writing,
 - try to spread erases out evenly across physical blocks of the SSD
 - Intel promises 100GB/day x 5 years for its SSD drives
- Log-structured filesystems:
 - convert random writes within a filesystem
 - to log appends on the SSD

SSD cost

- Capacity
 - today, flash SSD costs ~\$2.50/GB
 - 1TB drive costs around \$2500
 - 1TB hard drive costs around \$50
 - Data on cost trends is volatile/preliminary
- Energy
 - SSD is typically more energy efficient than a hard drive
 - 1-2 watts to power an SSD
 - ~10 watts to power a high performance hard drive
 - (can also buy a 1 watt lower-performance drive)

Summary

- Disk trends
- Memory Hierarchy
- Performance
- Scheduling
- SSDs
 - Read
 - Write
 - Performance
 - Cost
- Next lecture: Revision tutorial