

CS 450: Operating Systems

Ben Lenard

blenard@iit.edu

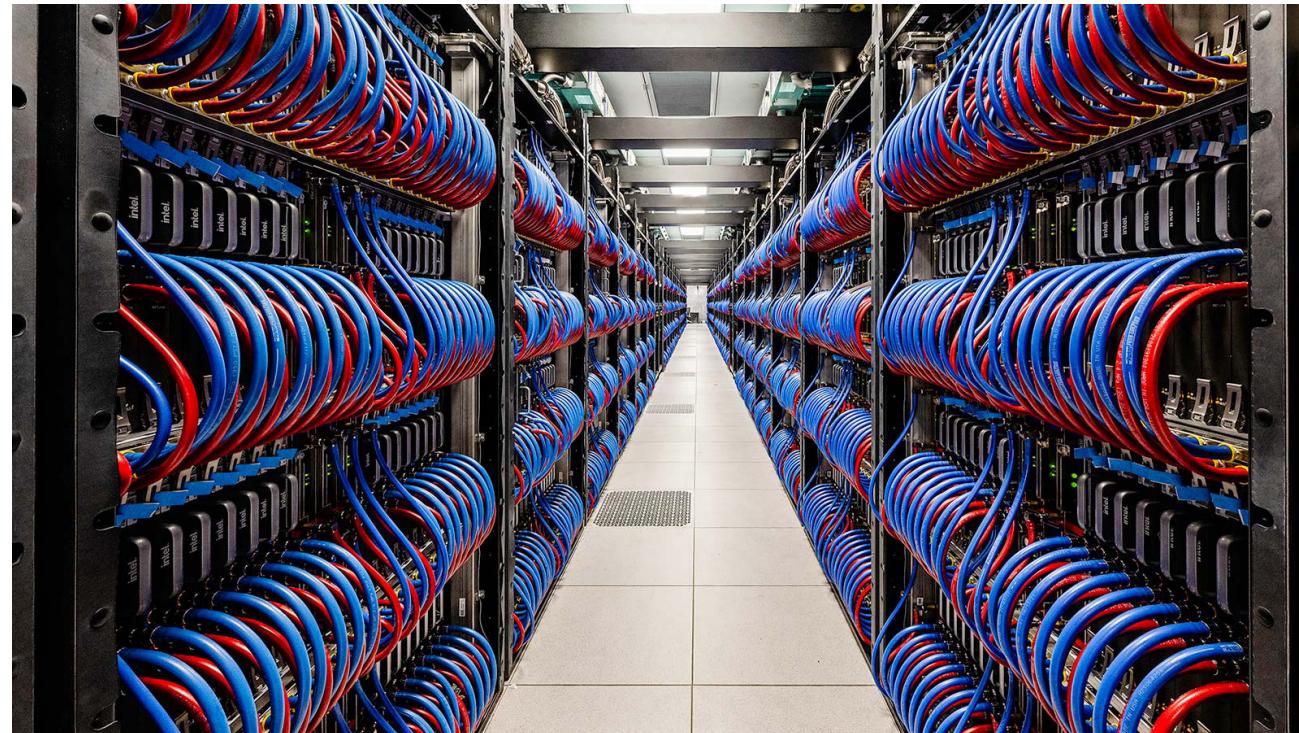
ILLINOIS TECH

College of Computing

Instructor

- Ben Lenard (Do not call me Dr or Professor)
- Email: blenard@iit.edu
- Office: ?
- Office hours: I am flexible to meet on Zoom, so email me for an appointment. Alternatively, I can meet before or after class.
- About me:
 - 14 years or so working for a financial giant, ran PeopleSoft for another university, and 8+ years working at Argonne's Leadership Computing environment.
 - 2 Master degrees (one is a MBA), and a PhD student at DePaul

One of the computers I work with



ILLINOIS TECH

College of Computing

Class schedule, TA's, and updates

- All class schedules and updates will be posted in Blackboard.
- The schedule may change slightly as the course proceeds into the term.
- No teaching assistances at this time

Agenda for tonight

- Prerequisites
- Resources:
 - Blackboard
 - Textbooks
 - Computing Environment
- Evaluation:
 - Assignments
 - Exams
 - Grading
- Class overview

ILLINOIS TECH

College of Computing

Prerequisites

ILLINOIS TECH

College of Computing

CS Essentials

- Essential algorithms & runtime analysis
- Data structures
- Data representation and manipulation

ILLINOIS TECH

College of Computing

Programming Knowledge

- Languages: Assembly (x86 or other), C (or other procedural)
- Compilation process (assembly, compilation, linking, etc.)
- Runtime stack usage and conventions
- Dynamic memory allocation

ILLINOIS TECH

College of Computing

Computer Organization

- Von Neumann model
- Instruction Set Architectures (RISC/CISC)
- Cache organization and operation
- Interrupt procedures

ILLINOIS TECH

College of Computing

Operating System API

- Knowledge of Unix syscalls
 - process management (fork/exec/wait)
 - memory management (sbrk/mmap)
 - I/O (open/close/read/write/seek)

ILLINOIS TECH

College of Computing

Tools

- QEMU / etc
- Command line – ssh / bash / zsh
- Debugger/Tracer - GDB
- Build automation - Make
- Version Control – git / svn

Resources

- Blackboard – schedule, assignments, useful links
- Textbooks – Free
- Discord for discussions and to help each other.
- Computing – what do people have? Let me know ASAP if you do not have your own computer.

Blackboard

ILLINOIS TECH

College of Computing

Operating Systems: Three Easy Pieces

<https://pages.cs.wisc.edu/~remzi/OSTEP/>

And now, the free online form of the book, in chapter-by-chapter form (now with chapter numbers!):

Intro	Virtualization		Concurrency	Persistence	Security
Preface	3 Dialogue	12 Dialogue	25 Dialogue	35 Dialogue	52 Dialogue
TOC	4 Processes	13 Address Spaces <small>code</small>	26 Concurrency and Threads <small>code</small>	36 I/O Devices	53 Intro Security
1 Dialogue	5 Process API <small>code</small>	14 Memory API	27 Thread API <small>code</small>	37 Hard Disk Drives	54 Authentication
2 Introduction <small>code</small>	6 Direct Execution	15 Address Translation	28 Locks <small>code</small>	38 Redundant Disk Arrays (RAID)	55 Access Control
	7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories	56 Cryptography
	8 Multi-level Feedback	17 Free Space Management	30 Condition Variables <small>code</small>	40 File System Implementation	57 Distributed
	9 Lottery Scheduling <small>code</small>	18 Introduction to Paging	31 Semaphores <small>code</small>	41 Fast File System (FFS)	
	10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling	Appendices
	11 Summary	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS) <small>Dialogue</small>	
		21 Swapping: Mechanisms	34 Summary	44 Flash-based SSDs	Virtual Machines
		22 Swapping: Policies		45 Data Integrity and Protection	Dialogue
		23 Complete VM Systems		46 Summary	Monitors
		24 Summary		47 Dialogue	Dialogue
				48 Distributed Systems	Lab Tutorial
				49 Network File System (NFS)	Systems Labs
				50 Andrew File System (AFS)	xv6 Labs
				51 Summary	

INSTRUCTORS: If you are using these free chapters, [please just link to them directly](#) (instead of making a copy locally); we make little improvements frequently and thus would like to provide the latest to whomever is using it. Also: we have made our own class-preparation notes available to those of you teaching from this book; please drop us a line at remzi@cs.wisc.edu if you are interested.

HOMEWORKS: Some of the chapters have homeworks at the end, which require simulators and other code. More details on that, including how to find said code, can be found here: [HOMEWORK](#)

PROJECTS: While the book should provide a good conceptual guide to key aspects of modern operating systems, no education is complete without projects. We are in the process of making the projects we use at the University of Wisconsin-Madison widely available; an initial link to project descriptions is available here: [PROJECTS](#). Coming soon: the automated testing framework that we use to grade projects.

BOOKS NEWS: Lots of new stuff to finally get to version 1.0. Track changes: [NEWS](#)

ILLINOIS TECH

College of Computing

Readings before class!

- Please read (or at least, skim) readings before associated lecture
 - I do like to relate the material to the real world and engage the students.
- Check website frequently in case schedule changes

ILLINOIS TECH

College of Computing

Grading / Evaluation

ILLINOIS TECH

College of Computing

Assignments

- 5-7 assignments = 50% of grade:
 - Question sets (quantitative analysis)
 - Machine problems (coding)

ILLINOIS TECH

College of Computing

Exams

- Two exams (midterm & final) @ 25% each
- Tentative midterm exam date: October

ILLINOIS TECH

College of Computing

Grading Scale

- A: $\geq 90\%$
- B: 80-89%
- C: 70-79%
- D: 60-69%
- E: < 60%

ILLINOIS TECH

College of Computing

Class Overview

ILLINOIS TECH

College of Computing

CS 450

- Capstone of the systems sequence (CS 350 → 351 → 450)
- Answer to “how do modern (general purpose) computers work under the hood?”
- The OS is the foundation for platforms of every type: Z, Unix, Linux, Windows, etc.

You should know

- What services are provided by an OS
- And how to invoke them (syscalls)
- How to use them effectively and efficiently

Lingering Questions

- How are processes actually created/tracked?
- How do processes safely & efficiently share resources (e.g., CPU/Mem)?
- How to correctly/safely leverage concurrency?
- How does the file system work?
- How are protection/security enforced?

Primary topics

- Kernel architecture
- Processes and Threads
- Scheduling
- Virtual memory
- I/O architectures and device programming
- File Systems
- Interprocess Communication
- Concurrency and Synchronization
- Security

ILLINOIS TECH

College of Computing

Theory vs. Implementation

- Grand academic debate
- Theory comes before implementation



ILLINOIS TECH

College of Computing

XV6

- During this class:
 - We'll read an existing OS codebase
 - make modifications and additions
- This is easier than writing millions of lines of code
- It is based on UNIXv6 — released in 1975, among the first preemptive multitasking OSes, and still a great software engineering blueprint!

Topic for next time: What is an OS?

- Before next time, please read OS:TEP chapters 1 & 2!

What is an OS?

Ben Lenard

blenard@iit.edu

ILLINOIS TECH

College of Computing

Agenda

- Road to the modern OS - OS responsibilities
- OS privileges
- OS organization
- Summary
- Overview of the first assignment

ILLINOIS TECH

College of Computing

Road to the modern OS

ILLINOIS TECH

College of Computing

1950s: Batch processing

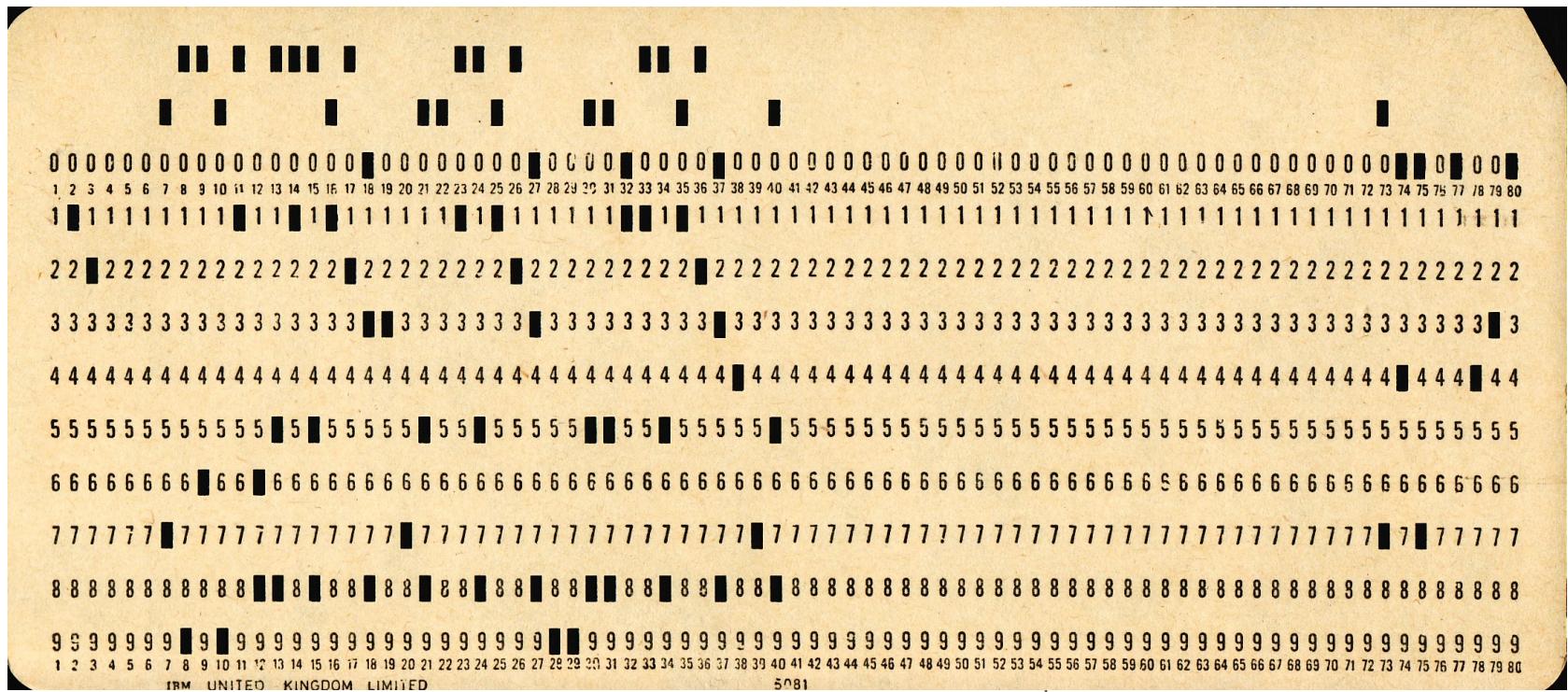
- A program is completely defined by a batch of punch cards
- Batches are manually fed into mainframes, which execute a single batch at a time (a “job”)
- Programmer defines any and all routines needed for the job
 - for controlling and accessing specific I/O devices



ILLINOIS TECH

College of Computing

A punch card



ILLINOIS TECH

College of Computing

1950s-1960s: Support libraries

- Useful, reusable routines, code, (e.g., for math, I/O) distributed as collections of punch cards
- These routines can be “linked” (manually) into programs without much modification
- First support libraries – prior to this people rewrote code over and over. What happens?

Cartons of punched cards stored in a United States National Archives Records Service facility in 1959. Each carton could hold 2,000 cards.



ILLINOIS TECH

College of Computing

1960s: Automatic batch processing

- - To keep up with faster processors, reading and starting/transitioning between jobs require automation to keep up with keeping the system busy.
- “Monitor” programs also keep track of usage, resources expended, etc. Why?
- Became runtime libraries that automatically manage the execution of multiple batches of jobs in sequence

Pros and Cons of Batch processing

Pros

- Full use of hardware
- Not worrying about other jobs during execution

Cons

- No interactivity
- No live debugging - only dumps
- No feedback loop – i.e. do based on input
- Can be poor hardware utilization
- Write and handle everything yourself

Mainframe Dump

Address	Memory Dump in Hexadecimal
11F48000	F5F2F2F3 F4F9F9F9 F9C3C3F1 F1F0F1F1
11F48020	40404040 40404040 40404040 40404040
11F48040	40404040 40404040 40404040 40404040
11F48060	F9F9F9C3 C3F1F1F0 F1F2F1F0 F0F0F0F7
11F48080	40404040 40404040 40404040 40404040
11F480A0	40404040 40404040 40404040 F0F0F0F0
11F480C0	F1F8F1F2 F1F0F0F0 F0F0F0F7 F6811556
11F480E0	40404040 40404040 40404040 40404040
11F48100	40404040 4040F0F0 F0F0F1F5 F0F00000
11F48120	00000000 00000000 00000000 00000000

Address	Memory Dump in Display
11F48000	*522349999CC11011100000088...p.
11F48020	*
11F48040	*
11F48060	*999CC11012100007994alpha
11F48080	*
11F480A0	00000200522349731CC2*
11F480C0	*1812100000076a....
11F480E0	*
11F48100	00001500.....*
11F48120*

6 address of 11F4805A
points to beginning of the
event record, hex F5 or
display value 5

Register 6 address of 11F4805A
Plus displacement of 19
= 11FA8073

The display value "alpha" is at this location

ILLINOIS TECH

College of Computing

1970s: Rise of Timesharing

- Timesharing was developed to allow multiple jobs run concurrently on the hardware.
- Timesharing software is needed to automatically change context between jobs.
 - For example:
 - Resources (e.g., CPU & memory) are virtualized
 - To ensure security, jobs are isolated from each other
- Timesharing resulted in system overhead, but offset by improved hardware utilization
- Development and availability of UNIX on mainframes and minicomputers – aka TSO on the mainframe

1970'S Mainframe



ILLINOIS TECH

College of Computing

1980s: Era of PC OSes

- Personal computers (microcomputers) become widely available
 - Underpowered compared to systems that ran contemporary timesharing OSes such as UNIX
- PC OSes (e.g., MS-DOS, Mac OS) were dumbed down in many ways
 - Lack of memory protection – often crashed
 - Cooperative multitasking vs. preemptive multitasking
 - Poor system stability, and chaos for developers!
 - The era of shareware 😊

1990s-Present: Modern OSes

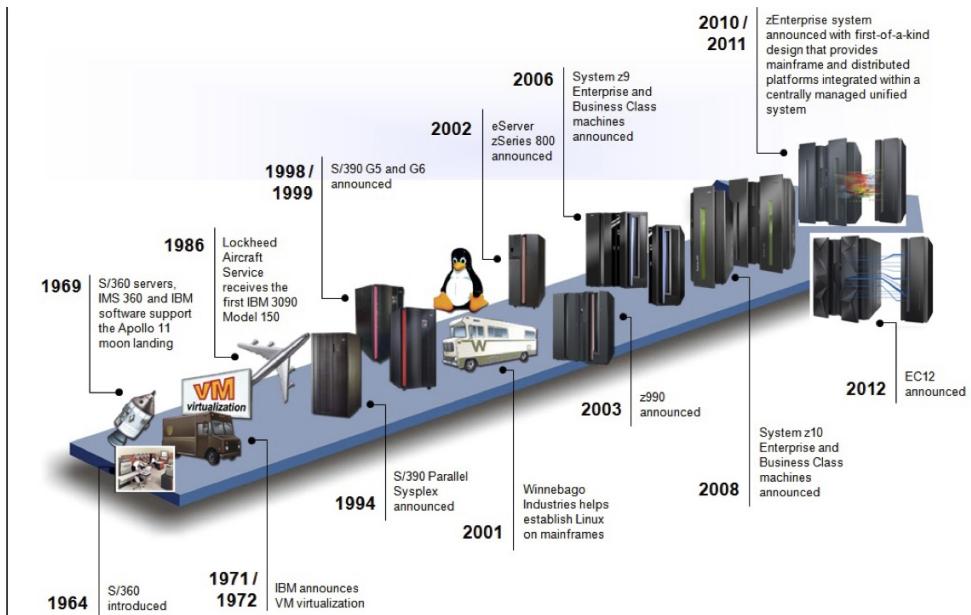
- More powerful PCs make preemptively multitasked OSes generally viable
- High degrees of virtualization, isolation, and concurrency
- Exploding market for varied I/O devices and peripherals
 - OS support for “plug and play” third-party device drivers - Large, sophisticated system call interfaces
- Standards are created for portability across OSes (i.e POSIX)

Yes or No: Are Mainframes still around?

ILLINOIS TECH

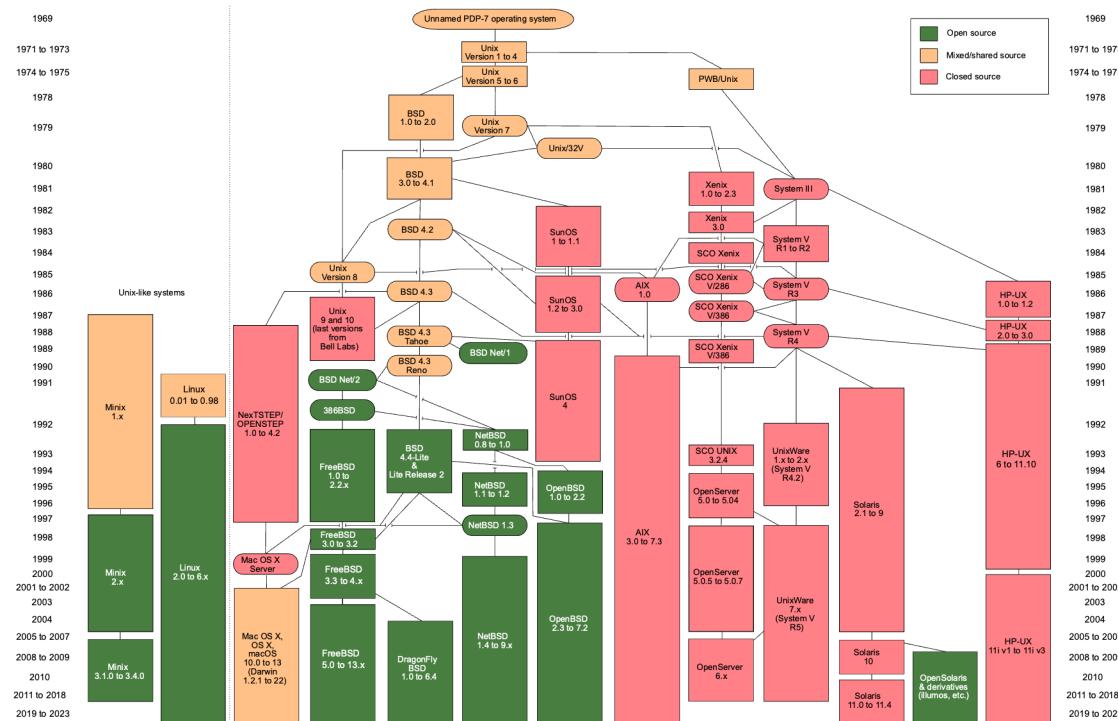
College of Computing

Yes



UNIX & AIX vs Linux

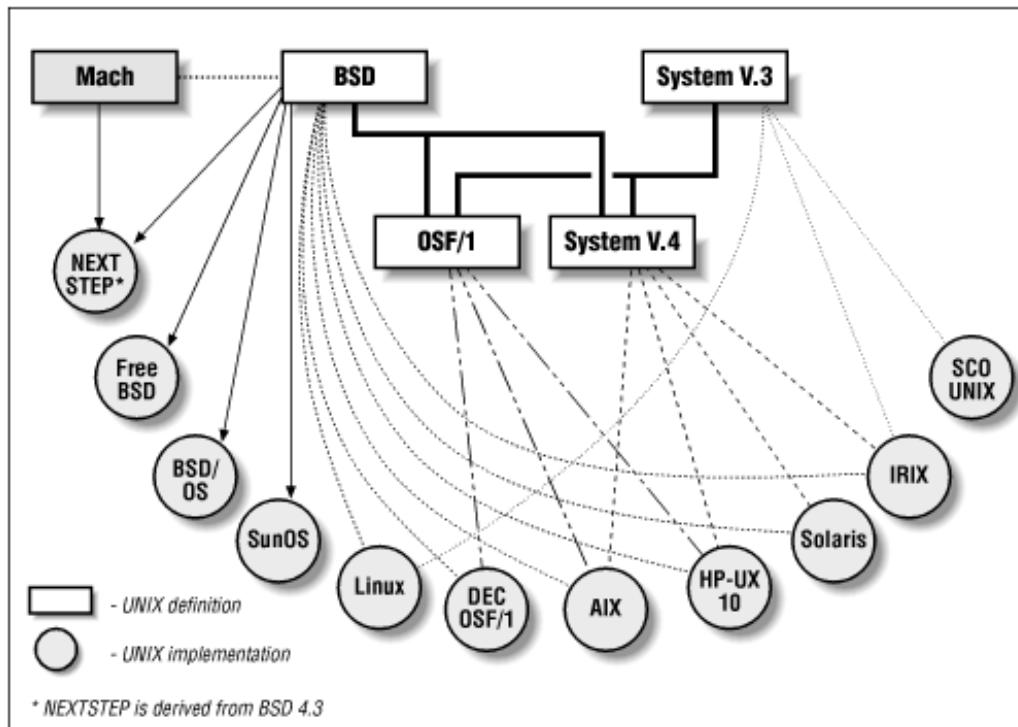
- UNIX – Commercial software, HPUX, AIX, Solaris, etc.
 - UNiplexed Information Computing System - Unix is not an acronym; it is a pun on "Multics". Multics is a large multi-user operating system that was being developed at Bell Labs shortly before Unix was created in the early '70s. Brian Kernighan is credited with the name.
 - AIX is developed by IBM and stands for Advanced Interactive eXecutive
 - Proprietary kernels
- Linux – Opensource with sometimes commercial support
 - Redhat, OpenSUSE, Fedora, Rocky, etc
 - Same kernel with tweaks



ILLINOIS TECH

College of Computing

Simple



ILLINOIS TECH

College of Computing

What does an OS do?

ILLINOIS TECH

College of Computing

Operating system

- An operating system (OS) is system software that manages computer hardware and software resources, and provides common services for computer programs. (Wikipedia)
- The software “master control application” that runs the computer. It is the first program loaded when the computer is turned on, and its main component, the kernel, resides in memory at all times. The operating system sets the standards for all application programs (such as the Web server) that run in the computer. The applications communicate with the operating system for most user interface and file management operations. (NIST)

Resource management

- CPU, Memory, I/O devices are limited resources
 - Allocate the processes to CPU cores
 - Handle the total memory required compared to physical RAM
 - Manage file accesses translated to disk read & writes
- OS acts as a high level resource manager
 - Processes can focus on their own tasks and not worry about security

Virtualization

- A powerful model for resource allocation is virtualization:
 - Each process behaves as though it is accessing its own private CPU(s), address space, I/O device, registers, etc.
 - Behind the scenes, the OS maintains this illusion by allocating and multiplexing resources across all concurrently executing processes
 - Effectively creates an independent machine for each process
 - Can be aided with processor extensions

ILLINOIS TECH

College of Computing

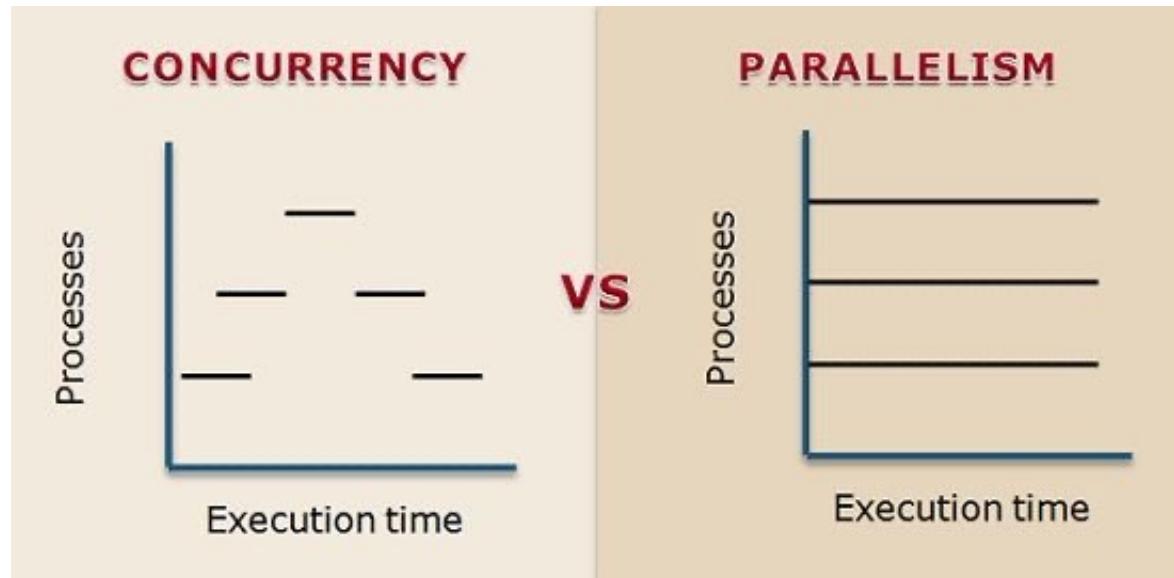
Concurrency

- Concurrency presents its own challenges and techniques for dealing with them
- Concurrent processes must be isolated from each other
- Nondeterministic execution and access to resources creates race conditions within the OS and between processes
- Dealing with these issues requires special tools and techniques
- Applies to Databases as well

ILLINOIS TECH

College of Computing

Difference Between Concurrency and Parallelism



ILLINOIS TECH

College of Computing

Persistence

- CPU and Memory state are volatile*
- I/O devices provide support for persistent storage as well as issues:
 - How to namespace persistent data?
 - What OS APIs are needed for accessing persistent data?
 - How to efficiently manage and access data on slow devices? (I/O scheduling and queueing)
 - If processes crash when updating persistent store, how to guarantee consistency?

How to achieve these requirements?

- - To implement virtualization, concurrency, persistence (and other goals), the OS relies on hardware assistance
- Modern hardware features that allow the OS to maintain exclusive access to privileged operations and structures. This is to prevent prevent accidental and/or malicious process behavior from interfering with other processes or the OS itself

OS organization

ILLINOIS TECH

College of Computing

What is privileged?

- Which portions of an OS will be run in “privileged” mode?
 - Standard OS modules:
 - Virtual memory
 - Scheduler
 - Device drivers
 - File system
 - IPC

The OS Kernel

- Privileged code / modules constitute the kernel of the operating system. Optimized code.
- Always loaded first into memory, and always memory-resident
- Handles all privileged operations
- Handles Hardware access
- Updating special and/or controls registers
- Running special instructions
- Works in close concert with architecture features (e.g., clock interrupt)

ILLINOIS TECH

College of Computing

Monolithic architecture

- Primary modules and I/O device drivers run in privileged mode
- Relatively large, permanent memory footprint
- No mode transitions when jumping between different pieces of the OS
- Very little system overhead
- Because the privileged codebase is very large, harder to verify and guarantee system robustness!
- If one piece of the OS crashes, all of it does

Microkernel architecture

- Only essential services are privileged, everything else runs in user mode
- Relatively small memory footprint
- Microkernel functions in part as a messenger between different modules running in user mode
- Jumping between different OS modules may require mode switch
- Higher system overhead
- Easier to verify and guarantee robustness
- If a user-level OS module crashes, just restart it

ILLINOIS TECH

College of Computing

And there's more...

- Hybrid
- Nano
- Exo
- Multi

ILLINOIS TECH

College of Computing

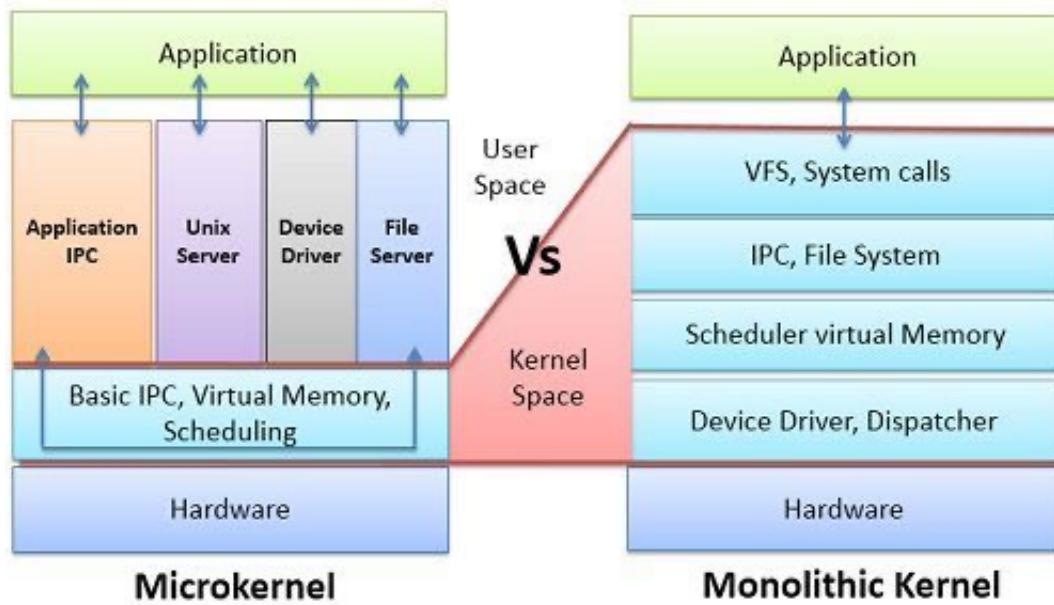
Who is right?

- It depends....

ILLINOIS TECH

College of Computing

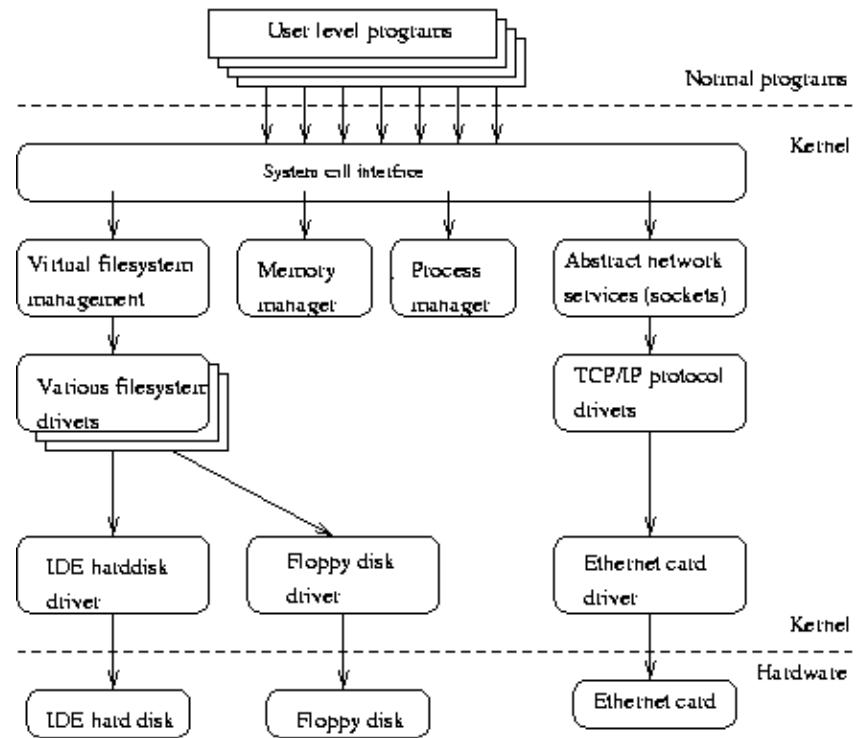
Micro vs Mono



ILLINOIS TECH

College of Computing

Linux Kernel implementation



ILLINOIS TECH

College of Computing

Discussion Questions

- What an operating system?
- What does it do?
- What is virtualization?
- How does an OS provide access to its features?
- What illusion does a virtualized CPU provide?
- How does this affect the user experience?
- What if the CPU were not virtualized?
- What is a memory address?
- What is memory virtualization?
- Why would we want this?
- What happens if you write a C++ program that writes past the end of an array?
- What is a thread?
Why would we ever write a multi-threaded program?
- Is a C++ statement atomic?
What does persistence mean?
- How does OS hard drive virtualization differ from CPU & memory virtualization?
- How does running multiple programs at the same time increase CPU efficiency?

Summary

- Why do we need an OS?
 - To facilitate process execution and simplify/control access to hardware
- What does an OS do?
 - Provide virtualization, concurrency, and persistence
- How is an OS organized?
 - Separation of privileged and user code — architecture of kernel is an exercise in tradeoffs!

What is Process?

Ben Lenard

blenard@iit.edu

ILLINOIS TECH

College of Computing

Agenda

- Recap of what an OS is?
- The Process: what is it and what's in it?
- Forms of Multitasking
- Tracking processes in the OS
- Context switches and Scheduling
- Process API

ILLINOIS TECH

College of Computing

Resource management

- CPU, Memory, I/O devices are limited resources
 - Allocate the processes to CPU cores
 - Handle the total memory required compared to physical RAM
 - Manage file accesses translated to disk read & writes
- OS acts as a high level resource manager
 - Processes can focus on their own tasks and not worry about security

process

- The definition is: a series of actions or steps taken in order to achieve a particular end.
- In computer science, a process is a program in execution
- - its behavior is largely defined by the program being executed

ILLINOIS TECH

College of Computing

Multitasking

- Modern general-purpose OSes typically run dozens to hundreds of processes simultaneously
- May collectively exceed capacity of hardware – they run concurrently
- *Virtualization* allows each process to ignore physical hardware limitations and let OS take care of details

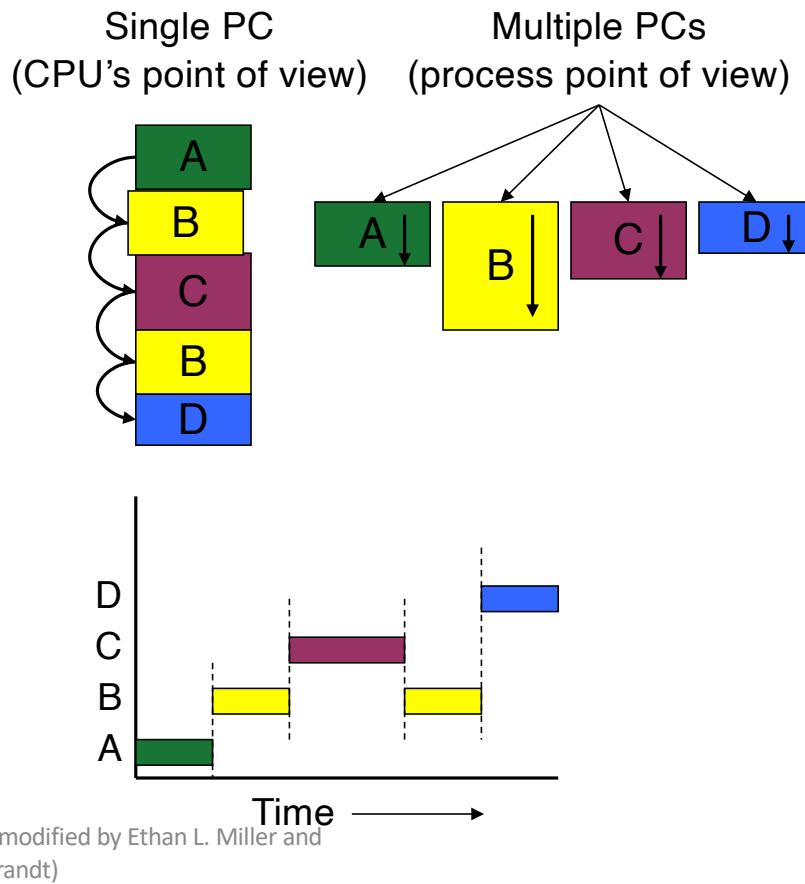
ILLINOIS TECH

College of Computing

CPU/Memory Virtualization

- Time-slicing of CPU(s) is performed to simulate concurrency
- Memory is partitioned and shared amongst processes
- But per-process view is of a uniform address space
- On-demand loading of processes lowers total burden

The process model



- Four programs
- Conceptual model
 - 4 independent processes
 - Processes run sequentially
- Only one program active at any instant!
 - That instant can be very short...
 - Each process has it's own heap,stack and code in memory

Remember Batch processing

- Without multitasking, each program is run from start to finish without interruption from other processes
- Including any I/O operations (what did I say day 1?)
- Ensures minimal overhead (but at what cost?)
- Is virtualization still necessary?

Pros/Cons of Multitasking

- Pro: may improve resource utilization if we can run some processes while others are blocking
- Pro: makes process interaction possible
- Con: virtualization introduces overhead (examples?)
- Con: possibly reduced overall throughput

Forms of Multitasking

- Cooperative multitasking: processes voluntarily give up control
- Preemptive multitasking: OS polices transitions (how?)
- Real-time systems: hard, fixed time constraints (Aircraft?)

What's in a process?

- Code, data, and stack
 - Usually (but not always) has its own address space
- Program state
 - CPU registers
 - Program counter (current location in the code)
 - Stack pointer
- Only one process can be running in the CPU at any given time!

ILLINOIS TECH

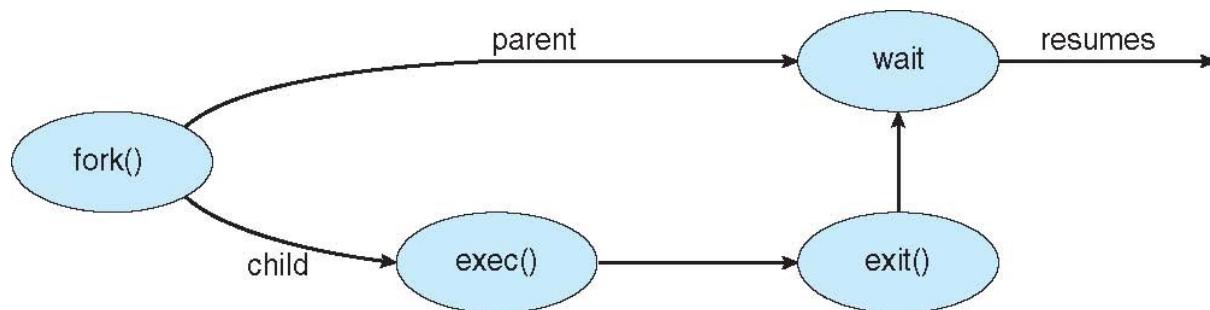
College of Computing

When is a process created?

- Processes can be created in two ways
 - System initialization: one or more processes created when the OS starts up
 - Execution of a process creation system call: something explicitly asks for a new process
- System calls can come from
 - User request to create a new process (system call executed from user shell)
 - Already running processes
 - User programs
 - System daemons

Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing options
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution options
 - Parent and children execute concurrently
 - Parent waits until children terminate



- Address space
 - Child duplicate of parent
 - Child has a program loaded into it
- UNIX examples
 - fork() system call creates new process
 - exec() system call used after a fork() to replace the process' memory space with a new program

Process Creation (Cont.)

ILLINOIS TECH

College of Computing

Metadata about a process

- Metadata examples:
 - PID, GID, UID
 - Allotted CPU time
 - Virtual → Physical memory mapping
 - Pending I/O operations

OS Data Structures

- Critical function of OS is to maintain data structures for keeping track of and managing all current processes
- Layout of many structures are dictated by hardware
 - e.g., VM structures, interrupt stack frame

When do processes end?

- Conditions that terminate processes can be
 - Voluntary
 - Involuntary
- Voluntary
 - Normal exit
 - Error exit
- Involuntary
 - Fatal error (only sort of involuntary)
 - Killed by another process

ILLINOIS TECH

College of Computing

Process Termination

- Process executes last statement and then asks the operating system to delete it using the exit() system call.
 - Returns status data from child to parent (via wait())
 - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the abort() system call. Some reasons for doing so:
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

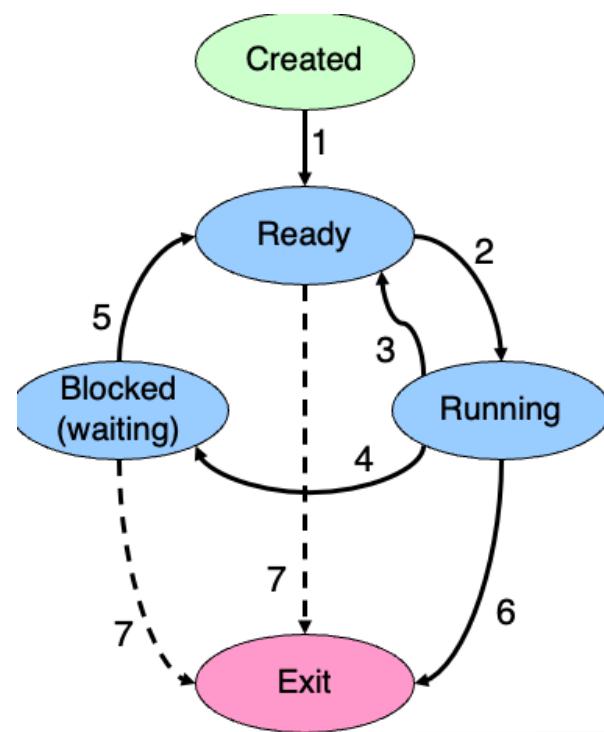
Process Termination (cont.)

- Some operating systems do not allow a child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
 - cascading termination. All children, grandchildren, etc. are terminated.
 - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process `pid = wait(&status);`
- If no parent waiting (did not invoke `wait()`) process is a zombie
- If parent terminated without invoking `wait`, process is an orphan

Process hierarchies

- Parent creates a child process
 - Child processes can create their own children
- Forms a hierarchy
 - UNIX calls this a “process group”
 - If a process exits, its children are “inherited” by the exiting process’s parent

Process states



Process in one of 5 states:

- Created
- Ready
- Running
- Blocked
- Exit

Transitions between states:

- 1 - Process enters ready queue
- 2 - Scheduler picks this process
- 3 - Scheduler picks a different process
- 4 - Process waits for event (such as I/O)
- 5 - Event occurs
- 6 - Process exits
- 7 - Process ended by another process

Process Control Block (PCB)

- Aggregate per-process data entry is referred to as the Process Control Block (PCB)
- Implementation likely consists of many disparate structures

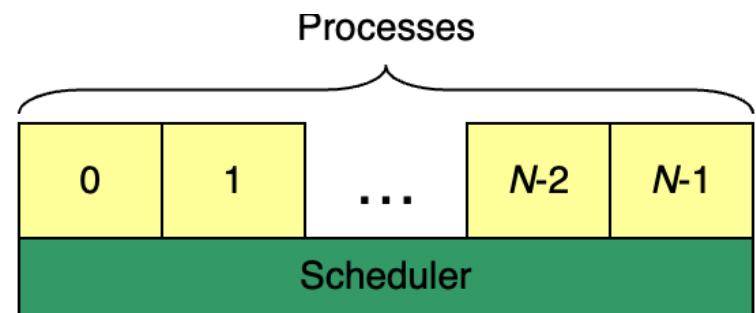
Part of the PCB

```
struct proc {  
    uint sz;  
    pde_t* pgdir;  
    char *kstack;  
    enum procstate state;  
    int pid;  
    later → struct proc *parent;  
    later → struct trapframe *tf;  
    struct context *context;  
    → void *chan;  
    int killed;  
    struct file *ofile[NFILE];  
    struct inode *cwd;  
    char name[16];  
};
```

Size of process memory
Page directory pointer for process
Kernel stack pointer
Files opened
Current working directory
Executable name

Processes in the OS

- Two “layers” for processes
- Lowest layer of process-structured OS handles interrupts, scheduling
- Above that layer are sequential processes
 - Processes tracked in the process table
 - Each process has a process table entry

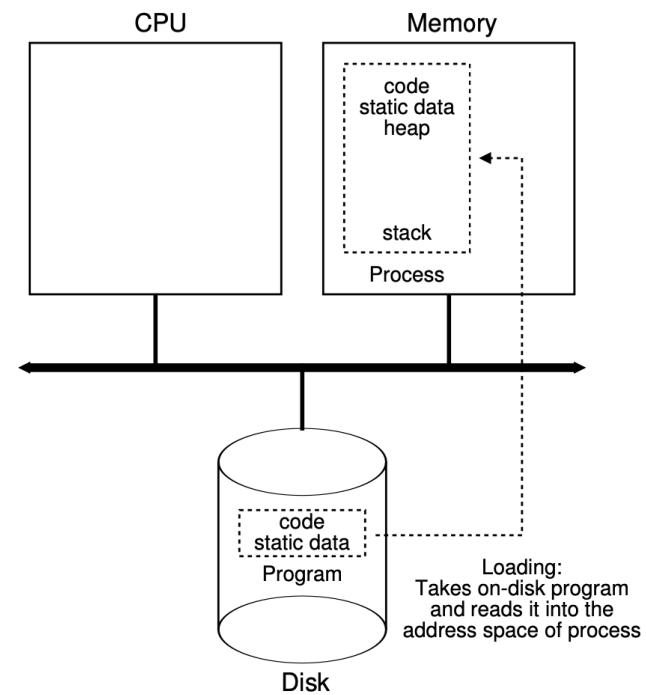


What's in a process table entry?

- Process management
 - Registers
 - Program counter
 - CPU status word
 - Stack pointer
 - Process state
- File management
 - Root directory
 - Working (current) directory...
- Memory management
 - Pointers to text, data, stack
 - Pointer to page table

Potatoes vs Potatoes

- Program vs process
- Database vs instance
- Program != process
- Database != instance



ILLINOIS TECH

College of Computing

Context Switches

- Multitasking via virtualization relies on seamlessly switching contexts between processes on hardware
 - Requires frequently saving/loading state to/from PCB
- At any point may have multiple processes ready to run
 - How does the scheduler pick the next process?

What happens on a trap/interrupt?

- Hardware saves program counter (on stack or in a special register)
- Hardware loads new PC, identifies interrupt
- Assembly language routine saves registers
- Assembly language routine sets up stack
- Assembly language calls C to run service routine
- Service routine calls scheduler
- Scheduler selects a process to run next (might be the one interrupted...)
- Assembly language routine loads PC & registers for the selected process

Scheduler

- Scheduler triggered to run when timer interrupt occurs or when running process is blocked on I/O
- Scheduler picks another process from the ready queue
- Performs a context switch

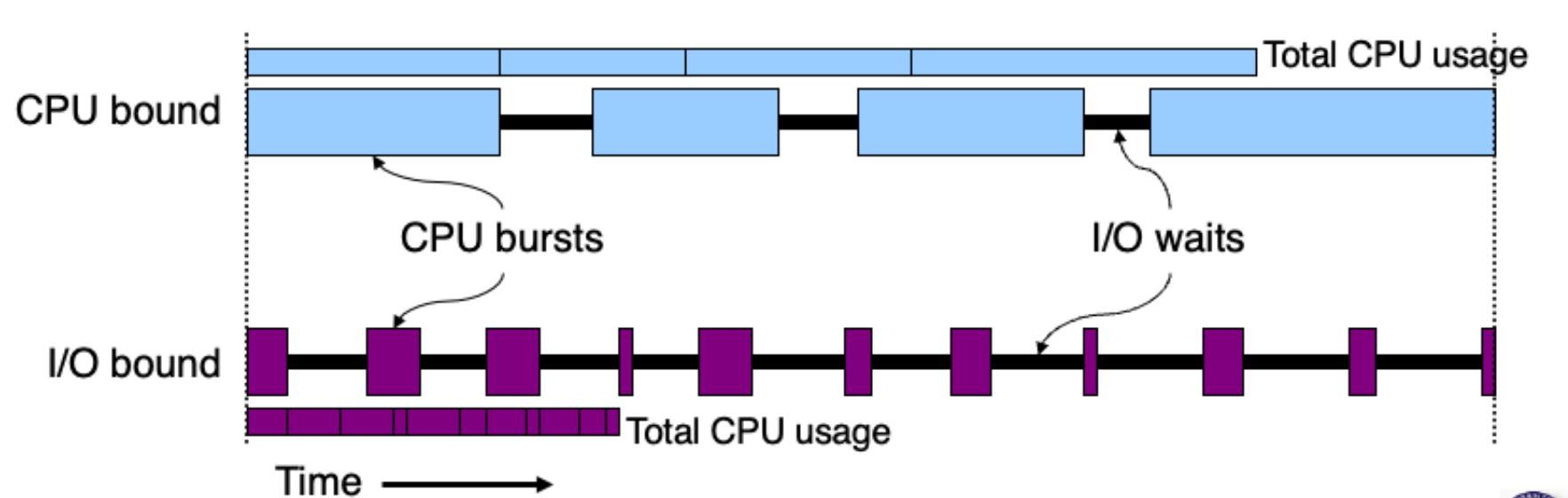
Why schedule processes?

- Bursts of CPU usage alternate with periods of I/O wait
- Some processes are CPU-bound: they don't make many I/O requests
- Other processes are I/O-bound and make many kernel requests

When are processes scheduled?

- At the time they enter the system
 - Common in batch systems
 - Two types of batch scheduling
 - Submission of a new job causes the scheduler to run
 - Scheduling only done when a job voluntarily gives up the CPU (i.e., while waiting for an I/O request)
- At relatively fixed intervals (clock interrupts)
 - Necessary for interactive systems
 - May also be used for batch systems
 - Scheduling algorithms at each interrupt, and picks the next process from the pool of “ready” processes

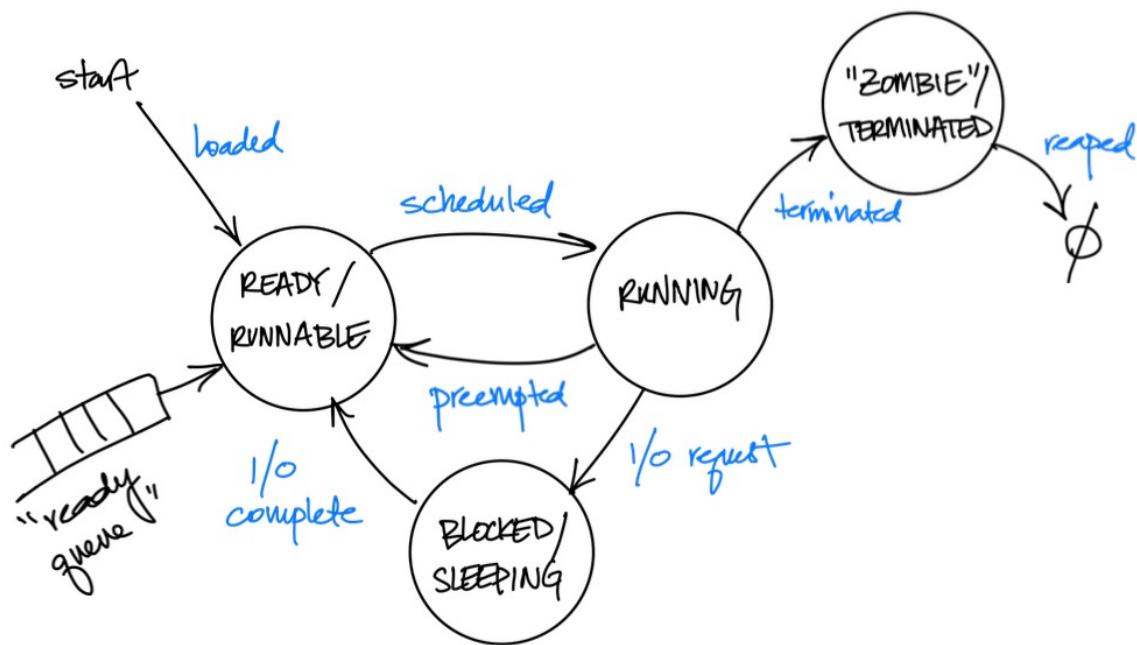
A Process over time



ILLINOIS TECH

College of Computing

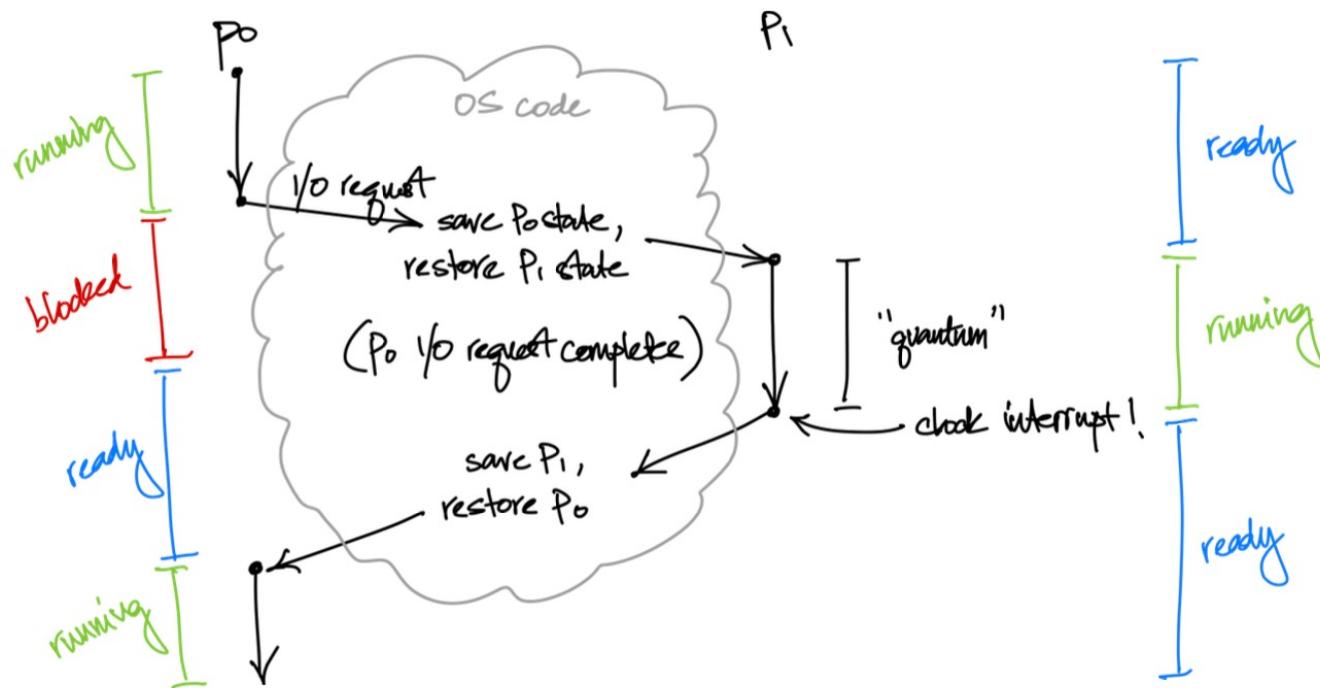
Processes over time and its life



ILLINOIS TECH

College of Computing

Showing how two processes switch



ILLINOIS TECH

College of Computing

The waiting game

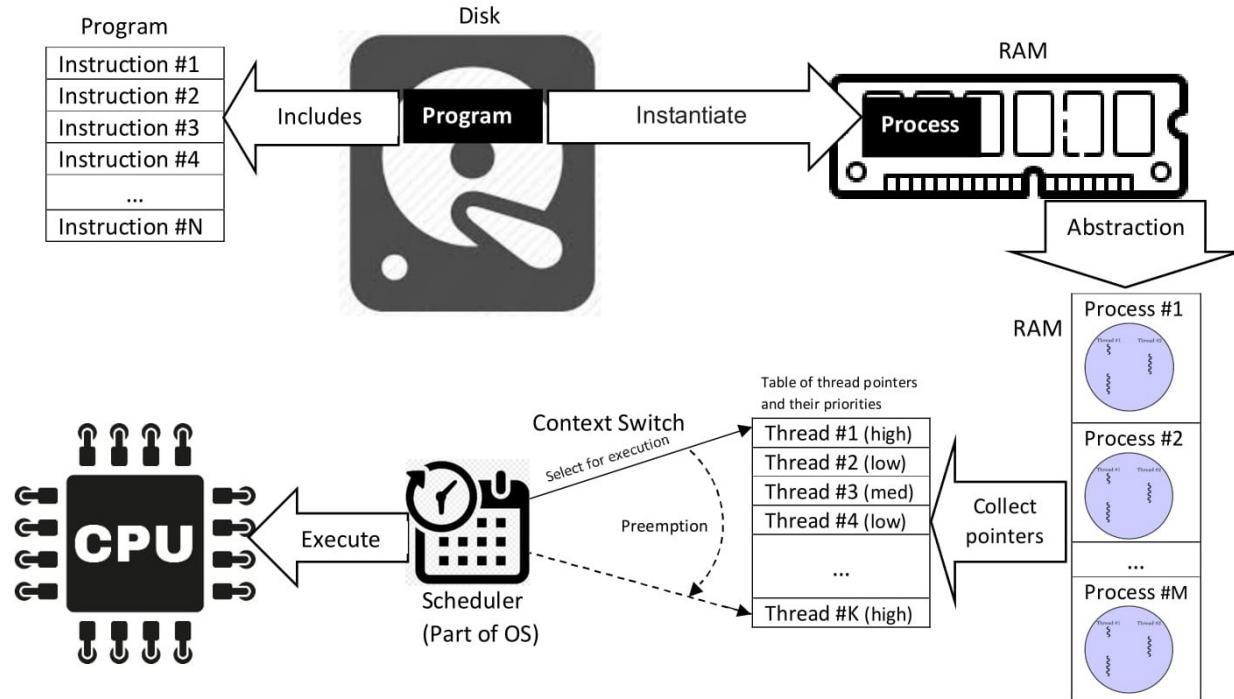
Time	Process ₀	Process ₁	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process ₀ now done
5	—	Running	
6	—	Running	
7	—	Running	
8	—	Running	Process ₁ now done

Time	Process ₀	Process ₁	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process ₀ initiates I/O
4	Blocked	Running	Process ₀ is blocked,
5	Blocked	Running	so Process ₁ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process ₁ now done
9	Running	—	
10	Running	—	Process ₀ now done

The overhead of fork()

- Why is there a lot of overhead when we fork?
- How do we combat this? Threads
 - A thread is a shared user space / address space
 - Lite weight compared to a fork()
 - Classic example webservers. Why?

Threads



ILLINOIS TECH

College of Computing

Questions

- What is a register, stack, heap, instruction pointer, stack pointer?
- What is a process?
 - what does it represent?
 - what does it abstract?
 - what is included in the abstraction?
- How does timesharing provide virtualization?
- What is the difference between the Ready and Running states?
- Why might you want to be able to create/destroy/suspend a process?
- Should there be a limit on the number of processes a user can run concurrently?

ILLINOIS TECH

College of Computing

CPU Virtualization Limited Direct Execution

Ben Lenard

blenard@iit.edu

ILLINOIS TECH

College of Computing

Agenda

- Recap of what a process?
- Central question: how to implement time-sharing?
- While maintaining OS control & maximizing performance
- “Limited direct execution”
- Mechanics of context switches

ILLINOIS TECH

College of Computing

Questions

- What is a register, stack, heap, instruction pointer, stack pointer?
- What is a process?
 - what does it represent?
 - what does it abstract?
 - what is included in the abstraction?
- How does timesharing provide virtualization?
- What is the difference between the Ready and Running states?
- Why might you want to be able to create/destroy/suspend a process?
- Should there be a limit on the number of processes a user can run concurrently?

ILLINOIS TECH

College of Computing

How to efficiently virtualize the CPU with control?

- The OS needs to share the physical CPU by time sharing.
- Issues:
 - Performance: How can we implement virtualization without adding excessive overhead to the system? Remember the whole goal is to increase utilization of the hardware.
 - Control: How can we run processes efficiently while retaining control over the CPU?

Direct Execution

- The OS loads process program, data, and arguments into predefined location(s), then points PC at entry point (e.g., main)
- When program terminates (e.g., return from main), OS cleans up process footprint (data/metadata)?
- How does the OS regain control or implement time sharing?

Direct Execution

1. Create entry for process list
2. Allocate memory for program
3. Load program into memory
4. Set up stack with argc / argv
5. Clear registers
6. Execute call main()
 - 7. Run main()
 - 8. Execute return from main()
9. Free memory of process
10. Remove from process list

Direct Execution

- Problems:
 - No concurrency
 - Process is unchecked – no real limits
 - How do we insure the process doesn't do something dangerous to the OS?
 - It's like handing some random person your car keys, when do you get it back?

Limited Direct Execution



ILLINOIS TECH

College of Computing

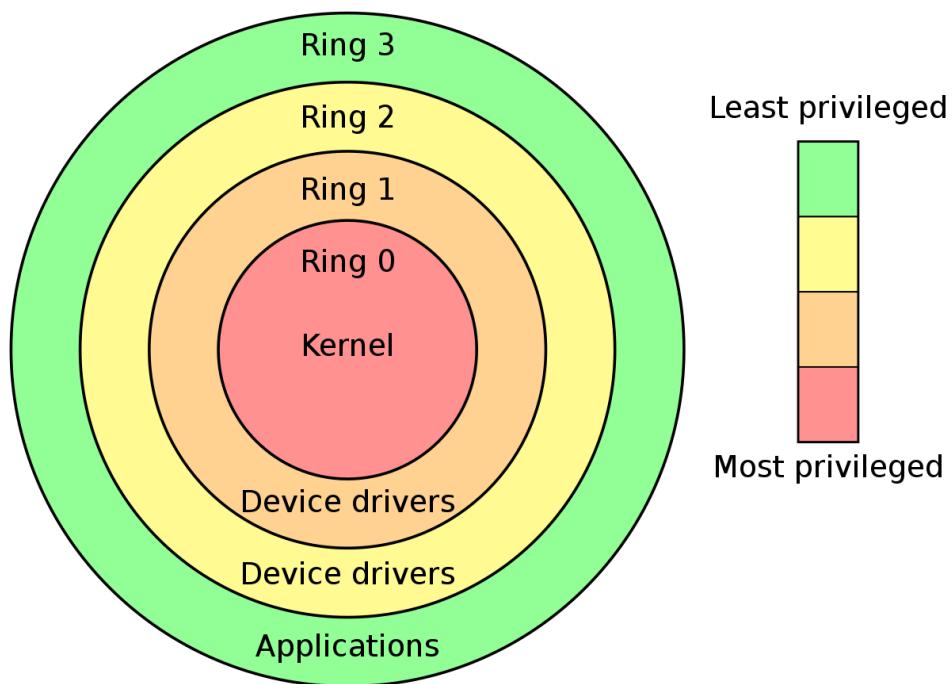
Limited Direct Execution

- Must prevent user from:
 - Accessing arbitrary memory addresses
 - Executing “dangerous” instructions
 - Access to I/O directly
 - System registers

Remember Kernel vs User Mode?

- Privileged instructions can only be executed in kernel mode
 - (what happens when user attempts to run?)
- On x86: ring/CPL flag in Code Selector register, 4 modes but 2 are commonly used: 0 = kernel, 3 = user
- After system boot, OS switches to user mode before delegating control to process

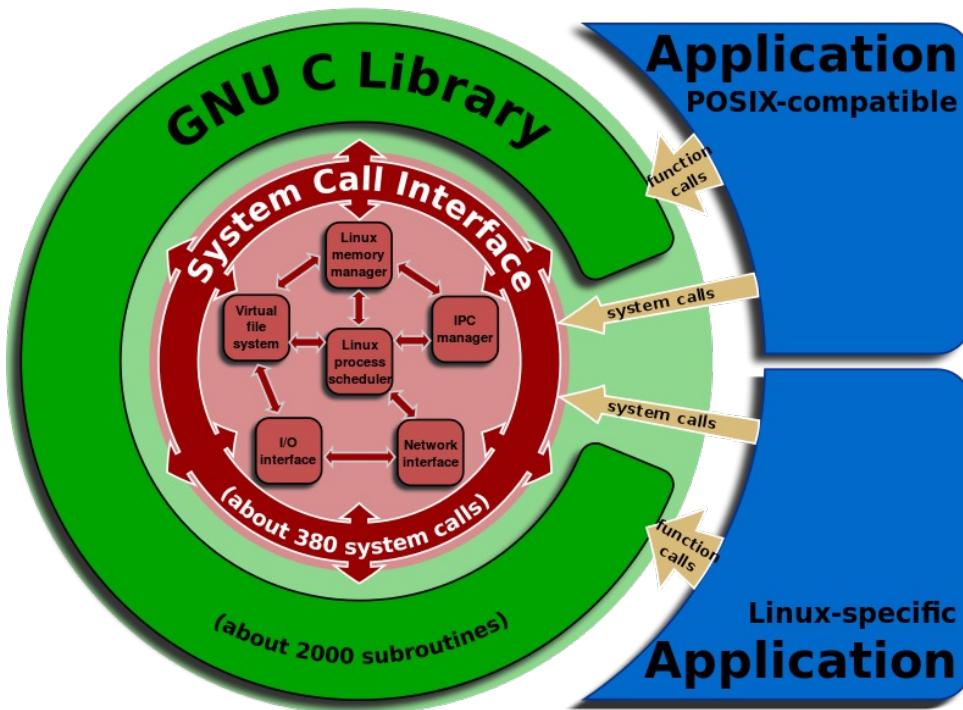
Protection Ring



ILLINOIS TECH

College of Computing

LINUX FOR EXAMPLE



ILLINOIS TECH

College of Computing

Restricted Operation

- What if a process wishes to perform some kind of restricted operation such as ...
 - Issuing an I/O request to a disk
 - Gaining access to more system resources such as CPU or memory
- Solution: Using protected control transfer (processor has to support it)
 - User mode: Applications do not have full access to hardware resources.
 - Kernel mode: The OS has access to the full resources of the machine

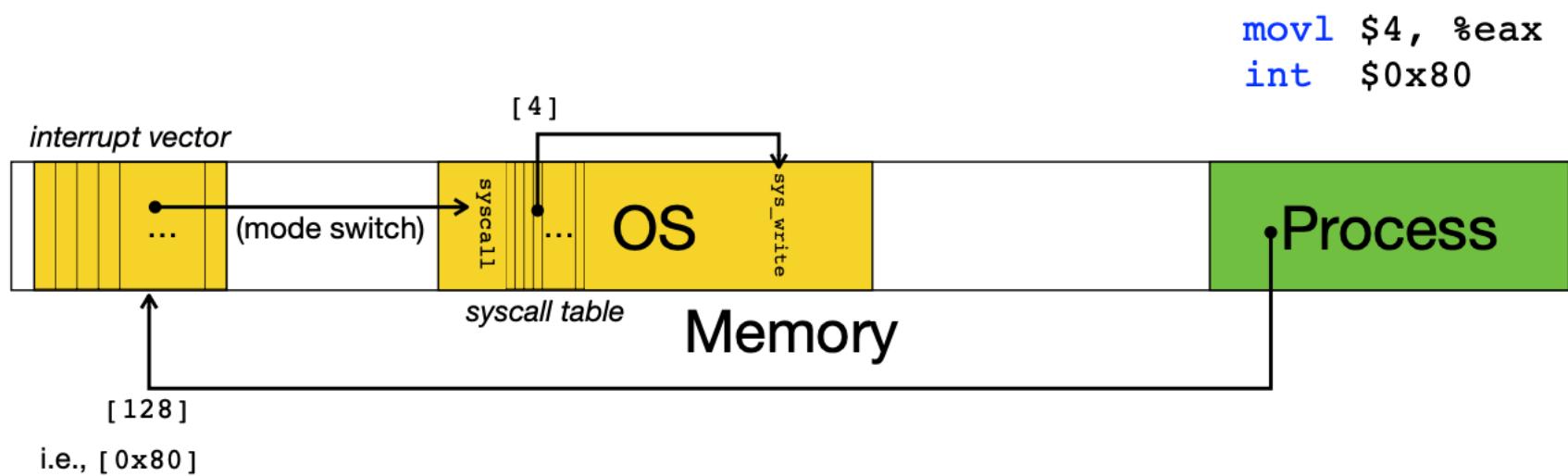
System Calls

- When user needs to perform I/O, invoke kernel-mode OS functions via system calls:
 - Accessing the file system
 - Creating and destroying processes
 - Communicating with other processes
 - Allocating more memory
- Looks like a regular function call, but isn't!

System Calls (Cont.)

char *str = "hello world";	movl len, %edx
int len = strlen(str);	movl str, %ecx
write(1, str, len);	movl \$1, %ebx
	movl \$4, %eax # syscall num
	int \$0x80 # trap instr

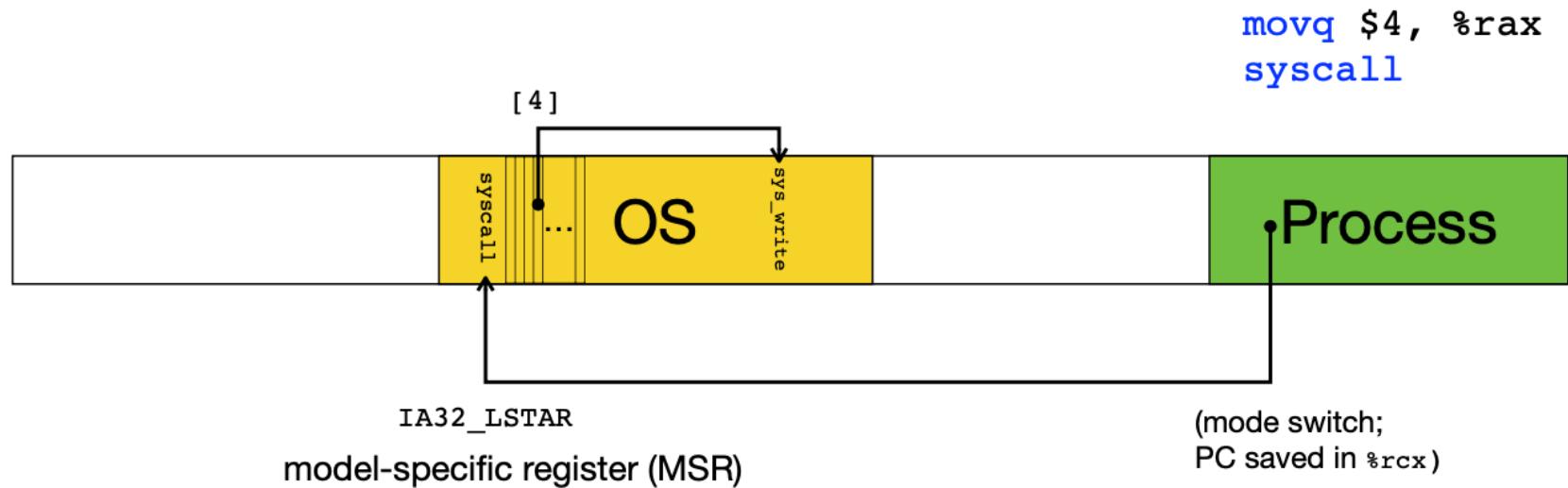
Trap Mechanism (x86)



ILLINOIS TECH

College of Computing

Trap Mechanism (x86-64) syscall added



ILLINOIS TECH

College of Computing

SYSTEM CALLS / TRAPS

- Defensive programming is key
- Why?
- Each call has its own number

ILLINOIS TECH

College of Computing

Who invented x86_64?

ILLINOIS TECH

College of Computing

AMD

- x86-64 is a 64-bit processing technology developed by AMD that debuted with the Opteron and Athlon 64 processor. x86-64 is also known as x64 and AMD64.

ILLINOIS TECH

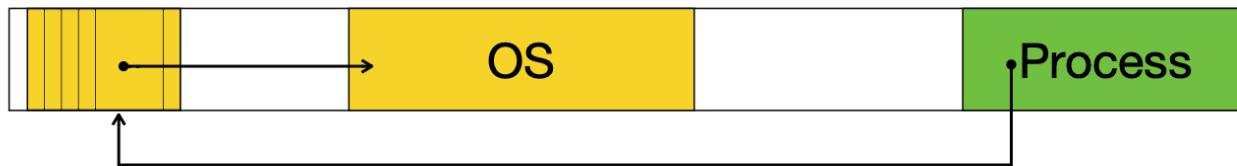
College of Computing

General Interrupt Mechanism

- IDTR (base address register) — populated by privileged lidtr instruction
 - 0-31 reserved for CPU-generated
 - 32-255 software configurable (for sw/hw interrupts) – not all are from User Mode

What to do?

- Problem: when transitioning to OS code, process state may be lost (e.g., PC, SP, etc.)
- Should save in case we return to process after servicing trap

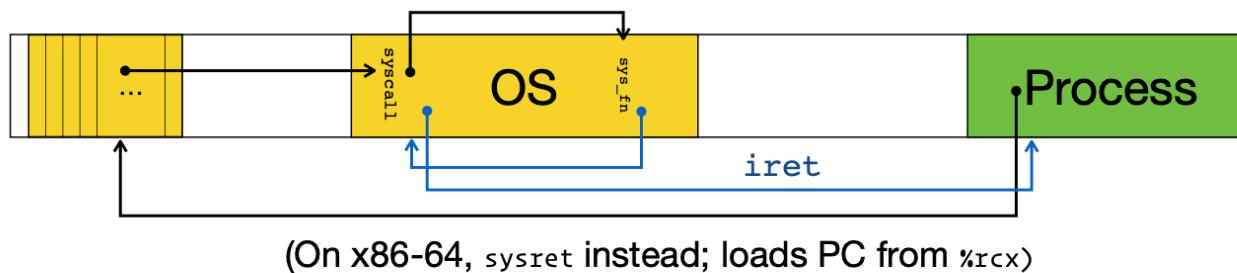


Saving Process State

- Hardware automatically saves current context during trap
- Where?
 - On kernel stack — automatically activated on mode switch
- Every process has its own separate kernel stack and it is used to keep track of kernel state (e.g., while handling I/O)

Restoring Process State

- “return from trap” instruction: iret — pops and restores trap frame and returns to process in user mode



Do we always immediately return to trapping process?

- Nope:
 - Process may be blocked (due to I/O request)
 - Scheduling decision

System Call (Cont.)

- Trap instruction
 - Jump into the kernel (how to tell where?)
 - Raise (the processor) privilege level to kernel mode
- Return-from-trap instruction
 - Return into the calling user program
 - Reduce (the processor) privilege level back to user mode

System Call (Cont.)

OS @ boot (kernel mode)	Hardware	
initialize trap table	remember address of ... syscall handler	
OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC return-from -trap	restore regs from kernel stack move to user mode jump to main	Run main() ... Call system trap into OS

ILLINOIS TECH

College of Computing

System Call (Cont.)

OS @ run (kernel mode)	Hardware	Program (user mode)
<i>(Cont.)</i>		
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall return-from-trap	restore regs from kernel stack move to user mode jump to PC after trap	
		... return from main trap (via exit())
Free memory of process Remove from process list		

ILLINOIS TECH

College of Computing

Switching Between Processes

- How can the OS regain control of the CPU so that it can switch between processes?
 - A cooperative Approach: Wait for system calls
 - A Non-Cooperative Approach: The OS takes control

A cooperative Approach: Wait for system calls

- Processes periodically give up the CPU by making system calls such as yield.
 - The OS decides to run some other task.
 - Application also transfer control to the OS when they do something illegal. i.e. Divide by zero
 - Try to access memory that it shouldn't be able to access
- Early versions of the Macintosh OS, The old Xerox Alto system

What can go wrong?

- A process gets stuck in an infinite loop = Reboot the machine

A Non-Cooperative Approach: OS Takes Control

- A timer interrupt
- During the boot sequence, the OS starts the timer (hardware).
- The timer raises an interrupt every so many milliseconds. (hardware)
- When the interrupt is raised :
 - The currently running process is halted.
 - Save enough of the state of the program.
 - A pre-configured interrupt handler in the OS runs.

What does this fix?

- The OS can jump back into execution.

ILLINOIS TECH

College of Computing

Context Switch

- A piece of assembly code
 - Save a few register values for the current process onto its kernel stack ¢ General purpose registers
 - kernel stack pointer
- Restore a few for the soon-to-be-executing process from its kernel stack
- Switch to the kernel stack for the soon-to-be-executing process

Limited Direct Execution (Timer)

OS @ boot (kernel mode)	Hardware	
initialize trap table	remember address of ... syscall handler timer handler	
start interrupt timer	start timer interrupt CPU in X ms	
OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A ...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	

ILLINOIS TECH

College of Computing

Timer (Cont.)

OS @ run (kernel mode)	Hardware	Program (user mode)
<i>(Cont.)</i>		
Handle the trap Call switch() routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) return-from-trap (into B)		restore regs(B) from k-stack(B) move to user mode jump to B's PC

ILLINOIS TECH

College of Computing

XV6 Context Switch code

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.

.globl swtch
swtch:
    movl 4(%esp), %eax
    movl 8(%esp), %edx

    # Save old callee-saved registers
    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi

    # Switch stacks
    movl %esp, (%eax)
    movl %edx, %esp

    # Load new callee-saved registers
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
~
```

ILLINOIS TECH

College of Computing

Questions?

- What does architecture mean?
 - How does it differ from OS?
- What is the key question/problem addressed in this chapter?
 - Why not just run processes directly on the CPU & let them yield when done?
 - Why not have the OS fetch, decode, & execute each instruction of a program instead?
- What is an interrupt?
- Restricted Operations
- Why are there restricted operations for processes?
- What is user mode? kernel mode?
 - What type of operations are restricted in user- mode?
 - who/what provides these two modes?

ILLINOIS TECH

College of Computing

x86 & xv6 overview

Ben Lenard

blenard@iit.edu

ILLINOIS TECH

College of Computing

Agenda

- Recap of what is CPU Virtualization and Limited Direct Execution?
- Motivation
- x86 ISA
- PC architecture
- UNIX
- xv6

ILLINOIS TECH

College of Computing

Questions

- What does architecture mean?
 - How does it differ from OS?
- What is the key question/problem addressed in this chapter?
 - Why not just run processes directly on the CPU & let them yield when done?
 - Why not have the OS fetch, decode, & execute each instruction of a program instead?
- What is an interrupt?
- Restricted Operations
- Why are there restricted operations for processes?
- What is user mode? kernel mode?

What type of operations are restricted in user- mode?

- who/what provides these two modes?

ILLINOIS TECH

College of Computing

- x86

ILLINOIS TECH

College of Computing

Documentation

- Intel IA-32 Software Developer's Manuals are complete references
 - Volume 1: Architectural Overview
 - Volume 2: Instruction Set Reference
 - **Volume 3: Systems Programming Guide**
- Many diagrams in slides taken from them

x86 coverage

- Timeline
- Syntax
- Registers
- Instruction operands
- Instructions and sample usage
- Processor modes
- Interrupt & Exception handling

ILLINOIS TECH

College of Computing

Timeline

- **1978:** Intel released 8086, a 16-bit CPU
- **1982:** 80186 and 80286 (still 16-bit)
- **1985:** 80386 was the first 32-bit x86 CPU (aka i386/IA-32)
- **2000:** AMD created x86-64: 64-bit ISA compatible with x86
- **2001:** Intel released IA-64 “Itanium” ISA, incompatible with x86
 - End-of-life announced in 2019 (i.e., official failure)
 - Came from a joint venture with HP/HPE
 - HP/HPE Used it in Nonstop / Tandom

x86 ISA

- xv6 uses the IA-32 ISA
 - But we can still build/run it on x86-64!
- x86 is a CISC ISA, so we have:
 - Memory operands for non-load/store instructions
 - Complex addressing modes
 - Relatively large number of instructions

ILLINOIS TECH

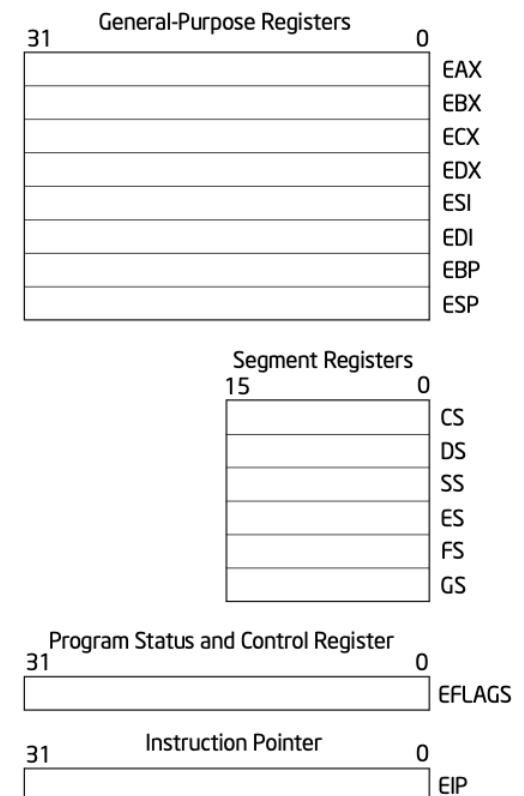
College of Computing

Syntax / Formatting

- Two common variants: Intel and AT&T syntax
- Intel syntax common in Windows world
 - e.g., `mov DWORD PTR [ebp-4], 10` ; format: OP DST, SRC
- AT&T syntax common in UNIX world (default GCC output)
 - e.g., `movl $10, -4(%ebp)` # format: OP SRC, DST

Registers

- 8 general-purpose registers
- 6 segment registers for addressing
- Status & Control register
- Program counter / Instruction pointer
- (Many others — including control registers — coming up later)



ILLINOIS TECH

College of Computing

General purpose registers

- Can be directly manipulated, but some have special applications
- Most can be accessed as full 32-bit values, or as 16/8-bit subvalues
- Each register is, by convention, volatile or non-volatile
 - A volatile register may be clobbered by a function call; i.e., its value should be saved — maybe on the stack — if it must be preserved
 - A non-volatile register is preserved (by callees) across function calls

General-Purpose Registers							
31	16	15	8	7	0		
	AH		AL				
	BH		BL				
	CH		CL				
	DH		DL				
	BP						
	SI						
	DI						
	SP						

16-bit 32-bit

AX	EAX
BX	EBX
CX	ECX
DX	EDX
	EBP
	ESI
	EDI
	ESP

Register	Purpose
%eax	Return value
%ebx	—
%ecx	Counter
%edx	—
%ebp	Frame/Base pointer
%esi	Source index (for arrays)
%edi	Destination index (for arrays)
%esp	Stack pointer

%eax, %ecx, %edx are volatile registers

ILLINOIS TECH

College of Computing

Instruction Operands

Mode	Example(s)	Meaning
Immediate	\$0x42, \$0xd00d	Literal value
Register	%eax, %esp	Value found in register
Direct	0x4001000	Value found in address
Indirect	(%esp)	Value found at address in register
Base-Displacement	8(%esp), -24(%ebp)	Given D(B), value found at address D+B (i.e., address in base register B + numeric offset D)
Scaled Index	8(%esp,%esi,4)	Given D(B,I,S), value found at address D+B+I×S $S \in \{1, 2, 4, 8\}$; D and I default to 0 if left out, S defaults to 1

Memory references

ILLINOIS TECH

College of Computing

Instructions

- Instructions have 0-3 operands
 - For many 2 operand instructions, one operand is both read and written
 - e.g., addl \$1, %eax # %eax = %eax + 1
- Instruction suffix indicates width of operands (l/w/b → 32/16/8 bits)
- Arithmetic operations populate EFLAGS register bits, including ZF (zero result), SF (signed/neg result), CF (carry-out of MSB occurred), OF (overflow occurred)
- Used by subsequent conditional instructions (e.g., jump if result = zero)

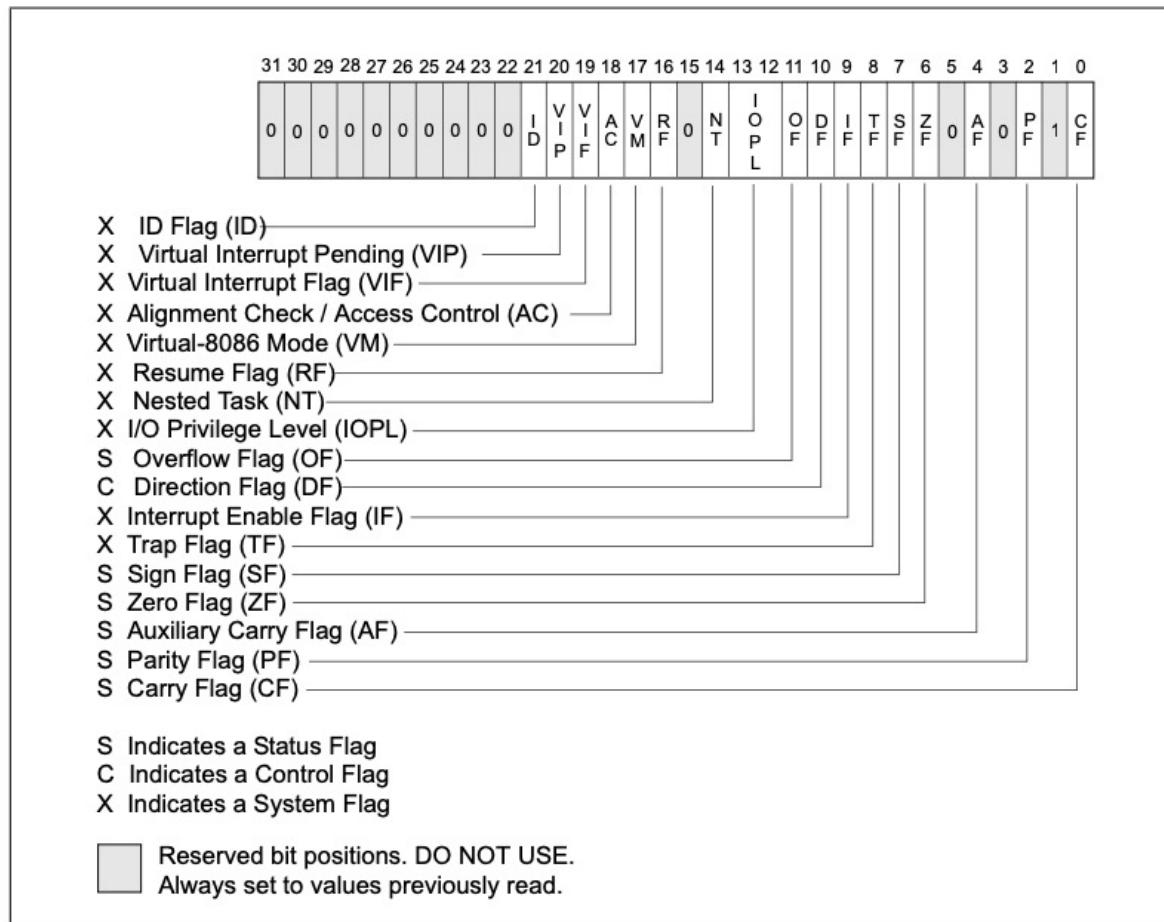


Figure 3-8. EFLAGS Register

ILLINOIS TECH

College of Computing

Arithmetic

Instruction(s)	Description
{add,sub,imul} src, dst	$dst = dst \{+,-,\times\} src$
neg dst	$dst = -dst$
{inc,dec} dst	$dst = dst \{+,-\} 1$
{sal,sar,shr} src, dst	$dst = dst \{<<,>>,>>>\} src$ (arithmetic & logical shifts)
{and,or,xor} src, dst	$dst = dst \{\&, ,^\}$ src (bitwise)
not dst	$dst = \sim dst$ (bitwise)

src can be an immediate, register, or memory operand; *dst* can be a register or memory operand.
But at most one memory operand!

ILLINOIS TECH

College of Computing

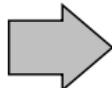
Conditions and Branches

Instruction(s)	Description
<code>cmp src, dst</code>	dst - src (discard result but set flags)
<code>test src, dst</code>	dst & src (discard result but set flags)
<code>jmp target</code>	Unconditionally jump to target (change %eip)
<code>{je,jne} target</code>	Jump to target if dst equal/not equal src (ZF=1 / ZF=0)
<code>{jl,jle} target</code>	Jump to target if dst </≤ src (SF≠OF / ZF=1 or SF=OF)
<code>{jg,jge} target</code>	Jump to target if dst >/≥ src (ZF=0 and SF=OF / SF≠OF)
<code>{ja,jb} target</code>	Jump to target if dst above/below src (CF=0 and ZF=0 / CF=1)

target is usually an address encoded as an immediate operand (e.g., `jmp $0x4001000`), but addresses may be stored in a register or memory, in which case *indirect addressing* is required, which uses the * symbol.
E.g., `jmp *%eax` (jump to address in `%eax`), `jmp *0x4001000` (jump to address found at address `0x4001000`)

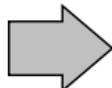
Basic Control Structures

```
if (cond) {  
    // if-clause  
} else {  
    // else-clause  
}  
...
```



```
testl %eax, %eax # %eax = cond  
je ELSE  
# if-clause  
jmp ENDIF  
ELSE:  
# else-clause  
ENDIF:  
# ...
```

```
while (cond) {  
    // loop-body  
}  
...
```



```
testl %eax, %eax # %eax = cond  
je ENDLOOP  
LOOP:  
# loop-body  
testl %eax, %eax  
jne LOOP  
ENDLOOP:  
# ...
```

ILLINOIS TECH

College of Computing

Data movement

Instruction(s)	Description
mov <i>src, dst</i>	Copy data from src to dst (memory→memory moves not possible)
movzbl <i>src, dst</i>	Copy 8-bit value to 32-bit target (& other variants), using zero-fill
movsbl <i>src, dst</i>	Copy 8-bit value to 32-bit target (& other variants), using sign-extension
{cmove/ne} <i>src, dst</i>	Move data from src to dst if ZF=1 / ZF=0
{cmovg/ge/l/le/a/b/...}	Conditionally move data from src to dst (per jump naming conventions)

Address computation

lea *address, dst*

dst = address (no memory access! just computes value of address)

ILLINOIS TECH

College of Computing

Functions and Call stack

Instruction(s)	Description
push src	Push src onto stack
pop dst	Pop top of stack into dst
call target	Push current %eip (address of instruction after call) onto stack, jump to target
leave	Restore frame pointer (%ebp) and clears stack frame
ret	Pop top of stack into %eip

All instructions above implicitly adjust %esp and access the stack.

target may use *indirect addressing* as well, e.g., `call *%eax` (call function whose address is in %eax)

ILLINOIS TECH

College of Computing

Function calls

- Functions make extensive use of the call stack — leads to convention-driven prologue and epilogue blocks in assembly code
- Typical function prologue:
 - Save old frame pointer and establish new frame pointer
 - Save non-volatile register values we might clobber (“callee-saved”)
 - Load needed parameters from prior stack frame
 - Allocate stack space for any local data

Function Calls (cont.)

- Typical function epilogue:
 - Place return value in %eax
 - Deallocate any space used for local data
 - Restore/Pop any clobbered non-volatile register values\
 - Restore/Pop old frame pointer
 - Return

```

int main() {
    int x=10, y=20;
    sum(x, y);
    return 0;
}

int sum(int a, int b) {
    int ret = a + b;
    return ret;
}

```

```

main:
    pushl %ebp
    movl %esp, %ebp
    subl $16, %esp
    movl $10, -4(%ebp)
    movl $20, -8(%ebp)
    movl -4(%ebp), %edi
    movl -8(%ebp), %esi
    call sum
    movl $0, %eax
    addl $16, %esp
    popl %ebp
    ret

```

```

sum: # unoptimized
    pushl %ebp
    movl %esp, %ebp
    movl %edi, -4(%ebp)
    movl %esi, -8(%ebp)
    movl -4(%ebp), %eax
    addl -8(%ebp), %eax
    movl %eax, -12(%ebp)
    movl -12(%ebp), %eax
    popl %ebp
    ret

sum: # optimized
    leal (%edi,%esi), %eax
    ret

```

ILLINOIS TECH

College of Computing

Processor modes

- - When an x86 system first boots up, it runs in 16-bit real mode (8086 compatible) — all addresses reference “real” memory locations
- - 16/32-bit protected modes add privilege levels, virtual memory, and other mechanisms useful to the OS (e.g., for multitasking)
- - 64-bit long mode removes some instructions and adds 64-bit registers and addressing

Real mode addressing

- Only 16-bit registers, but support for 20-bit addresses (1MB address space) through the use of segment registers: CS, DS, ES, SS
 - Left-shift segment number by 4 (i.e., $\times 16$) to obtain base address, and add to offset to compute 20-bit physical address
- Code (via IP) and Stack (via SP and BP) accesses automatically use CS (code segment) and SS (stack segment) to compute addresses
 - e.g., if IP=0x4000 and CS=0x1100, CS:IP refers to physical address $0x1100 \times 16 + 0x4000 = 0x15000$

Protected mode

- Segment registers (expanded to CS, DS, SS, ES, FS, GS) no longer hold base addresses, but selectors
 - Selectors are used to load segment descriptors from a descriptor table which describe location/size/status/etc. of segments
- CS selector contains a 2-bit CPL in addition to selector value
 - Recall: privileged instructions are only available when CPL=0

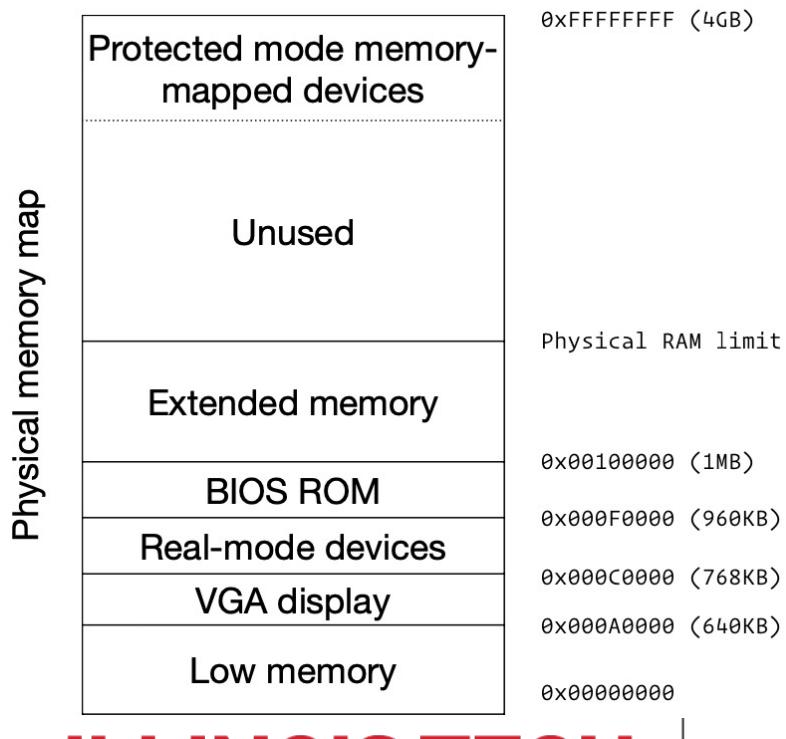
PC Architecture

ILLINOIS TECH

College of Computing

What else?

- Memory + memory layout
- Persistent store (disk)
- Text/graphics display
- Keyboard/Mouse + other I/O devices and controllers
- BIOS, Clock



ILLINOIS TECH

College of Computing

Startup & BIOS

- On startup, PC transfers control to address FFFF:0000 (real mode)
- BIOS executes power on self test, initializes video card, disk controller, and sets up basic interrupt routines for simple I/O
- If boot drive is found, load boot sector (512 bytes, tagged with ending 0x55AA marker) from drive at address 0000:7C00

Bootloader Responsibilities

- Set up minimal execution environment (stack, protected mode)
- Scans disk for kernel image (may load second-stage bootloader to navigate partitions, file system, executable formats, etc.)
- Load kernel image at predetermined location in memory
- Transfer control to kernel
- - Bootloaders can get very complicated!
- - E.g., multistage boot loaders like Linux Loader (LILO) and Grand Unified Bootloader (GRUB) understand file systems and executable file formats

QEMU

ILLINOIS TECH

College of Computing

Full System Emulator

- Emulates the behavior of a real x86 PC in software
- Simulates physical memory map and I/O devices
- Supports up to 255 CPUs (speed dependent on host machine)
- Simple to debug, and won't break your actual OS!
- Can connect to GDB to “step” through instructions

XV6

ILLINOIS TECH

College of Computing

XV6 system calls

System call

fork()
exit()
wait()
kill(pid)
getpid()
sleep(n)
exec(filename, *argv)
sbrk(n)
open(filename, flags)
read(fd, buf, n)
write(fd, buf, n)
close(fd)
dup(fd)

Description

Create process
Terminate current process
Wait for a child process to exit
Terminate process pid
Return current process's id
Sleep for n seconds
Load a file and execute it
Grow process's memory by n bytes
Open a file; flags indicate read/write
Read n bytes from an open file into buf
Write n bytes to an open file
Release open file fd
Duplicate fd

ILLINOIS TECH

College of Computing

File Descriptors – basic cat

```
char buf[512];
int n;
for(;;){
    n = read(0, buf, sizeof buf);
    if(n == 0)
        break;
    if(n < 0){
        fprintf(2, "read error\n");
        exit(); }
    if(write(1, buf, n) != n){
        fprintf(2, "write error\n");
        exit(); }
}
```

ILLINOIS TECH

College of Computing

FD's (cont).

```
mkdir("/dir");
fd = open("/dir/file", O_CREATE|O_WRONLY);
close(fd);
mknod("/console", 1, 1);
```

Syscall numbers – syscall.h

```
/* System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep   13
#define SYS_uptime 14
#define SYS_open    15
#define SYS_write   16
#define SYS_mknod   17
#define SYS_unlink  18
#define SYS_link    19
#define SYS_mkdir   20
#define SYS_close   21
~
```

Syscall.c

```
// Fetch the nth word-sized system call argument as a
// string.  Check that the pointer is valid and the string is
// (There is no shared writable memory, so the string
// between this check and being used by the kernel.)
int
argstr(int n, char ***pp)
{
    int addr;
    if(argint(n, &addr) < 0)
        return -1;
    return fetchstr(addr, pp);
}

extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
};
```

ILLINOIS TECH

College of Computing

Sysproc.c

```
// return how many clock tick interrupts have occurred
// since start.
int
sys_uptime(void)
{
    uint xticks;
    acquire(&tickslock);
    xticks = ticks;
    release(&tickslock);
    return xticks;
```

ILLINOIS TECH

College of Computing

Usys.S

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
.globl name; \
name: \
    movl $SYS_ ## name, \
    int $T_SYSCALL; \
ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL	mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
~
```

ILLINOIS TECH

College of Computing

User.h

```
struct stat;
struct rtcdate;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
```

ILLINOIS TECH

College of Computing

Homework 2

- These file are what you'll need to edit

ILLINOIS TECH

College of Computing

Questions?

ILLINOIS TECH

College of Computing

Security Part 1

Ben Lenard

blenard@iit.edu

ILLINOIS TECH

College of Computing

Agenda

- Recap of Homework2 and Questions
- Our TA - Subramanya Ganesh
- Security Part 1

ILLINOIS TECH

College of Computing

Homework2

- Any questions since I reposted it on Friday?

ILLINOIS TECH

College of Computing

Our TA

- Subramanya Ganesh
- sganesh10@hawk.iit.edu
- Tuesday and Thursday Office hours

ILLINOIS TECH

College of Computing

Security

ILLINOIS TECH

College of Computing

Intro

- Security has been a vital topic as money has been lost due to security exploits.
- Everything software-wise runs on top of the OS; therefore the OS needs to be secure.
- Even if your software doesn't contain flaws, an OS flaw can be exploited to gain access to your data
 - Consider a database and reading file contents
- What controls the hardware?
- Would you build a house on sand?

ILLINOIS TECH

College of Computing

Consider the OS

- Consider the OS
 - Multiple running applications
 - Multiple users
 - Multiple applications that could interact with one another
- One set of physical hardware
- Multi Tenancy

ILLINOIS TECH

College of Computing

TPM and Security Enclaves

- Trusted Platform Modules are developed to ensure that the desired OS and Kernel are booted
- General hardware elements have tried to control aspects of the machine such as cryptography.
 - These hardware elements are called security enclaves, since they are meant to allow only safe use of this data, even by the most powerful, trusted code in the system the operating system itself.

Security Goals at a high level

- Confidentiality – Only showing data to the people/apps with the correct permissions – this is different then chmod 777
 - Integrity – Ensure something is correct and not altered
 - Availability – Ensuring apps/people have access to the data.
-
- These areas are covered on the CISSP exam

ILLINOIS TECH

College of Computing

Non-Repudiation

- All about the proof
- Proof that someone received a message or file and cannot say they never received it.
- This will come back with service accounts

Designing Secure Systems

1. **Economy of mechanism** – KISS
2. **Fail-safe defaults** – Default to security, not insecurity. Similar to the credit card networks, default is decline
3. **Complete mediation** – This is a security term meaning that you should check if an action to be performed meets security policies every single time the action is taken. Often ignored.
4. **Open design** – Assume your adversary knows every detail of your design.
5. **Separation of privilege** – Require separate parties or credentials to perform critical actions. Similar to SOX in banking? Why do we have SOX?
6. **Least privilege** – Smaller attack vectors. Similar to why we have userspace.
7. **Least common mechanism** – For different users or processes, use separate data structures or mechanisms to handle them. Aka don't share. We'll talk about containers and switch root.
8. **Acceptability** – If your users won't use it, your system is worthless. Aka the sales pitch

ILLINOIS TECH

College of Computing

The Basics of OS Security

- System calls provide a method for the OS to provide security since the calls that require elevated privileges are funneled through one spot.
- For example in a process, the OS can use access control methods to decide if a process is authorized to perform a task
- XV6 does not have the notion of users
- Why is using ‘root’ bad?

What does sudo do?

- In your homework, I have you use sudo, what does it do?
- How does root's ulimits differ from a user?
- Do file permissions apply to root?

ILLINOIS TECH

College of Computing

The Principal

- Not someone from school
- Someone / entities who is requesting something
- Output from Kafka for example:
 - Adding ACLs for resource `ResourcePattern(resourceType=TOPIC, name=SIRIUS.pcm-monitoring, patternType=LITERAL)`:
 - (principal=User:BLAH, host=*, operation=DESCRIBE, permissionType=ALLOW)
 - (principal=User:BLAH, host=*, operation=READ, permissionType=ALLOW)

ILLINOIS TECH

College of Computing

The Agent

- Not 007
- When the OS performs a system call on behalf of a process, principal, is called the agent.
- In other words, the agent does the verification process before requesting a service on behalf of the principal.

The Object and credential

- In security terms, the object is the target of the request made by the broker.
- Look at Kafka
- The credential is the access decision that the OS keeps for future use.

Processes and identities

- Remember XV6 does not have users?
- Most if not all OS's have users even iOS
- What is a UID/GID?
- How does the OS know which user owns what process?
- The fork systemcall has the ability to set the UID & GID in the PCB

Authentication By What You Know

- Authentication by what you know is most commonly performed by using passwords.
- A password is a secret known only to the party to be authenticated.
- Passwords are stored as a hash
 - Consider /etc/shadow and /etc/passwd

Passwords and Dictionary Attacks

- Why when we are asked to create passwords now, we're asked for numbers and symbols?
- Dictionary Attacks
 - As it sounds, trying known words to break your password
- Most applications or OS's now lock the account after 5 failed logins or so
- At the bank we had to change our passwords every 90 days
- At Argonne we have multiple MFA's

Authentication by What You Have

- For example an ATM card
- For a computer, you can have:
 - A PIV/CAC card with an SSL certificate on it identifying you
 - A USB Key identifying you
 - RSA token

Authentication by What You Are

- Consider the iphone and face reader
 - How does that work?

ILLINOIS TECH

College of Computing

Authentication of Applications

- Passwords
- Tokens via a 3rd party

ILLINOIS TECH

College of Computing

Applications and user accounts

- Why not just run the application as root?
- Service accounts / application accounts
- Book talks about Apache
- Ulimits for the Application
- Why is it a bad idea to run an application as a real user id? (for Ben - CUG paper)

Ulimits – Linux – A user

```
$ ulimit -a
core file size      (blocks, -c) 0
data seg size       (kbytes, -d) unlimited
scheduling priority (-e) 0
file size           (blocks, -f) unlimited
pending signals     (-i) 384666
max locked memory   (kbytes, -l) 64
max memory size     (kbytes, -m) unlimited
open files          (-n) 1024
pipe size           (512 bytes, -p) 8
POSIX message queues (bytes, -q) 819200
real-time priority   (-r) 0
stack size           (kbytes, -s) 8192
cpu time             (seconds, -t) unlimited
max user processes    (-u) 4096
virtual memory        (kbytes, -v) unlimited
file locks            (-x) unlimited
```

ILLINOIS TECH

College of Computing

Ulimits - root

```
ulimit -a
core file size      (blocks, -c) 0
data seg size       (kbytes, -d) unlimited
scheduling priority (-e) 0
file size           (blocks, -f) unlimited
pending signals     (-i) 384666
max locked memory   (kbytes, -l) 64
max memory size     (kbytes, -m) unlimited
open files          (-n) 1024
pipe size           (512 bytes, -p) 8
POSIX message queues (bytes, -q) 819200
real-time priority   (-r) 0
stack size           (kbytes, -s) 8192
cpu time             (seconds, -t) unlimited
max user processes   (-u) 384666
virtual memory        (kbytes, -v) unlimited
file locks            (-x) unlimited
```

ILLINOIS TECH

College of Computing

SeLinux and App Armor

- Examples of mandatory access control (MAC) refers to a type of access control by which the operating system or database constrains the ability of a subject or initiator to access or generally perform some sort of operation on an object or target.

N Tier

- Consider an application where there's a webserver (Apache/Nginx), Application Server (Tomcat/Websphere), and database server (Oracle/Db2/Postgres).
- Why would it be a bad idea to run all three on the same server? Let's assume the server has 128 cores and 2TB of ram, so resource consumption isn't a concern.
- What is shared?

VM's for separation

- If we implement N Tier architecture, we either need VM's or physical servers.
- Think about the OS overhead

ILLINOIS TECH

College of Computing

Containers?

- Less overhead - Containers require less system resources than traditional or hardware virtual machine environments because they don't include operating system images.
- Increased portability - Applications running in containers can be deployed easily to multiple different operating systems and hardware platforms.
- Greater efficiency - Containers allow applications to be more rapidly deployed, patched, or scaled.

ILLINOIS TECH

College of Computing

Questions?

ILLINOIS TECH

College of Computing

Security Part 2

Ben Lenard

blenard@iit.edu

ILLINOIS TECH

College of Computing

Agenda

- Our TA - Subramanya Ganesh
- Security Part 2

ILLINOIS TECH

College of Computing

Our TA

- Subramanya Ganesh
- sganesh10@hawk.iit.edu
- Tuesday and Thursday Office hours 12pm to 1pm

ILLINOIS TECH

College of Computing

Security Part 2

ILLINOIS TECH

College of Computing

Access Control

- Basically it boils down to:
 - Figure out if the request fits within our security policy.
 - If it does, perform the operation. If not, make sure it isn't done.

Aspects Of The Access Control

- Subjects - A subject is the entity that wants to perform the access, perhaps a user or a process.
- Objects - An object is the thing the subject wants to access, perhaps a file or a device.
- Access - Access is some particular mode of dealing with the object, such as reading it or writing it.
- We sometimes refer to the process of determining if a particular subject is allowed to perform a particular form of access on a particular object as authorization.

Virtualization and Processes

- Since the OS uses virtualization aspects such as virtual memory, and allow the owning process to have control of that virtual memory segment, the OS is able to let process freely use that memory.

Access Control Lists (ACLs) Systems

- Consider /tmp/text
- User blenard wants to access it
- The OS has to lookup if blenard has an ACL to access it.
- Linux uses ACLs

Capability-Based Systems

- Consider /tmp/text, again
- User blenard wants to access it
- Blenard can only access it if the OS granted the capability to access it

Mandatory And Discretionary Access Control

- Mandatory access control (MAC) is a model is granted on a need to know basis: users have to prove a need for information before gaining access. MAC implements zero-trust principles with its control mechanisms. Scalability and Maintainability becomes a pain.
- Discretionary access control is an identity-based access control model that provides users with a certain amount of control over their data. Consider you issuing chmod and chown

Role-Based Access Control (RBAC)

- Give the role the permissions then add the user
- Consider Ben with 50+ dbs – easier on then adding users db by db
- Consider an enterprise with 10000's of systems and employees – my prior employer
- Furthermore, systems such as KeON can be deployed to enforce/deploy RBAC at the OS level

Privilege Escalation

- As it sounds, when a user needs privileges greater than what they have.
- In other words, corner case handling
- Consider sudo, do you need sudo all the time?

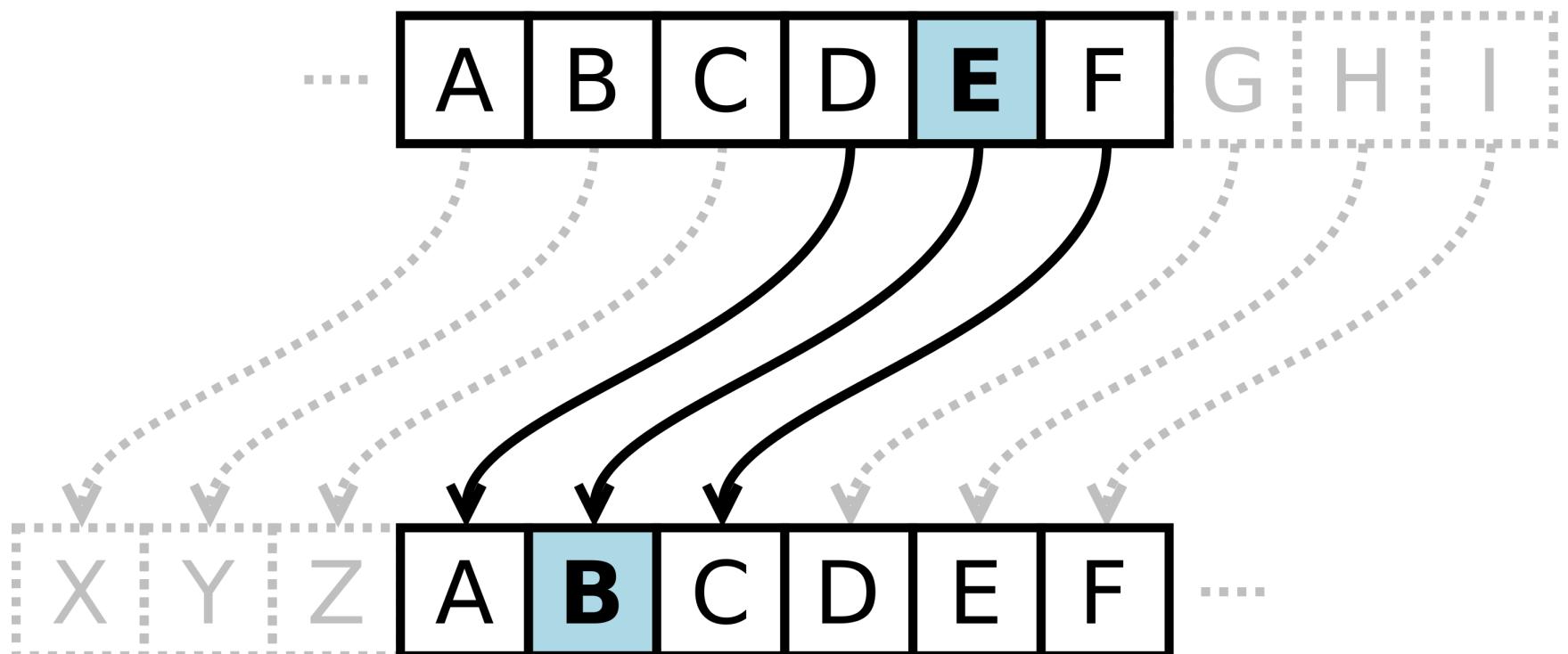
ILLINOIS TECH

College of Computing

Protecting Information With Cryptography

- Cryptography is technique of securing information and communications through use of codes so that only those person for whom the information is intended can understand it and process it. Thus preventing unauthorized access to information. The prefix “crypt” means “hidden” and suffix “graphy” means “writing”.
- Dates back to the Caesar cipher which is a shift cipher

Shift Cipher



ILLINOIS TECH

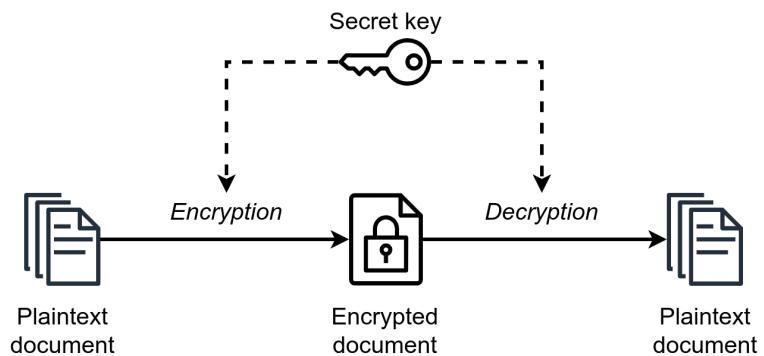
College of Computing

Basic Idea Behind Cryptography

- Consider: $C = E(P, K)$
 - C is Ciphertext
 - P is Plaintext
 - K is for the Key
 - E is the algorithm
- Similarly, to decrypt $P = D(C, K)$

DES and AES

- DES - Data Encryption Standard was the standard until 1999 when it was broken in 22 hours.
- AES - Advanced Encryption Standard became the new standard in 2001
- Both are symmetric-key algorithms



ILLINOIS TECH

College of Computing

Cryptographic Hash

- The cost of encrypting and decrypting data can be high, however if we need to verify integrity of data, hashing is a cheaper alternative.
- Cryptographic hashes are a special category of hash functions with several important properties:
 - It is computationally infeasible to find two inputs that will produce the same hash value.
 - Any change to an input will result in an unpredictable change to the resulting hash value.
 - It is computationally infeasible to infer any properties of the input based only on the hash value.

ILLINOIS TECH

College of Computing

Remember

- THE CRYPTOGRAPHY'S BENEFIT RELIES ENTIRELY ON THE SECRECY OF THE KEY.

ILLINOIS TECH

College of Computing

What does this have to do with the OS?

- Consider encrypting network traffic to and from the OS?
- Consider validating passwords? Are passwords kept in cleartext?
- What about sensitive data on the filesystem? There's PGP for files, Encrypted filesystems. How does this help to protect data?
- Data at rest is a concern. Why? What if someone gains access?
- Did you know we can ensure data deletion with encryption?

Distributed System Security

- An operating system can only control its own machine's resources.
- The issue becomes:
 - The other machines in the distributed system might not properly implement the security policies you want, or they might be adversaries impersonating trusted partners. We cannot control remote systems, but we still have to be able to trust validity of the credentials and capabilities they give us.
 - Machines in a distributed system communicate across a network that none of them fully control and that, generally, cannot be trusted. Adversaries often have equal access to that network and can forge, copy, replay, alter, destroy, and delay our messages, and generally interfere with our attempts to use the network.

ILLINOIS TECH

College of Computing

The Role of Authentication

ILLINOIS TECH

College of Computing

SSL/TLS

- As we know, SSL/TLS is used when you go to a website so your credit card information is secure.
- It can be used for verification of a server's identity.
- Assume we trust a certificate authority (CA)
- The CA will issue a certificate with the FQDN of the server
- The other party knows the server is legit since the certificate matches the FQDN and created by the trusted CA

Public Key Authentication For Distributed Systems

- Public Key authentication in distributed systems
- Similar to the steps in the SSL slide.
- My ELK cluster only talks to each other this way.
- Sometimes it's referred to as x509 certs

Passwords Authentication For Distributed Systems

- A password is a secret that only the authorized user should have.
- Quite often this authentication method is simpler
- For example, Kafka is composed of independent servers in a distributed fashion. Passwords can be used to authenticate to each other.

Man-in-the-middle Attacks

1. Alice sends a message to Bob, which is intercepted by Mallory:

Alice "Hi Bob, it's Alice. Give me your key." → Mallory Bob

2. Mallory relays this message to Bob; Bob cannot tell it is not really from Alice:

Alice Mallory "Hi Bob, it's Alice. Give me your key." → Bob

3. Bob responds with his encryption key:

Alice Mallory ← [Bob's key] Bob

4. Mallory replaces Bob's key with her own, and relays this to Alice, claiming that it is Bob's key:

Alice ← [Mallory's key] Mallory Bob

5. Alice encrypts a message with what she believes to be Bob's key, thinking that only Bob can read it:

Alice "Meet me at the bus stop!" [encrypted with Mallory's key] → Mallory Bob

6. However, because it was actually encrypted with Mallory's key, Mallory can decrypt it, read it, modify it (if desired), re-encrypt with Bob's key, and forward it to Bob:

Alice Mallory "Meet me at the van down by the river!" [encrypted with Bob's key] → Bob

7. Bob thinks that this message is a secure communication from Alice.

14 / 1

ILLINOIS TECH

College of Computing

Questions?

ILLINOIS TECH

College of Computing

Scheduling

Ben Lenard

blenard@iit.edu

ILLINOIS TECH

College of Computing

Agenda

- Homework 2
- Overview
 - Not just one scheduler – one size does not fit all
- Scheduling metrics
 - “Interactive” jobs and responsiveness
- Scheduling policies
 - FCFS, SJF, PSJF, RR, HPRN
 - MLQ, MLFQ

ILLINOIS TECH

College of Computing

Homework 2

- Any questions?

ILLINOIS TECH

College of Computing

Overview

ILLINOIS TECH

College of Computing

Definition

- Scheduling: policies & mechanisms used to allocate limited resources to some set of entities
- Initial focus: resource & entities = CPU & processes (aka jobs) - other possibilities:
 - resources: memory, I/O bus/devices
 - entities: threads, users, groups
- schedulers for the above may exist in an OS (and must play nice with each other)!

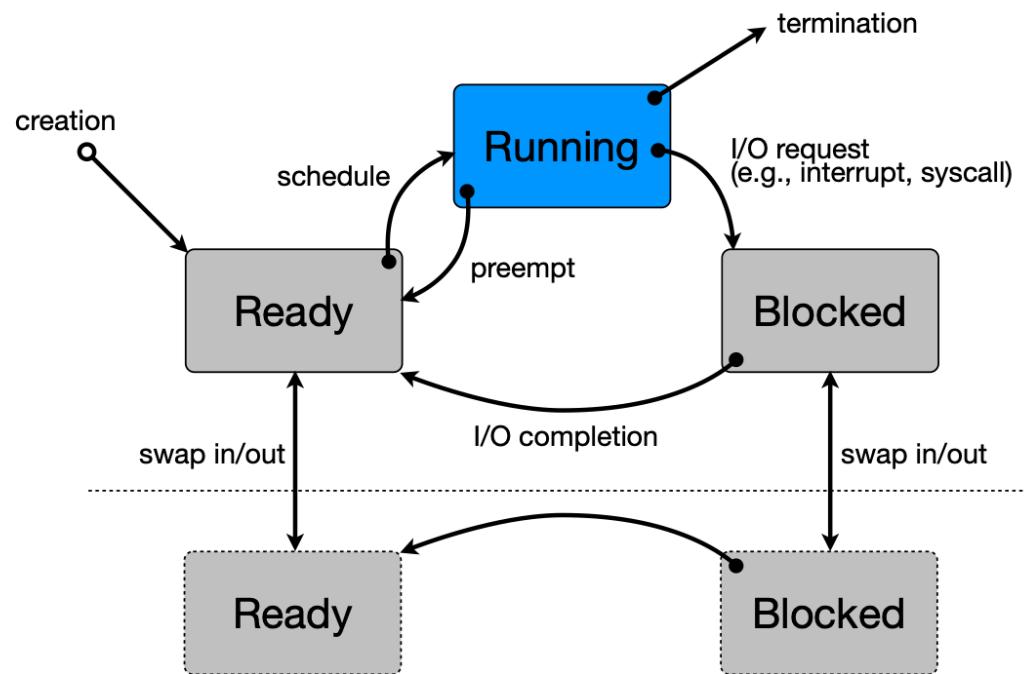
Policy

- high-level “what”
- scheduling disciplines
 - e.g., FCFS, SJF, RR, etc.
- driven by a variety of potentially conflicting goals
 - e.g., performance and fairness

Mechanism

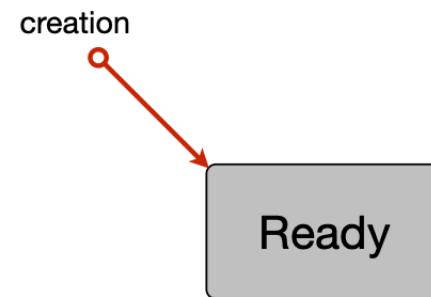
- low-level “how”
- combination of HW/SW
 - e.g., clock interrupt, high precision timer, PCB
- scattered throughout kernel codebase

Schedulers are concerned with transitions between process states in this context



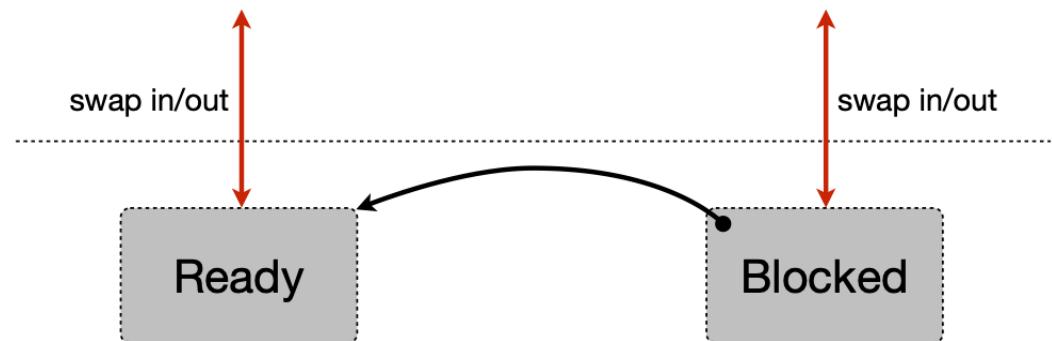
Domain of the “long-term” scheduler

- choose which jobs are admitted to the system
- may control mix of jobs (e.g., I/O vs. CPU bound) - not common in general-purpose, time-shared OSes



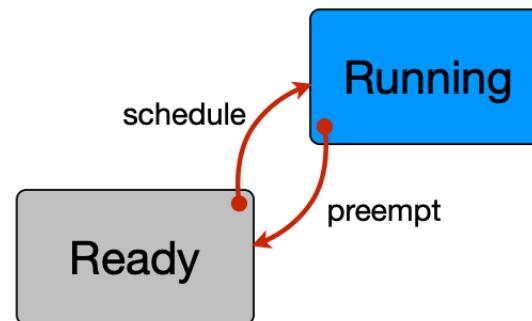
Domain of the “medium-term” scheduler

- swaps processes out to disk to make room for others
- active when there is insufficient memory
- runs much less frequently (slower!) than CPU scheduler



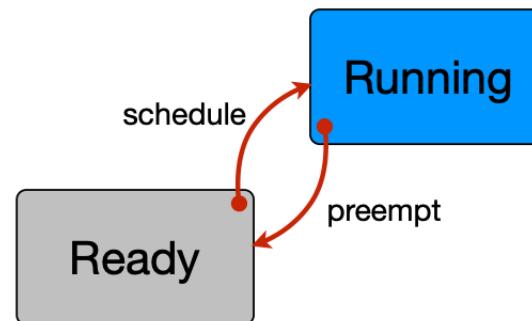
Domain of the “short-term” scheduler, i.e., the CPU scheduler

- chooses between in-memory, ready processes to run on CPU
- invoked to carry out scheduling policies after interrupts/traps



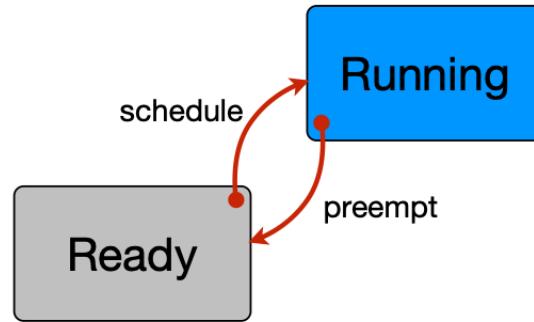
Domain of the “short-term” scheduler, i.e., the CPU scheduler

- chooses between in-memory, ready processes to run on CPU
- invoked to carry out scheduling policies after interrupts/traps



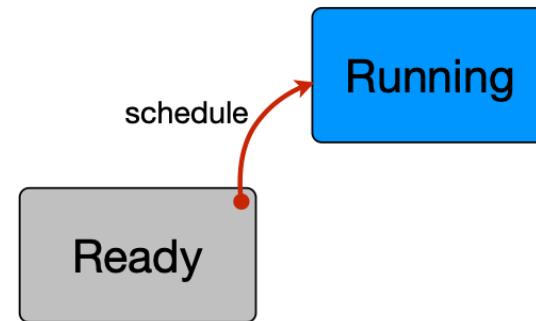
preemptive scheduling

- preemptive scheduling
- relies on clock interrupt
 - (to regain control of CPU)



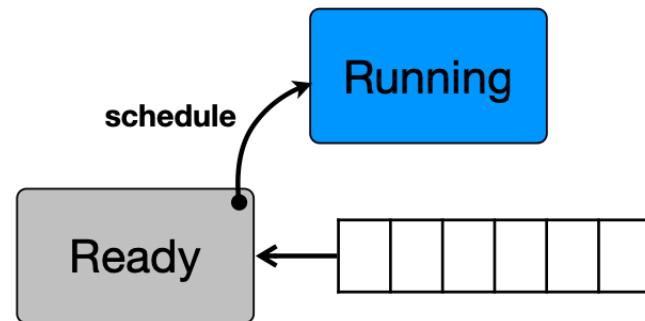
non-preemptive scheduling

- non-preemptive scheduling
- once a job starts, it continues until it terminates/blocks



Ready queue isn't always FIFO

- convenient to envision a ready queue (though not necessarily FIFO!)
- the scheduling policy decides which job to select from the set of ready (runnable) jobs to run next



High-level policy considerations

- Preemptive vs. Non-preemptive
- Information available for making informed decisions
- Depends on lower-level mechanisms available
 - Scheduling goals
- Based on optimizing/tuning scheduling metrics

Scheduling Metrics

ILLINOIS TECH

College of Computing

Some scheduling metrics

- Turnaround time
- Wait time
- Response time
- Throughput
- Utilization

Turnaround time

- Turnaround time
 - turnaround = completion - creation
 - i.e., total time to complete job
- Useful metric for a CPU-bound process — how much time is required to carry out a lengthy computation?
- Not generally a great yardstick for evaluating a scheduler!
 - What if job is I/O-bound?
 - What if job never “completes”?

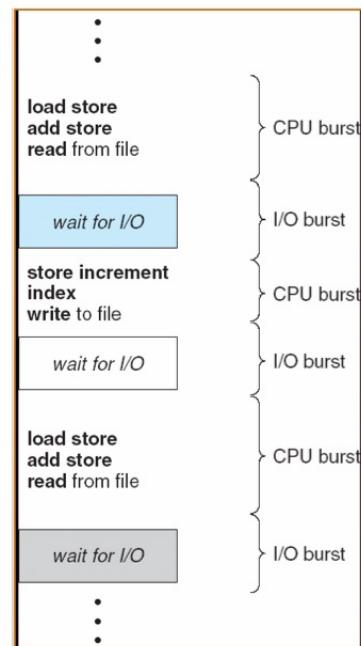
Wait time

- Time spent in ready queue
 - i.e., runnable, but not actually running
- CPU is busy doing other things
 - this is not an ideal state for a process!
- Minimizing wait time is a possible goal for a scheduling policy

Interactive processes

- Turnaround & Wait time may be measured over the entire course of a job
- Not a very relevant metric for interactive processes! (why?)
 - Interactive jobs have “bursty” execution — alternate between bursts of CPU and I/O activity
 - May never terminate! (e.g., consider browser, email client, etc.)
- Can compute turnaround/wait times on a per-burst basis
 - i.e., how long does a burst (of CPU activity) need to complete/wait before getting to the next I/O burst?

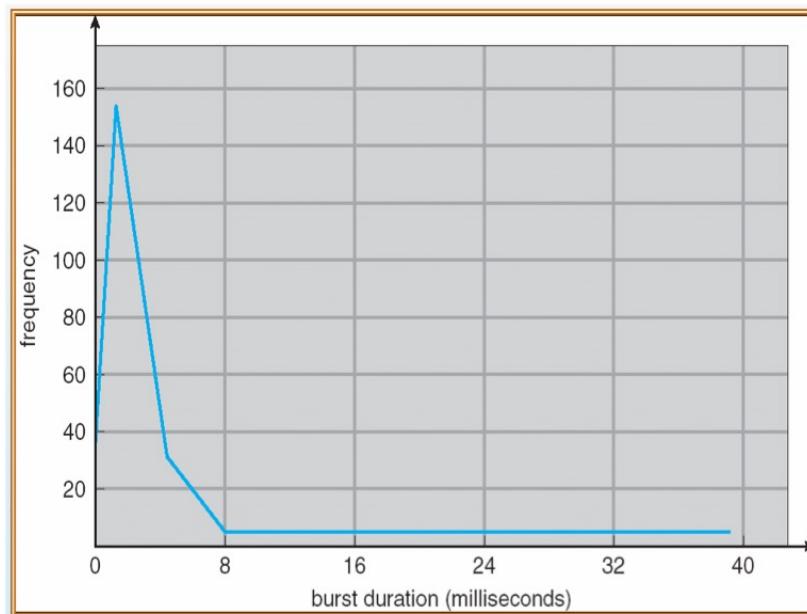
“bursty” execution



ILLINOIS TECH

College of Computing

burst length histogram



ILLINOIS TECH

College of Computing

Responsiveness

- For interactive jobs, improving responsiveness is arguably more important than optimizing total turnaround/wait times
 - How to quantify this?
- Response time: response = firstrun - arrival
 - i.e., how soon is a job given a chance to run after becoming ready?
- What's wrong with this? (consider requirements for “interaction”)
- How might we improve this metric?

Throughout & Utilization

- Aggregate metrics
- Throughput: # of completed jobs or bursts per unit time
 - e.g., 5 processes / minute, 25 CPU bursts / second
- Utilization: percentage of time CPU is busy running jobs
 - Context switch time counts against utilization!
 - CPU can be idle if there are no active jobs or if all jobs are blocked

Fairness

- What does it mean?
- How to measure/quantify it?
- Is it useful?
- How to enforce it?
- Prioritizing fairness may lower performance — which is more important? It Depends – consider ALCF.

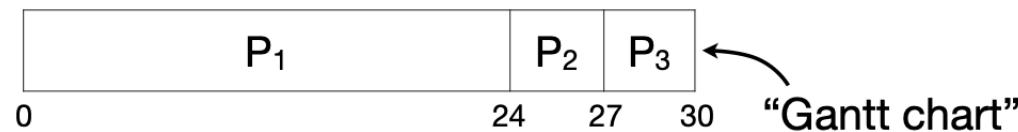
Scheduling Policies

ILLINOIS TECH

College of Computing

First come first served (FCFS)

Process	Arrival Time	Burst Time
P ₁	0	24
P ₂	0	3
P ₃	0	3



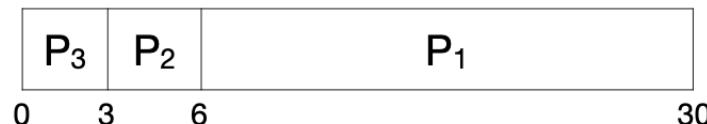
Wait times: P₁ = 0, P₂ = 24, P₃ = 27

Average: (0 + 24 + 27) / 3 = 17

REORDERED FCFS

Process	Arrival Time	Burst Time
P ₃	0	3
P ₂	0	3
P ₁	0	24

(better for everyone) →



Wait times: P₁ = 6, P₂ = 3, P₃ = 0

Average: $(6 + 3 + 0) / 3 = 3$

ILLINOIS TECH

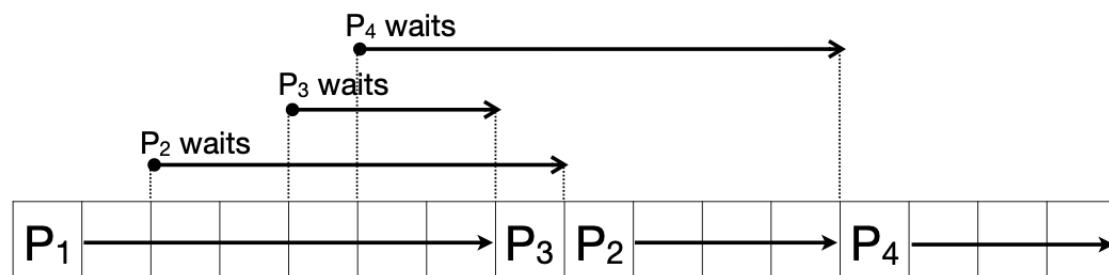
College of Computing

Shortest Job First (SJF)

- “Obvious” improvement to FCFS
- What metric(s) are we improving?
- Still a non-preemptive policy — i.e., once a job starts executing a CPU burst, it runs until it blocks (or completes)

SJF (CONT.)

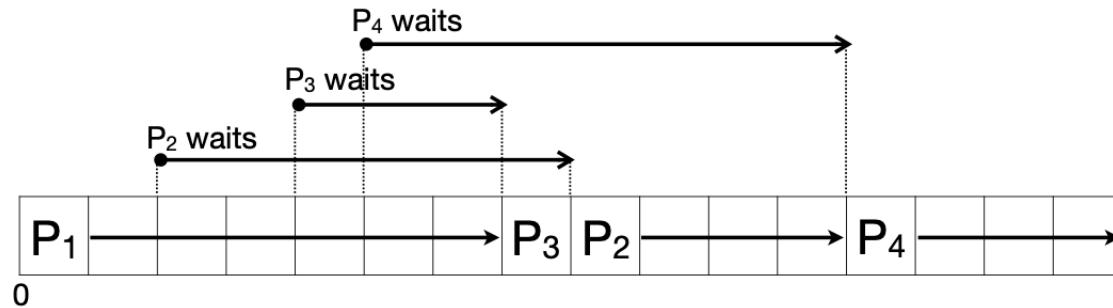
<i>Process</i>	<i>Arrival Time</i>	<i>Burst Time</i>
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4



ILLINOIS TECH

College of Computing

SJF (CONT.)



Wait times: $P_1 = 0$, $P_2 = 6$, $P_3 = 3$, $P_4 = 7$

Average: $(0 + 6 + 3 + 7) / 4 = 4$

(can we do better?)

ILLINOIS TECH

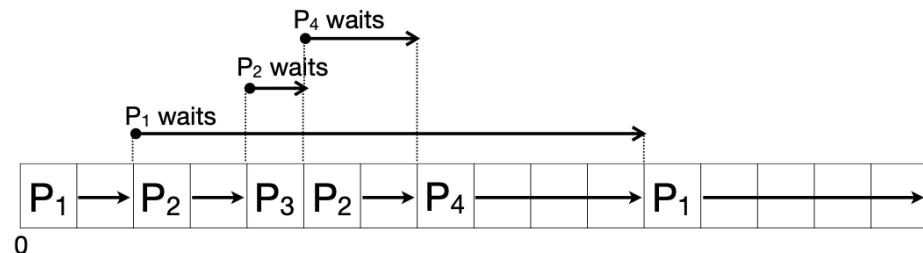
College of Computing

Preemptive SJF (PSJF)

- aka “Shortest Time-to-Completion First” (STCF)
- aka “Shortest Remaining-Time First” (SRTF)
- May preempt running job to schedule a different (ready) job

PSJF (CONT.)

Process	Arrival Time	Burst Time
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4



Wait times: P₁ = 9, P₂ = 1, P₃ = 0, P₄ = 2

Average: (9 + 1 + 0 + 2) / 4 = 3 (vs SJF @ 4)

ILLINOIS TECH

College of Computing

Greedy algorithms

- SJF/PSJF are greedy algorithms
 - i.e., they select the best choice at the moment (“local maximum”)
- Greedy algorithms don’t always produce globally maximal results
 - e.g., naive hill-climbing algorithm (only take a step if it brings me to higher ground) doesn’t always find the tallest peak!
- Are SJF/PSJF optimal?

Is It Optimal?

- Consider 4 jobs with burst lengths t_0, t_1, t_2, t_3 that just became ready
- What is the average wait time if scheduled in the order given?
 - $= (3 \cdot t_0 + 2 \cdot t_1 + t_2) / 4$
 - Weighted average — clearly minimized by running shortest jobs first!
- SJF/PSJF are provably optimal with respect to average wait time
 - But at what cost?
 - Potential CPU starvation! (e.g., longer jobs keep getting put off)

No way to tell the future

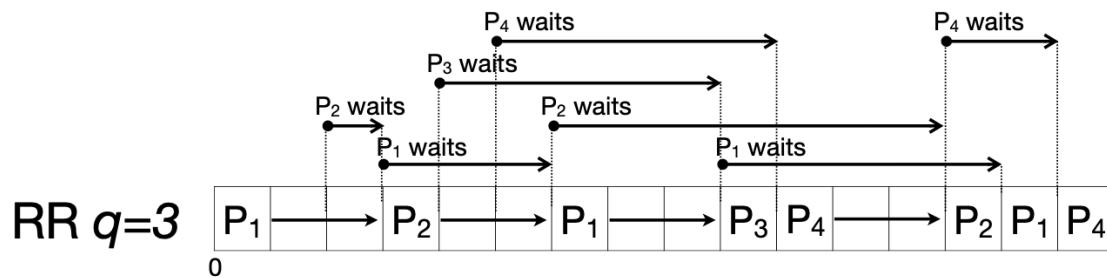
- We've been assuming that job/burst lengths are known in advance
- May be possible in rare circumstances (e.g., repeated jobs, job profiling), but unlikely in practice
- Common approach: predict future burst lengths based on past behavior
 - Simple moving average (sliding window of past values)
 - Exponentially weighted moving average (EMA)

Round Robin (RR)

- The “fairest” of them all
- Uses a FIFO queue:
 - Each job runs for a maximum fixed time quantum q
 - If unfinished, re-enter queue at the tail end
- Given time quantum q and n jobs:
 - max wait time (per cycle) = $q \cdot (n - 1)$
 - each job receives $1/n$ timeshare
- RR is also used for other applications such as load balancing and dns

RR (cont.)

Process	Arrival Time	Burst Time
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4



Wait times: P₁ = 8, P₂ = 8, P₃ = 5, P₄ = 7

Average: $(8 + 8 + 5 + 7) / 4 = 7$

RR (cont.)

Process	Arrival Time	Burst Time
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4

	Avg. Turnaround	Avg. Wait Time
RR $q=7$	8.75	4.75
RR $q=4$	9	5
RR $q=3$	11	7
RR $q=1$	9.75	5.75

(FCFS)

ILLINOIS TECH

College of Computing

RR (cont.)

<i>Process</i>	<i>Arrival Time</i>	<i>Burst Time</i>
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4

	Throughput	Utilization
RR q=7	0.25	1.0
RR q=4	0.25	1.0
RR q=3	0.25	1.0
RR q=1	0.25	1.0

ILLINOIS TECH

College of Computing

Context Switch Time

- CST = interrupt + context switch + scheduler
- $\sim 1 \mu\text{s}$ in Linux on recent hardware
- Each time we preempt a job we introduce systemic overhead (i.e., costs not incurred by the job itself) and reduce utilization
- Longer quantum times help amortize the cost of CSTs
- Just measuring CST oversimplifies the cost of context switches
 - E.g., cache perturbation significantly affects execution efficiency

<i>Process</i>	<i>Arrival Time</i>	<i>Burst Time</i>
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4

(CST=1)	Avg. Turnaround	Avg. Wait Time
RR q=7	10.25	6.25
RR q=4	11.5	7.25
RR q=3	16.25	11.25
RR q=1	20.25	13.25

ILLINOIS TECH

College of Computing

<i>Process</i>	<i>Arrival Time</i>	<i>Burst Time</i>
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4

(CST=1)	Throughput	Utilization
RR q=7	0.2	0.8
RR q=4	0.19	0.762
RR q=3	0.167	0.667
RR q=1	0.125	0.5

ILLINOIS TECH

College of Computing

Priority Schedulers

- Can implement more fine-grained scheduling policies by introducing a system of arbitrary priorities, gathered/computed by the scheduler
 - Process with maximum priority is scheduled
- SJF/PSFJ are priority schedulers! (priority = 1 / predicted burst length)
- Starvation due to priority scheduling may be combatted by aging
 - But there may be other insidious issues!

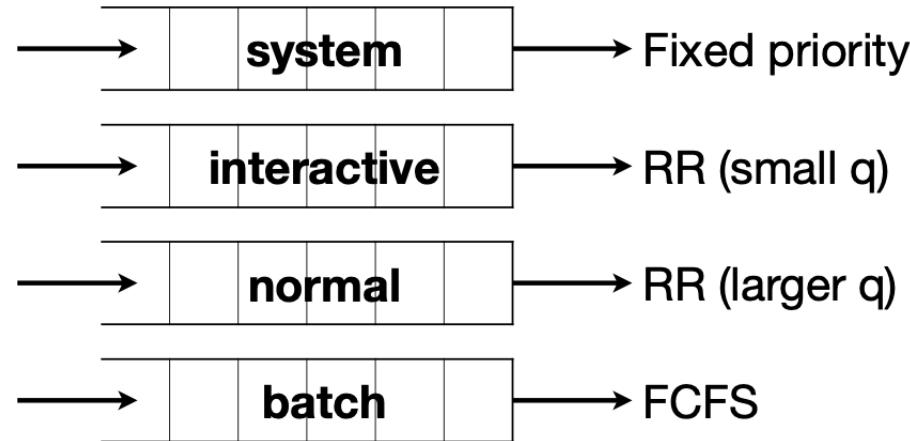
Highest Penalty Ratio Next (HPRN)

- Example of a priority scheduler that implements aging
- Two statistics maintained by scheduler for each job:
 - “wall clock” age, t
 - total CPU execution time, e
- Priority is the “penalty ratio” = t / e
 - ∞ when job is first ready, decreases as job receives CPU time
- In practice would incur too many context switches!
- Can institute a minimum execution quantum (is this RR?)

Scheduling is rocket science!

- Jobs are unpredictable, and interactions between jobs even more so
- Priority-based scheduling is useful, as it may help us optimize different scheduling metrics. But there are potential downsides:
 - Starvation and Priority inversion
 - Not all jobs require the same sort of optimization!
 - E.g., CPU-bound vs. interactive jobs
- Would like a mechanism that allows us to optimize for different metrics across separate groups of processes

Multi-Level Queue

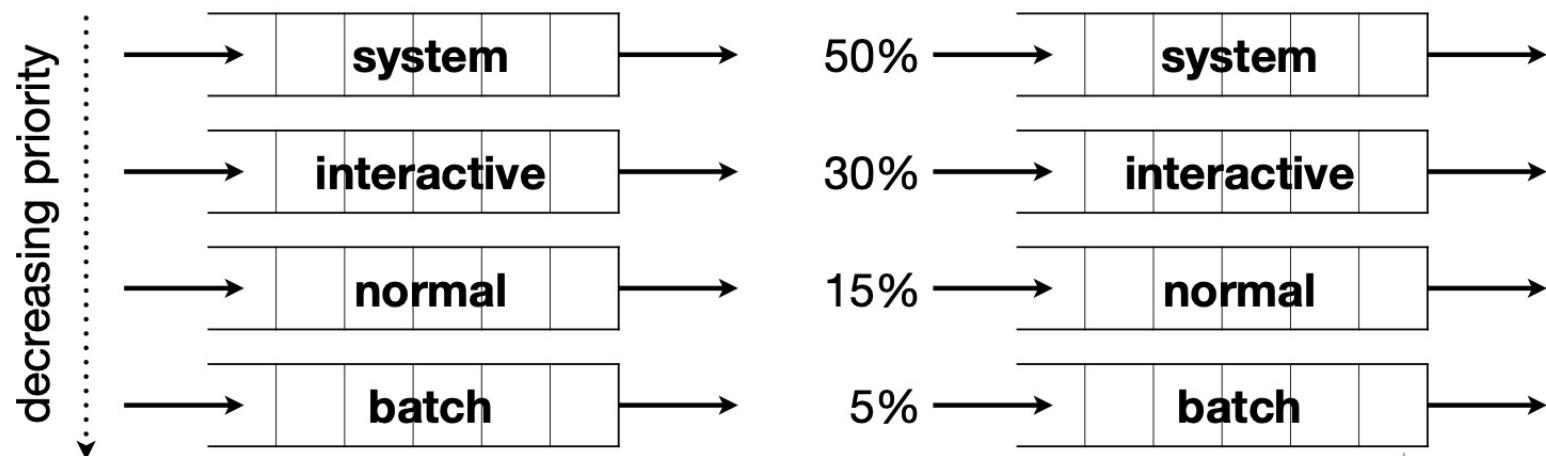


ILLINOIS TECH

College of Computing

Multi-Level Queue

- Requires a *queue arbitration* policy, i.e., which queue to select jobs from?
- Approach 1: select jobs from top, non-empty queue
- Approach 2: allocate macro time slices to each queue



MLQ (CONT.)

- Which jobs go in which queues?
- Can be self-declared/assigned
 - e.g., UNIX “nice” value
 - Can jobs be trusted?
- Jobs can be profiled based on initial burst(s)
 - e.g., short, periodic CPU bursts → classify as interactive job
- May be gamed by programmers looking for better treatment

Shifting requirements?

- More important issue: what if job requirements change dynamically?
 - E.g., photo editor: tool selection (interactive) → apply filter (CPU-bound) → simple edits (interactive) → apply compression (CPU-bound) ...
- Scheduler should respond to changes in job requirements by applying appropriate policies
- While maximizing responsiveness and efficiency where possible!

Multi-Level Feedback Queue (MLFQ)

- Supports movement between queues after initial assignment
- Based on dynamic job characteristics (mostly discerned from burst lengths relative to allocated quanta)
 - e.g., 3 RR queues with different q

MLFQ summary

- Many parameters may be needed to fine-tune an MLFQ scheduler
 - Behavior may be driven by a combination of heuristics and mathematical/algorithmic optimization
 - Hard to avoid the use of “magic numbers” that work for specific systems and workloads
- MLFQ helps dynamically identify and balance interactive and CPU-bound jobs — a popular choice in modern operating systems!

Questions?

ILLINOIS TECH

College of Computing

Scheduling Part 2

Ben Lenard

blenard@iit.edu

ILLINOIS TECH

College of Computing

Agenda

- Homework 2
- Recap of Monday
 - Scheduling policies
 - FCFS, SJF, PSJF, RR, HPRN
 - MLQ, MLFQ
 - Scheduling: Proportional Share
 - Multiprocessor Scheduling

ILLINOIS TECH

College of Computing

Homework 2

- Any questions?

ILLINOIS TECH

College of Computing

Recap

ILLINOIS TECH

College of Computing

Policy

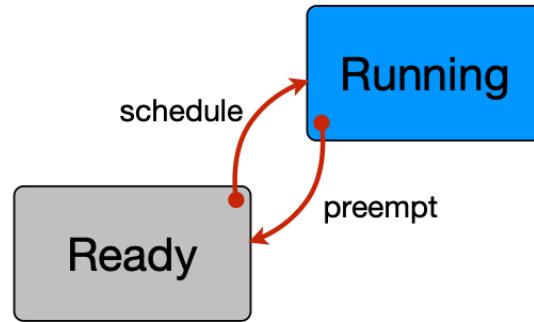
- high-level “what”
- scheduling disciplines
 - e.g., FCFS, SJF, RR, etc.
- driven by a variety of potentially conflicting goals
 - e.g., performance and fairness

Mechanism

- low-level “how”
- combination of HW/SW
 - e.g., clock interrupt, high precision timer, PCB
- scattered throughout kernel codebase

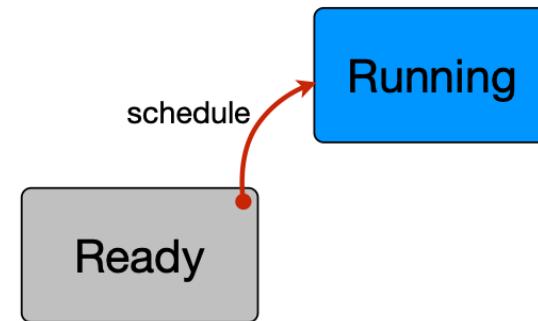
preemptive scheduling

- preemptive scheduling
- relies on clock interrupt
 - (to regain control of CPU)



non-preemptive scheduling

- non-preemptive scheduling
- once a job starts, it continues until it terminates/blocks



High-level policy considerations

- Preemptive vs. Non-preemptive
- Information available for making informed decisions
- Depends on lower-level mechanisms available
 - Scheduling goals
- Based on optimizing/tuning scheduling metrics

Policies for processes vs other things like IO

- Scheduling policies are more for processes
 - FCFS, SJF, PSJF, RR, HPRN
 - MLQ, MLFQ
- For example IO:
 - bfq (Budget Fair Queuing) (Multiqueue)
 - kyber (Multiqueue)
 - none (Multiqueue)
 - mq-deadline (Multiqueue)
- Why so many?
- We'll talk more about them later this term

Scheduling for HPC

- It relates and can be easier to visualize then P1,P2,...
- <https://status.alcf.anl.gov/>

ILLINOIS TECH

College of Computing

Scheduling: Proportional Share

ILLINOIS TECH

College of Computing

Scheduling: Proportional Share

- proportional-share scheduler or fair-share scheduler.
- Proportional-share is based around a simple concept: instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain percentage of CPU time.

Lottery Scheduling

- Simple Idea, at a given interval, hold a lottery to determine which process should get to run next; processes that should run more often should be given more chances to win the lottery.
- How can we design a scheduler to share the CPU in a proportional manner?
- What are the key mechanisms for doing so?
- How effective are they?

Tickets Represent Your Share

- Tickets represent your share
- Consider 10 people each one gets 1 ticket, verses 2,3,4,5 for one person
- Let's say we add timeslices, or cpu time
- Lottery Scheduling is probabilistic not deterministic
- It uses randomness

Tickets Represent Your Share (cont.)

- A gets 0 – 74
- B gets 75 - 99

63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43 0 49 12

Here is the resulting schedule:

A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
B		B							B		B								

ILLINOIS TECH

College of Computing

Ticket Currency

- User A & B get 100 tickets each

User A → 500 (A's currency) to A1 → 50 (global currency)

→ 500 (A's currency) to A2 → 50 (global currency)

User B → 10 (B's currency) to B1 → 100 (global currency)

ILLINOIS TECH

College of Computing

Other ticketing mechanisms

- Ticket Transfer - a process can temporarily hand off its tickets to another process
- Ticket Inflation – a way to control demand at peak times

Implementation

- Simplicity
 - Random number generator to pick the winning ticket,
 - A data structure to track the processes of the system and their allocation
 - The total number of tickets

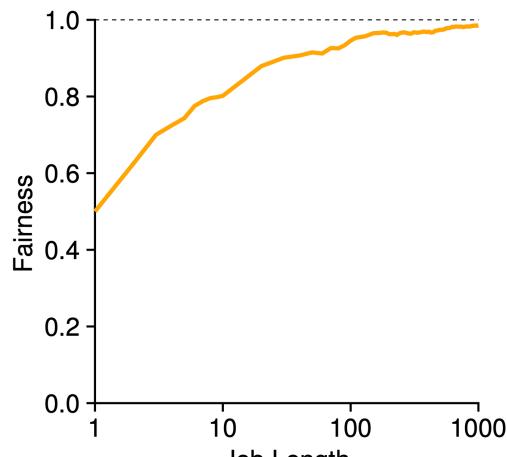


Figure 9.2: Lottery Fairness Study

ILLINOIS TECH

College of Computing

Stride Scheduling

- Assume A has 100, B 50, and C 250 tickets
- Divide those values into a larger value like 10,000
- The stride for A is 100, 200, and 40.
- The basic idea is simple:
 - At any given time, pick the process to run that has the lowest pass value so far
 - When you run a process, increment its pass counter by its stride.

Stride Scheduling (cont.)

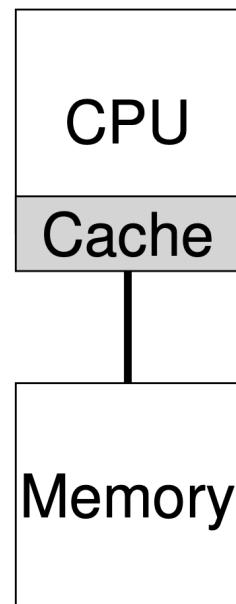
Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Multiprocessor scheduling

ILLINOIS TECH

College of Computing

Until this point...



ILLINOIS TECH

College of Computing

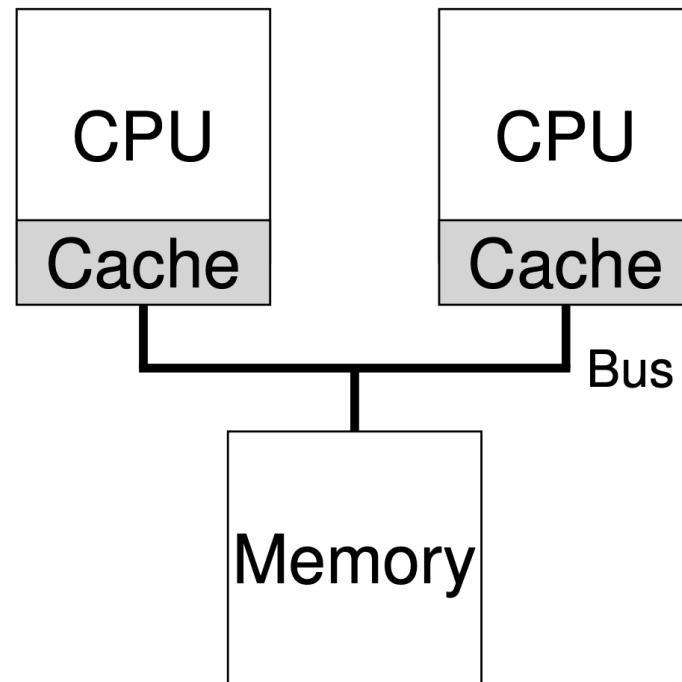
Multiprocessor

- Once upon a time, processors were single cores
- We had servers with multiple processors
- As we couldn't make processors faster, we added more cores
- We introduced challenges such as parallelization of tasks

Understanding the Problem

- Single-CPU hardware
 - Hierarchy of hardware caches that in general help the processor run programs faster.
 - Caches are small, fast memories that hold copies of popular data that is found in the main memory of the system.
 - Main memory, in contrast, holds all of the data, but access to this larger memory is slower.
- By keeping frequently accessed data in a cache, the system can make the large, slow memory appear to be a fast one.

Multiprocessor



ILLINOIS TECH

College of Computing

Types of Cache

- Cache is locality based mean is closest to the core or processor
- Two types:
 - Temporal locality - a piece of data is accessed, it is likely to be accessed again in the near future
 - Spatial locality - if a program accesses a data item at address x, it is likely to access data items near x as well

Cache Coherence

- Imagine, for example, that a program running on CPU 1 reads a data item (with value D) at address A; because the data is not in the cache on CPU 1, the system fetches it from main memory, and gets the value D. The program then modifies the value at address A, just updating its cache with the new value D'; writing the data through all the way to main memory is slow, so the system will (usually) do that later. Then assume the OS decides to stop running the program and move it to CPU 2. The program then re-reads the value at address A; there is no such data in CPU 2's cache, and thus the system fetches the value from main memory, and gets the old value D instead of the correct value D'. Oops!

How do we fix this?

- hardware: by monitoring memory accesses, hardware can ensure that basically the “right thing” happens and that the view of a single shared memory is preserved.

Synchronization

- Consider locking and lock free structures.
- We will talk about this later in the term
- Multi processors introduce concurrency later in the term – do we know what ACID is?

Cache Affinity

- Basically the scheduler should be aware of where the running process is and continue to schedule a given task at the given location

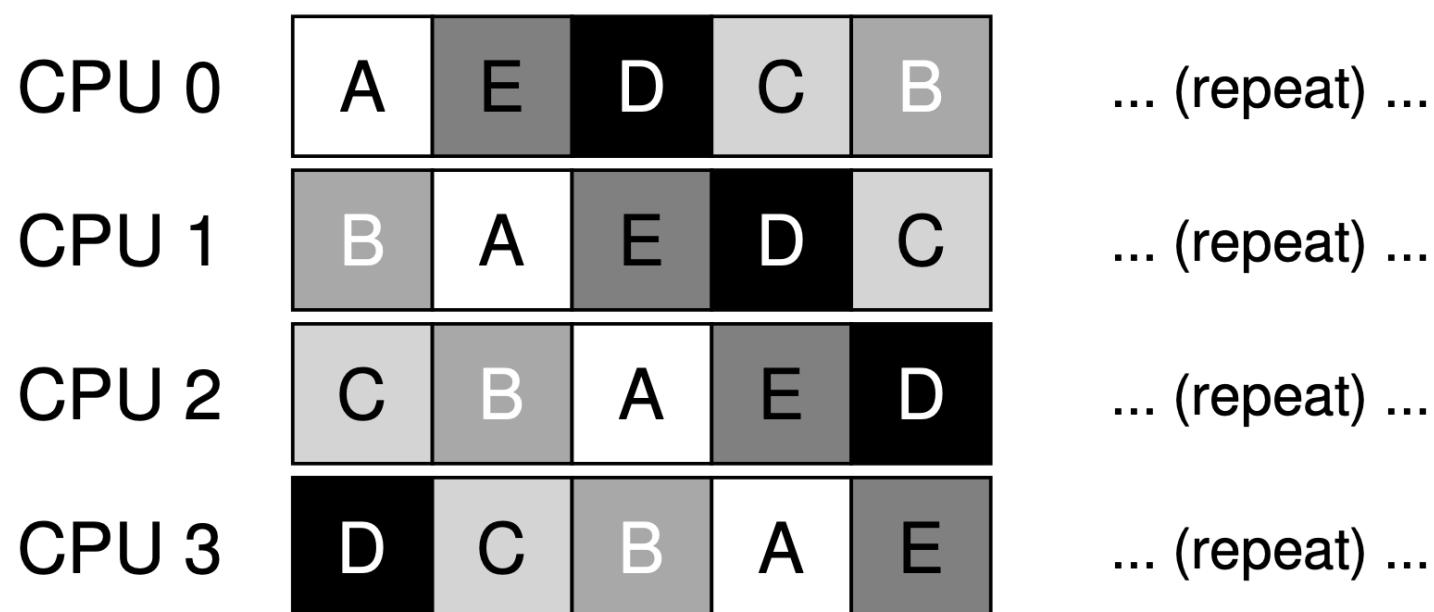
ILLINOIS TECH

College of Computing

Single-Queue Scheduling

- No cache affinity
- Locking would need to be added
- Does not scale

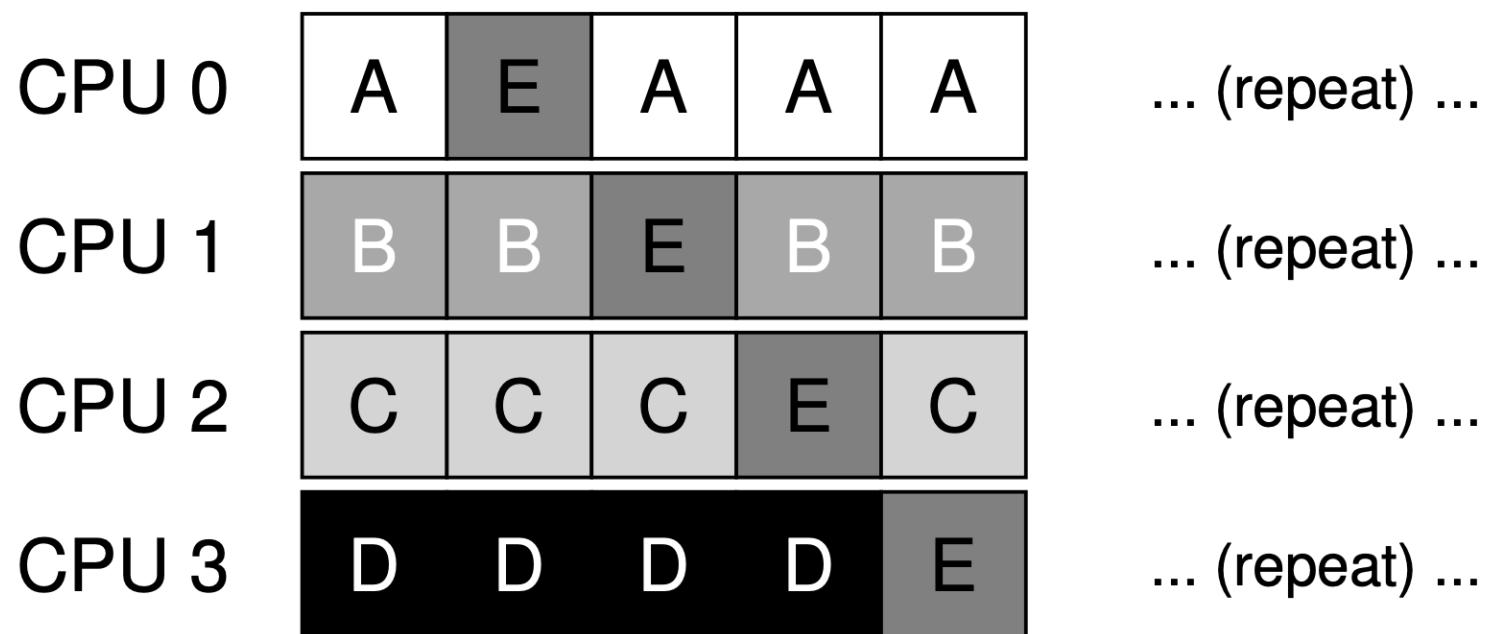
Single-Queue Scheduling (cont.)



ILLINOIS TECH

College of Computing

Single-Queue Scheduling with Affinity

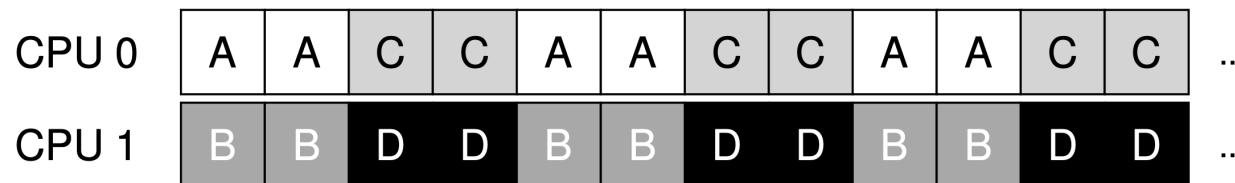


ILLINOIS TECH

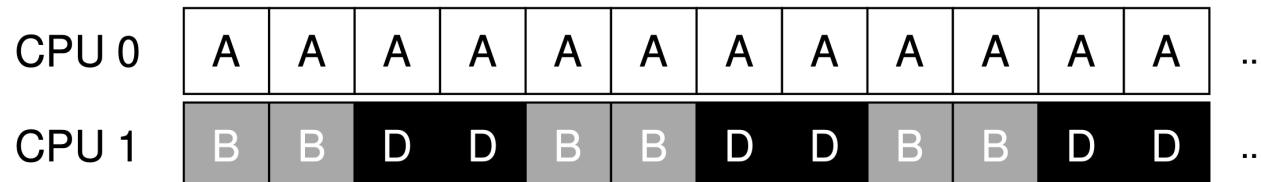
College of Computing

Multi-Queue Scheduling

- One queue per CPU



- Could lead to a poor balance



ILLINOIS TECH

College of Computing

Linux Multiprocessor Schedulers

- the O(1) scheduler
- the Completely Fair Scheduler (CFS)
- BF Scheduler (BFS)

ILLINOIS TECH

College of Computing

Questions?

ILLINOIS TECH

College of Computing

Virtual Memory

Ben Lenard

blenard@iit.edu

ILLINOIS TECH

College of Computing

Agenda

- Address spaces
- Segmentation
- Paging
- Swapping
- User space memory management
- Case study: Linux+x86 VM

ILLINOIS TECH

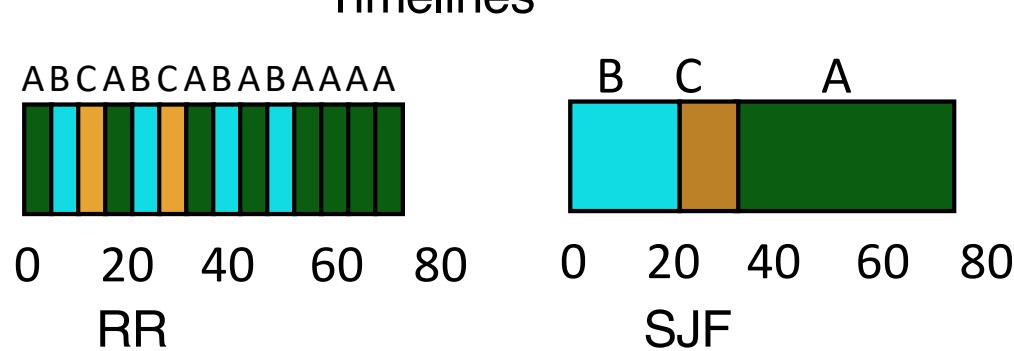
College of Computing

Scheduling Policy Review :

Workload

JOB	arrival	run
A	0	40
B	0	20
C	5	10

Timelines



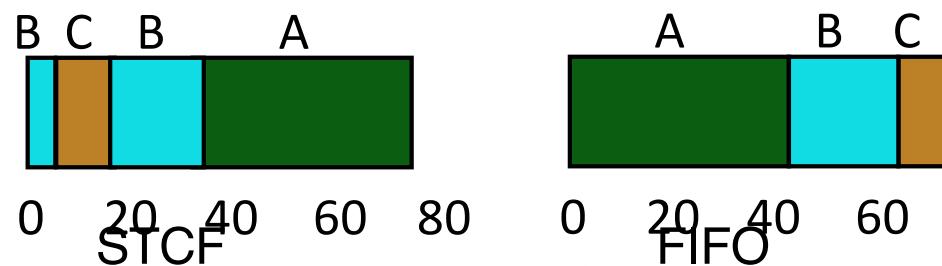
Schedulers:

FIFO

SJF

STCF

RR



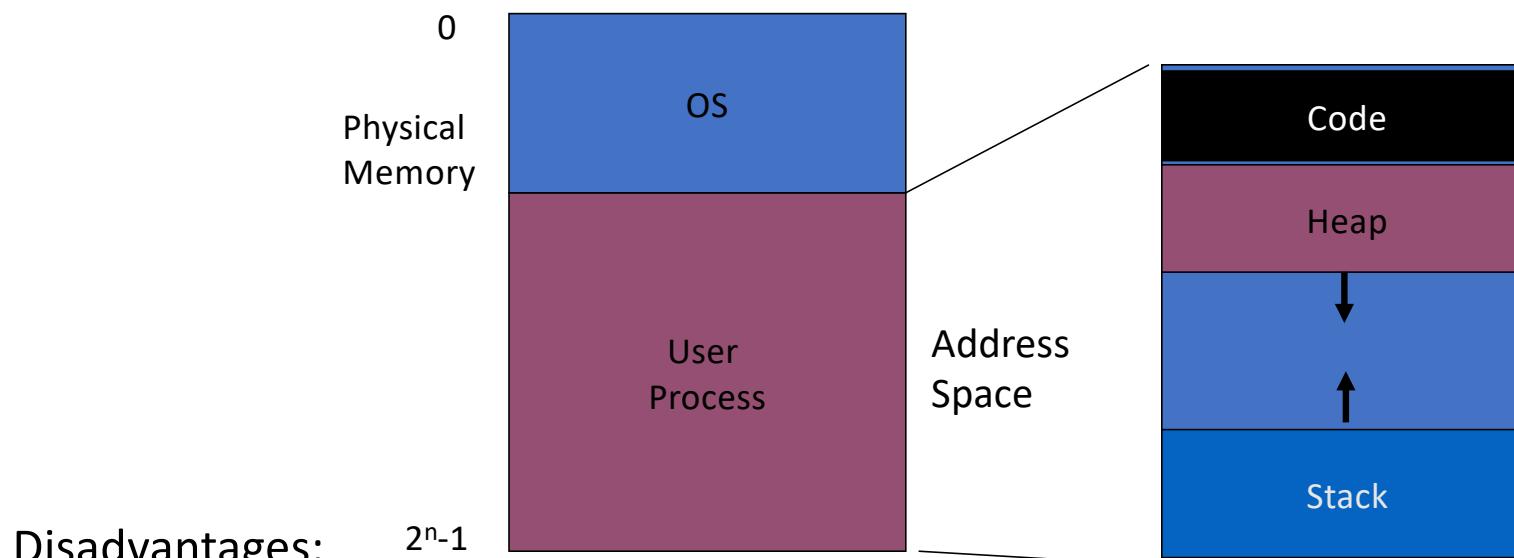
ILLINOIS TECH

College of Computing

- Questions answered in this lecture:
- What is in the address space of a process (review)?
- What are the different ways that that OS can virtualize memory?
 - Time sharing, static relocation, dynamic relocation
 - (base, base + bounds, segmentation)
 - What hardware support is needed for dynamic relocation?

Motivation for Virtualization

Uniprogramming: One process runs at a time



- Only one process runs at a time
- Process can destroy OS

ILLINOIS TECH

College of Computing

Multiprogramming Goals

Transparency

- Processes are not aware that memory is shared
- Works regardless of number and/or location of processes

Protection

- Cannot corrupt OS or other processes
- Privacy: Cannot read data of other processes

Efficiency

- Do not waste memory resources (minimize fragmentation)

Sharing

- Cooperating processes can share portions of address space

ILLINOIS TECH

College of Computing

Address spaces

ILLINOIS TECH

College of Computing

Programs → Processes

- Compilers transform programs into binary images that contains executable machine code and static data (e.g., constants/globals)
- The kernel can turn these binaries into active, running processes
 - Via limited direct execution and the scheduler/dispatcher, multiple processes can run concurrently on timeshared CPUs
- But how do processes share memory?
- How are memory addresses encoded into programs?

Abstraction: Address Space

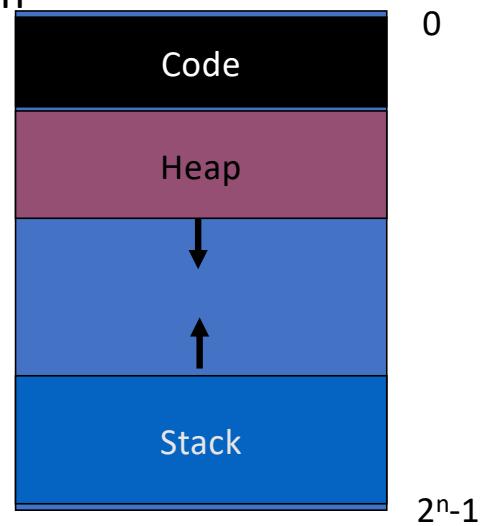
Address space: Each process has set of addresses that map to bytes

Problem: How can OS provide illusion of private address space to each process?

Review: What is in an address space?

Address space has static and dynamic components

- Static: Code and some global variables
- Dynamic: Stack and Heap



```
#include <stdio.h>

unsigned int glob = 0xdeadbeef;

int main() {
    printf("0x%lx\n", (unsigned long)&main);
    printf("0x%lx\n", (unsigned long)&glob);
    printf("0x%x\n", glob);
    return 0;
}
```

0x40052d
0x601040
0xdeadbeef

> objdump -d a.out

<pre>000000000040052d <main>: 40052d: 55 40052e: 48 89 e5 400531: b8 2d 05 40 00 400536: 48 89 c6 400539: bf 04 06 40 00 40053e: b8 00 00 00 00 400543: e8 c8 fe ff ff 400548: b8 40 10 60 00 40054d: 48 89 c6 400550: bf 04 06 40 00 400555: b8 00 00 00 00 40055a: e8 b1 fe ff ff 40055f: 8b 05 db 0a 20 00 400565: 89 c6 400567: bf 0b 06 40 00 40056c: b8 00 00 00 00 400571: e8 9a fe ff ff 400576: b8 00 00 00 00 40057b: 5d 40057c: c3</pre>	<pre>push %rbp mov %rsp,%rbp mov \$0x40052d,%eax mov %rax,%rsi mov \$0x400604,%edi mov \$0x0,%eax callq 400410 mov \$0x601040,%eax mov %rax,%rsi mov \$0x400604,%edi mov \$0x0,%eax callq 400410 mov 0x200adb(%rip),%eax # 601040 <glob mov %eax,%esi mov \$0x40060b,%edi mov \$0x0,%eax callq 400410 mov \$0x0,%eax pop %rbp retq</pre>
---	---

text (code) addresses

global data address

> objdump -h a.out

a.out: file format elf64-x86-64

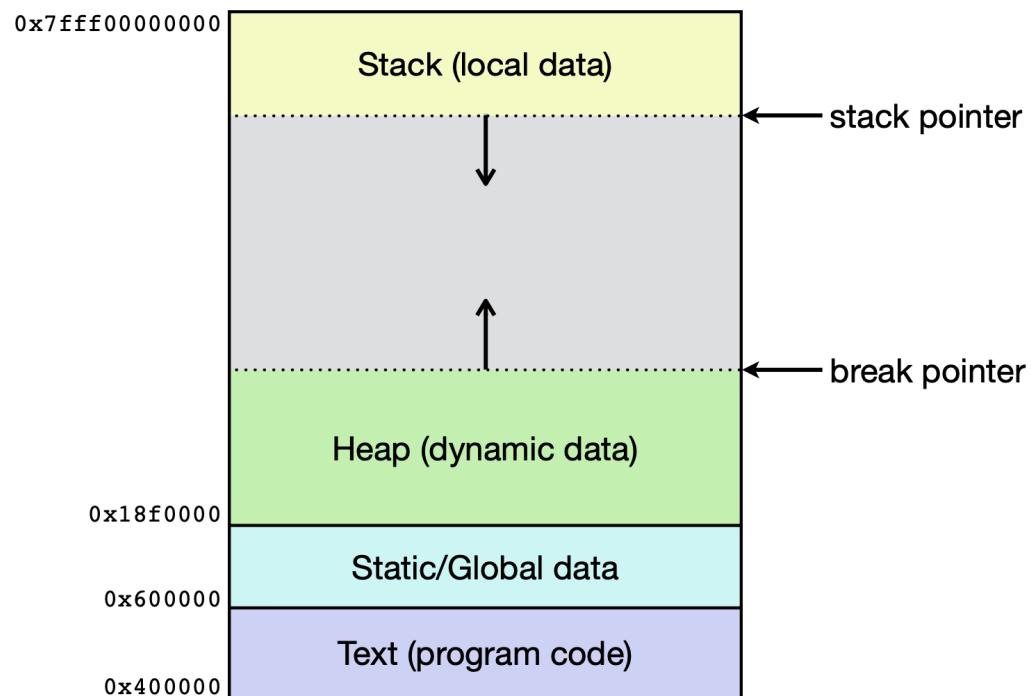
Sections:

Idx	Name	Size	VMA	LMA	File off	Alg
12	.text	000001b2	0000000000400440	0000000000400440	00000440	2**
			CONTENTS, ALLOC, LOAD, READONLY, CODE			
23	.data	00000014	0000000000601030	0000000000601030	00001030	2**
			CONTENTS, ALLOC, LOAD, DATA			
24	.bss	00000004	0000000000601044	0000000000601044	00001044	2**
			ALLOC			

“Hardcoded” addresses

- At compile time, the linker embeds fixed addresses into the binary
 - E.g., for function calls, global variables/constants, jump tables, etc.
- When exec-ed, the OS loads sections of the binary into memory at these pre-established locations
 - Text & Data sections are initialized with contents of binary
 - BSS section is zero-initialized
 - Starting addresses for initially empty stack & heap are also established

Uniform process address space



ILLINOIS TECH

College of Computing

Stack Organization

Definition: Memory is freed in opposite order from allocation

```
alloc(A);  
alloc(B);  
alloc(C);  
free(C);  
alloc(D);  
free(D);  
free(B);  
free(A);
```

Simple and efficient implementation:

Pointer separates allocated and freed space

Allocate: Increment pointer

Free: Decrement pointer

No fragmentation

ILLINOIS TECH

College of Computing

Where Are stacks Used?

OS uses stack for procedure call frames (local variables and parameters)

```
main () {
    int A = 0;
    foo (A);
    printf("A: %d\n", A);
}

void foo (int Z) {
    int A = 2;
    Z = 5;
    printf("A: %d Z: %d\n", A, Z);
}
```

Heap Organization

Definition: Allocate from any random location: malloc(), new()

- Heap memory consists of allocated areas and free areas (holes)
- Order of allocation and free is unpredictable

Advantage

- Works for all data structures

Disadvantages

- Allocation can be slow
 - End up with small chunks of free space - fragmentation
 - Where to allocate 12 bytes? 16 bytes? 24 bytes??
- What is OS's role in managing heap?
 - OS gives big chunk of free memory to process; library manages individual allocations



Memory Accesses

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int x;
    x = x + 3;
}
```



```
0x10: movl 0x8(%rbp), %edi
0x13: addl $0x3, %edi
0x19: movl %edi, 0x8(%rbp)
```

%rbp is the base pointer:
points to base of current stack frame

Address space = a lie!

- If processes are simultaneously accessing physical memory ... - Not all text sections can begin at 0x400000
 - Not all data sections can begin at 0x600000
 - Not all heaps can begin at 0x18f0000
 - Not all stacks can begin at 0x7fff00000000
- Uniform process address spaces are an illusion created by the kernel
 - To simplify program loading and execution (among other reasons)

How to Virtualize Memory?

Problem: How to run multiple processes simultaneously?

Addresses are “hardcoded” into process binaries

How to avoid collisions?

Possible Solutions for Mechanisms (covered today):

2. Static Relocation
3. Base
4. Base+Bounds
5. Segmentation

Timesharing?

Try similar approach to how OS virtualizes CPU

Observation:

OS gives illusion of many virtual CPUs by saving **CPU registers** to **memory** when a process isn't running

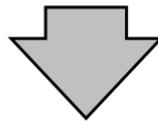
Could give illusion of many virtual memories by saving **memory** to **disk** when process isn't running

Software relocation

```
00000000040052d <main>:
```

```
40052d: 55  
40052e: 48 89 e5  
400531: b8 2d 05 40 00  
400536: 48 89 c6  
400539: bf 04 06 40 00
```

```
...
```



```
push %rbp  
mov %rsp,%rbp  
mov $0x40052d,%eax  
mov %rax,%rsi  
mov $0x400604,%edi
```

```
00000000060052d <main>:
```

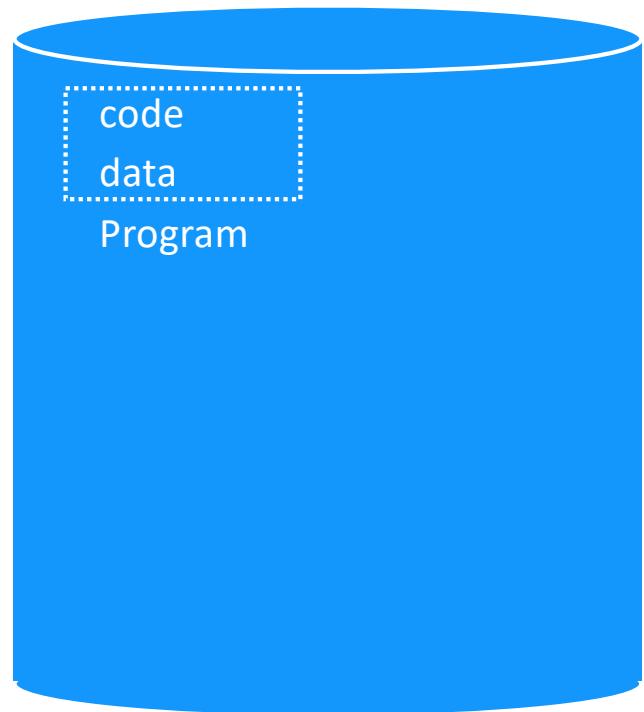
```
60052d: 55  
60052e: 48 89 e5  
600531: b8 2d 05 60 00  
600536: 48 89 c6  
600539: bf 04 06 60 00
```

```
...
```

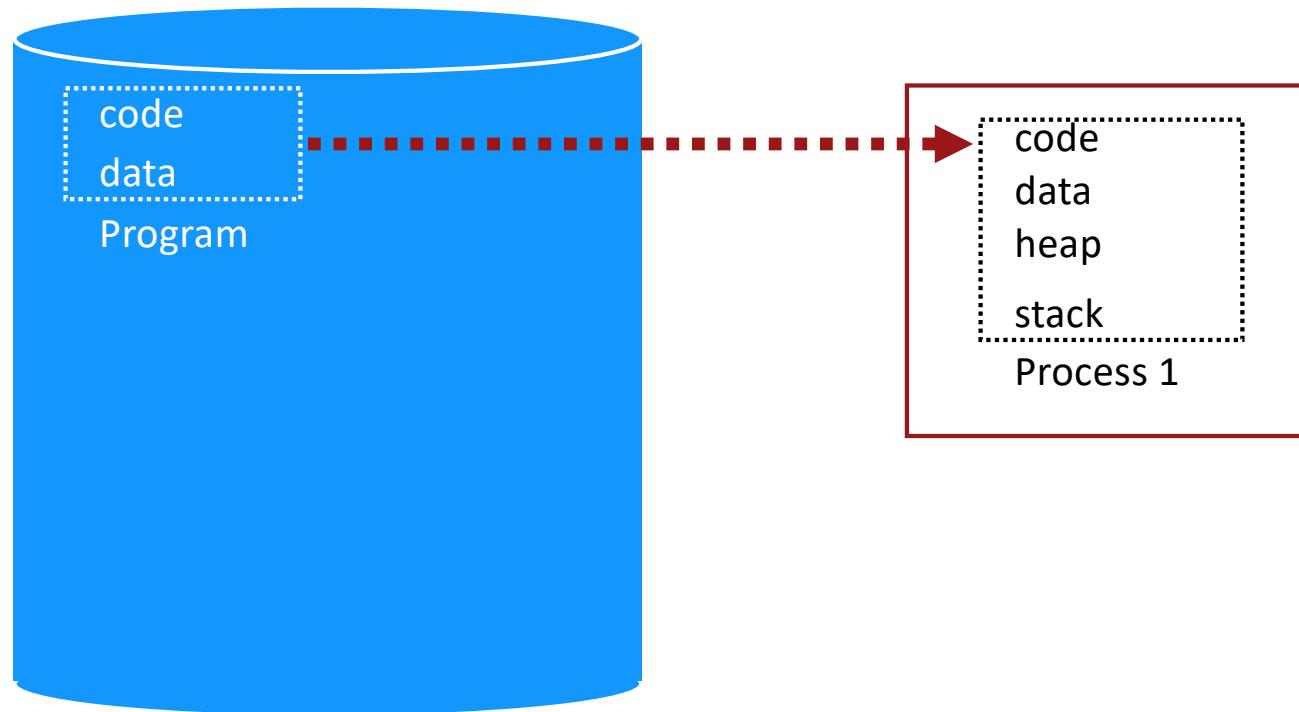
```
push %rbp  
mov %rsp,%rbp  
mov $0x60052d,%eax  
mov %rax,%rsi  
mov $0x600604,%edi
```

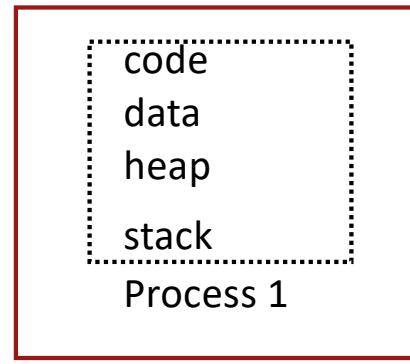
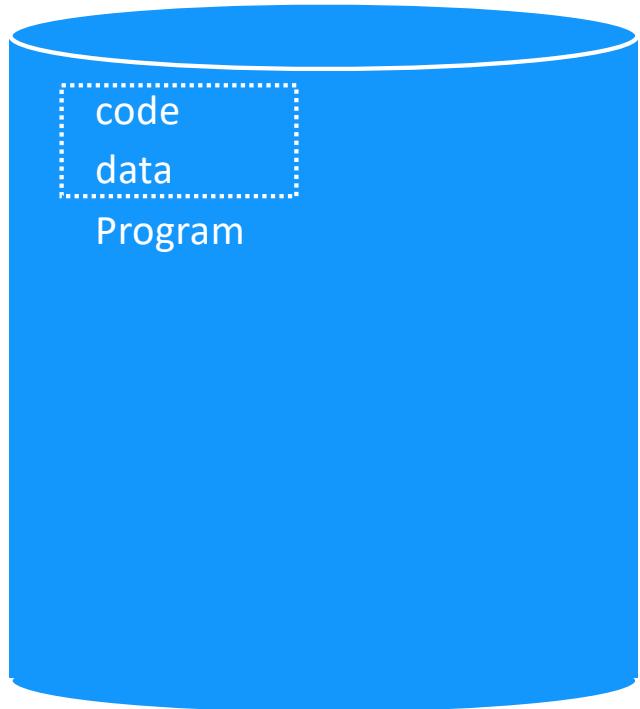
Software relocation

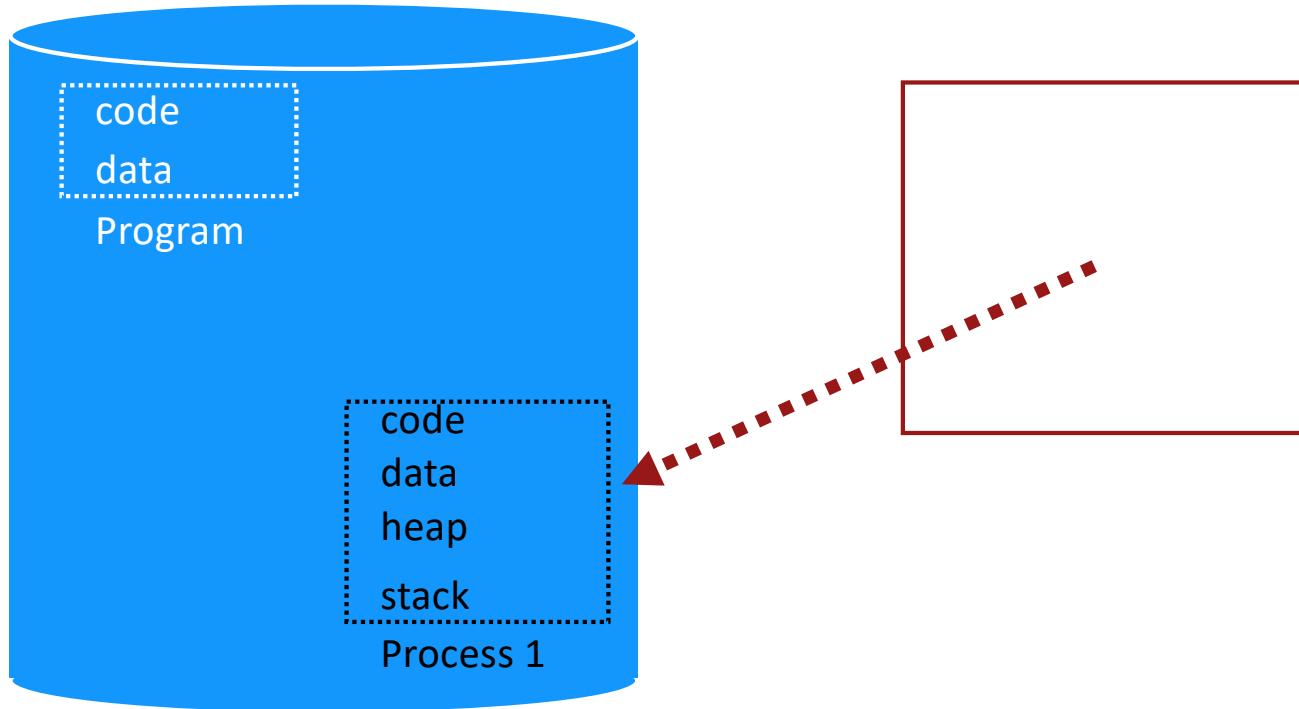
- Simple option: rewrite all addresses at load-time (in software), so that processes can occupy memory simultaneously (space-sharing)
- Once loaded, cannot easily relocate process in memory
 - Software relocation would be complex and time-consuming (and perhaps impossible, without runtime type information)
- If a process accidentally/maliciously uses a bad address, it could access another process's (or the kernel's) address space!
 - Pure software relocation makes address space protection difficult

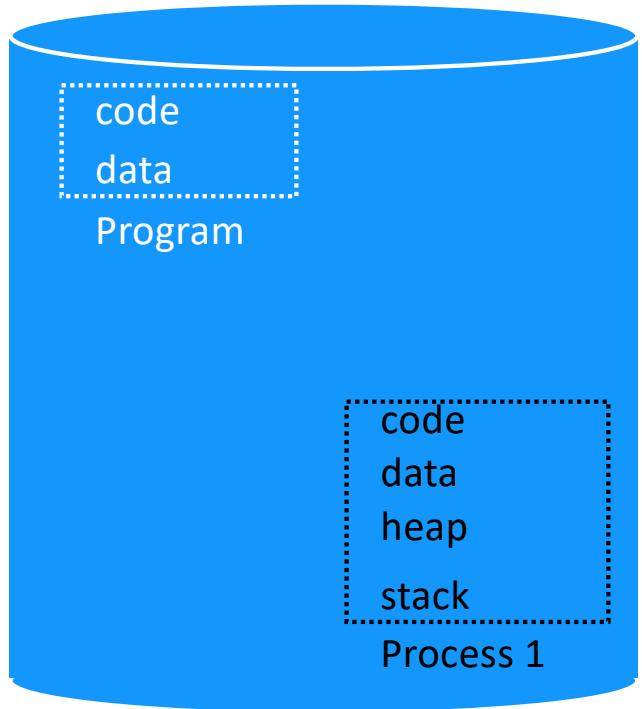


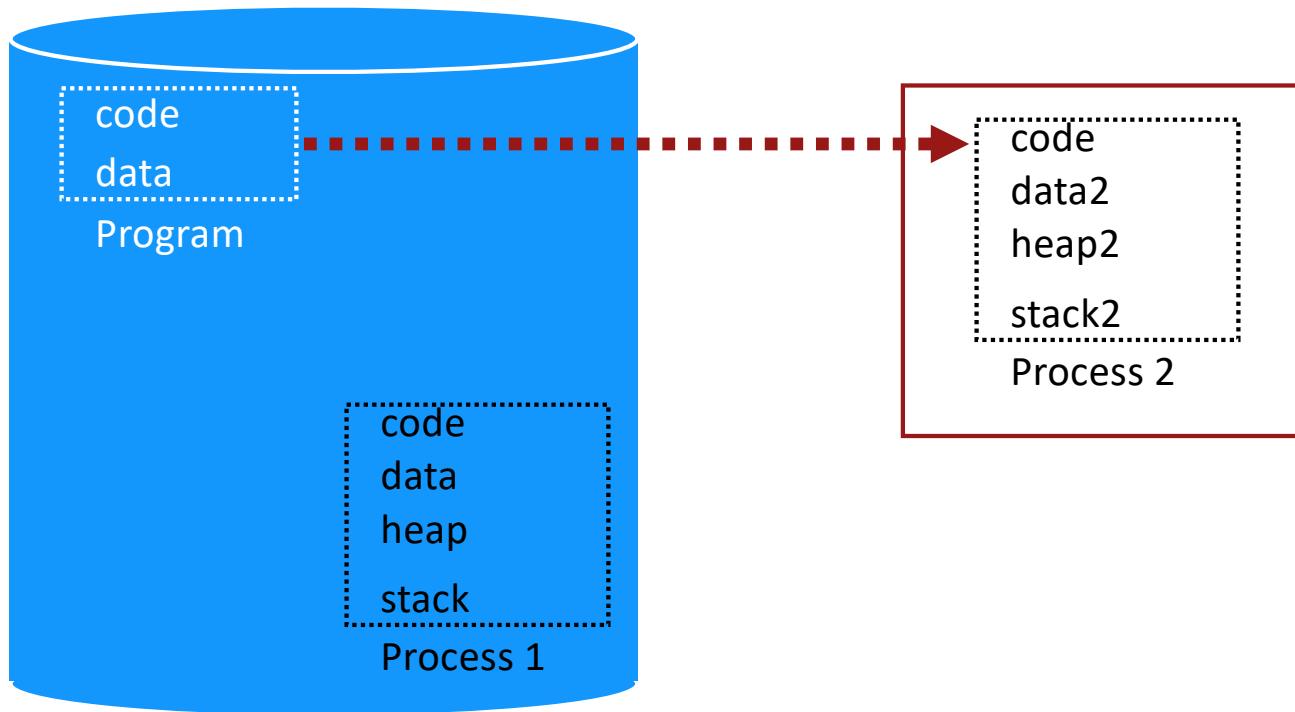
Time Share Memory: Example

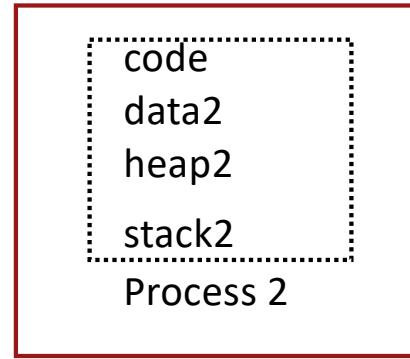
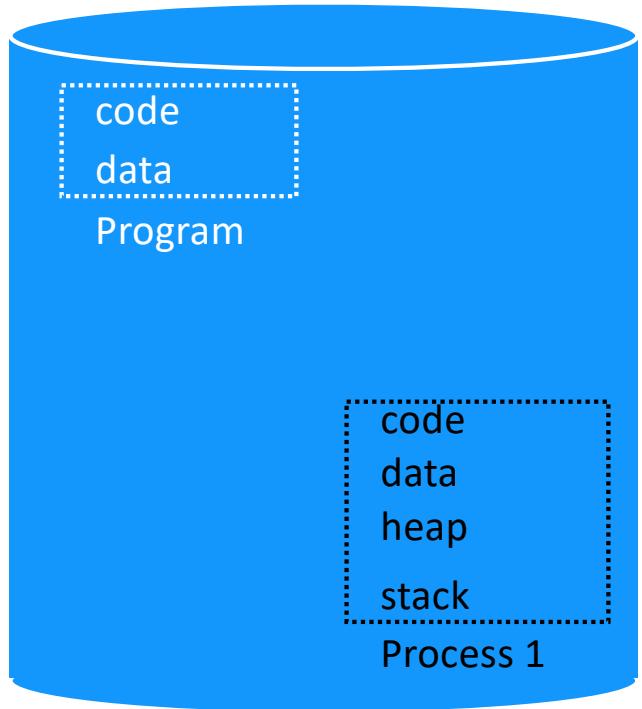


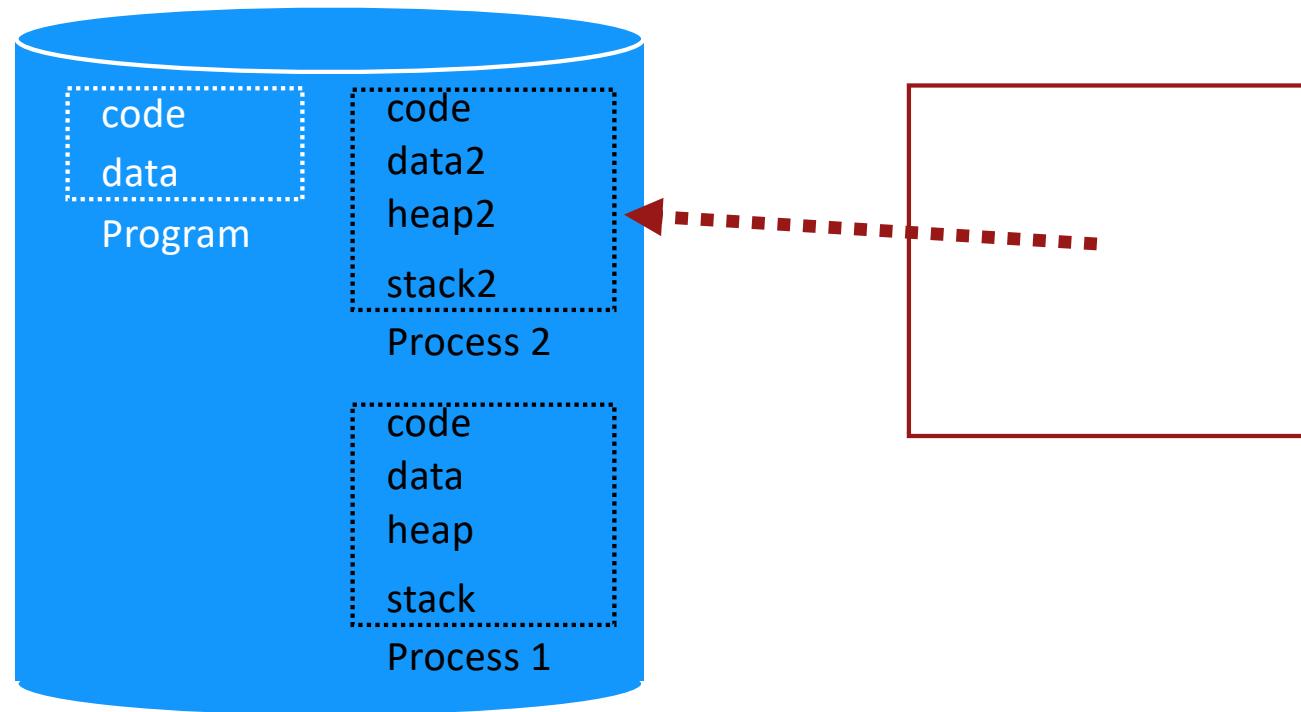


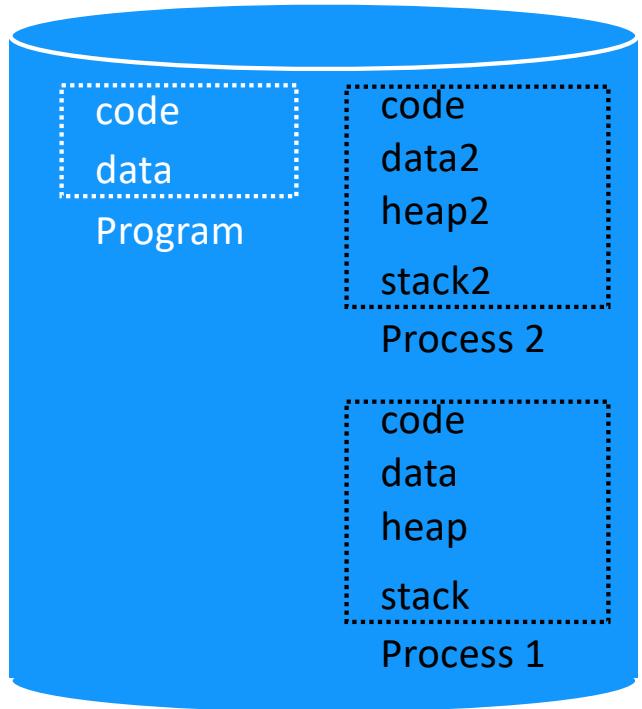


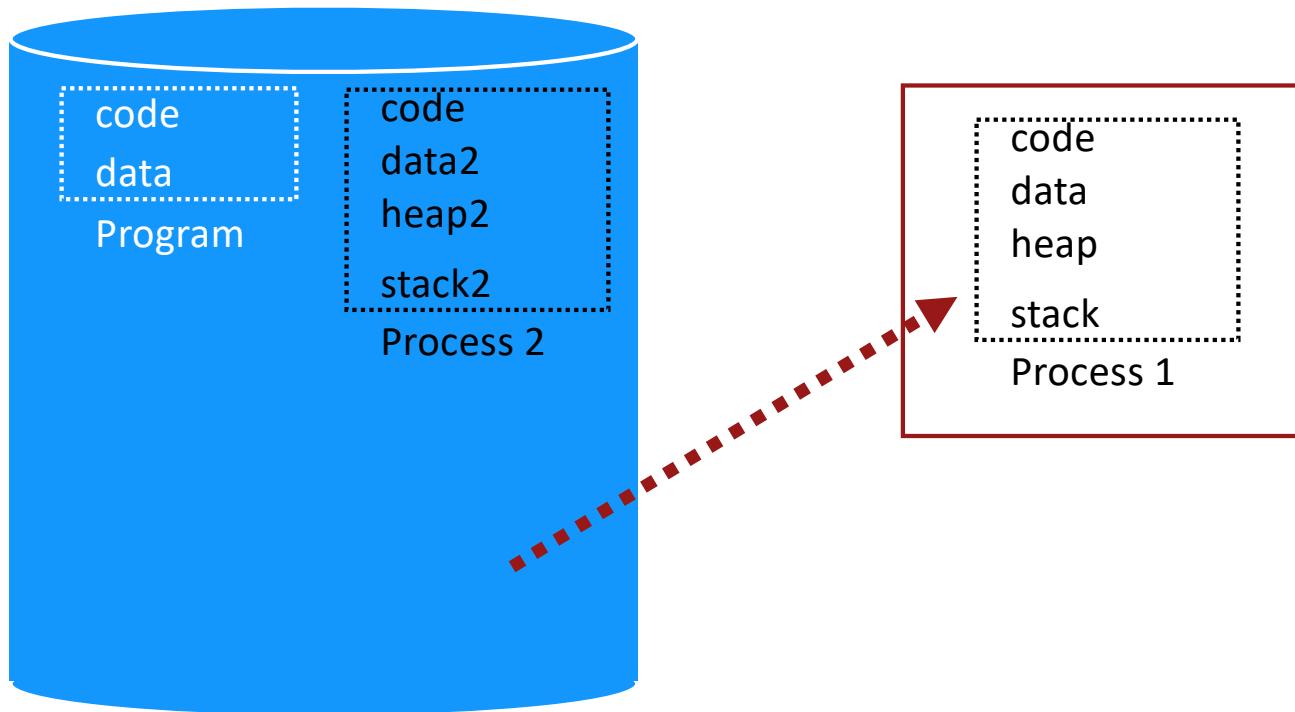


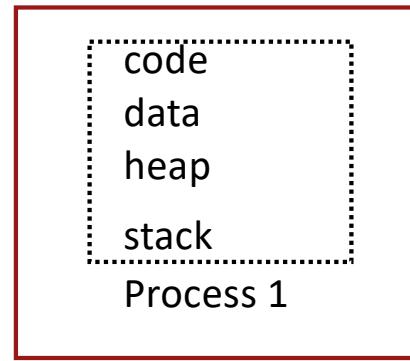
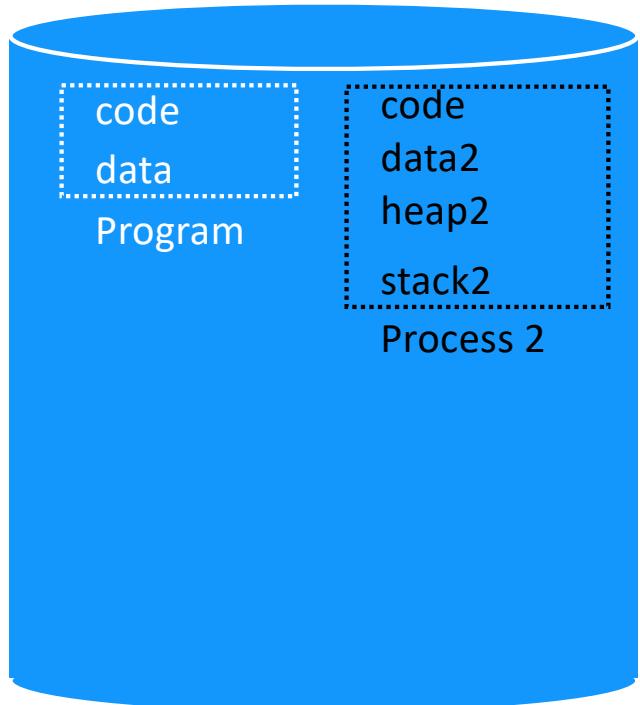












Problems with Time Sharing Memory

Problem: Ridiculously poor performance

Better Alternative: space sharing

- At same time, space of memory is divided across processes

Remainder of solutions all use space sharing

2) Static Relocation

- 0x10: movl 0x8(%rbp), %edi
- 0x13: addl \$0x3, %edi
- 0x19: movl %edi, 0x8(%rbp)

rewrite

```
0x1010: movl 0x8(%rbp), %edi
0x1013: addl $0x3, %edi
0x1019: movl %edi, 0x8(%rbp)
```

rewrite

```
0x3010: movl 0x8(%rbp), %edi
0x3013: addl $0x3, %edi
0x3019: movl %edi, 0x8(%rbp)
```

Static: Layout in Memory



```
0x1010: movl 0x8(%rbp), %edi  
0x1013: addl $0x3, %edi  
0x1019: movl %edi, 0x8(%rbp)
```

```
0x3010: movl 0x8(%rbp), %edi  
0x3013: addl $0x3, %edi  
0x3019: movl %edi, 0x8(%rbp)
```

Static Relocation: Disadvantages

No protection

- Process can destroy OS or other processes
- No privacy

Cannot move address space after it has been placed

- May not be able to allocate new process

3) Dynamic Relocation

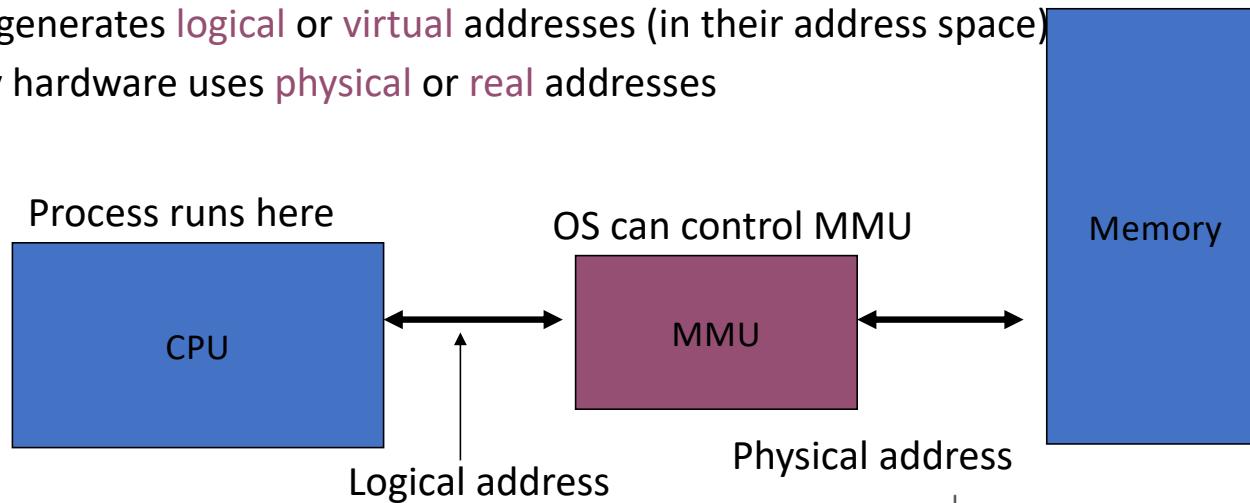
Goal: Protect processes from one another

Requires hardware support

- Memory Management Unit (MMU)

MMU dynamically changes process address at every memory reference

- Process generates **logical** or **virtual** addresses (in their address space)
- Memory hardware uses **physical** or **real** addresses



ILLINOIS TECH

College of Computing

Hardware Support for Dynamic Relocation

Two operating modes

- Privileged (protected, kernel) mode: OS runs
 - When enter OS (trap, system calls, interrupts, exceptions)
 - Allows certain instructions to be executed
 - Can manipulate contents of MMU
 - **Allows OS to access all of physical memory**
- User mode: User processes run
 - Perform translation of logical address to physical address

Minimal MMU contains **base register** for translation

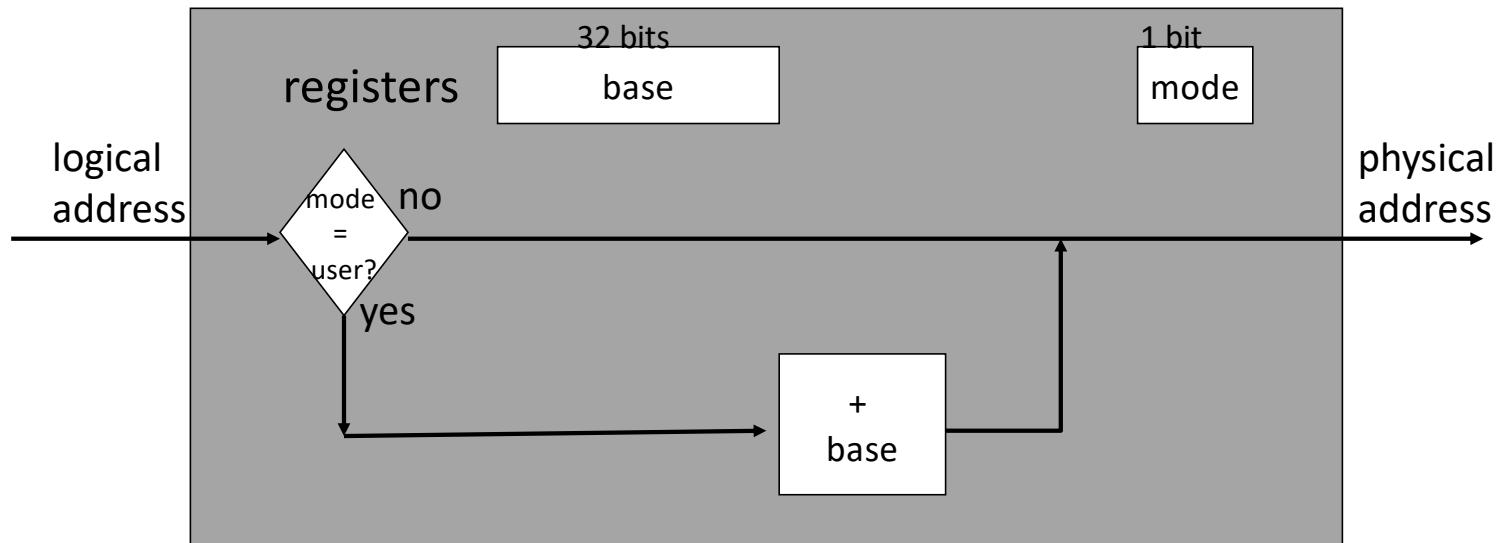
- base: start location for address space

Implementation of Dynamic Relocation: BASE REG

Translation on every memory access of user process

- MMU adds base register to logical address to form physical address

MMU



ILLINOIS TECH

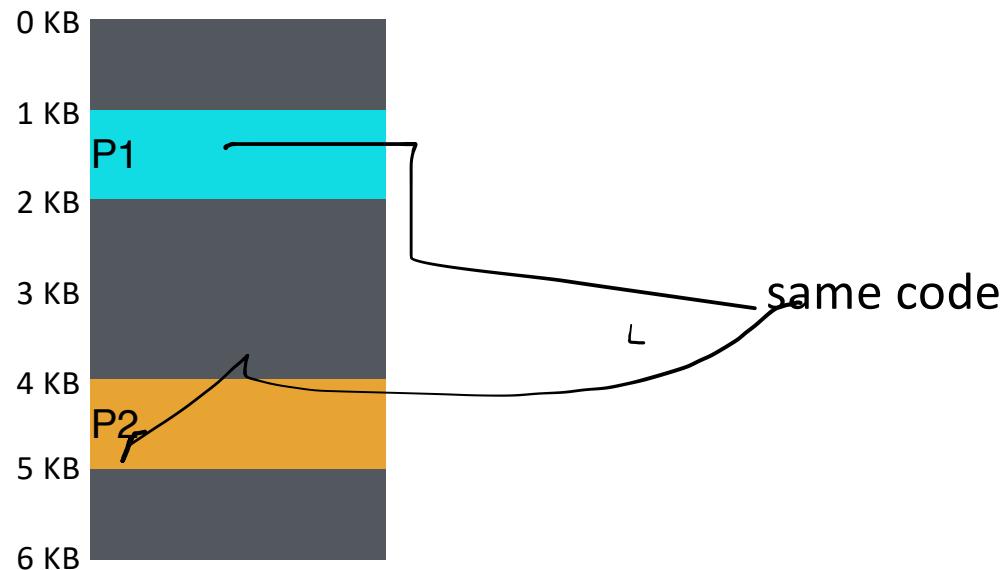
College of Computing

Dynamic Relocation with Base Register

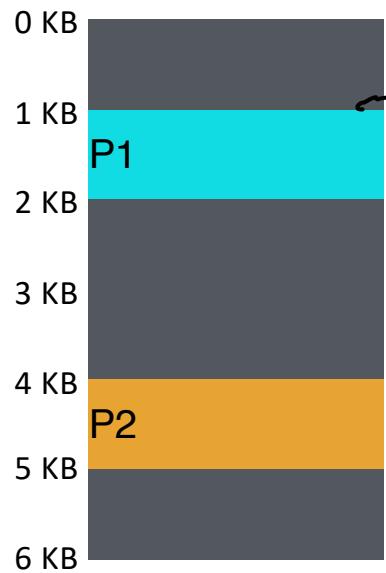
Idea: translate virtual addresses to physical by adding a fixed offset each time.

Store offset in base register

Each process has different value in base register

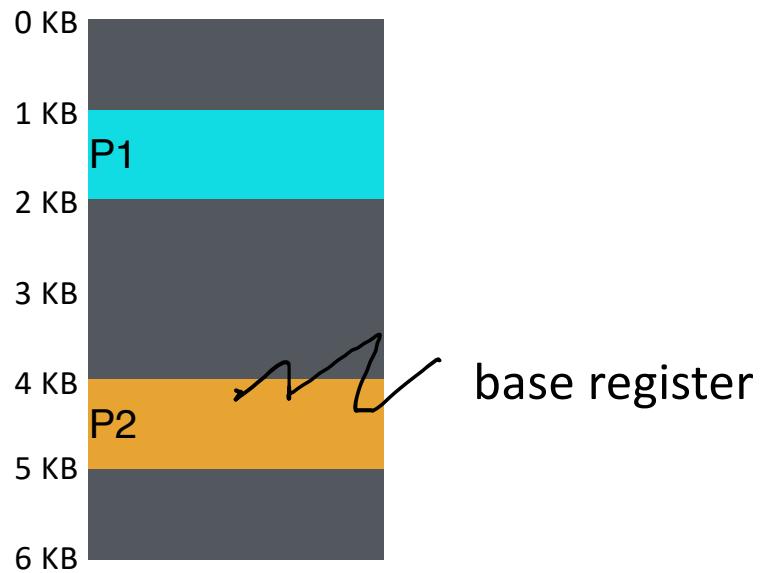


VISUAL Example of DYNAMIC RELOCATION:
BASE REGISTER



base register

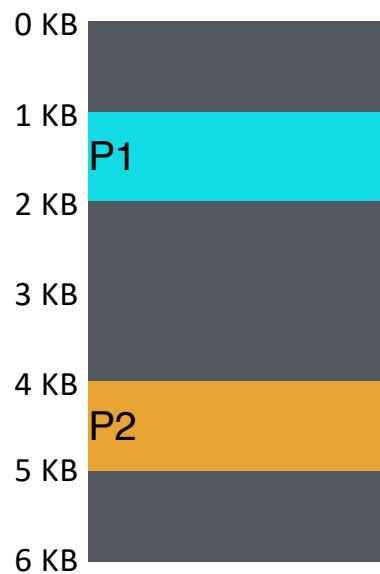
P1 is running



P2 is running

base register

(Decimal notation)



Virtual

Physical

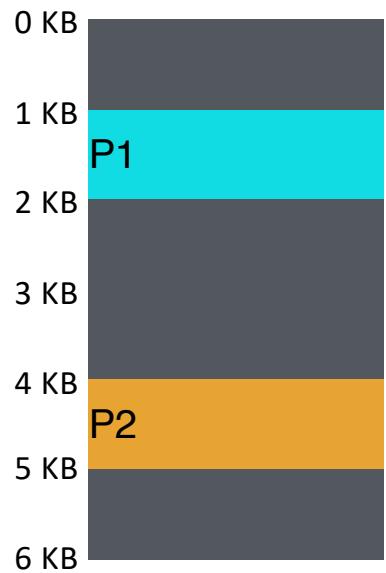
P1: load 100, R1



Virtual
P1: load 100, R1

Physical
load 1124, R1

(1024 + 100)



Virtual

P1: load 100, R1

P2: load 100, R1

Physical

load 1124, R1



Virtual

P1: load 100, R1

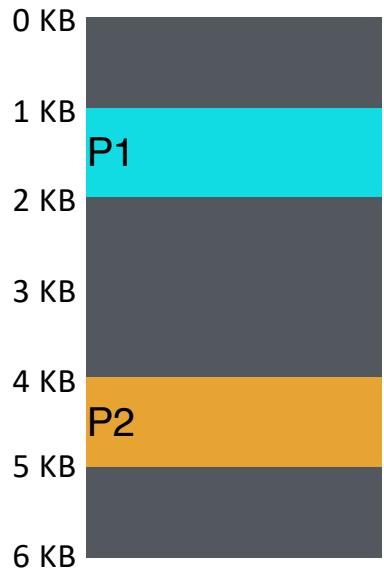
P2: load 100, R1

Physical

load 1124, R1

load 4196, R1

$(4096 + 100)$



Virtual

P1: load 100, R1

P2: load 100, R1

P2: load 1000, R1

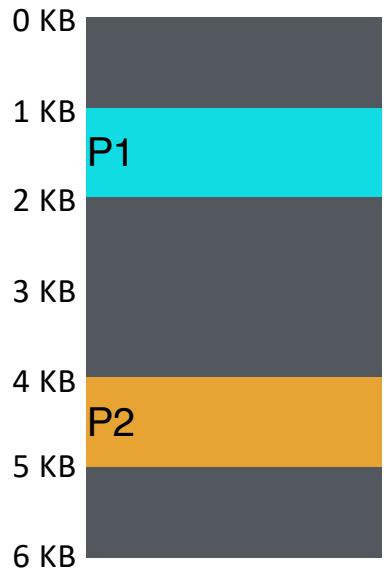
Physical

load 1124, R1

load 4196, R1



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	



Virtual

P1: load 100, R1

P2: load 100, R1

P2: load 1000, R1

P1: load 1000, R1

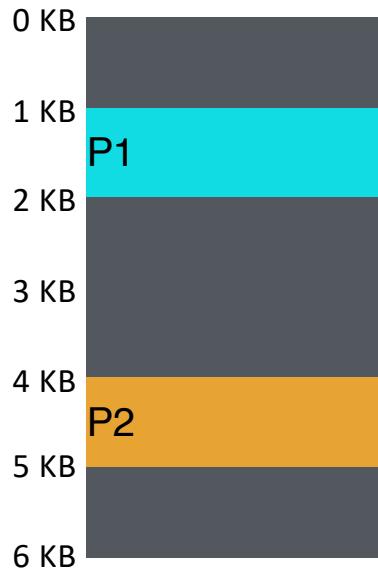
Physical

load 1124, R1

load 4196, R1

load 5196, R1

load 2024, R1



Virtual

P1: load 100, R1

P2: load 100, R1

P2: load 1000, R1

P1: load 100, R1

Physical

load 1124, R1

load 4196, R1

load 5196, R1

load 2024, R1

Can P2 hurt P1?

Can P1 hurt P2?

How well does dynamic relocation do with base register for protection?



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	store 4096, R1 (3072 + 1024)

Can P2 hurt P1?
Can P1 hurt P2?

How well does dynamic relocation do with base register for protection?

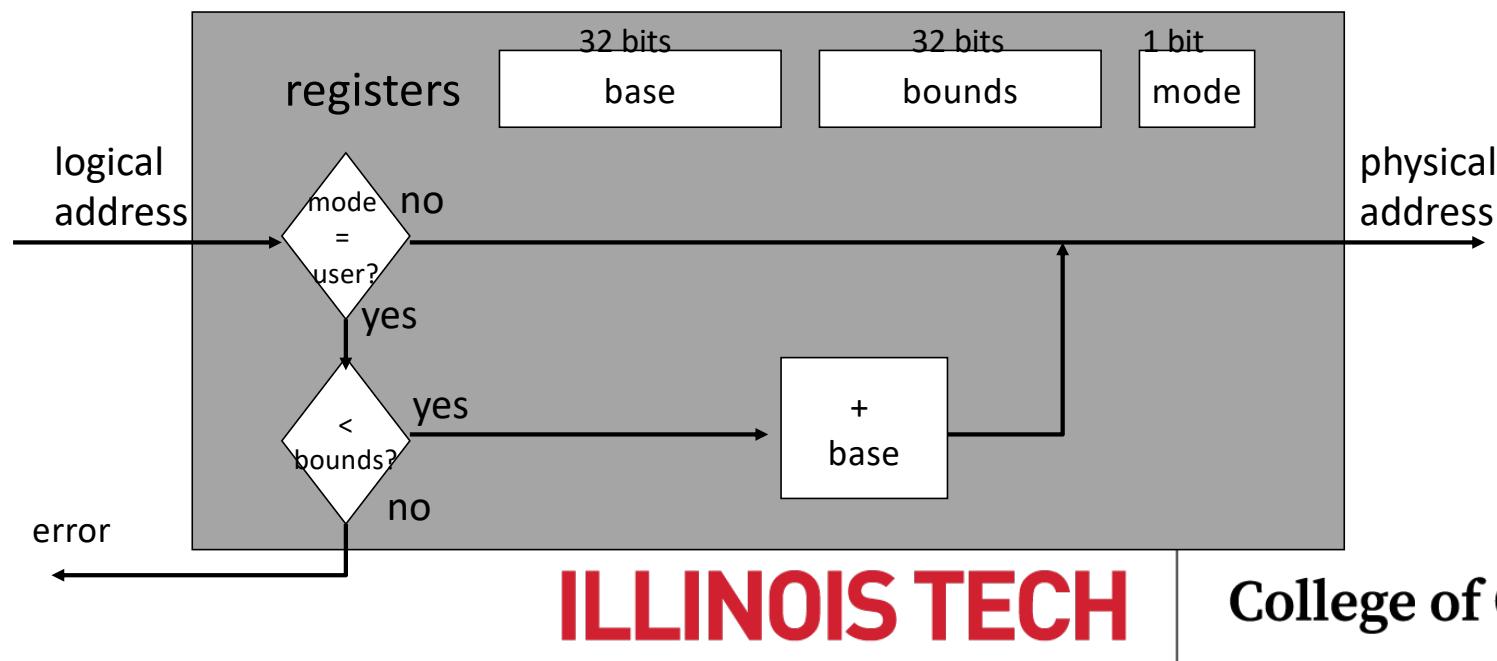
4) Dynamic with Base+Bounds

- Idea: limit the address space with a bounds register
- Base register: smallest physical addr (or starting location)
- Bounds register: size of this process's virtual address space
 - Sometimes defined as largest physical address (base + size)
- OS kills process if process loads/stores beyond bounds

Implementation of BASE+BOUNDS

Translation on every memory access of user process

- MMU compares logical address to bounds register
 - if logical address is greater, then generate error
- MMU adds base register to logical address to form physical address



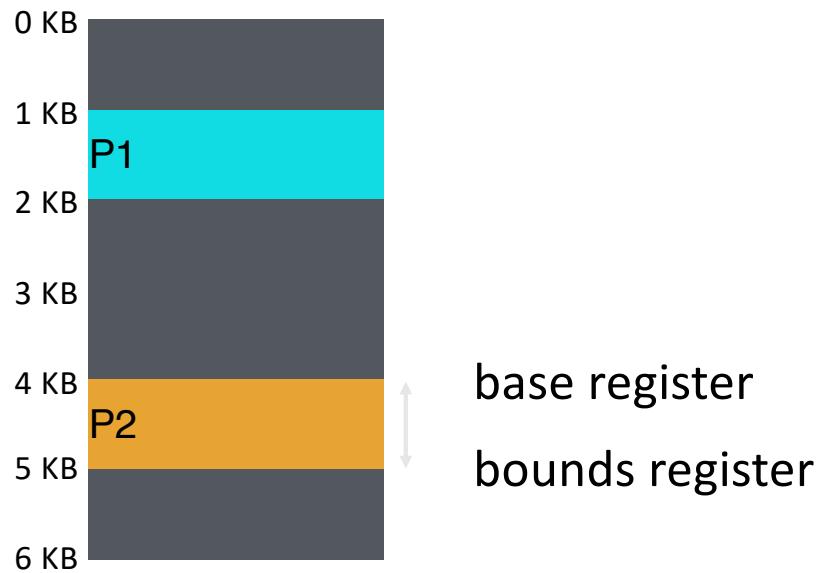
ILLINOIS TECH

College of Computing

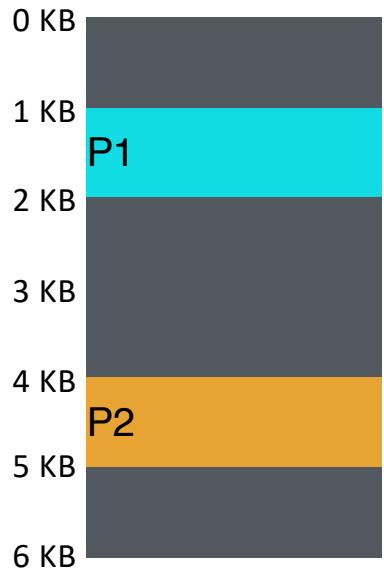


base register
bounds register

P1 is running

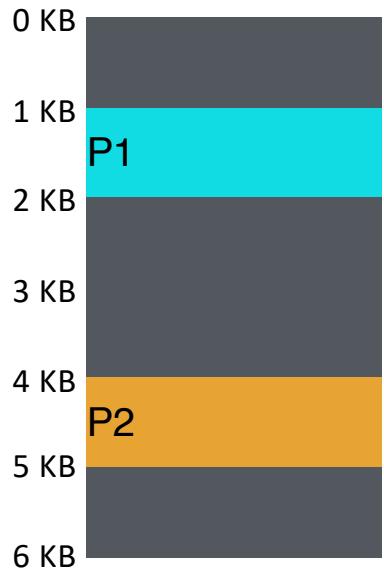


P2 is running



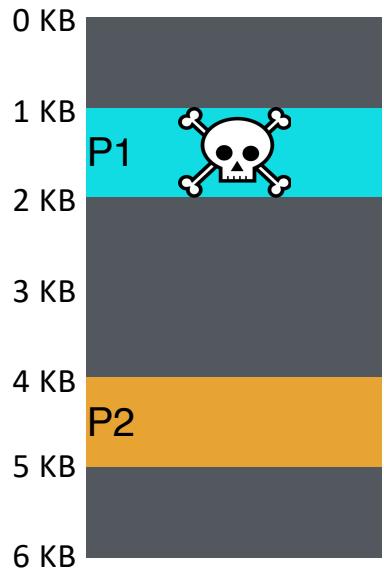
Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	

Can P1 hurt P2?



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	interrupt OS!
	3072 > 1024

Can P1 hurt P2?



Virtual	Physical
P1: load 100, R1	load 1124, R1
P2: load 100, R1	load 4196, R1
P2: load 1000, R1	load 5196, R1
P1: load 100, R1	load 2024, R1
P1: store 3072, R1	interrupt OS!

Can P1 hurt P2?

Managing Processes with Base and Bounds

Context-switch

- Add base and bounds registers to PCB
- Steps
 - Change to privileged mode
 - Save base and bounds registers of old process
 - Load base and bounds registers of new process
 - Change to user mode and jump to new process

What if don't change base and bounds registers when switch?

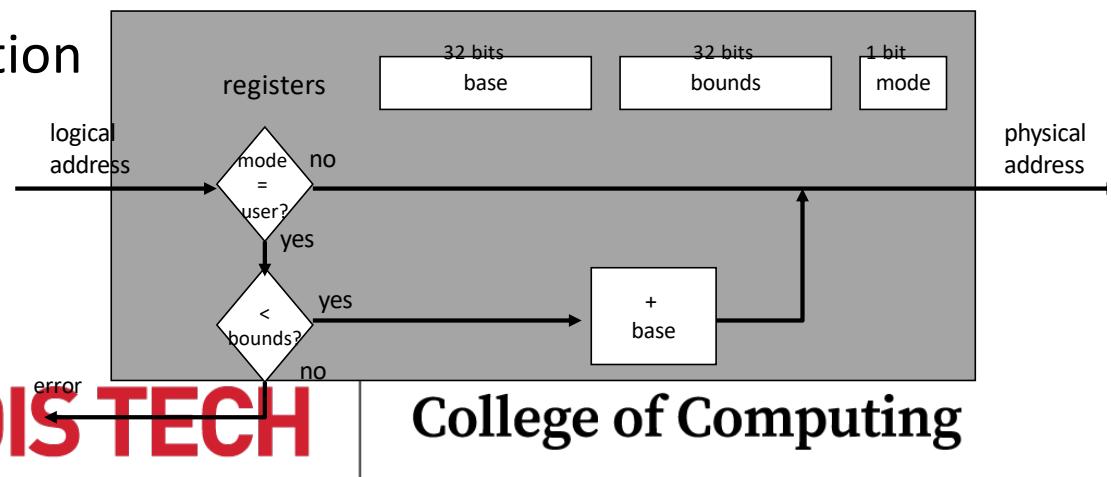
Protection requirement

- User process cannot change base and bounds registers
- User process cannot change to privileged mode

Base and Bounds Advantages

Advantages

- Provides protection (both read and write) across address spaces
- Supports dynamic relocation
 - Can place process at different locations initially and also move address spaces
- Simple, inexpensive implementation
 - Few registers, little logic in MMU
- Fast
 - Add and compare in parallel



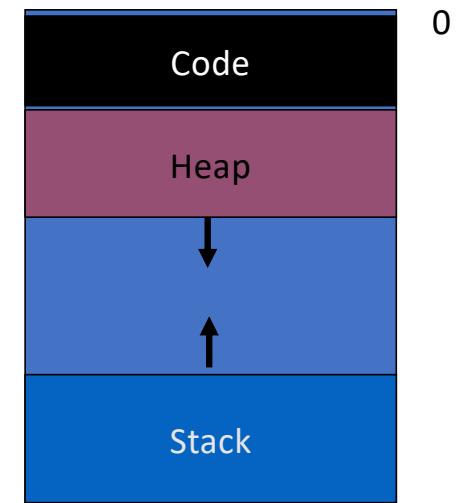
ILLINOIS TECH

College of Computing

Base and Bounds DISADVANTAGES

Disadvantages

- Each process must be allocated contiguously in physical memory
 - Must allocate memory that may not be used by process
- No partial sharing: Cannot share limited parts of address space



ILLINOIS TECH

College of Computing $_{2^{n-1}}$

Hardware address translation

- To support fast, dynamic translation and address space protection, hardware support for address translation is needed
- Idea:
 - All process memory requests (on the CPU) are for virtual addresses
 - Hardware translates each request to a physical address
 - Kernel sets up mapping from virtual address spaces to physical memory
 - Translation hardware is the memory management unit (MMU)

Primary goals

- Transparency
 - Processes aren't involved in (or aware of) the translation process
- Efficiency
 - Time (speed/throughput) & Space (utilization)
- Protection
 - Processes cannot access data outside their own address spaces
 - Isolated from each other and the kernel

ILLINOIS TECH

College of Computing

Simple implementation: Relocation

- Assumptions:
 - Fixed size process address spaces
 - Process address space < Physical memory
 - Relocation requires a uniform shift for every request

Base address

- Kernel maintains base address of each process in PCB
 - Load into base (address) register in MMU on each context switch
 - Relocation = register access + addition
- Problem: protection not guaranteed!

Base + Limit registers

- Incorporate a limit register to enforce memory protection
- Assertion failure triggers fault (software exception) and loads kernel

Questions?

ILLINOIS TECH

College of Computing

Segmentation and Paging

Ben Lenard

blenard@iit.edu

ILLINOIS TECH

College of Computing

Agenda

- Homework 3
- Address spaces recap
- Base + Bounds
- Segmentation
- Paging

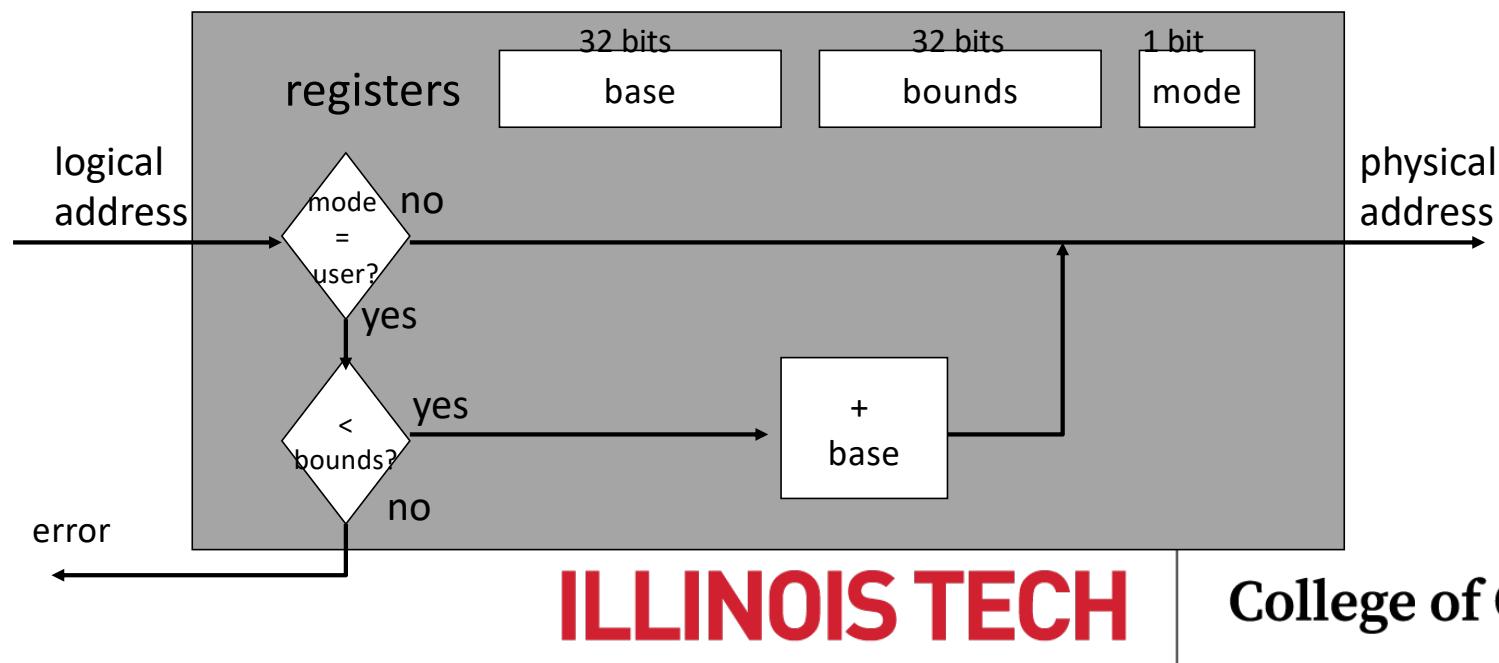
ILLINOIS TECH

College of Computing

Implementation of BASE+BOUNDS

Translation on every memory access of user process

- MMU compares logical address to bounds register
 - if logical address is greater, then generate error
- MMU adds base register to logical address to form physical address



ILLINOIS TECH

College of Computing

Base address

- Kernel maintains base address of each process in PCB
 - Load into base (address) register in MMU on each context switch
 - Relocation = register access + addition
- Problem: protection not guaranteed!

Base + Limit registers

- Incorporate a limit register to enforce memory protection
- Assertion failure triggers fault (software exception) and loads kernel

Managing Processes with Base and Bounds

Context-switch

- Add base and bounds registers to PCB
- Steps
 - Change to privileged mode
 - Save base and bounds registers of old process
 - Load base and bounds registers of new process
 - Change to user mode and jump to new process

What if don't change base and bounds registers when switch?

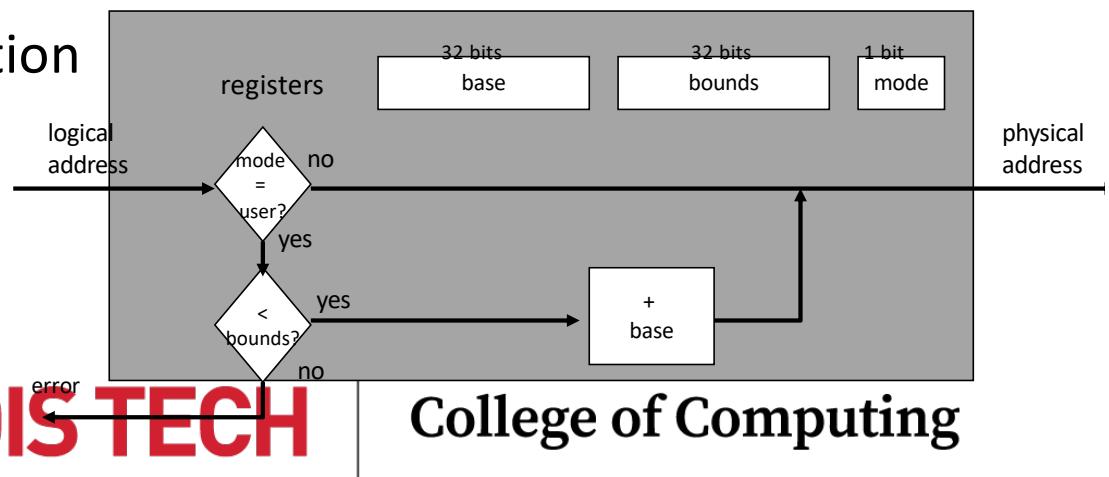
Protection requirement

- User process cannot change base and bounds registers
- User process cannot change to privileged mode

Base and Bounds Advantages

Advantages

- Provides protection (both read and write) across address spaces
- Supports dynamic relocation
 - Can place process at different locations initially and also move address spaces
- Simple, inexpensive implementation
 - Few registers, little logic in MMU
- Fast
 - Add and compare in parallel



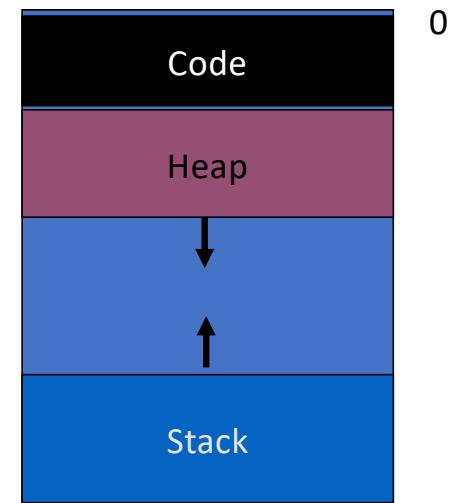
ILLINOIS TECH

College of Computing

Base and Bounds DISADVANTAGES

Disadvantages

- Each process must be allocated contiguously in physical memory
 - Must allocate memory that may not be used by process
- No partial sharing: Cannot share limited parts of address space



ILLINOIS TECH

College of Computing $_{2^{n-1}}$

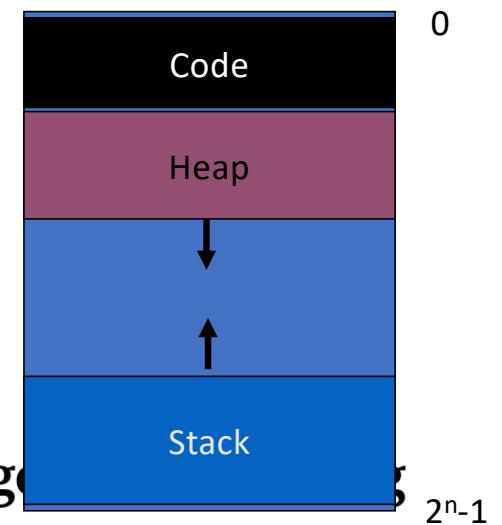
Segmentation

Divide address space into logical segments

- Each segment corresponds to logical entity in address space
 - code, stack, heap

Each segment can independently:

- be placed separately in physical memory
- grow and shrink
- be protected (separate read/write/execute protection bits)



Segmentation (Cont.)

- Partition virtual address space into multiple disjoint segments
 - Individually map onto physical memory with separate base/limit registers
 - Address space info stored in PCB and restored on context switch - Requires that memory requests are for segmented addresses
- Consist of segment selector and offset into segment
- Alternatively: segment can be implied by instruction (e.g., PC always refers to code segment)

Segmented Addressing

Process now specifies segment and offset within segment

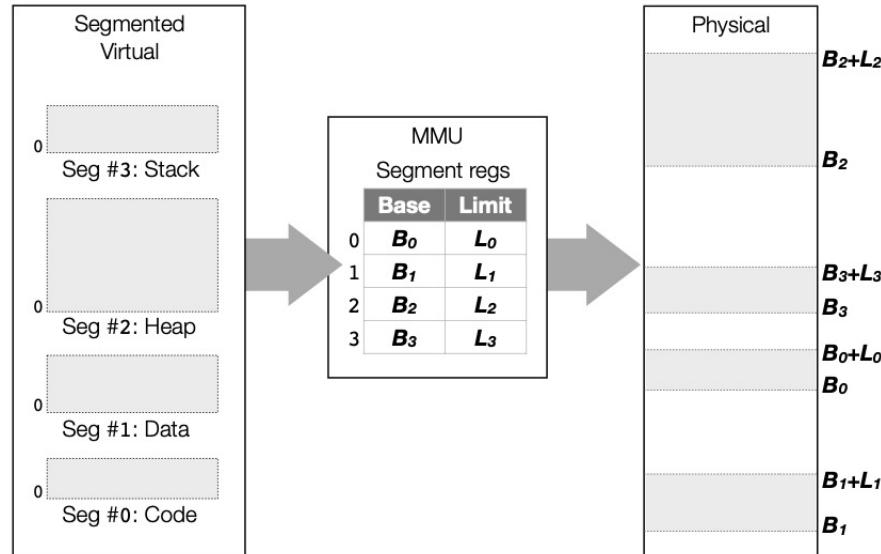
How does process designate a particular segment?

- Use part of logical address
 - Top bits of logical address select segment
 - Low bits of logical address select offset within segment

What if small address space, not enough bits?

- Implicitly by type of memory reference
- Special registers

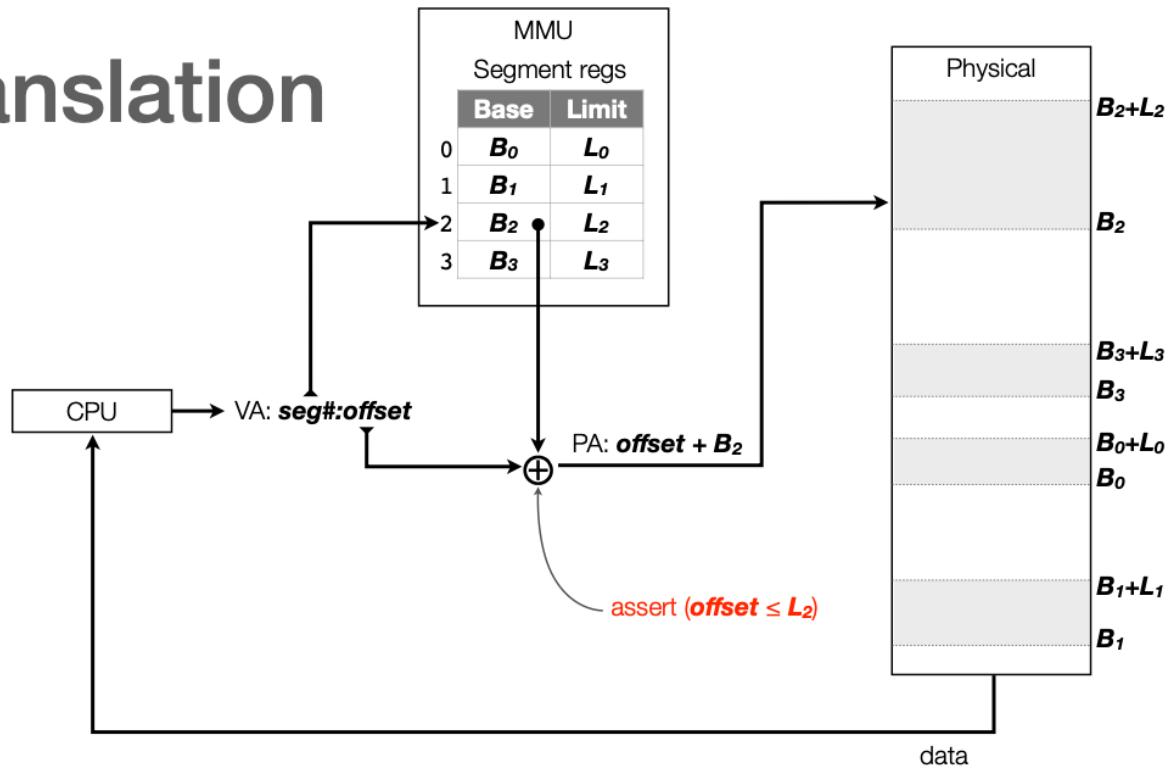
E.g., logical segments



ILLINOIS TECH

College of Computing

E.g., translation

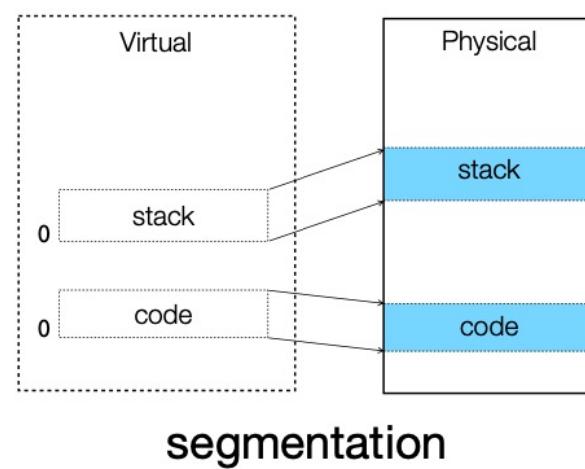
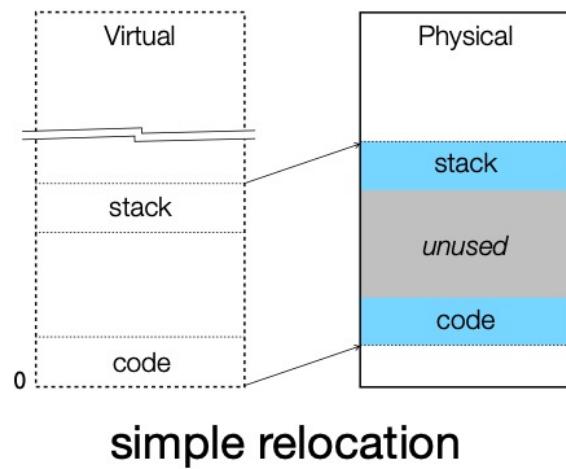


ILLINOIS TECH

College of Computing

Improved utilization

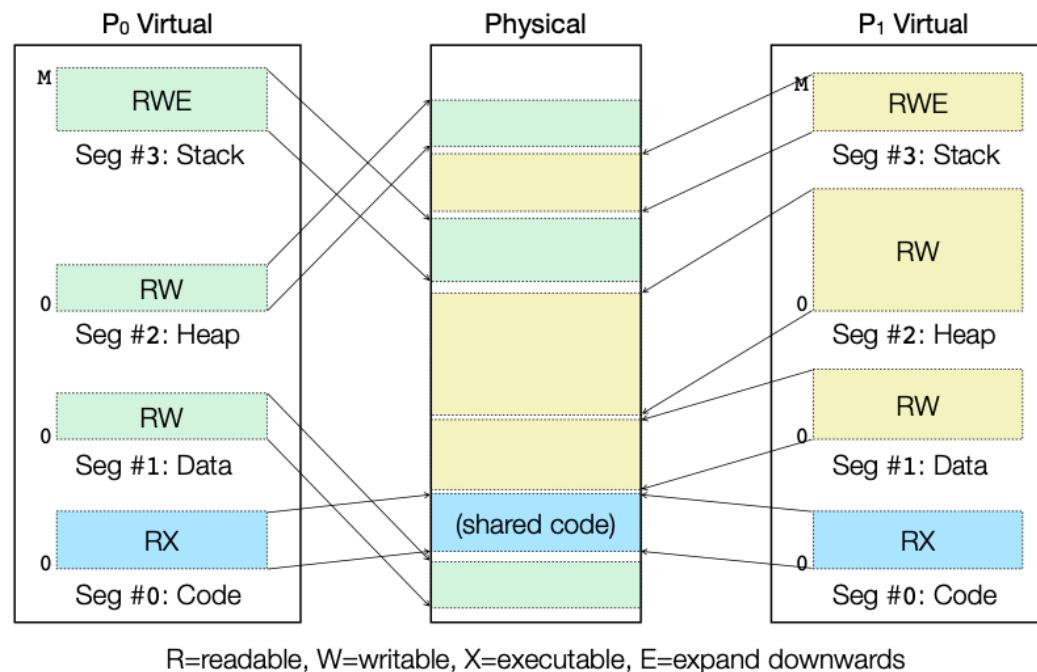
- Mapping individual segments avoids reserving memory for unused space between segments in the linear address space



Segment sharing and metadata

- Segments may be shared between processes to reduce memory usage (and improve caching behavior)
- Segments may have additional metadata to control and limit access -
Read-only / Non-executable segments
 - Privilege-level based access control (kernel vs. user)
 - Direction of growth (e.g., downward from max offset for stacks)

E.g., shared code and metadata



ILLINOIS TECH

College of Computing

Segmentation fault

- A segmentation fault can be generated by the MMU when:
 - Limit check fails (access beyond ends of segment)
 - Access control assertion fails (e.g., illegal operation)
 - Privilege assertion fails (e.g., insufficient privilege)
- Fault transfers control to kernel (to alert/terminate offending process)

Segmentation Implementation

MMU contains Segment Table (per process)

- Each segment has own base and bounds, protection bits
- Example: 14 bit logical address, 4 segments; how many bits for segment? How many bits for offset?

Segment	Base	Bounds	R	W
0	0x2000	0x6ff	1	0
1	0x0000	0x4ff	1	1
2	0x3000	0xffff	1	1
3	0x0000	0x000	0	0

remember:
1 hex digit->4 bits

ILLINOIS TECH

College of Computing

Quiz: Address Translations with Segmentation

MMU contains Segment Table (per process)

- Each segment has own base and bounds, protection bits
- Example: 14 bit logical address, 4 segments; how many bits for segment? How many bits for offset?

Segment	Base	Bounds	R	W
0	0x2000	0x6ff	1	0
1	0x0000	0x4ff	1	1
2	0x3000	0xffff	1	1
3	0x0000	0x000	0	0

remember:
1 hex digit->4 bits

Translate logical addresses (in hex) to physical addresses

0x0240:

0x1108:

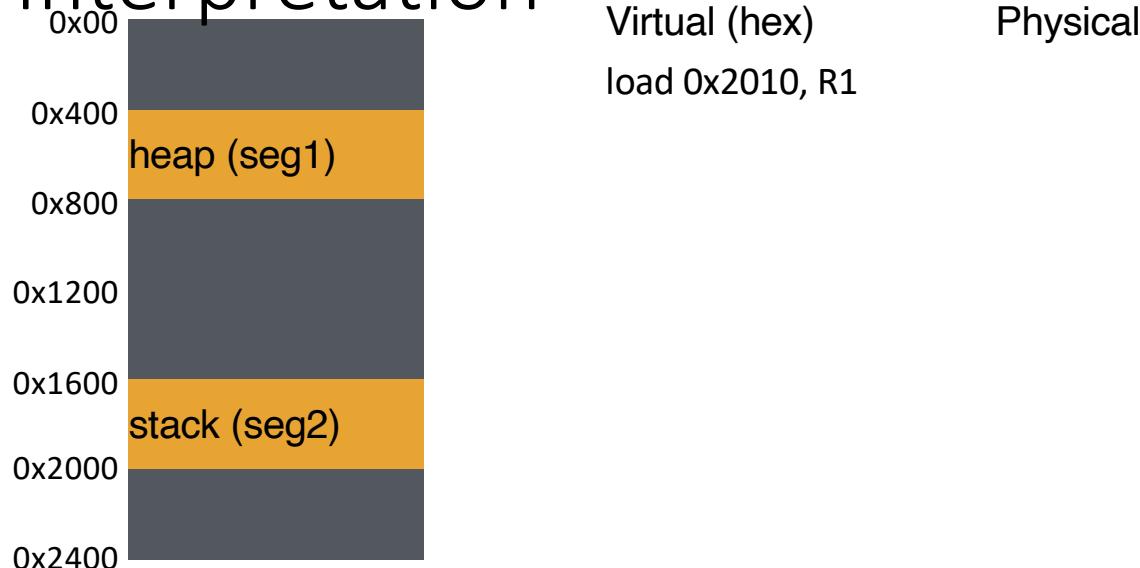
0x265c:

0x3002:

ILLINOIS TECH

College of Computing

Visual Interpretation



Segment numbers:

0: code+data

1: heap

2: stack



Virtual (hex)

load 0x2010, R1

Physical

$0x1600 + 0x010 = 0x1610$

Segment numbers:

0: code+data

1: heap

2: stack



Segment numbers:

0: code+data

1: heap

2: stack



Virtual (hex)

load 0x2010, R1

load 0x1010, R1

Physical

$0x1600 + 0x010 = 0x1610$

$0x400 + 0x010 = 0x410$

Segment numbers:

0: code+data

1: heap

2: stack

	Virtual	Physical
0x00		
0x400	load 0x2010, R1	$0x1600 + 0x010 = 0x1610$
0x800	load 0x1010, R1	$0x400 + 0x010 = 0x410$
0x1200		
0x1600	load 0x1100, R1	
0x2000		
0x2400		

heap (seg1)

stack (seg2)

Segment numbers:

0: code+data

1: heap

2: stack

	Virtual	Physical
0x00		
0x400	load 0x2010, R1	$0x1600 + 0x010 = 0x1610$
	heap (seg1)	
0x800	load 0x1010, R1	$0x400 + 0x010 = 0x410$
0x1200		
0x1600	load 0x1100, R1	$0x400 + 0x100 = 0x500$
	stack (seg2)	
0x2000		
0x2400		

Segment numbers:

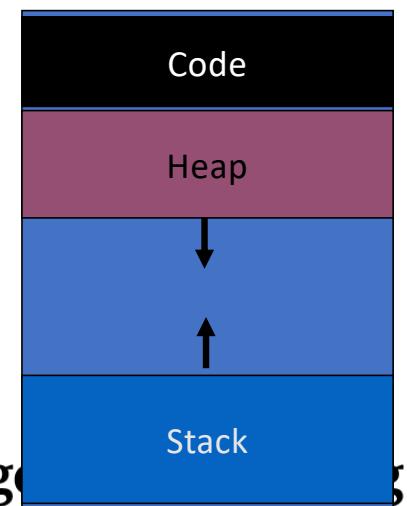
0: code+data

1: heap

2: stack

Advantages of Segmentation

- Enables sparse allocation of address space
 - Stack and heap can grow independently
 - Heap: If no data on free list, dynamic memory allocator requests more from OS (e.g., UNIX: malloc calls sbrk())
 - Stack: OS recognizes reference outside legal segment, extends stack implicitly
- Different protection for different segments
 - Read-only status for code
- Enables sharing of selected segments
- Supports dynamic relocation of each segment



ILLINOIS TECH

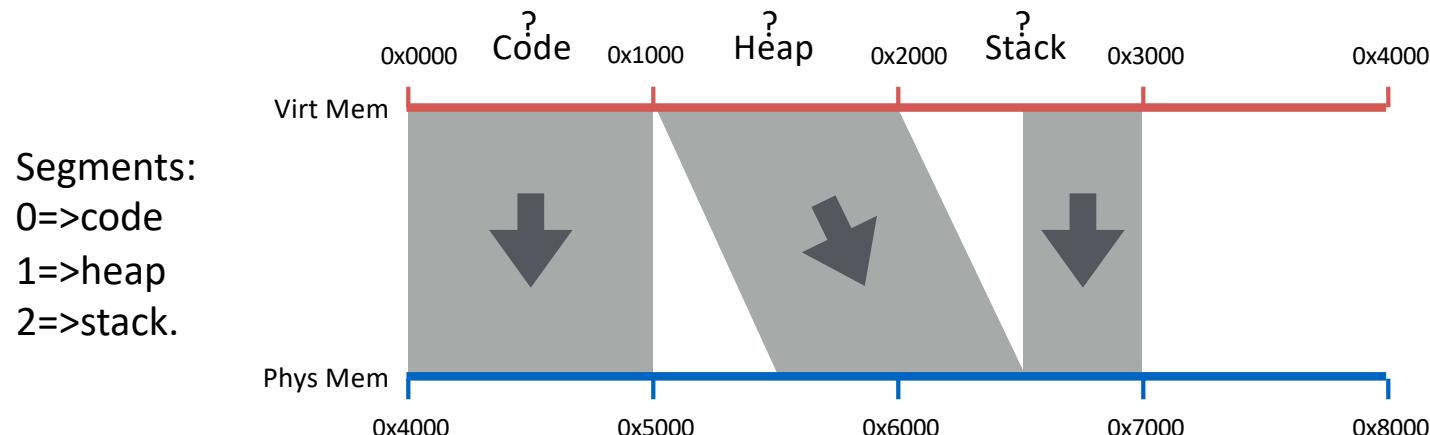
College

of

Computers

Segmentation

Assume 14-bit virtual addresses, high 2 bits indicate segment



Where does segment table live?

All registers, MMU

Problem: Fragmentation

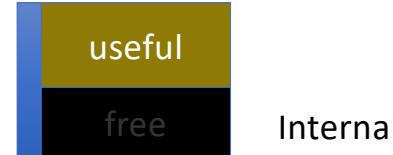
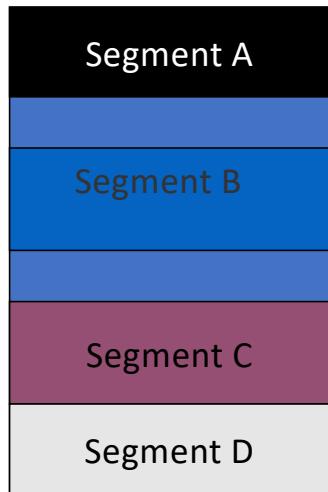
Definition: Free memory that can't be usefully allocated

Why?

- Free memory (hole) is too small and scattered
- Rules for allocating memory prohibit using this free space

Types of fragmentation

- External: Visible to allocator (e.g., OS)
- Internal: Visible to requester (e.g., if must allocate at some granularity)

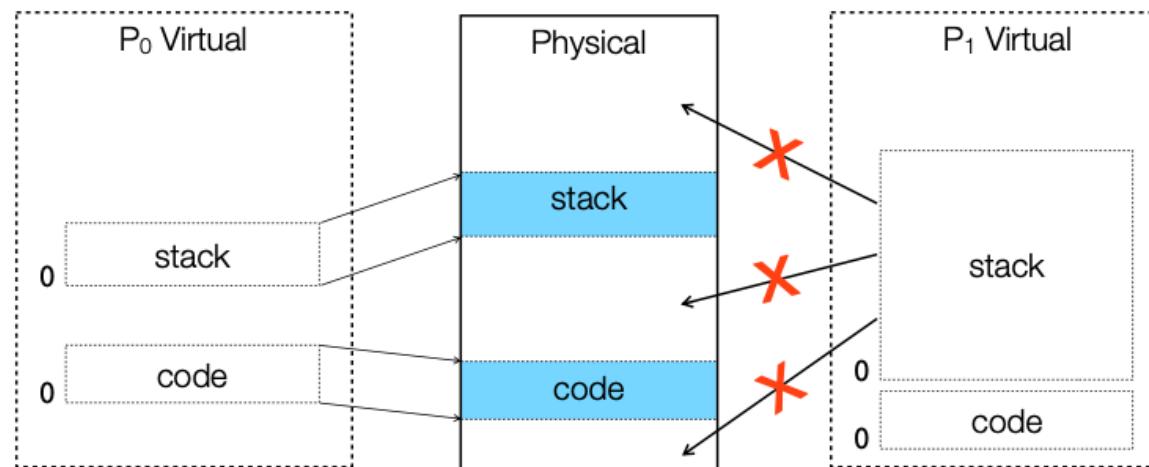


ILLINOIS TECH

College of Computing

Disadvantages of Segmentation

- Variable segment sizes make placement and free space search non-trivial
 - Memory may be defragmented via compaction, but this is expensive!
- Also, large segments still loaded monolithically (coarse-grained mapping)



ILLINOIS TECH

College of Computing

Conclusion

- Fast translation via base register + offset
- Protection enforced via limits
- Improved memory utilization over monolithic mapping
- Access control and sharing via additional segment metadata
- But variable, monolithic segments create external fragmentation, making free space search and/or compaction necessary

Addressing segmentation issues

- Variable segment sizes cause external fragmentation
 - Instead, partition address space and main memory into fixed-size pages
- Large, monolithic segments may reduce utilization, as only a fraction of a segment may be needed at a given time
 - Instead, reduce the granularity of mapping with smaller pages

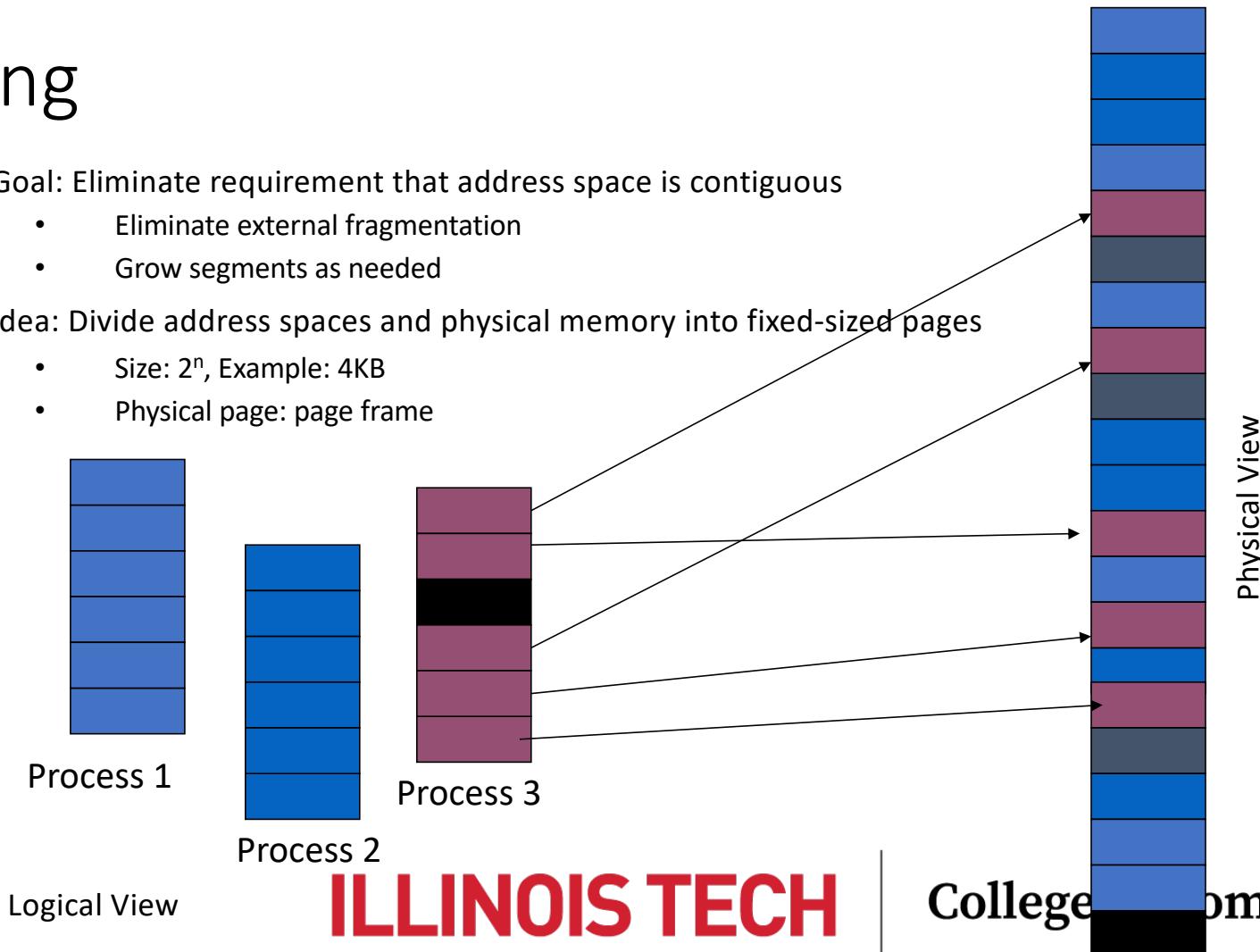
Paging

Goal: Eliminate requirement that address space is contiguous

- Eliminate external fragmentation
- Grow segments as needed

Idea: Divide address spaces and physical memory into fixed-sized pages

- Size: 2^n , Example: 4KB
- Physical page: page frame



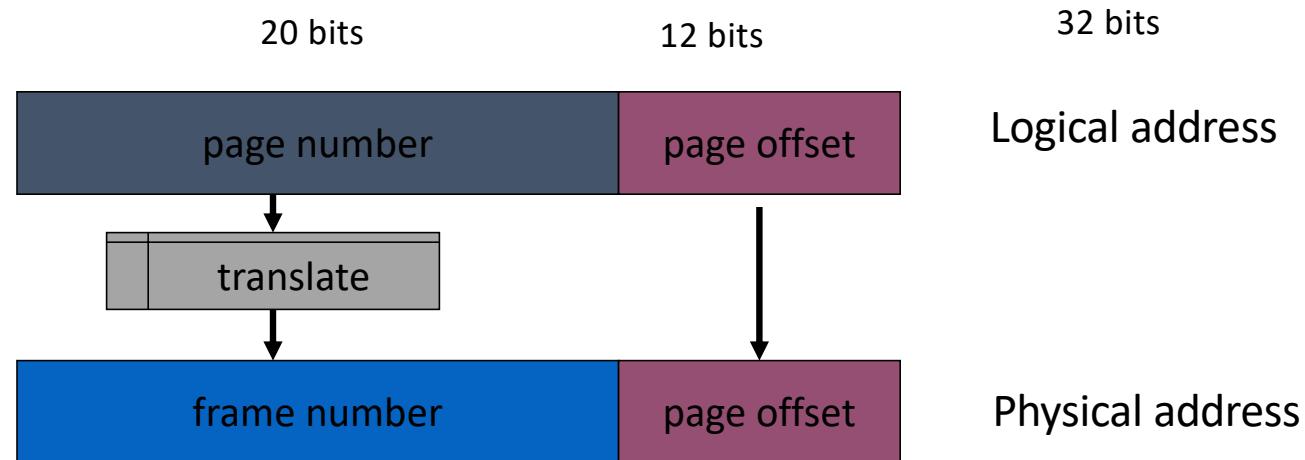
ILLINOIS TECH

College of Computing

Translation of Page Addresses

- How to translate logical address to physical address?
 - High-order bits of address designate page number
 - Low-order bits of address designate offset within page

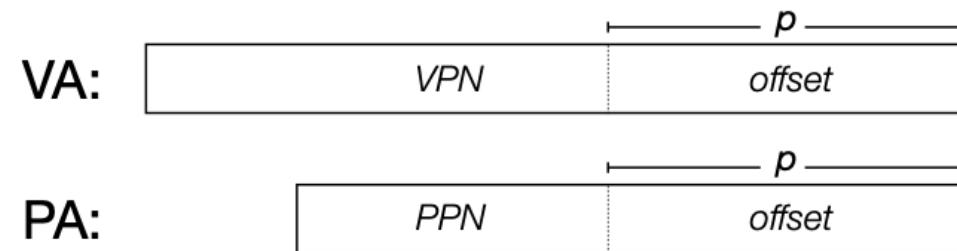
How does format of address space determine number of pages and size of pages?



No addition needed; just append bits correctly...

Translation of Page Addresses (cont.)

- A virtual address is broken down into a virtual page number and offset
- Mapping problem: virtual page number (VPN) \rightarrow physical page number (PPN)
(latter aka physical frame number (PFN))

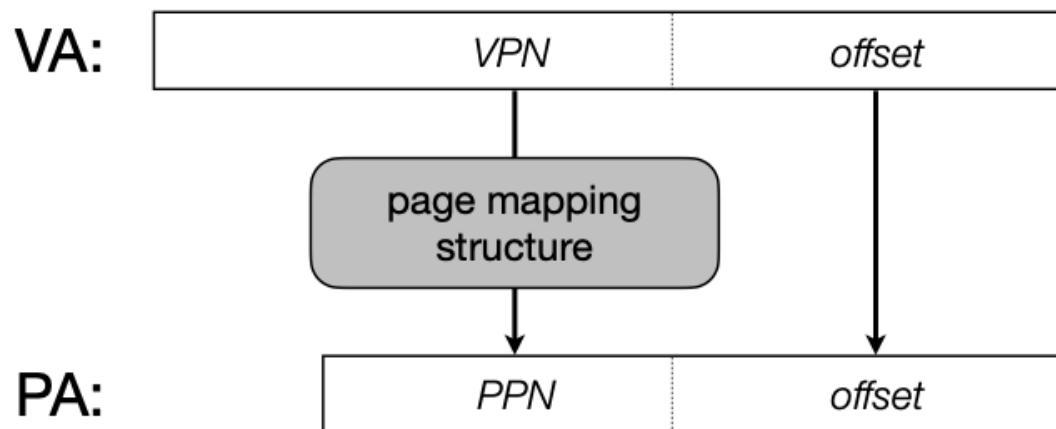


ILLINOIS TECH

College of Computing

Modified mapping problem

- Issue: how to store mappings?
 - I.e., what data structure to use for representing virtual address spaces?

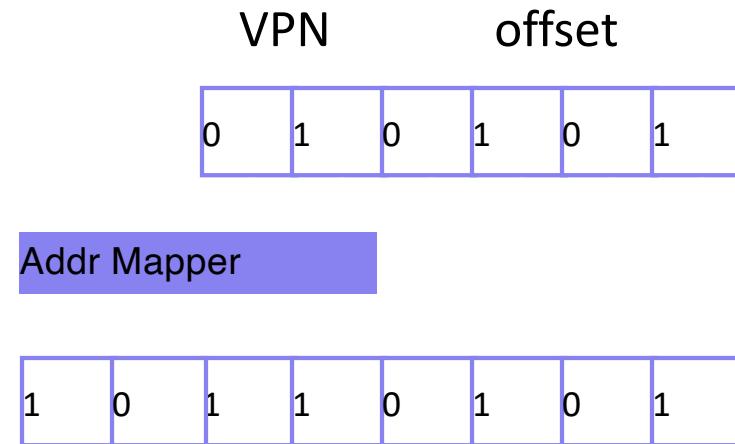


ILLINOIS TECH

College of Computing

VirtUAL => Physical PAGE Mapping

Number of bits in virtual address format does not need to equal number of bits in physical address format



How should OS translate VPN to PPN?

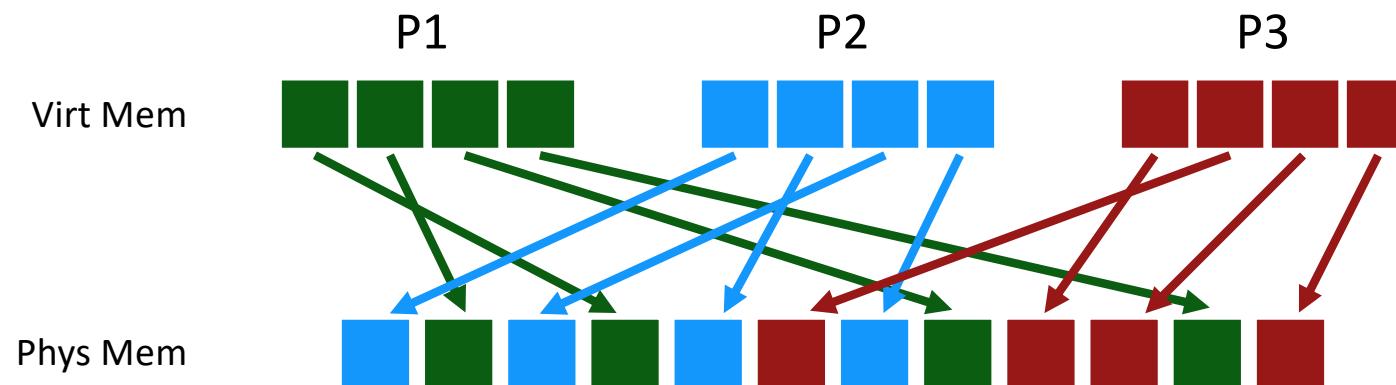
For segmentation, OS used a formula (e.g., phys addr = virt_offset + base_reg)

For paging, OS needs more general mapping mechanism

What data structure is good?

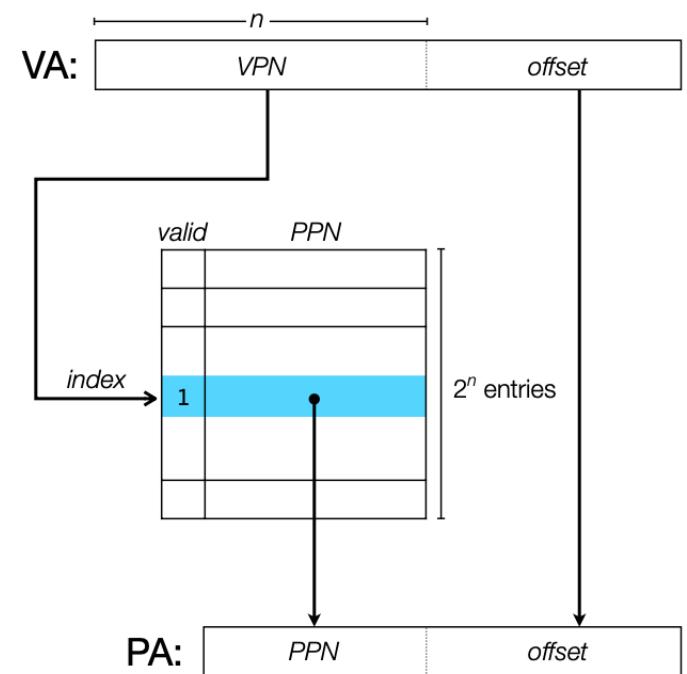
Big array: pagetable

The Mapping



Page table

- VPN from virtual address acts as an index into the page table
- PPN (if valid and mapped) is concatenated with offset from VA to form



ILLINOIS TECH

College of Computing

Where Are Pagetables Stored?

How big is a typical page table?

- assume **32-bit** address space
- assume 4 KB pages
- assume 4 byte entries

Final answer: $2^{(32 - \log(4KB))} * 4 = 4 \text{ MB}$

- Page table size = Num entries * size of each entry
- Num entries = num virtual pages = $2^{\text{bits for vpn}}$
- Bits for vpn = 32 – number of bits for page offset
 $= 32 - \lg(4KB) = 32 - 12 = 20$
- Num entries = $2^{20} = 1 \text{ MB}$
- Page table size = Num entries * 4 bytes = 4 MB

Implication: Store each page table in memory

- Hardware finds page table base with register (e.g., CR3 on x86)

What happens on a context-switch?

- Change contents of page table base register to newly scheduled process
- Save old page table base register in PCB of descheduled process

Other PT info

What other info is in pagetable entries besides translation?

- valid bit
- protection bits
- present bit (needed later)
- reference bit (needed later)
- dirty bit (needed later)

Pagetable entries are just bits stored in memory

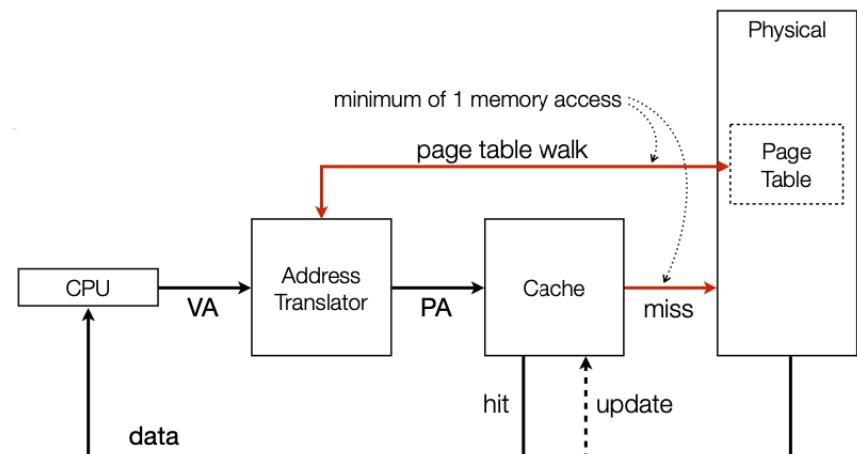
- Agreement between hw and OS about interpretation

Page table walk

- The page table is too large to fit into the MMU, so resides in memory
- Translating a VPN → PPN requires indexing into the page table (known as a page table walk)
 - Performed by MMU
 - Page table is managed by the kernel for each process
 - Current process page table is selected by kernel on each context switch (e.g., by pointing a page table base register at it)

Page table translations are slow!

- Most modern caching systems are physically addressed, so we cannot avoid translation before cache lookup
- I.e., each VA access requires up to two memory accesses!

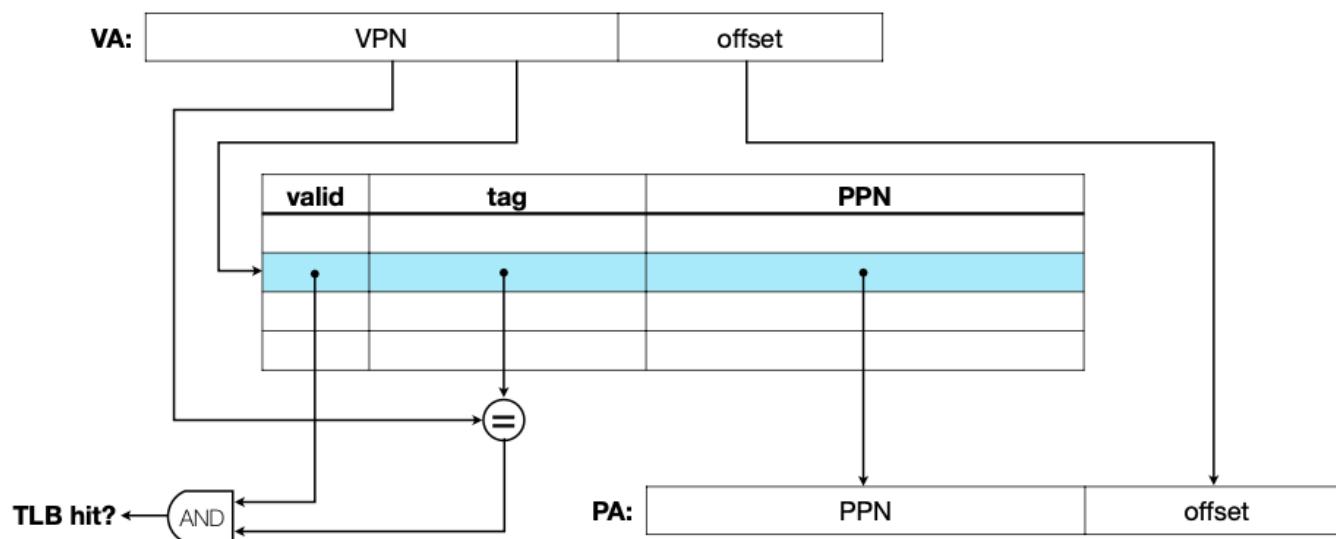


ILLINOIS TECH

College of Computing

Translation Lookaside Buffer (TLB)

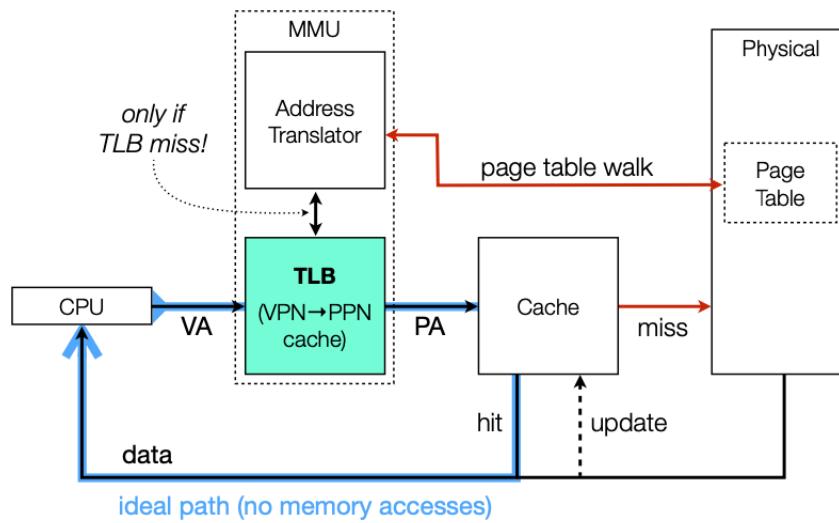
- Solution: dedicated cache for VPN → PPN translations
 - Page table walk only performed on TLB miss



ILLINOIS TECH

College of Computing

TLB / Cache / PT interaction



ILLINOIS TECH

College of Computing

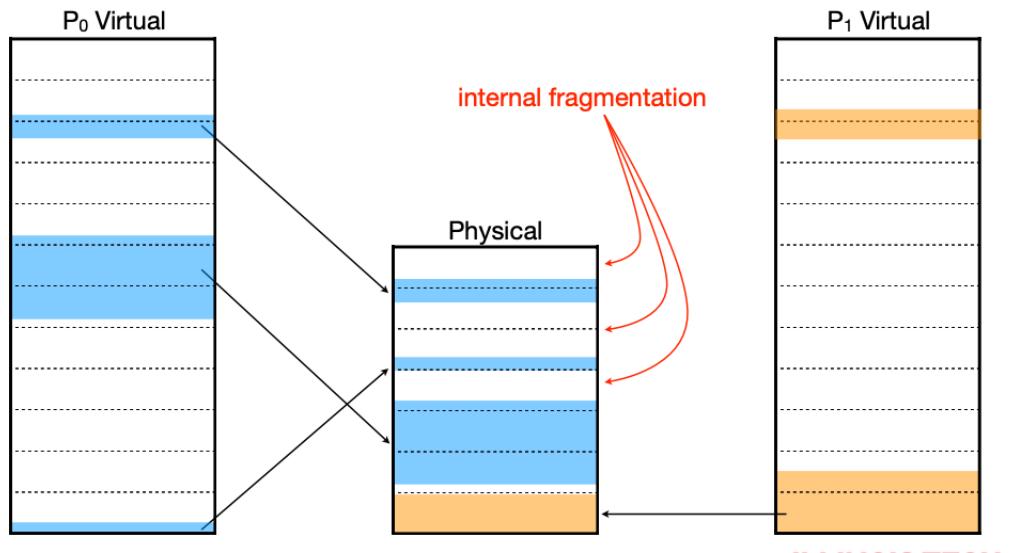
TLB issues

- TLB mappings are process specific — requires flush on context switch
 - Some architectures store address space identifier per cache line
- TLB only caches a few thousand mappings, at most
 - vs. orders of magnitude more per process, potentially!
 - Effectiveness of TLB can be “tuned” by adjusting number of pages (larger page size = smaller number of pages)
 - Downside to large pages?

Internal fragmentation

- Large pages result in coarser mapping granularity
 - I.e., larger “chunks” carved out of physical memory at a time
 - May lower utilization, if large portions of pages are not used — known as internal fragmentation
- Must balance TLB effectiveness against memory utilization
- Depends on the application and workload

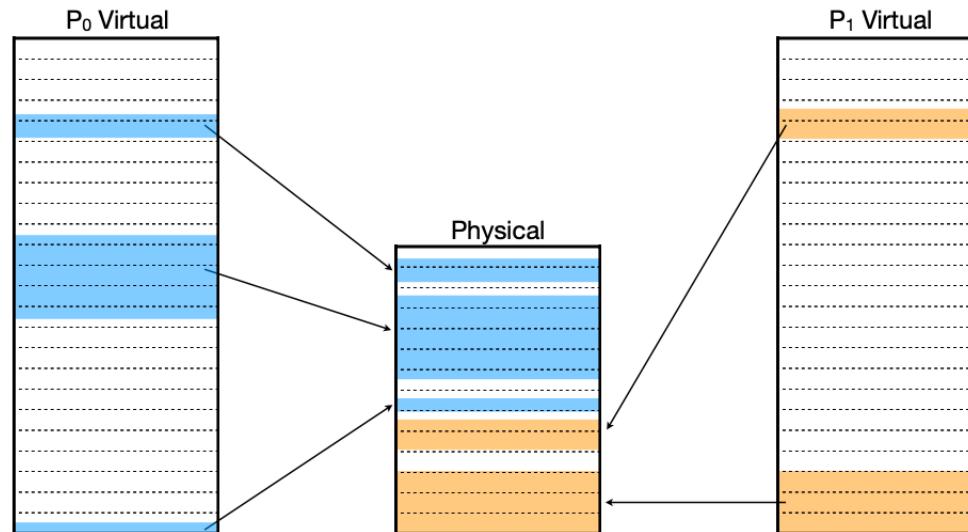
E.g., large(-ish) pages



ILLINOIS TECH

College of Computing

E.g., small(-ish) pages



ILLINOIS TECH

College of Computing

Memory Accesses with Pages

```
0x0010: movl 0x1100, %edi  
0x0013: addl $0x3, %edi  
0x0019: movl %edi, 0x1100
```

Assume PT is at phys addr 0x5000

Assume PTE's are 4 bytes

Assume 4KB pages

How many bits for offset? 12

Simplified view
of page table



Old: How many mem refs with segmentation?

5 (3 instrs, 2 movl)

Physical Memory Accesses with Paging?

1) Fetch instruction at logical addr 0x0010; vpn?

- Access page table to get ppn for vpn 0
- Mem ref 1: 0x5000
- Learn vpn 0 is at ppn 2
- Fetch instruction at 0x2010 (Mem ref 2)

Exec, load from logical addr 0x1100; vpn?

- Access page table to get ppn for vpn 1
- Mem ref 3: 0x5004
- Learn vpn 1 is at ppn 0
- Movl from 0x0100 into reg (Mem ref 4)

Pagetable is slow!!! Doubles memory references

Advantages of Paging

No external fragmentation

- Any page can be placed in any frame in physical memory

Fast to allocate and free

- Alloc: No searching for suitable free space
- Free: Doesn't have to coalesce with adjacent free space
- Just use bitmap to show free/allocated page frames

Simple to swap-out portions of memory to disk (later lecture)

- Page size matches disk block size
- Can run process when some pages are on disk
- Add "present" bit to PTE

ILLINOIS TECH

College of Computing

Disadvantages of Paging

Internal fragmentation: Page size may not match size needed by process

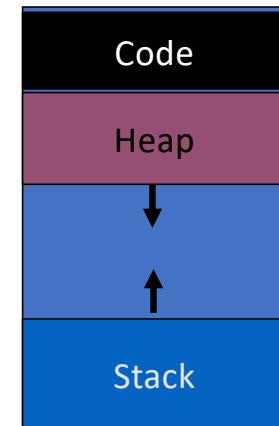
- Wasted memory grows with larger pages
- **Tension?**

Additional memory reference to page table --> Very inefficient

- Page table must be stored in memory
- MMU stores only base address of page table
- Solution: Add TLBs (future lecture)

Storage for page tables may be substantial

- Simple page table: Requires PTE for all pages in address space
 - Entry needed even if page not allocated
- Problematic with dynamic stack and heap within address space
- Page tables must be allocated contiguously in memory
- Solution: Combine paging and segmentation (future lecture)



Questions?

ILLINOIS TECH

College of Computing

Pages, Multilevel Page Tables, and Swapping

Ben Lenard

blenard@iit.edu

ILLINOIS TECH

College of Computing

Agenda

- Homework 3 questions
- Midterm update
- Review Paging
- Multilevel Paging
- Swapping

ILLINOIS TECH

College of Computing

Homework 3

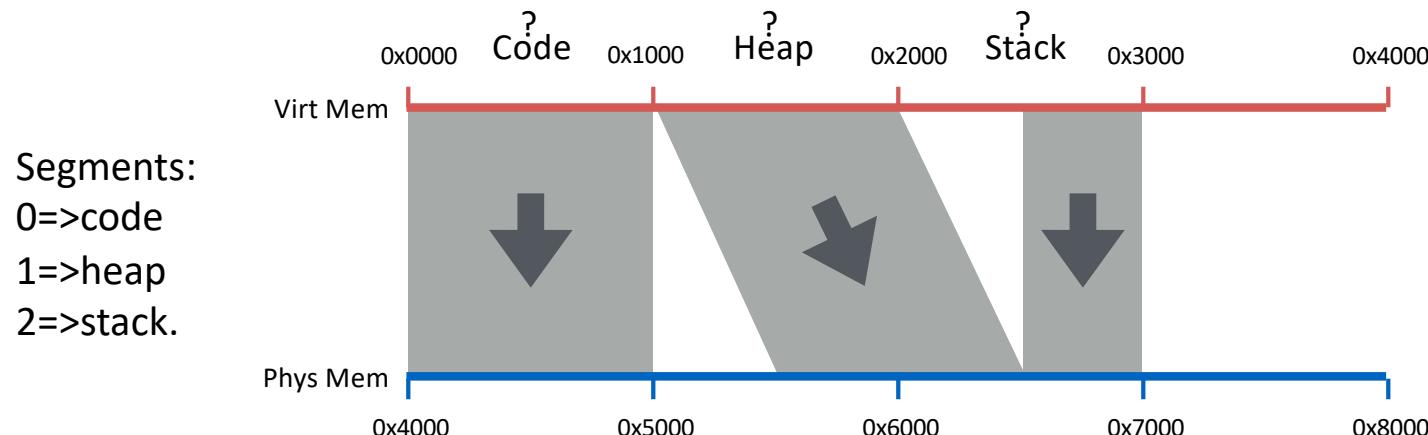
- Questions?

ILLINOIS TECH

College of Computing

Review: Segmentation

Assume 14-bit virtual addresses, high 2 bits indicate segment



Where does segment table live?

All registers, MMU	0x4000	0xffff
	0x5800	0xffff
	0x6800	0x7ff

Review: Memory Accesses

```
0x0010: movl 0x1100, %edi  
0x0013: addl $0x3, %edi  
0x0019: movl %edi, 0x1100
```

%rip: 0x0010

Physical Memory Accesses?

- 1) Fetch instruction at logical addr 0x0010
 - Physical addr: 0x4010Exec, load from logical addr 0x1100
 - Physical addr: 0x5900
- 2) Fetch instruction at logical addr 0x0013
 - Physical addr: 0x4013Exec, no load
- 3) Fetch instruction at logical addr 0x0019
 - Physical addr: 0x4019Exec, store to logical addr 0x1100
 - Physical addr: 0x5900

Total of 5 memory references (3 instruction fetches, 2 movl)

Problem: Fragmentation

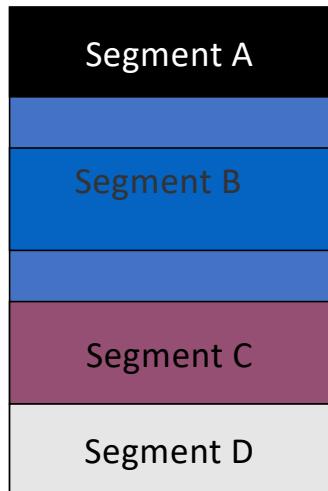
Definition: Free memory that can't be usefully allocated

Why?

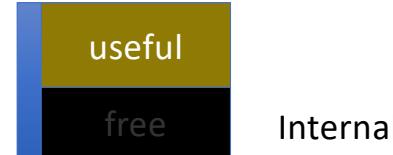
- Free memory (hole) is too small and scattered
- Rules for allocating memory prohibit using this free space

Types of fragmentation

- External: Visible to allocator (e.g., OS)
- Internal: Visible to requester (e.g., if must allocate at some granularity)



No contiguous space!



ILLINOIS TECH

College of Computing

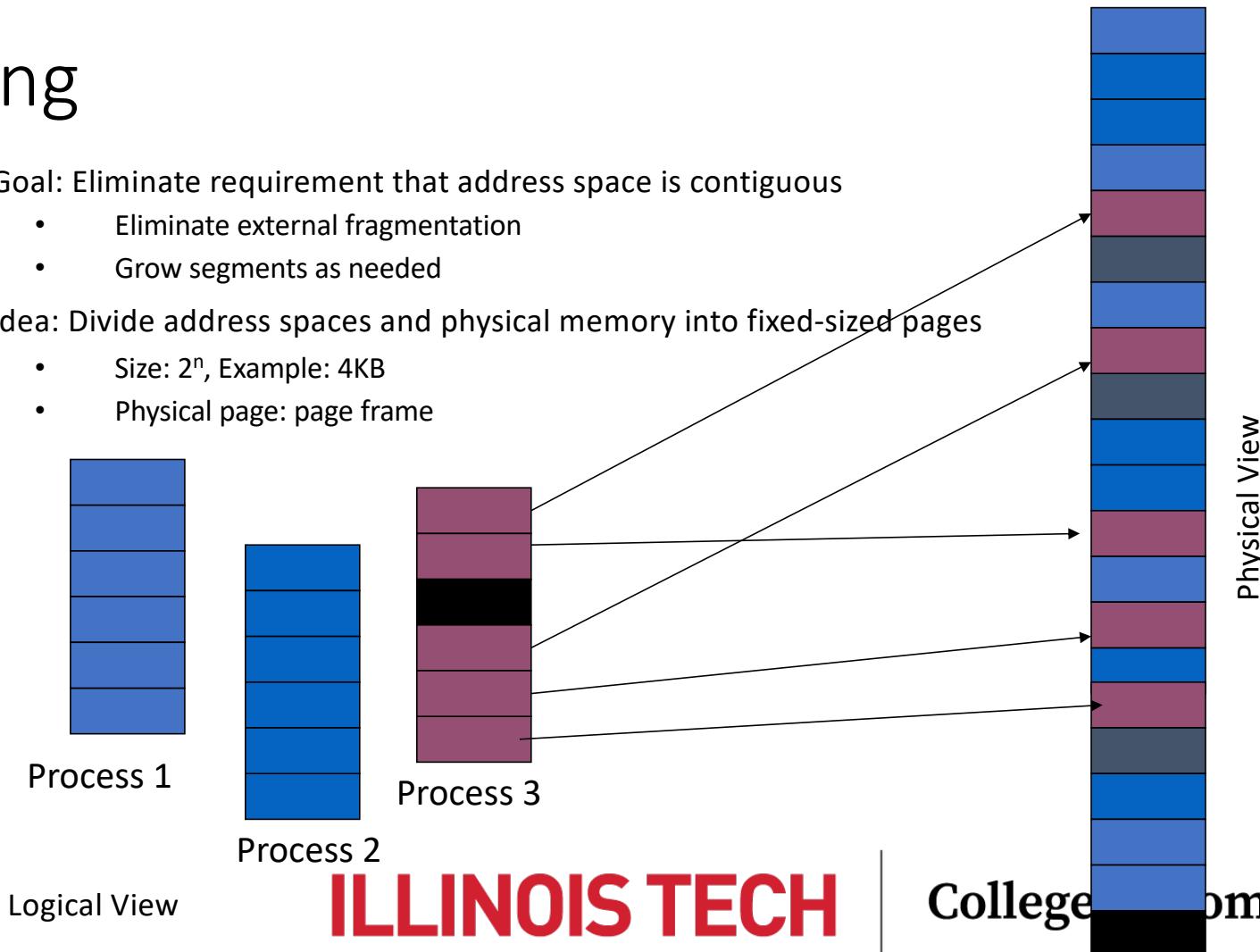
Paging

Goal: Eliminate requirement that address space is contiguous

- Eliminate external fragmentation
- Grow segments as needed

Idea: Divide address spaces and physical memory into fixed-sized pages

- Size: 2^n , Example: 4KB
- Physical page: page frame

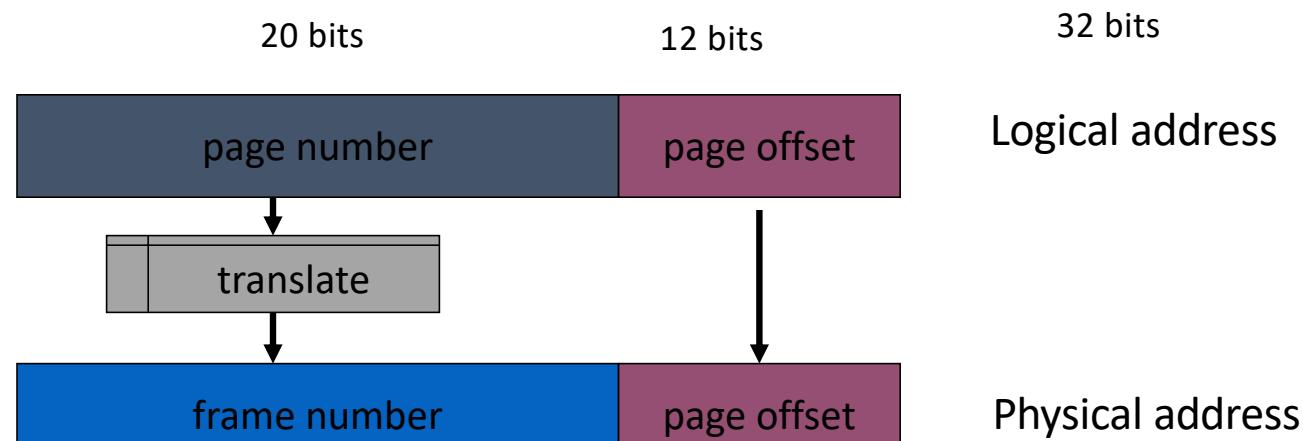


ILLINOIS TECH

College of Computing

Translation of Page Addresses

- How to translate logical address to physical address?
 - High-order bits of address designate page number
 - Low-order bits of address designate offset within page



No addition needed; just append bits correctly...

How does format of address space determine number of pages and size of page?

ILLINOISTECH

College of Computing

Where Are Pagetables Stored?

How big is a typical page table?

- assume **32-bit** address space
- assume 4 KB pages
- assume 4 byte entries

Final answer: $2^{(32 - \log(4KB))} * 4 = 4 \text{ MB}$

- Page table size = Num entries * size of each entry
- Num entries = num virtual pages = $2^{\text{bits for vpn}}$
- Bits for vpn = 32 – number of bits for page offset
 $= 32 - \lg(4KB) = 32 - 12 = 20$
- Num entries = $2^{20} = 1 \text{ MB}$
- Page table size = Num entries * 4 bytes = 4 MB

Implication: Store each page table in memory

- Hardware finds page table base with register (e.g., CR3 on x86)

What happens on a context-switch?

- Change contents of page table base register to newly scheduled process
- Save old page table base register in PCB of descheduled process

Other PT info

What other info is in pagetable entries besides translation?

- valid bit
- protection bits
- present bit (needed later)
- reference bit (needed later)
- dirty bit (needed later)

Pagetable entries are just bits stored in memory

- Agreement between hw and OS about interpretation

Advantages of Paging

No external fragmentation

- Any page can be placed in any frame in physical memory

Fast to allocate and free

- Alloc: No searching for suitable free space
- Free: Doesn't have to coalesce with adjacent free space
- Just use bitmap to show free/allocated page frames

Simple to swap-out portions of memory to disk (later lecture)

- Page size matches disk block size
- Can run process when some pages are on disk
- Add "present" bit to PTE

ILLINOIS TECH

College of Computing

Disadvantages of Paging

Internal fragmentation: Page size may not match size needed by process

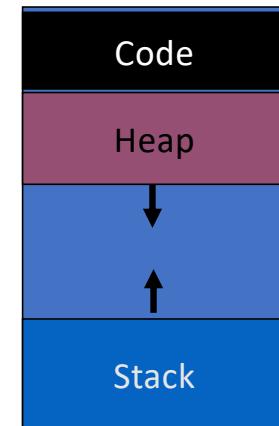
- Wasted memory grows with larger pages
- **Tension?**

Additional memory reference to page table --> Very inefficient

- Page table must be stored in memory
- MMU stores only base address of page table
- Solution: Add TLBs (future lecture)

Storage for page tables may be substantial

- Simple page table: Requires PTE for all pages in address space
 - Entry needed even if page not allocated
- Problematic with dynamic stack and heap within address space
- Page tables must be allocated contiguously in memory
- Solution: Combine paging and segmentation (future lecture)



ILLINOIS TECH

College of Computing

Translation Steps

H/W: for each mem reference:

1. extract **VPN** (virt page num) from **VA** (virt addr)
(cheap)
2. calculate addr of **PTE** (page table entry)
(cheap)
3. read **PTE** from memory
(expensive)
4. extract **PFN** (page frame num)
(cheap)
5. build **PA** (phys addr)
(cheap)
6. read contents of **PA** from memory into register
(expensive)

Which steps are expensive?

Which expensive step will we avoid in today's lecture?

- 3) Don't always have to read PTE from memory!

Example: Array Iterator

	What virtual addresses?	What physical addresses?
int sum = 0;		
for (i=0; i<N; i++) {	load 0x3000	load 0x100C
sum += a[i];	load 0x3004	load 0x7000
}		load 0x100C
Assume 'a' starts at 0x3000	load 0x3008	load 0x7004
Ignore instruction fetches	load 0x300C	load 0x100C
	...	load 0x7008
		load 0x100C
		load 0x700C

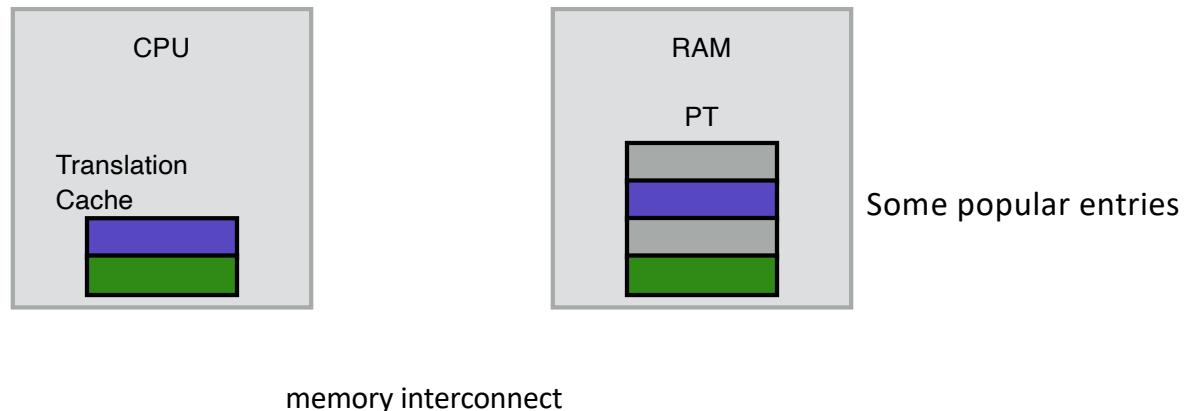
Aside: What can you infer?

- ptbr: 0x1000; PTE 4 bytes each
- VPN 3 -> PPN 7

Observation:

Repeatedly access same PTE because program repeatedly
accesses same virtual page

Strategy: Cache Page Translations



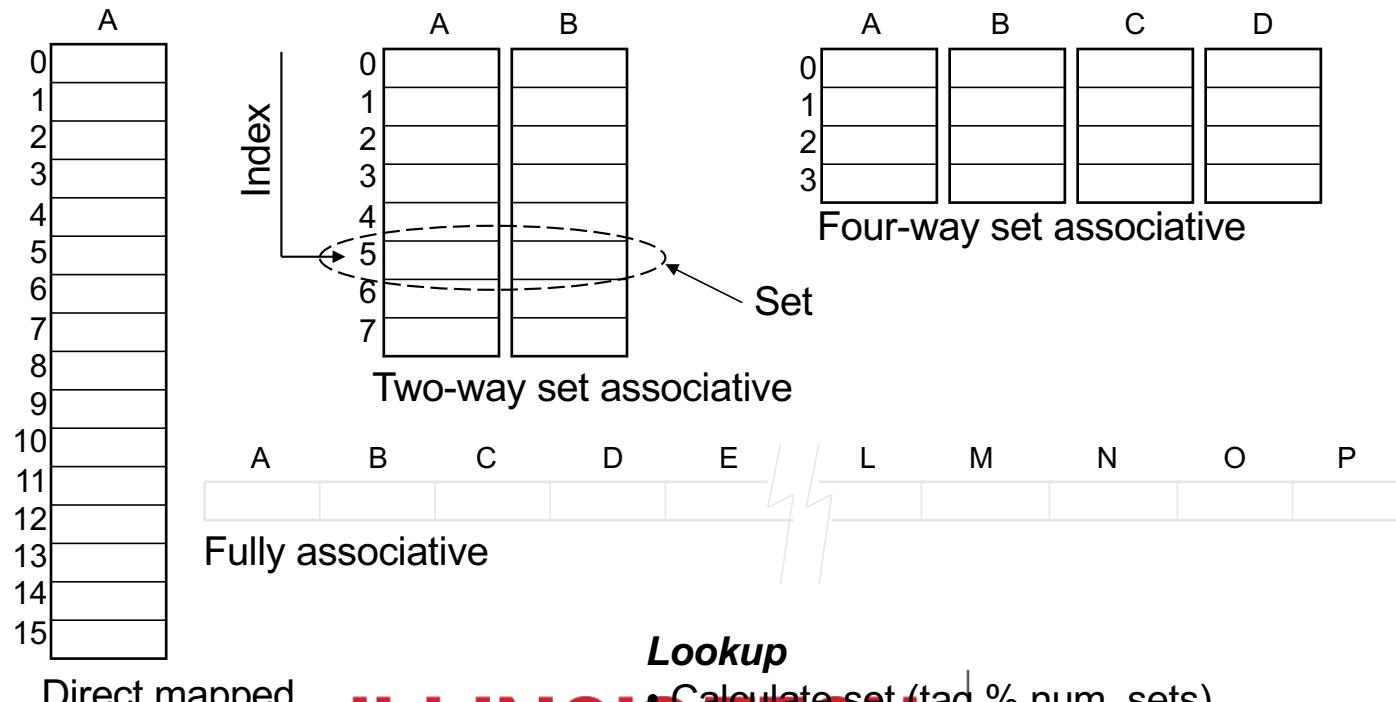
TLB: Translation Lookaside Buffer
(yes, a poor name!)

TLB Organization

TLB Entry

Tag (virtual page number)	Physical page number (page table entry)
---------------------------	---

Various ways to organize a 16-entry TLB (artificially small)



TLB Associativity Trade-offs

Higher associativity

- + Better utilization, fewer collisions
- Slower
- More hardware

Lower associativity

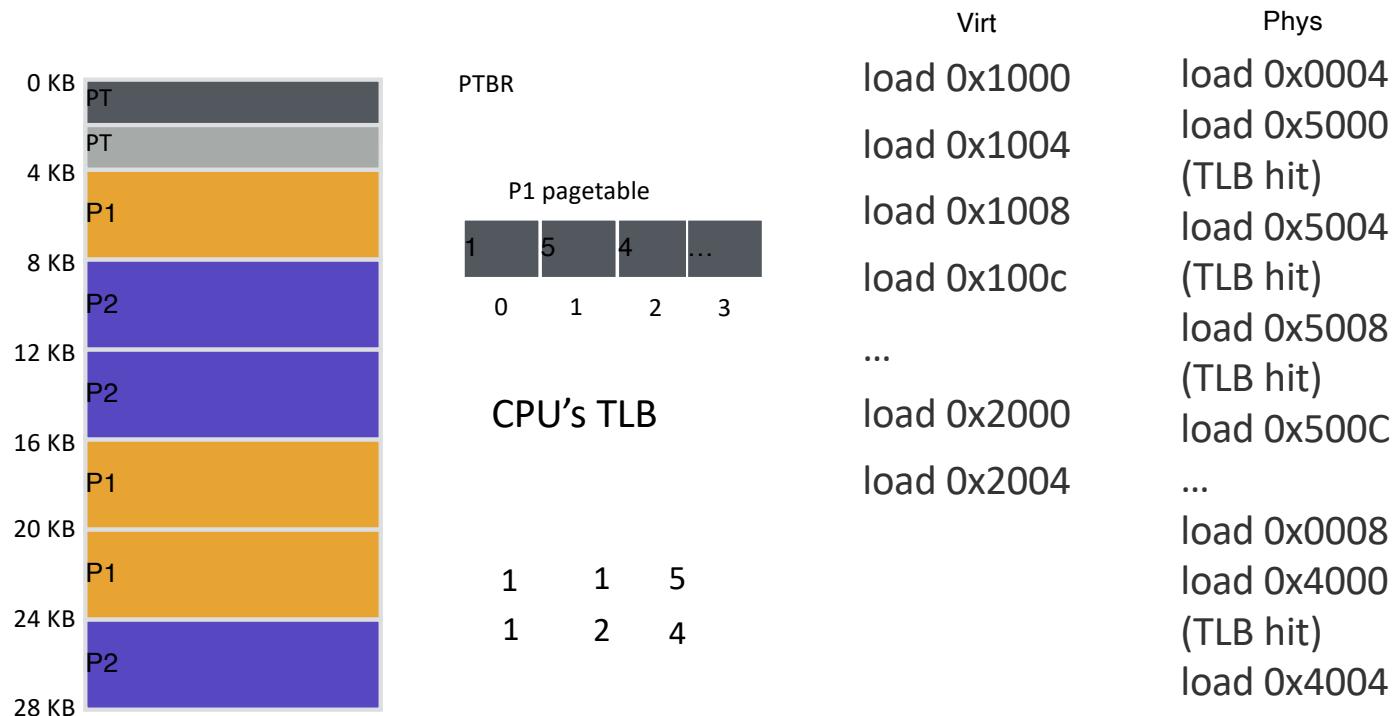
- + Fast
- + Simple, less hardware
- Greater chance of collisions

TLBs usually fully associative

ILLINOIS TECH

College of Computing

TLB Accesses: SEQUENTIAL Example



PERFORMANCE OF TLB?

Calculate miss rate of TLB for data:

TLB misses / # TLB lookups

TLB lookups?

= number of accesses to a = 2048

```
int sum = 0;
```

```
for (i=0; i<2048; i++) {  
    sum += a[i];  
}
```

TLB misses?

= number of unique pages accessed

= 2048 / (elements of 'a' per 4K page)

= 2K / (4K / sizeof(int)) = 2K / 1K

= 2

Miss rate?

2/2048 = 0.1%

Hit rate? (1 – miss rate)

99.9%

Would hit rate get better or worse with smaller pages?

Worse

TLB PERFORMANCE

How can system improve TLB performance (hit rate) given fixed number of TLB entries?

Increase page size

Fewer unique page translations needed to access same amount of memory

TLB Reach:

Number of TLB entries * Page Size

ILLINOIS TECH

College of Computing

TLB PERFORMANCE with Workloads

Sequential array accesses almost always hit in TLB

- Very fast!

What access pattern will be slow?

- Highly random, with no repeat accesses

Workload acCESS PATTERNS

Workload A

```
int sum = 0;  
  
for (i=0; i<2048; i++) {  
    sum += a[i];  
}
```

Workload B

```
int sum = 0;  
srand(1234);  
for (i=0; i<1000; i++) {  
    sum += a[rand() % N];  
}  
srand(1234);  
for (i=0; i<1000; i++) {  
    sum += a[rand() % N];  
}
```

Spatial Locality: future access will be to nearby addresses

Temporal Locality: future access will be repeats to the same data

What TLB characteristics are best for each type?

Spatial:

- Access same page repeatedly; need same vpn->ppn translation
- Same TLB entry re-used

Temporal:

- Access same address near in future
- Same TLB entry re-used in near future
- How near in future? How many TLB entries are there?

TLB Replacement policies

LRU: evict Least-Recently Used TLB slot when needed

(More on LRU later in policies next week)

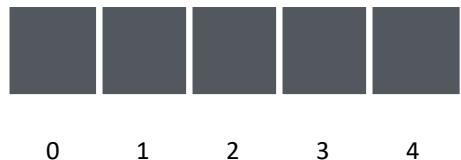
Random: Evict randomly chosen entry

Which is better?



LRU Troubles

virtual addresses:



Workload repeatedly accesses same offset across 5 pages (strided access),
but only 4 TLB entries

What will TLB contents be over time?

How will TLB perform?

TLB Replacement policies

LRU: evict Least-Recently Used TLB slot when needed

(More on LRU later in policies next week)

Random: Evict randomly chosen entry

Sometimes random is better than a “smart” policy!

TLB PERFORMANCE

How can system improve TLB performance (hit rate) given fixed number of TLB entries?

Increase page size

Fewer unique translations needed to access same amount of memory)

ILLINOIS TECH

College of Computing

Context Switches

What happens if a process uses cached TLB entries from another process?

Solutions?

1. Flush TLB on each switch
 - Costly; lose all recently cached translations
2. Track which entries are for which process
 - Address Space Identifier
 - Tag each TLB entry with an 8-bit ASID
 - how many ASIDs do we get?
 - why not use PIDs?

TLB Example with ASID



TLB Performance

Context switches are expensive

Even with ASID, other processes “pollute” TLB

- Discard process A's TLB entries for process B's entries

Architectures can have multiple TLBs

- 1 TLB for data, 1 TLB for instructions
- 1 TLB for regular pages, 1 TLB for “super pages”

HW and OS Roles

Who Handles TLB MISS? **H/W** or **OS**?

H/W: CPU must know where pagetables are

- CR3 register on x86
- Pagetable structure fixed and agreed upon between HW and OS
- HW “walks” the pagetable and fills TLB

OS: CPU traps into OS upon TLB miss

- “Software-managed TLB”
- OS interprets pagetables as it chooses
- Modifying TLB entries is privileged
 - otherwise what could process do?

Need same protection bits in TLB as pagetable

- rwx

Disadvantages of Paging

1. Additional memory reference to look up in page table
 - Very inefficient
 - Page table must be stored in memory
 - MMU stores only base address of page table
 - **Avoid extra memory reference for lookup with TLBs (previous lecture)**
2. Storage for page tables may be substantial
 - Simple page table: Requires PTE for all pages in address space
 - Entry needed even if page not allocated
 - Problematic with dynamic stack and heap within address space (today)

QUIZ: How big are page Tables?

1. PTE's are **2 bytes**, and **32** possible virtual page numbers

$$32 * 2 \text{ bytes} = 64 \text{ bytes}$$

2. PTE's are **2 bytes**, virtual addrs are **24 bits**, pages are **16 bytes**

$$2 \text{ bytes} * 2^{(24 - \lg 16)} = 2^{21} \text{ bytes (2 MB)}$$

3. PTE's are **4 bytes**, virtual addrs are **32 bits**, and pages are **4 KB**

$$4 \text{ bytes} * 2^{(32 - \lg 4K)} = 2^{22} \text{ bytes (2 MB)}$$

4. PTE's are **4 bytes**, virtual addrs are **64 bits**, and pages are **4 KB**

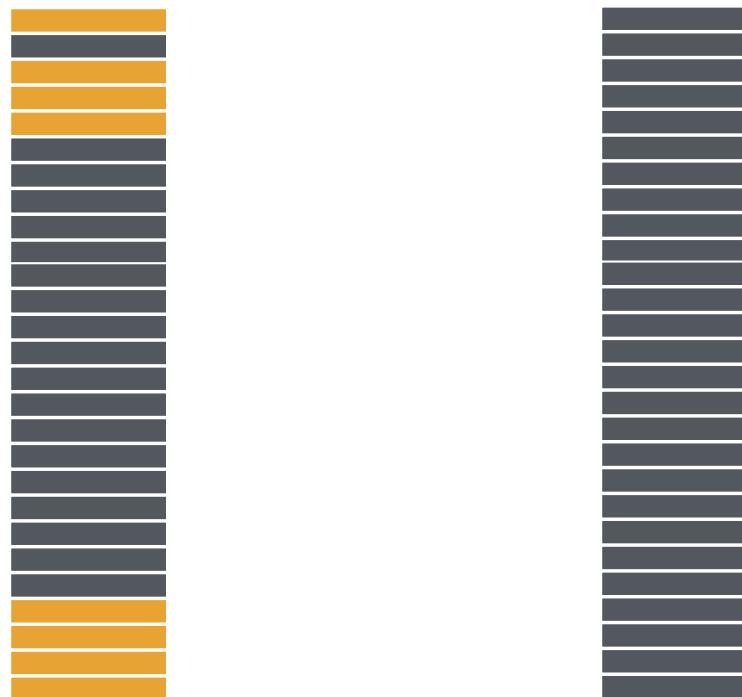
$$4 \text{ bytes} * 2^{(64 - \lg 4K)} = 2^{54} \text{ bytes}$$

How big is each page table?

ILLINOIS TECH

College of Computing

Why ARE Page Tables so Large?



Large address spaces = Large tables

- On 64-bit systems, virtual address spaces are up to 2^{48} bytes in size
 - Given 4KB page size and 8 byte page table entries
 - Page table size = $(2^{48} \div 2^{12}$ pages) $\times 2^3$ bytes/page
 - = 2^{39} bytes = **512GB**
- Most of the address space will be unmapped — i.e., the page table is a very *sparse* data structure
- How to reduce the size of page tables (without increasing page size)?

Avoid simple linear Page Table

Use more complex page tables, instead of just big array

Any data structure is possible with software-managed TLB

- Hardware looks for vpn in TLB on every memory access
- If TLB does not contain vpn, TLB miss
 - Trap into OS and let OS find vpn->ppn translation
 - OS notifies TLB of vpn->ppn for future accesses

Let's reduce that size

ILLINOIS TECH

College of Computing

Approach 1: Inverted Page Table

Inverted Page Tables

- Only need entries for virtual pages w/ valid physical mappings

Naïve approach:

Search through data structure <ppn, vpn+asid> to find match

- Too much time to search entire table

Better: Find possible matches entries by hashing vpn+asid

- Smaller number of entries to search for exact match

Managing inverted page table requires software-controlled TLB

For hardware-controlled TLB, need well-defined, simple approach

Other Approaches

1. Inverted Pagetables
2. Segmented Pagetables
3. Multi-level Pagetables
 - Page the page tables
 - Page the pagetables of page tables...

valid Ptes are Contiguous

how to avoid
storing these?

PFN	valid	prot	
-		rx	Note “hole” in addr space:
23	1		valids vs. invalids are clustered
-		rw-	
-		0	
-		0	
-		0	
		-	How did OS avoid allocating holes in phys memory?
	...many more invalid...	0	
		0	Segmentation
		0	
		0	
28	1		
4		rw-	
		rw-	

Combine Paging and Segmentation

Divide address space into segments (code, heap, stack)

- Segments can be variable length

Divide each segment into fixed-sized pages

Logical address divided into three portions



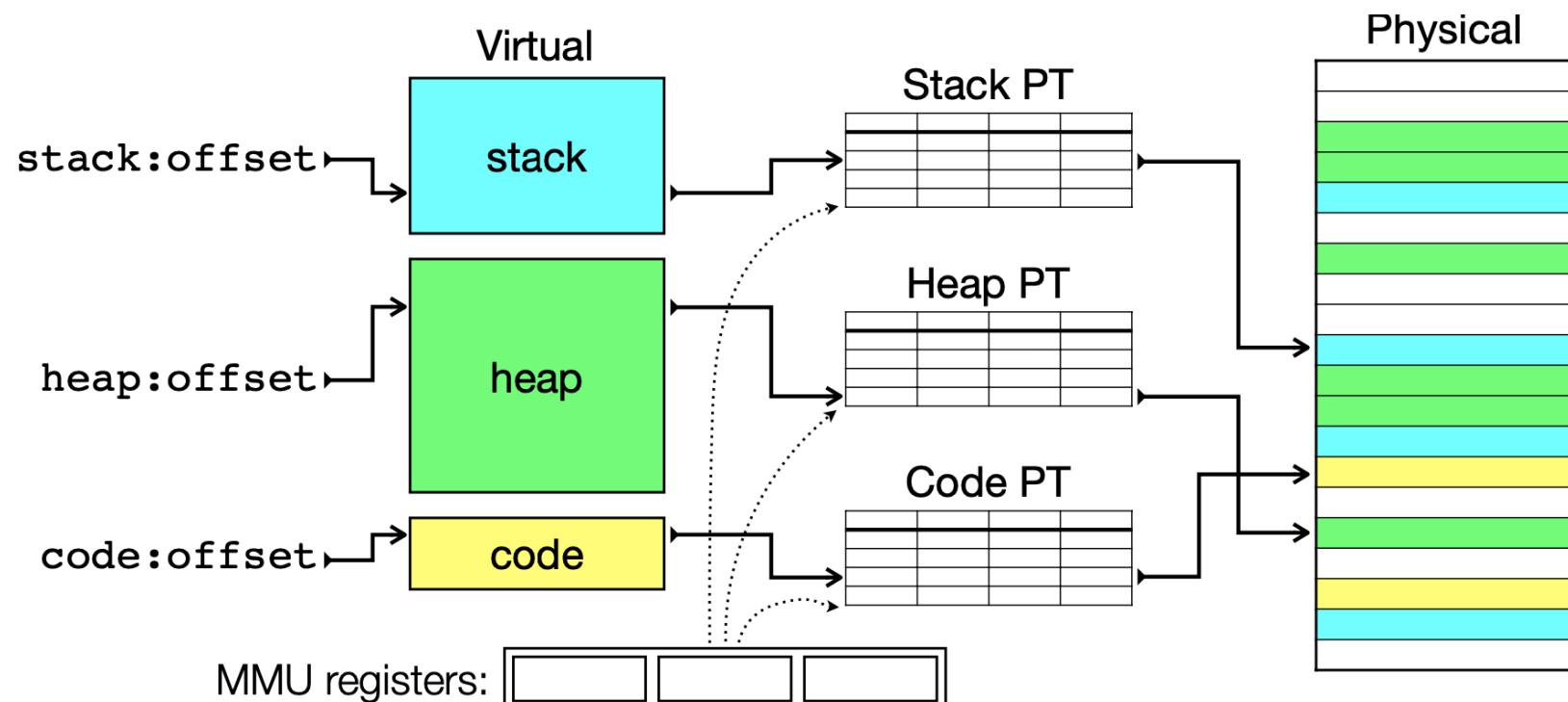
Implementation

- Each segment has a page table
- Each segment tracks base (physical address) and bounds of **page table** for that segment

ILLINOIS TECH

College of Computing

Segmentation + Paging



ILLINOIS TECH

College of Computing

Advantages of Paging and Segmentation

Advantages of Segments

- Supports sparse address spaces
 - Decreases size of page tables
 - If segment not used, not need for page table

Advantages of Pages

- No external fragmentation
- Segments can grow without any reshuffling
- Can run process when some pages are swapped to disk (next lecture)

Advantages of Both

- Increases flexibility of sharing
 - Share either single page or entire segment
 - How?

ILLINOIS TECH

College of Computing

Disadvantages of Paging and Segmentation

Potentially large page tables (for each segment)

- Must allocate each page table contiguously
- More problematic with more address bits
- Page table size?
 - Assume 2 bits for segment, 18 bits for page number, 12 bits for offset

Each page table is:

$$\begin{aligned} &= \text{Number of entries} * \text{size of each entry} \\ &= \text{Number of pages} * 4 \text{ bytes} \\ &= 2^{18} * 4 \text{ bytes} = 2^{20} \text{ bytes} = 1 \text{ MB!!!} \end{aligned}$$

ILLINOIS TECH

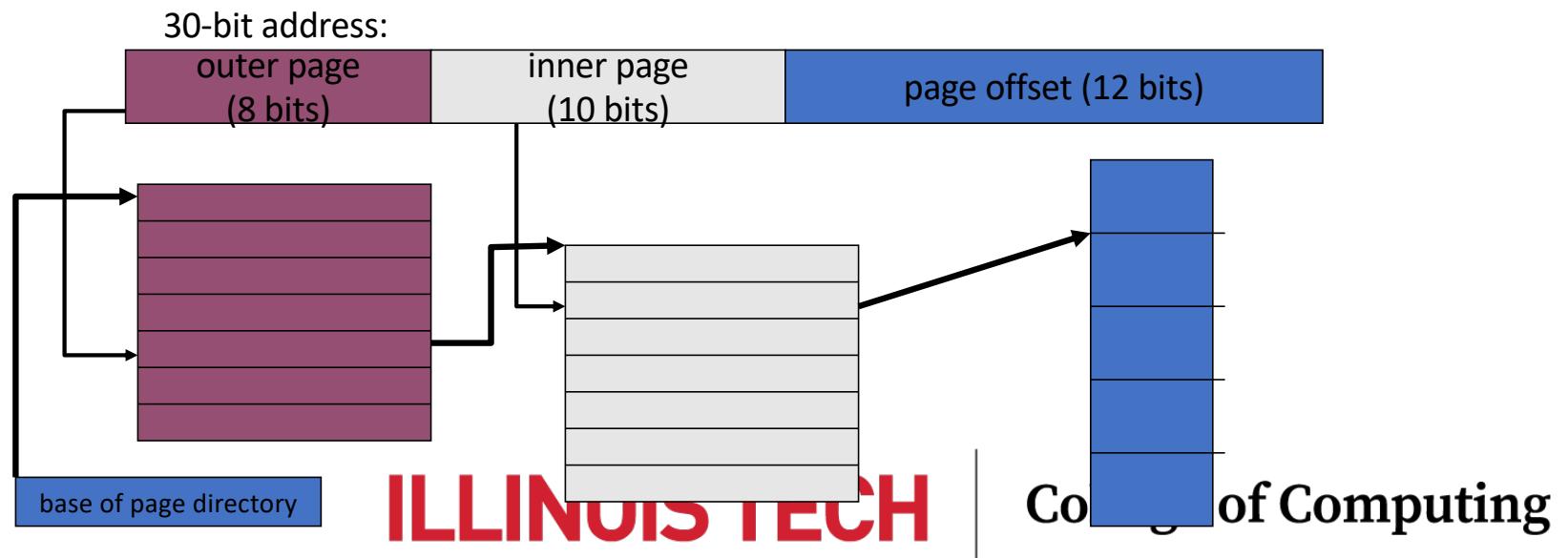
College of Computing

Multilevel Page Tables

Goal: Allow each page tables to be allocated non-contiguously

Idea: Page the page tables

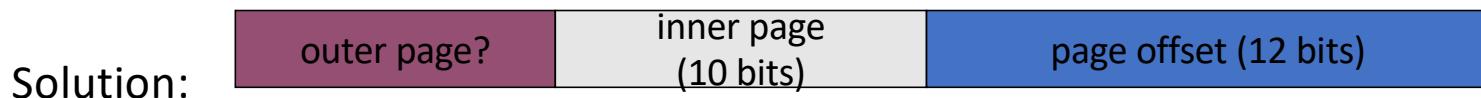
- Creates multiple levels of page tables; outer level “page directory”
- Only allocate page tables for pages in use
- Used in x86 architectures (hardware can walk known structure)



Problem with 2 levels?

Problem: page directories (outer level) may not fit in a page

64-bit address:



- Split page directories into pieces
- Use another page dir to refer to the page dir pieces.



How large is virtual address space with 4 KB pages, 4 byte PTEs,
each page table fits in page given 1, 2, 3 levels?

4KB / 4 bytes → 1K entries per level

1 level: $1K * 4K = 2^{12} = 4 \text{ MB}$

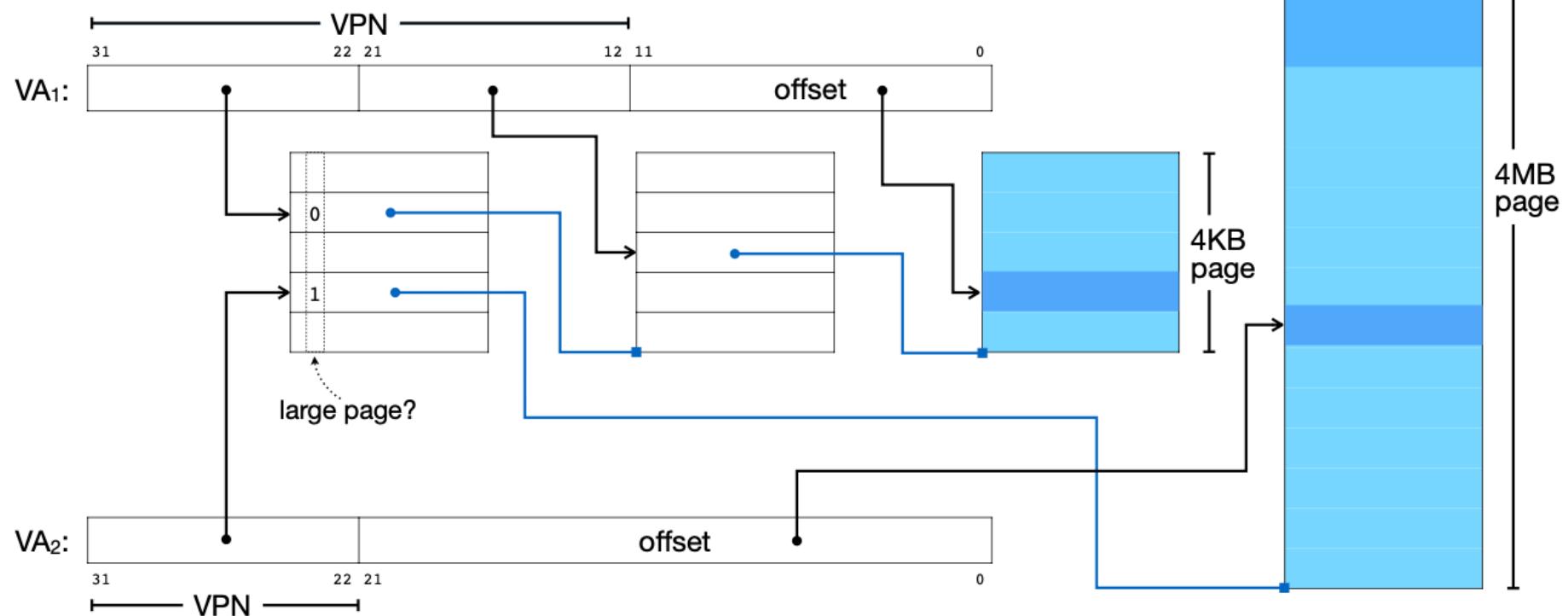
2 levels: $1K * 1K * 4K = 2^{12} \approx 4 \text{ GB}$

3 levels: $1K * 1K * 1K * 4K = 2^{18} \approx 4 \text{ TB}$

Multiple (fixed) page sizes

- Multi-level paging makes it possible to accommodate multiple page sizes
- Each level of paging partitions physical memory into smaller pieces (intuitively, fewer bits lefts over for offset field)
- “Skipping” one or more levels results in mapping larger pages from virtual to physical space
- Mapping large pages may greatly improve TLB effectiveness

E.g., 4KB and 4MB pages



Multi-level page table pros & cons

Pro's

- May reduce page table footprint
- Allocate levels as needed
- Multiple page sizes may coexist
- Large pages help TLB while reducing PT size

Con's

- Page table walk is expensive!
- Requires multiple memory accesses for translation
- More complex to access/manage
- Kernel must maintain PT data structures for each process

Summary: Better PAGE TABLES

Problem:

Simple linear page tables require too much contiguous memory

Many options for efficiently organizing page tables

If OS traps on TLB miss, OS can use any data structure

- Inverted page tables (hashing)

If Hardware handles TLB miss, page tables must follow specific format

- Multi-level page tables used in x86 architecture
- Each page table fits within a page

Next Topic:

What if desired address spaces do not fit in physical memory?

.

Swapping Motivation

OS goal: Support processes when not enough physical memory

- Single process with very large address space
- Multiple processes with combined address spaces

User code should be independent of amount of physical memory

- Correctness, if not performance

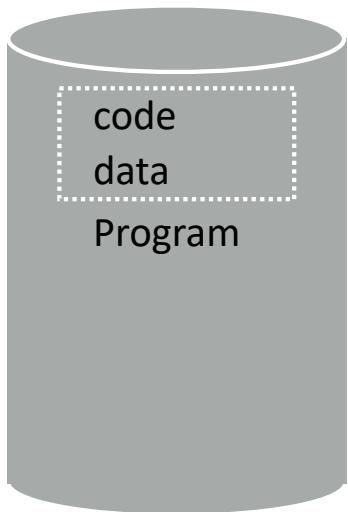
Virtual memory: OS provides illusion of more physical memory

Why does this work?

- Relies on key properties of user processes (workload) and machine architecture (hardware)

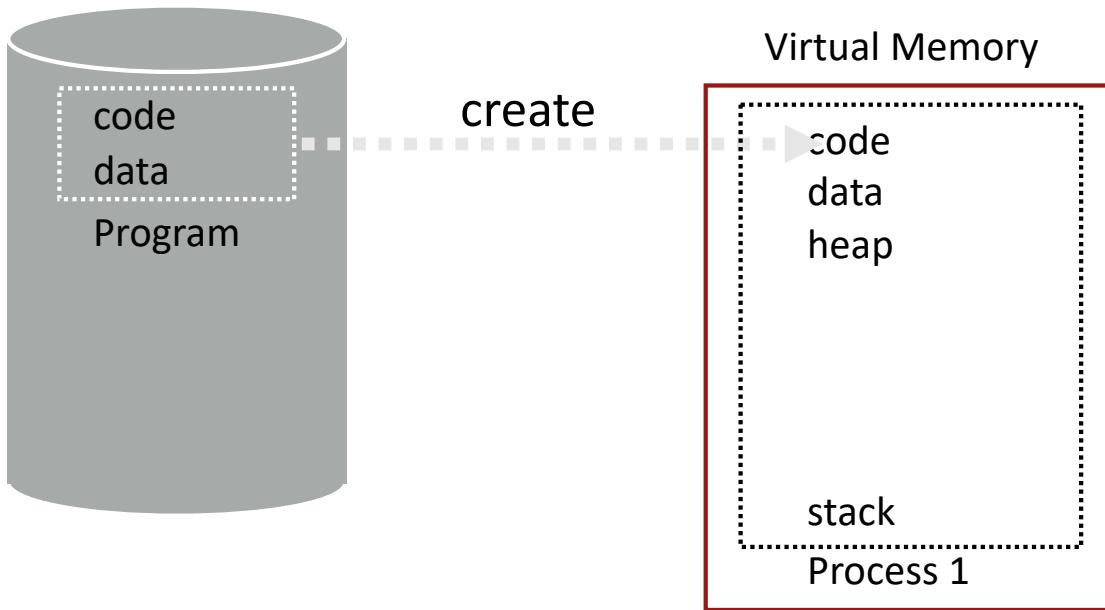
ILLINOIS TECH

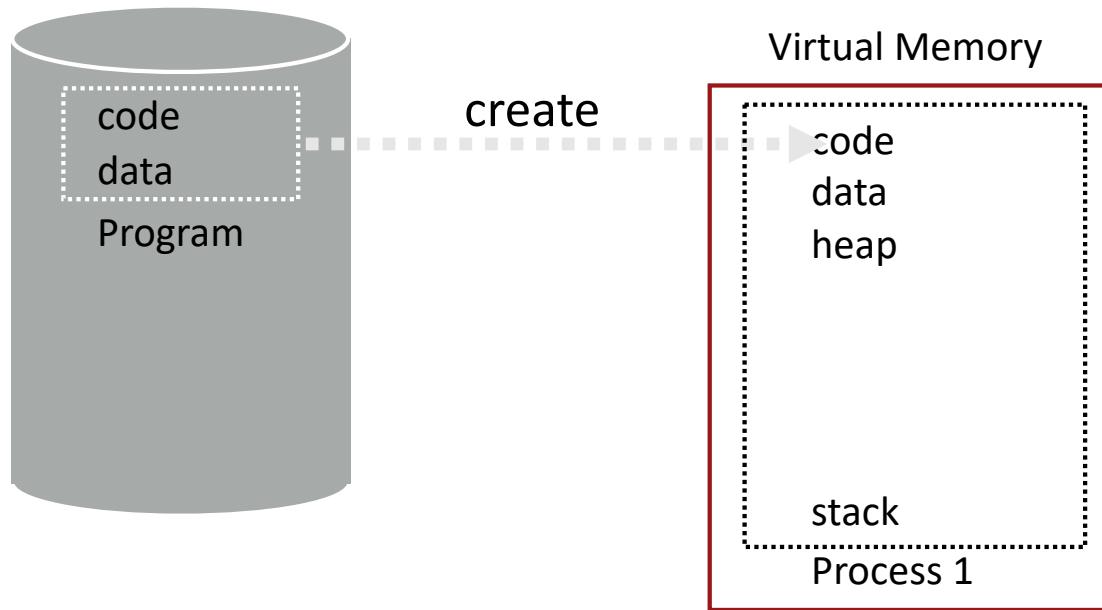
College of Computing



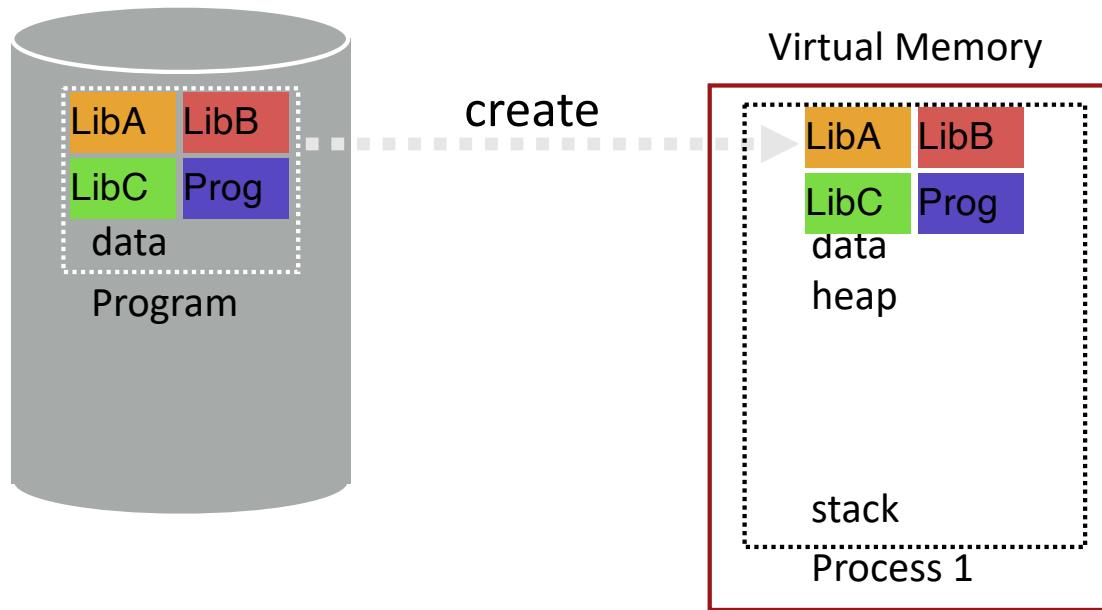
Virtual Memory





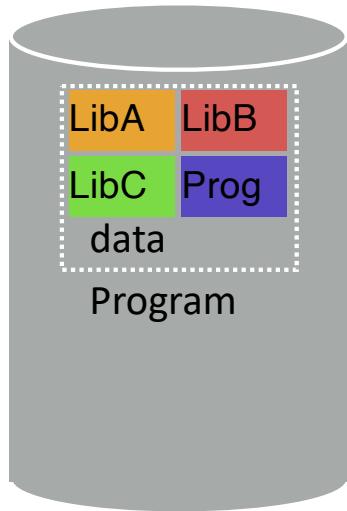


what's in code?



many large libraries, some
of which are rarely/never used

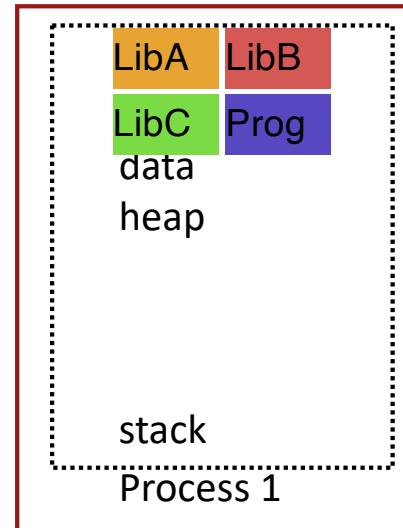
How to avoid wasting physical pages to back rarely used virtual pages?

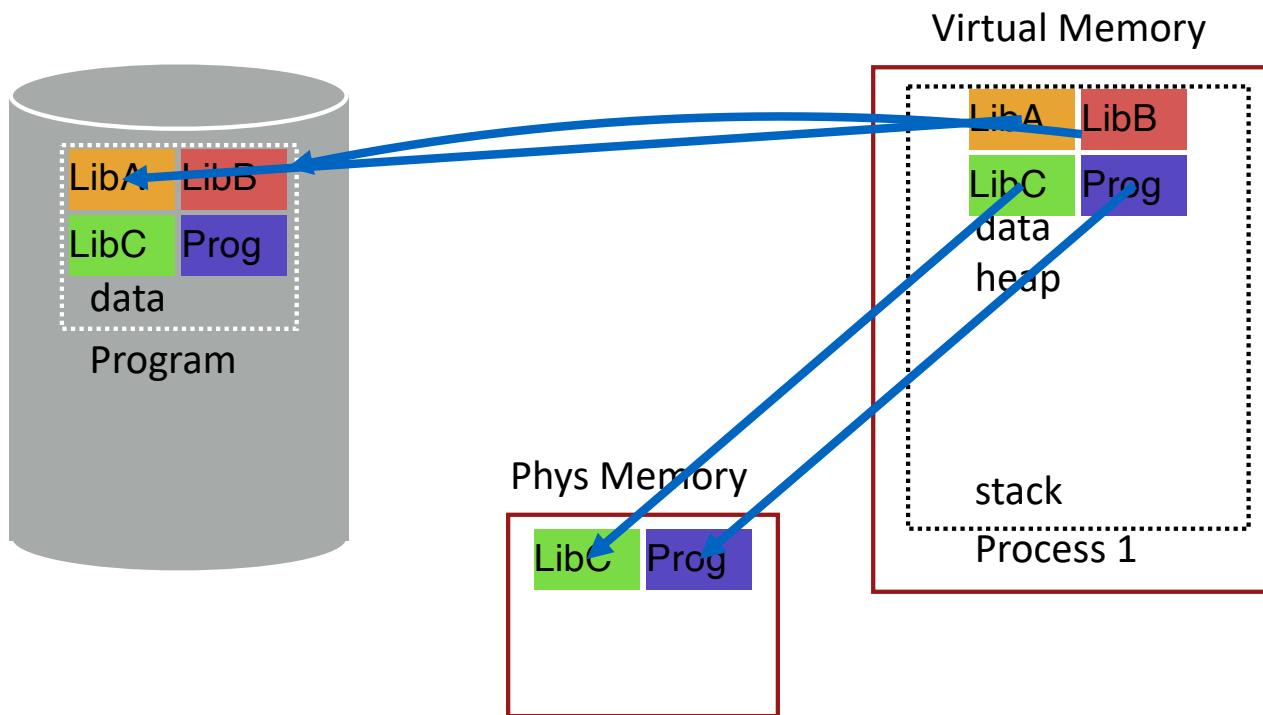


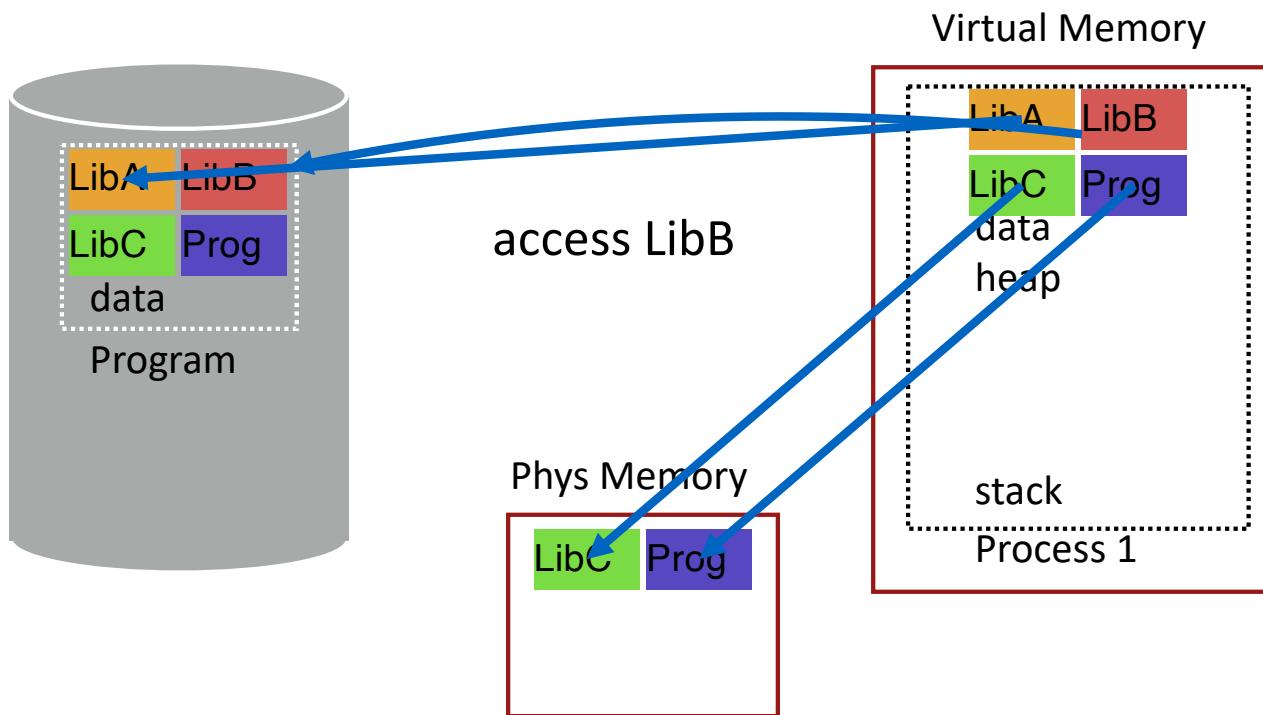
Phys Memory

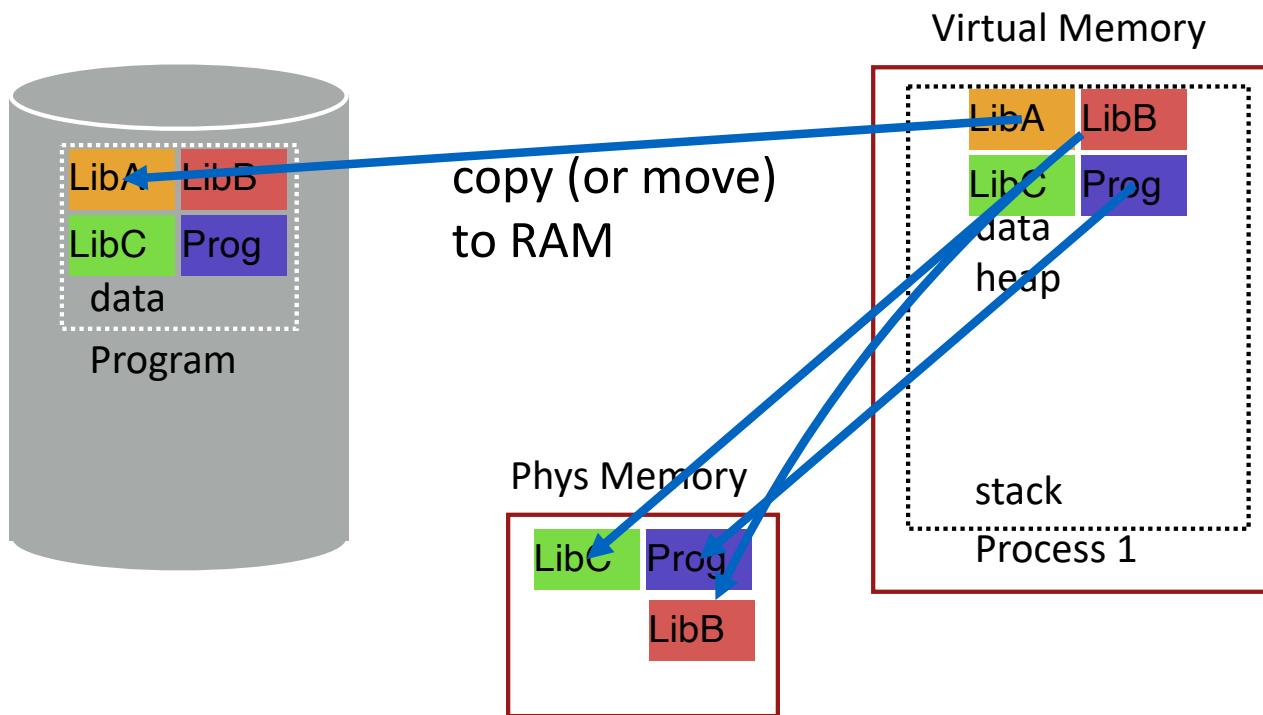


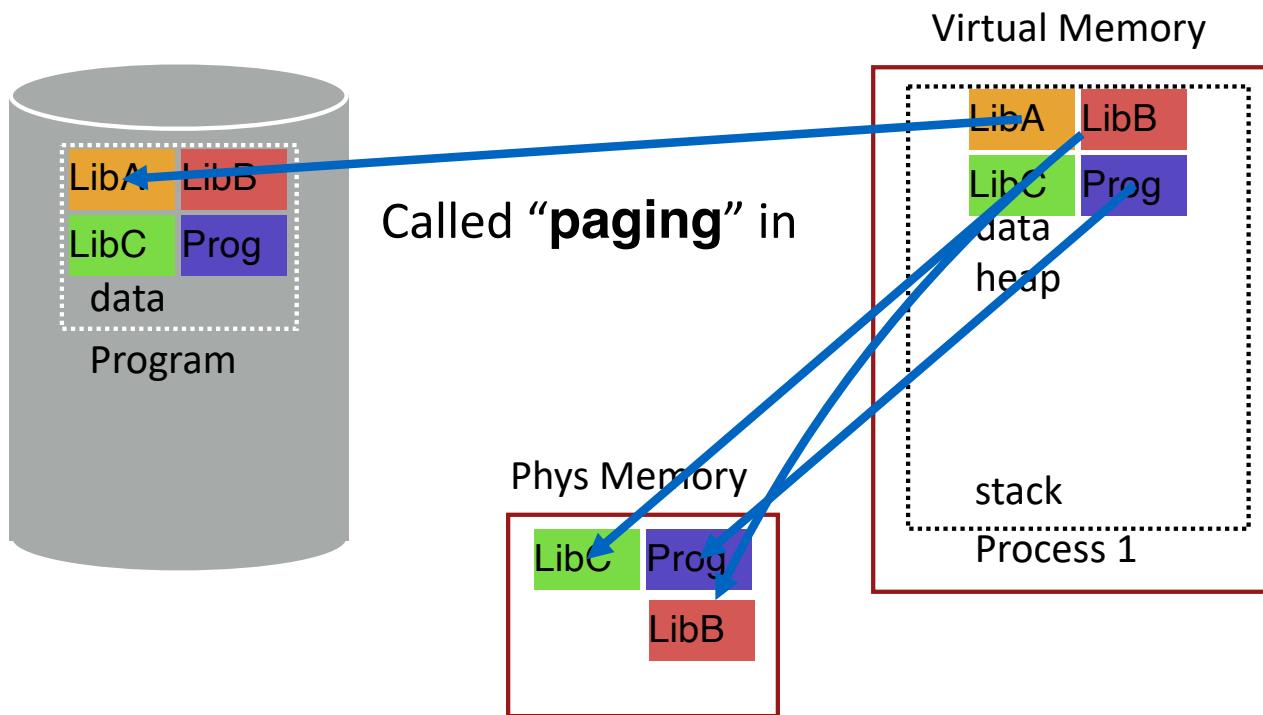
Virtual Memory











Locality of Reference

Leverage **locality of reference** within processes

- **Spatial:** reference memory addresses **near** previously referenced addresses
- **Temporal:** reference memory addresses that have referenced in the past
- Processes spend majority of time in small portion of code
 - Estimate: 90% of time in 10% of code

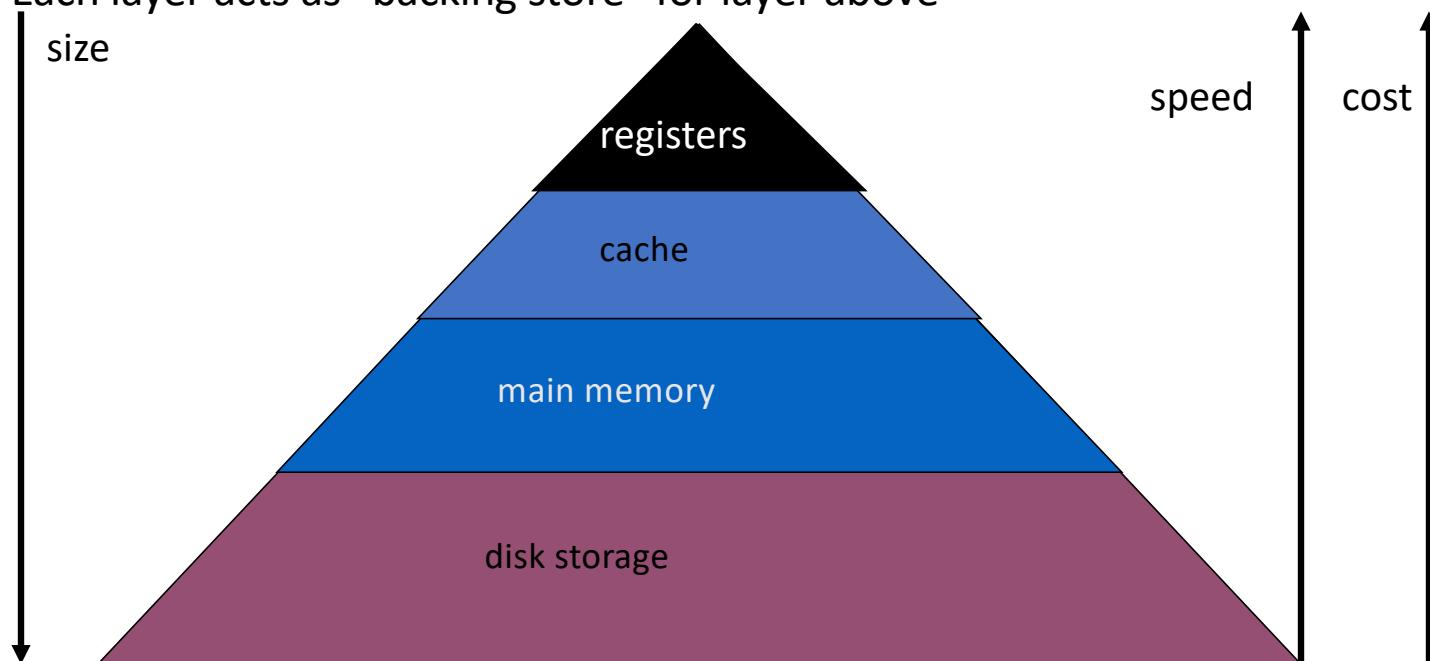
Implication:

- Process only uses small amount of address space at any moment
- Only small amount of address space must be resident in physical memory

Memory Hierarchy

Leverage **memory hierarchy** of machine architecture

Each layer acts as “backing store” for layer above



ILLINOIS TECH

College of Computing

Manual vs. Automatic swapping

- Option 1: user decides what to keep in memory and what resides on disk - Swapping is a manual task
 - Most control, but painful for any non-trivial application!
- Option 2: kernel automatically swaps data into and out of memory as needed by processes
 - Users can ignore physical memory constraints (to an extent)
 - Common approach: use pages as the unit of swapping

Virtual Memory Intuition

Idea: OS keeps unreferenced pages on disk

- Slower, cheaper backing store than memory

Process can run when not all pages are loaded into main memory

OS and hardware cooperate to provide illusion of large disk as fast as main memory

- Same behavior as if all of address space in main memory
- Hopefully have similar performance

Requirements:

- OS must have **mechanism** to identify location of each page in address space → in memory or on disk
- OS must have **policy** for determining which pages live in memory and which on disk

ILLINOIS TECH

College of Computing

Virtual Address Space Mechanisms

Each page in virtual address space maps to one of three locations:

- Physical main memory: Small, fast, expensive
- Disk (backing store): Large, slow, cheap
- Nothing (error): Free

Extend page tables with an extra bit: present

- permissions (r/w), valid, present
- Page in memory: present bit set in PTE
- Page on disk: present bit cleared
 - PTE points to block on disk
 - Causes trap into OS when page is referenced

Virtual Memory Mechanisms

Hardware and OS cooperate to translate addresses

First, hardware checks TLB for virtual address

- if TLB hit, address translation is done; page in physical memory

If ...

- Hardware or OS walk page tables
- If PTE designates page is present, then page in physical memory

If (i.e., present bit is cleared)

- Trap into OS (not handled by hardware)
- OS selects victim page in memory to replace
 - Write victim page out to disk if modified (add dirty bit to PTE)
- OS reads referenced page from disk into memory
- Page table is updated, present bit is set
- Process continues execution

What should scheduler do?

ILLINOIS TECH

College of Computing

Mechanism for Continuing a Process

Continuing a process after a page fault is tricky

- Want page fault to be transparent to user
- Page fault may have occurred in middle of instruction
 - When instruction is being fetched
 - When data is being loaded or stored
- Requires hardware support
 - *precise interrupts*: stop CPU pipeline such that instructions before faulting instruction have completed, and those after can be restarted

Complexity depends upon instruction set

- Can faulting instruction be restarted from beginning?
 - Example: `move + (SP), R2`
 - Must track side effects so hardware can undo

Virtual Memory Policies

Goal: Minimize number of page faults

- Page faults require milliseconds to handle (reading from disk)
- Implication: Plenty of time for OS to make good decision

OS has two decisions

- Page selection
 - **When** should a page (or pages) on disk be **brought into** memory?
- Page replacement
 - **Which** resident page (or pages) in memory should be **thrown out** to disk?

Page Selection

When should a page be brought from disk into memory?

Demand paging: Load page only when page fault occurs

- Intuition: Wait until page must absolutely be in memory
- When process starts: No pages are loaded in memory
- Problems: Pay cost of page fault for every newly accessed page

Prepaging (anticipatory, prefetching): Load page before referenced

- OS predicts future accesses (*oracle*) and brings pages into memory early
- Works well for some access patterns (e.g., sequential)
- Problems?

Hints: Combine above with user-supplied hints about page references

- User specifies: may need page in future, don't need this page anymore, or sequential access pattern, ...
- Example: `madvise()` in Unix

Page Replacement

Which page in main memory should selected as victim?

- Write out victim page to disk if modified (dirty bit set)
- If victim page is not modified (clean), just discard

OPT: Replace page not used for longest time in future

- Advantages: Guaranteed to minimize number of page faults
- Disadvantages: Requires that OS predict the future; Not practical, but good for comparison

FIFO: Replace page that has been in memory the longest

- Intuition: First referenced long time ago, done with it now
- Advantages: Fair: All pages receive equal residency; Easy to implement (circular buffer)
- Disadvantage: Some pages may always be needed

LRU: Least-recently-used: Replace page not used for longest time in past

- Intuition: Use past to predict the future
- Advantages: With locality, LRU approximates OPT
- Disadvantages:
 - Harder to implement, must track which pages have been accessed
 - Does not handle all workloads well

Page Replacement Example

Page reference string: ABCABDADBCB

Metric: Miss count	OPT	FIFO	LRU	Three pages of physical memory
ABC				
A				
B				
5, 7, 5 misses				
D				
A				
D				
B				
C				
B				

Page Replacement Comparison

Add more physical memory, what happens to performance?

- LRU, OPT: Add more memory, guaranteed to have fewer (or same number of) page faults
 - Smaller memory sizes are guaranteed to contain a subset of larger memory sizes
 - Stack property: smaller cache always subset of bigger
- FIFO: Add more memory, usually have fewer page faults
 - Belady's anomaly: May actually have [more](#) page faults!

Problems with LRU-based Replacement

LRU does not consider frequency of accesses

- Is a page accessed **once** in the past equal to one accessed **N** times?
- Common workload problem:
 - Scan (sequential read, never used again) one large data region flushes memory

Solution: Track frequency of accesses to page

Pure LFU (Least-frequently-used) replacement

- Problem: LFU can never forget pages from the far past

Examples of other more sophisticated algorithms:

- LRU-K and 2Q: Combines recency and frequency attributes
- Expensive to implement, LRU-2 used in databases

Implementing LRU

Software Perfect LRU

- OS maintains ordered list of physical pages by reference time
- When page is referenced: Move page to front of list
- When need victim: Pick page at back of list
- Trade-off: Slow on memory reference, fast on replacement

Hardware Perfect LRU

- Associate timestamp register with each page
- When page is referenced: Store system clock in register
- When need victim: Scan through registers to find oldest clock
- Trade-off: Fast on memory reference, slow on replacement (especially as size of memory grows)

In practice, do not implement Perfect LRU

- LRU is an approximation anyway, so approximate more
- Goal: Find an old page, but not necessarily the very old one

Clock Algorithm

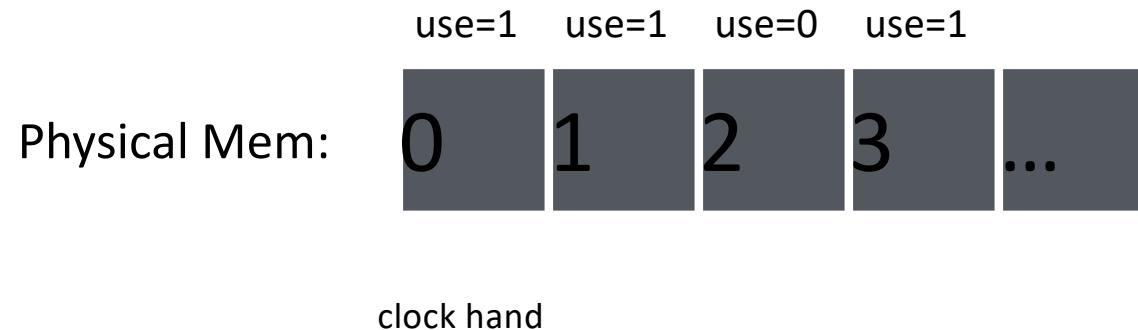
Hardware

- Keep `use` (or `reference`) bit for each page frame
- When page is referenced: set `use` bit

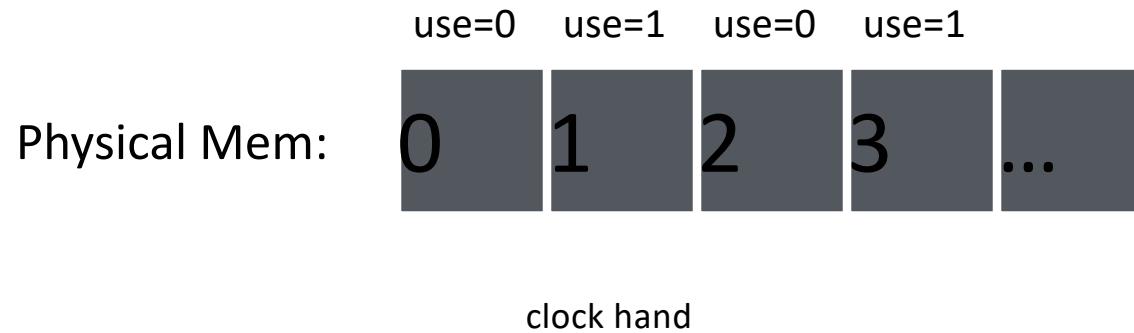
Operating System

- Page replacement: Look for page with `use` bit cleared
(has not been referenced for awhile)
- Implementation:
 - Keep pointer to last examined page frame
 - Traverse pages in circular buffer
 - Clear `use` bits as search
 - Stop when find page with already cleared `use` bit, replace this page

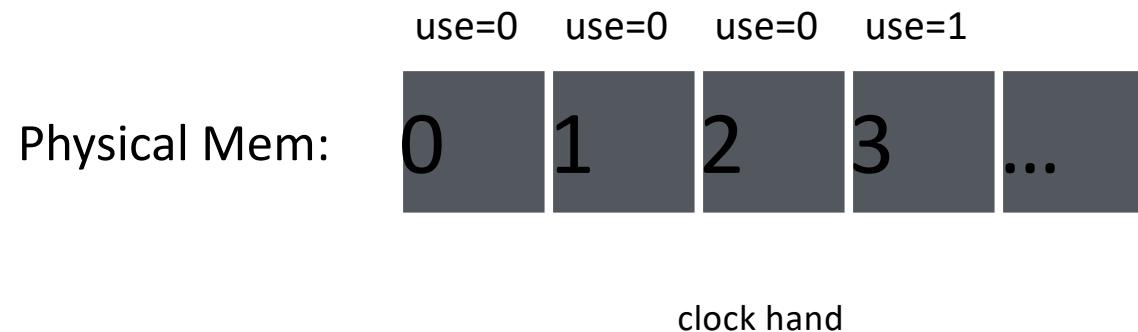
Clock: Look For a Page



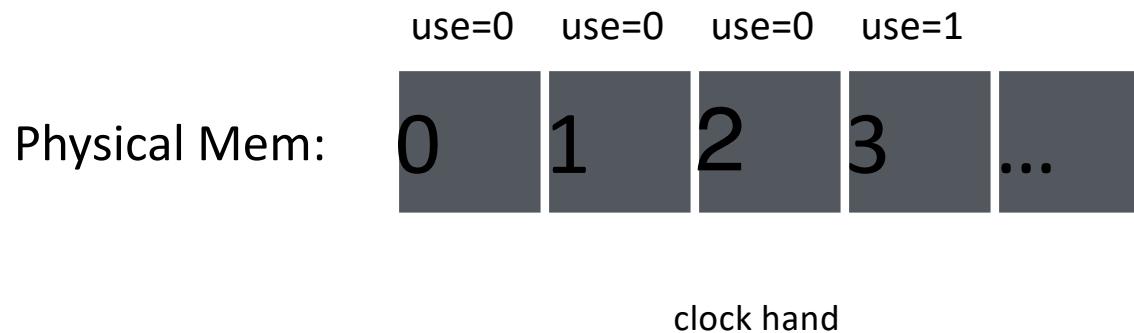
Clock: Look For a Page



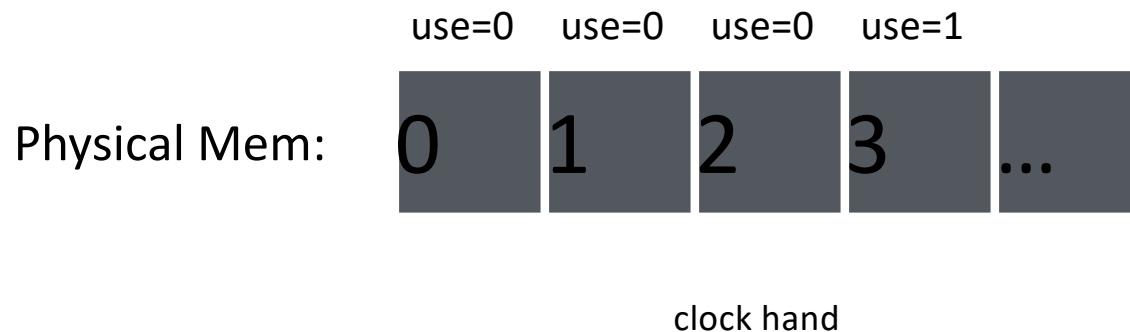
Clock: Look For a Page



Clock: Look For a Page

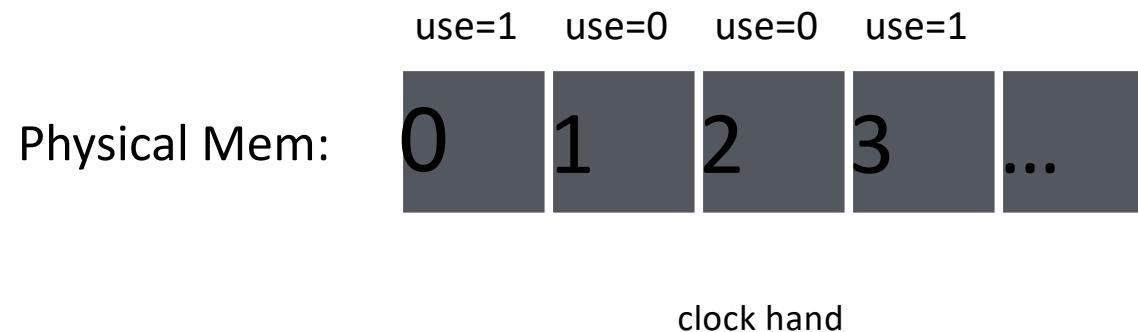


Clock: Look For a Page



page 0 is accessed...

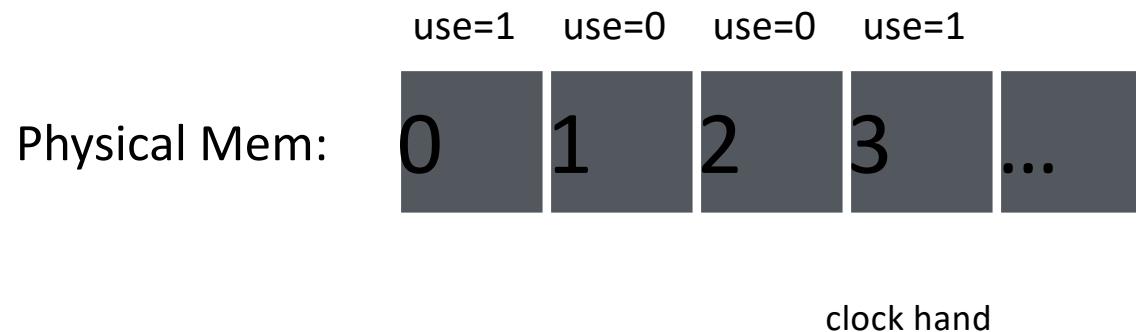
Clock: Look For a Page



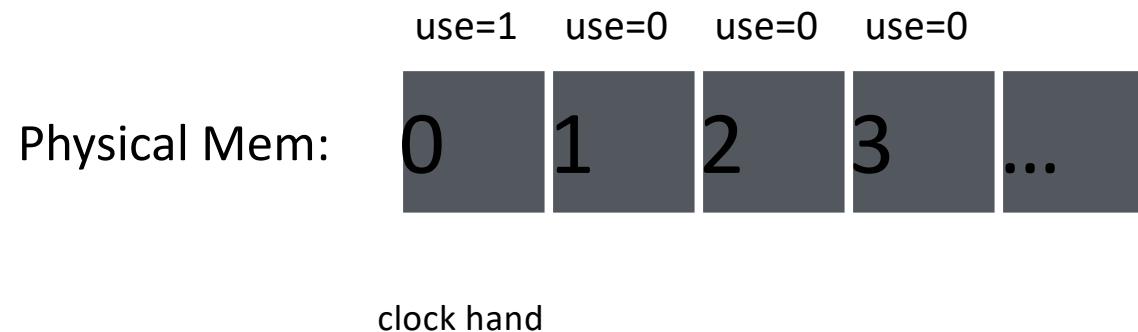
Clock: Look For a Page



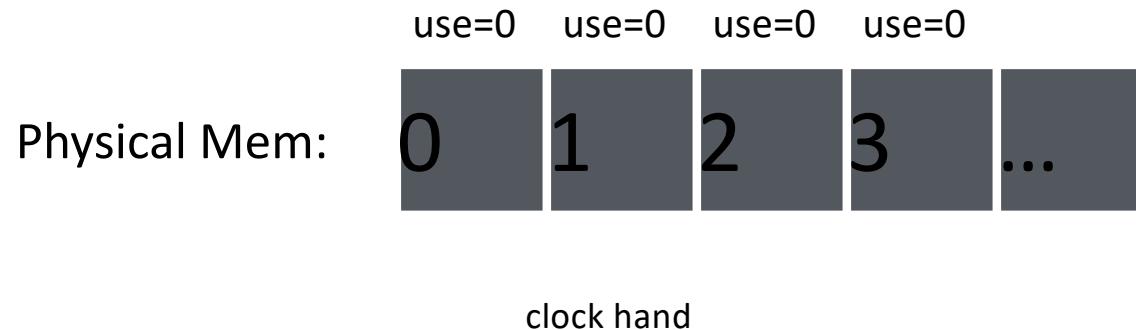
Clock: Look For a Page



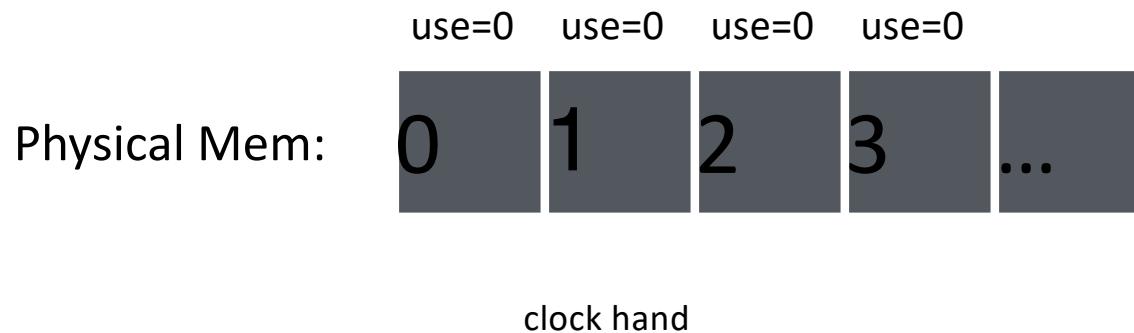
Clock: Look For a Page



Clock: Look For a Page



Clock: Look For a Page



evict **page 1** because it has not been recently used

Clock Extensions

Replace multiple pages at once

- Intuition:
Expensive to run replacement algorithm and to write single block to disk
- Find multiple victims each time and track free list

Add software counter (“chance”)

- Intuition: Better ability to differentiate across pages (how much they are being accessed)
- Increment software counter if `use` bit is 0
- Replace when chance exceeds some specified limit

Use dirty bit to give preference to dirty pages

- Intuition: More expensive to replace dirty pages
 - Dirty pages must be written to disk, clean pages do not
- Replace pages that have `use` bit and `dirty` bit cleared

ILLINOIS TECH

College of Computing

What if no Hardware Support?

What can the OS do if hardware does not have use bit
(or dirty bit)?

- Can the OS “emulate” these bits?

Leading question:

- How can the OS get control (i.e., generate a trap) every time use bit should be set? (i.e., when a page is accessed?)

Questions?

ILLINOIS TECH

College of Computing

Virtual Memory as a whole, XV6, X86, and Linux

Ben Lenard

blenard@iit.edu

ILLINOIS TECH

College of Computing

Agenda

- Homework 3 questions
- Midterm update
- Review Swapping
- X86 and xv6 and Linux

ILLINOIS TECH

College of Computing

Homework 3 questions?

ILLINOIS TECH

College of Computing

Midterm

The Midterm will be online from 6:25pm to 7:45pm October 16th 2023
online - I gave you 5 extra minutes for opening it etc.

Join Zoom Meeting for questions

[https://iit-
edu.zoom.us/j/82470358553?pwd=eEI1SHcrWHArQnkzZkw4VlBDMnV
kZz09](https://iit-edu.zoom.us/j/82470358553?pwd=eEI1SHcrWHArQnkzZkw4VlBDMnVkZz09)

ILLINOIS TECH

College of Computing

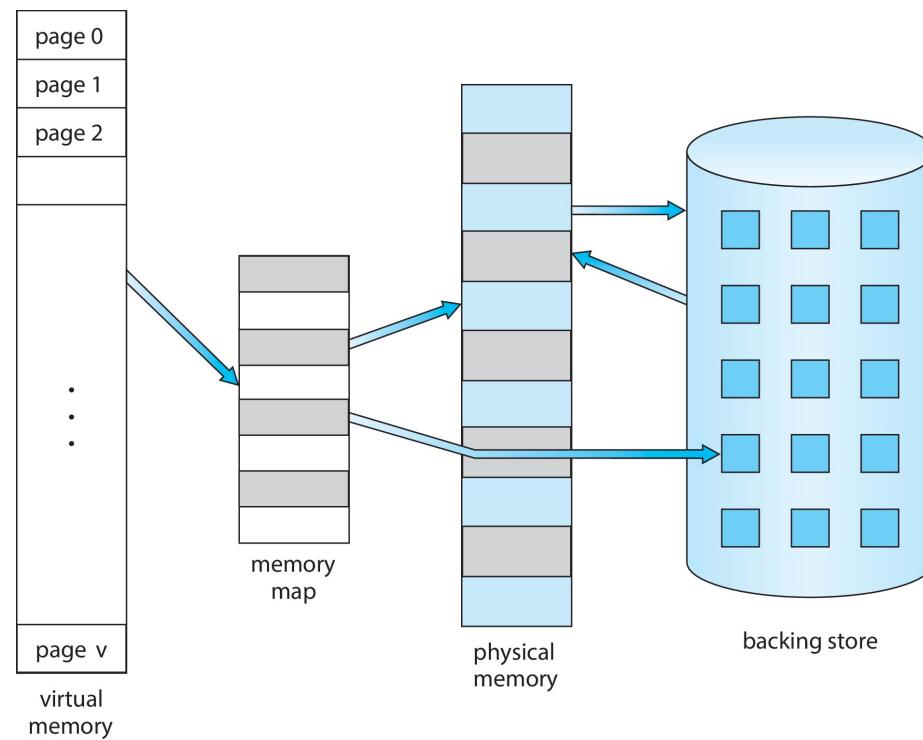
Virtual memory

- **Virtual memory** – separation of user logical memory from physical memory
 - Only part of the program needs to be in memory for execution
 - Logical address space can therefore be much larger than physical address space
 - Allows address spaces to be shared by several processes
 - Allows for more efficient process creation
 - More programs running concurrently
 - Less I/O needed to load or swap processes

Virtual memory (Cont.)

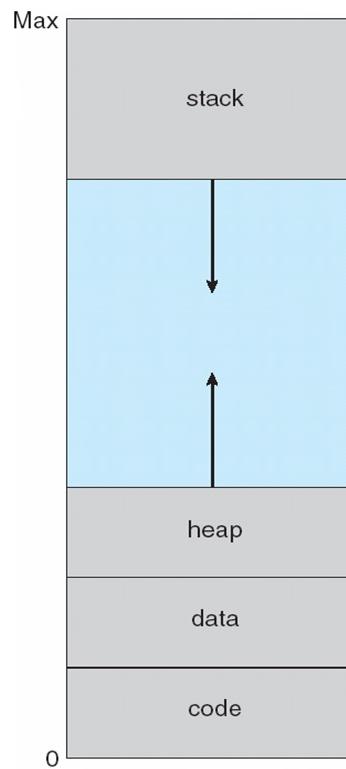
- **Virtual address space** – logical view of how process is stored in memory
 - Usually start at address 0, contiguous addresses until end of space
 - Meanwhile, physical memory organized in page frames
 - MMU must map logical to physical
- Virtual memory can be implemented via:
 - Demand paging
 - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

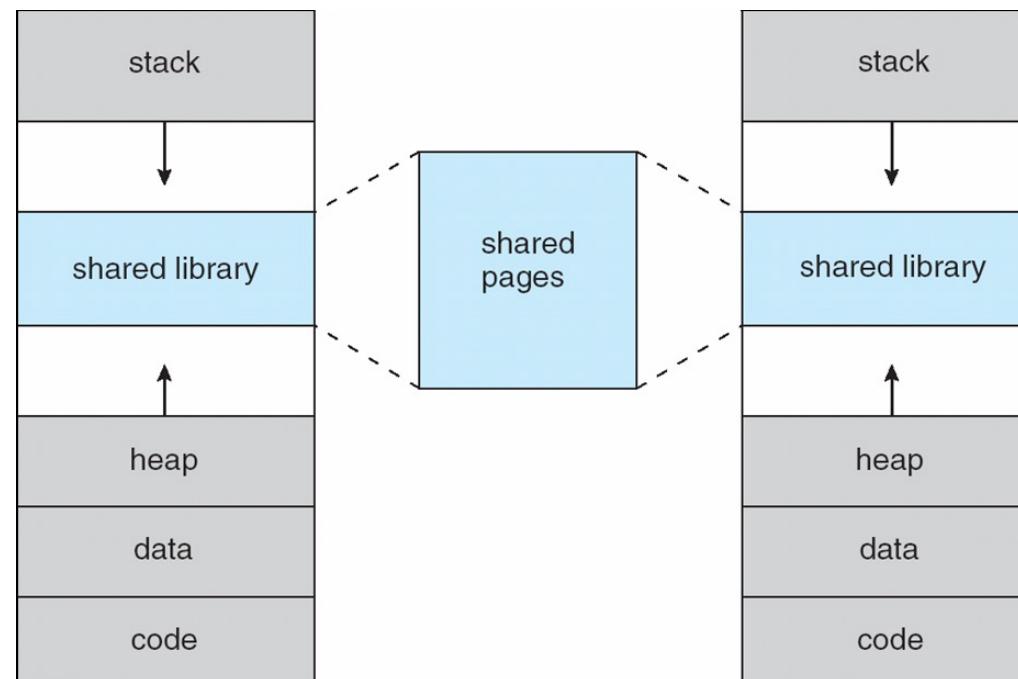


Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
 - Maximizes address space use
 - Unused address space between the two is hole
 - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



Shared Library Using Virtual Memory



Demand Paging

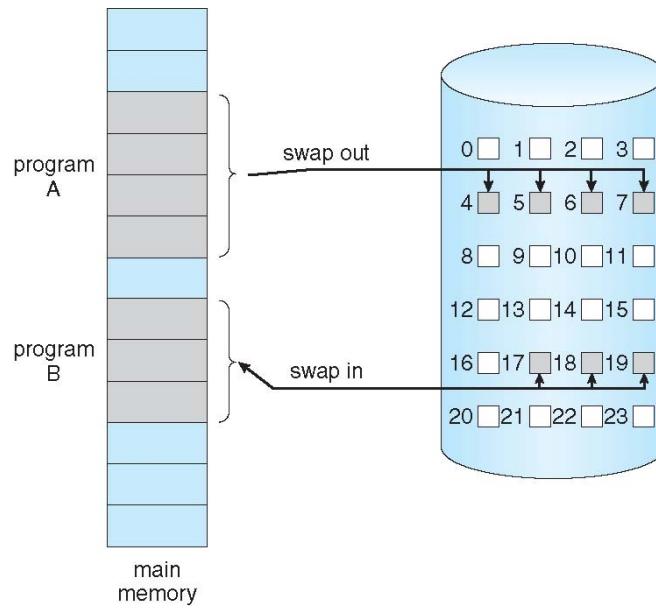
- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
 - Less I/O needed, no unnecessary I/O
 - Less memory needed
 - Faster response
 - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed \Rightarrow reference to it
 - invalid reference \Rightarrow abort
 - not-in-memory \Rightarrow bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
 - Swapper that deals with pages is a **pager**

ILLINOIS TECH

College of Computing

Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
 - Less I/O needed, no unnecessary I/O
 - Less memory needed
 - Faster response
 - More users
- Similar to paging system with swapping (diagram on right)



ILLINOIS TECH

College of Computing

```
Tasks: 312 total, 1 running, 311 sleeping, 0 stopped, 0 zombie
%Cpu(s): 18.9 us, 0.9 sy, 0.0 ni, 79.7 id, 0.5 wa, 0.0 hi, 0.1 si, 0.0 st
KiB Mem : 98497520 total, 5417132 free, 36884788 used, 56195596 buff/cache
KiB Swap: 5242876 total, 5182716 free, 60160 used. 60616540 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
20288	elastic+	20	0	175.9g	34.0g	1.4g	S	308.3	36.2	7:25.52	java
1923	root	20	0	2029716	150740	6868	S	1.0	0.2	188:03.98	filebeat
9	root	20	0	0	0	0	S	0.3	0.0	22:19.07	rcu_sched
1296	root	20	0	90668	2056	2056	S	0.3	0.0	10:44.70	rngd

ILLINOIS TECH

College of Computing

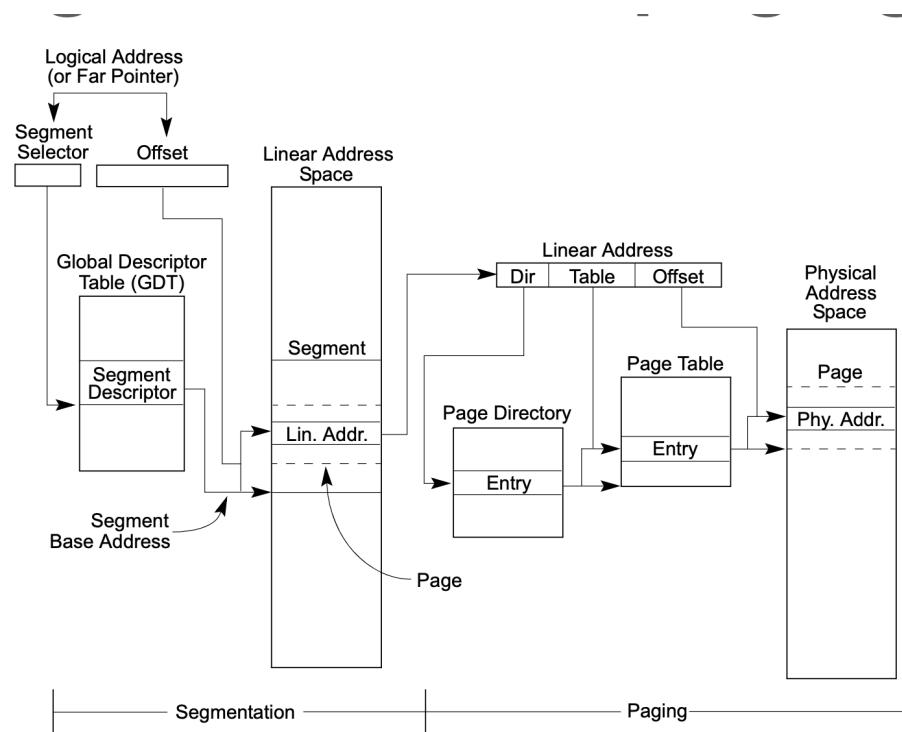
Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
 - Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
 - No difference from non demand-paging
- If page needed and not memory resident
 - Need to detect and load the page into memory from storage
 - Without changing program behavior
 - Without programmer needing to change code

x86 VM support

- x86 (aka IA-32) supports segmentation & paging in 32-bit protected mode
- x86-64 (aka IA-32e) introduces 64-bit (nominal) mode
 - Segmentation is mostly deprecated in favor of paging
 - Support for coexisting normal and “huge” pages

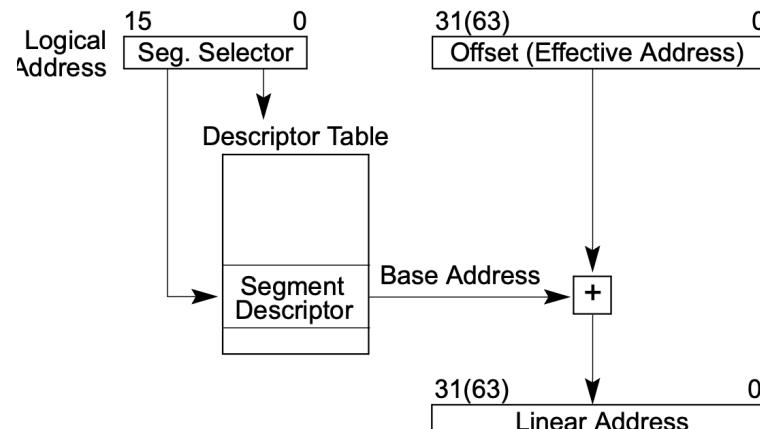
32-bit segmentation + paging



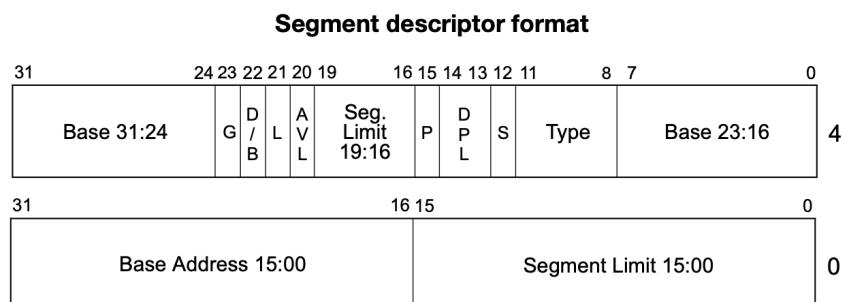
ILLINOIS TECH

College of Computing

32-bit segmentation



Segmentation registers	
Visible Part	Hidden Part
Segment Selector	Base Address, Limit, Access Information
	CS
	SS
	DS
	ES
	FS
	GS

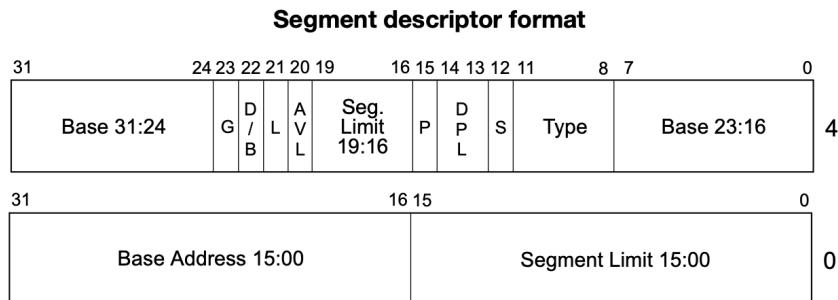


- L — 64-bit code segment (IA-32e mode only)
- AVL — Available for use by system software
- BASE — Segment base address
- D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
- DPL — Descriptor privilege level
- G — Granularity
- LIMIT — Segment Limit
- P — Segment present
- S — Descriptor type (0 = system; 1 = code or data)
- TYPE — Segment type

ILLINOIS TECH

College of Computing

32-bit xv6 segment initialization



L — 64-bit code segment (IA-32e mode only)

AVL — Available for use by system software

BASE — Segment base address

D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)

DPL — Descriptor privilege level

G — Granularity

LIMIT — Segment Limit

P — Segment present

S — Descriptor type (0 = system; 1 = code or data)

TYPE — Segment type

```
struct segdesc {
    uint lim_15_0 : 16; // Low bits of segment limit
    uint base_15_0 : 16; // Low bits of segment base address
    uint base_23_16 : 8; // Middle bits of segment base address
    uint type : 4; // Segment type (see STS_constants)
    uint s : 1; // 0 = system, 1 = application
    uint dpl : 2; // Descriptor Privilege Level
    uint p : 1; // Present
    uint lim_19_16 : 4; // High bits of segment limit
    uint avl : 1; // Unused (available for software use)
    uint rsv1 : 1; // Reserved
    uint db : 1; // 0 = 16-bit segment, 1 = 32-bit segment
    uint g : 1; // Granularity: limit scaled by 4K when set
    uint base_31_24 : 8; // High bits of segment base address
};

#define SEG(type, base, lim, dpl) (struct segdesc) \
{ ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff, \
((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
(uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
```

32-bit xv6 segment initialization

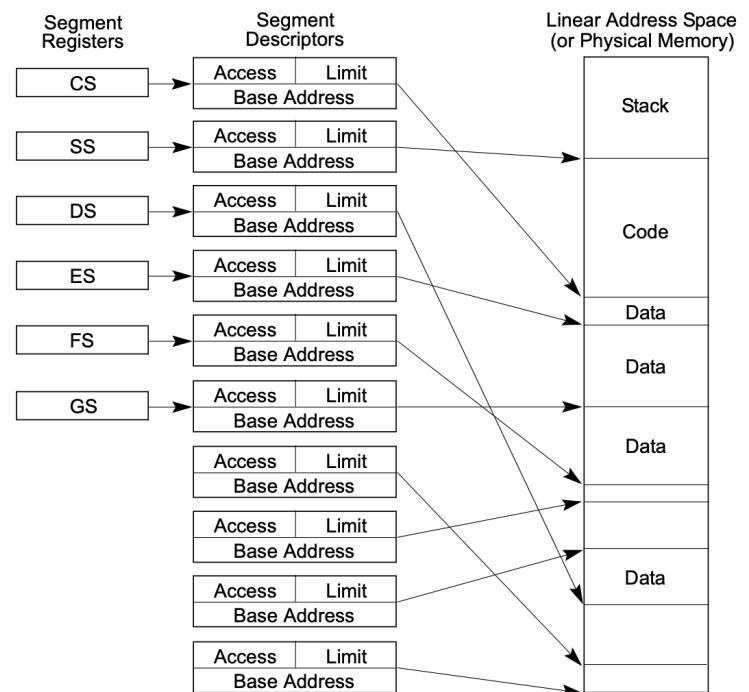
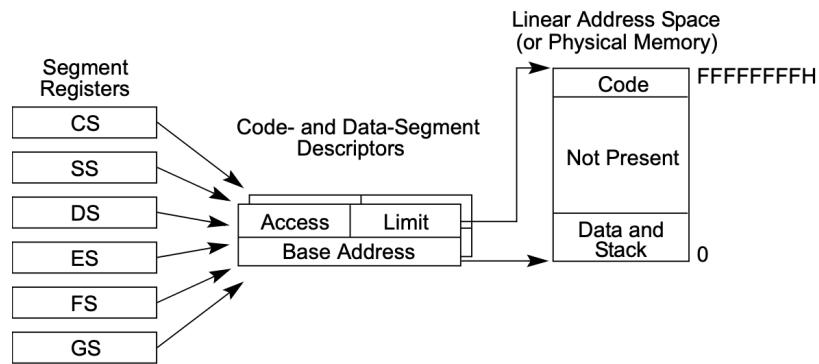
```
struct segdesc {
    uint lim_15_0 : 16; // Low bits of segment limit
    uint base_15_0 : 16; // Low bits of segment base address
    uint base_23_16 : 8; // Middle bits of segment base address
    uint type : 4; // Segment type (see STS_constants)
    uint s : 1; // 0 = system, 1 = application
    uint dpl : 2; // Descriptor Privilege Level
    uint p : 1; // Present
    uint lim_19_16 : 4; // High bits of segment limit
    uint avl : 1; // Unused (available for software use)
    uint rsv1 : 1; // Reserved
    uint db : 1; // 0 = 16-bit segment, 1 = 32-bit segment
    uint g : 1; // Granularity: limit scaled by 4K when set
    uint base_31_24 : 8; // High bits of segment base address
};

#define SEG(type, base, lim, dpl) (struct segdesc) \
{ ((lim) >> 12) & 0xffff, (uint)(base) & 0xffff, \
  ((uint)(base) >> 16) & 0xff, type, 1, dpl, 1, \
  (uint)(lim) >> 28, 0, 0, 1, 1, (uint)(base) >> 24 }
```

```
void
seginit(void)
{
    struct cpu *c;

    c = &cpus[cpuid()];
    c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
    c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
    c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
    c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
    lgdt(c->gdt, sizeof(c->gdt));
}
```

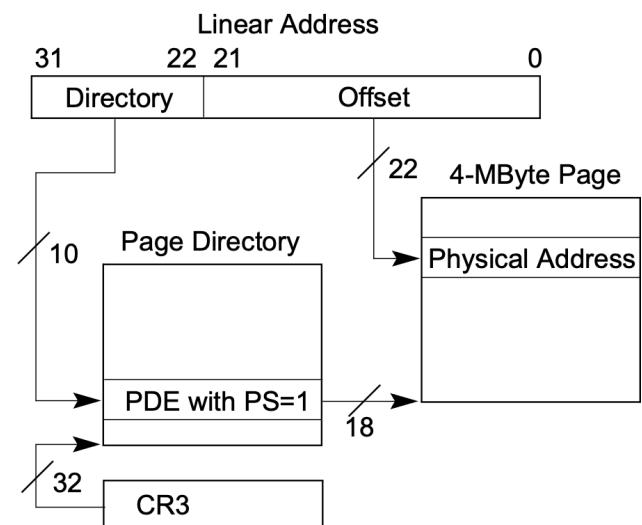
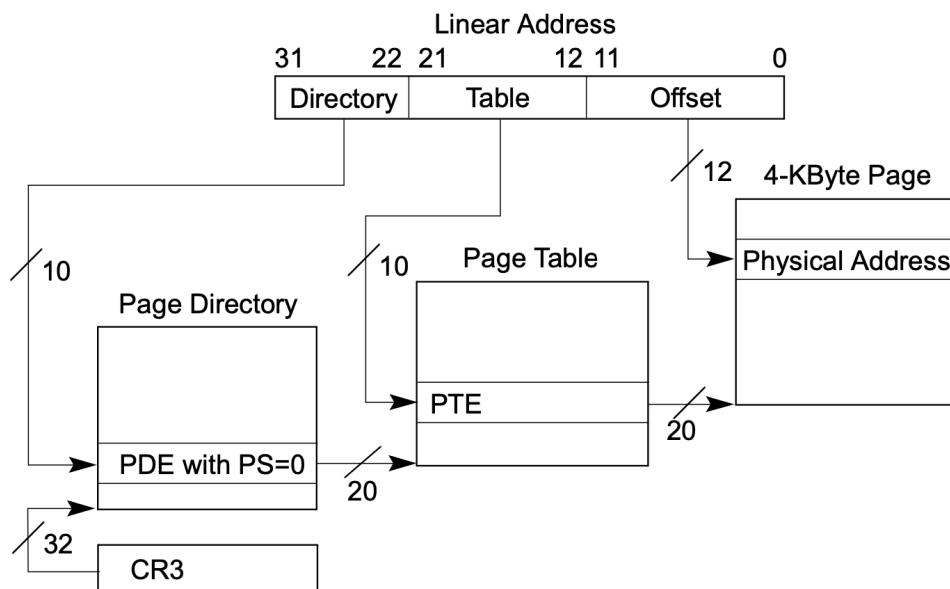
Flat vs. Multi-segment models



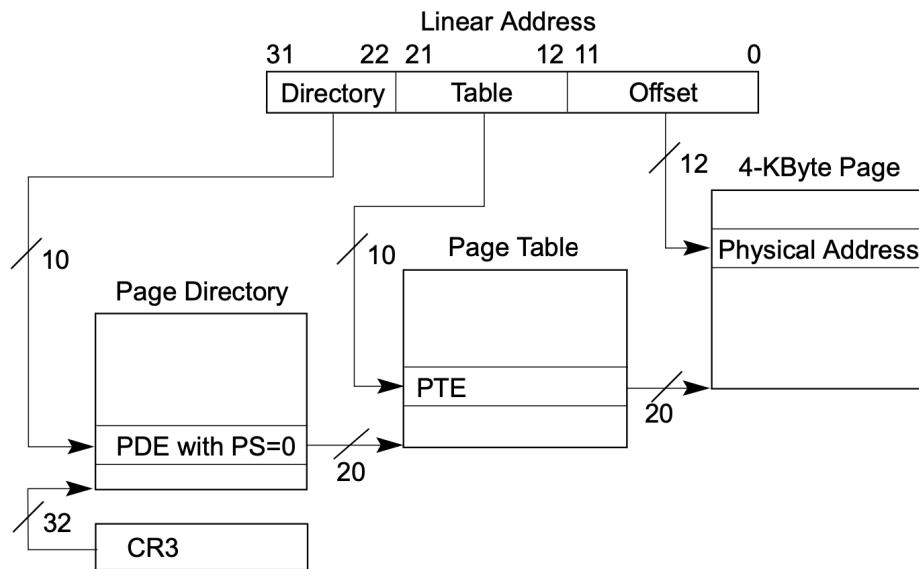
ILLINOIS TECH

College of Computing

32-bit 4KB vs. 4MB pages



32-bit 4KB xv6 page table walk/alloc



```
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
    pde_t *pde;
    pte_t *pgtab;

    pde = &pgdir[PDX(va)];
    if(*pde & PTE_P){
        pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
    } else {
        if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
            return 0;
        memset(pgtab, 0, PGSIZE);
        *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
    }
    return &pgtab[PTX(va)];
}
```

xv6 paging structure initialization

```
#define EXTMEM 0x100000          // Start of extended memory
#define PHYSTOP 0xE000000          // Top physical memory
#define DEVSPACE 0xFE000000         // Other devices are at high addresses

#define KERNBASE 0x80000000        // First kernel virtual address
#define KERNLINK (KERNBASE+EXTMEM) // Address where kernel is linked

#define PGSIZE      4096           // bytes mapped by a page
#define PGROUNDDUP(sz) (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))

#define V2P(a) (((uint)(a)) - KERNBASE)
#define P2V(a) ((void *)(((char*)(a)) + KERNBASE))

// This table defines the kernel's mappings, present in every process
static struct kmap {
    void *virt;
    uint phys_start;
    uint phys_end;
    int perm;
} kmap[] = {
    { (void*)KERNBASE, 0,          EXTMEM,     PTE_W}, // I/O space
    { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata
    { (void*)data,     V2P(data),   PHYSTOP,    PTE_W}, // kern data+memory
    { (void*)DEVSPACE, DEVSPACE,   0,          PTE_W}, // more devices
}
```

```
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
    char *a, *last;
    pte_t *pte;

    a = (char*)PGROUNDDOWN((uint)va);
    last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
    for(;;){
        if((pte = walkpgdir(pgdir, a, 1)) == 0)
            return -1;
        if(*pte & PTE_P)
            panic("remap");
        *pte = pa | perm | PTE_P;
        if(a == last)
            break;
        a += PGSIZE;
        pa += PGSIZE;
    }
    return 0;
}
```

ILLINOIS TECH

College of Computing

xv6 paging structure initialization

```
// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
    pde_t *pgdir;
    struct kmap *k;

    pgdir = (pde_t*)kalloc();

    memset(pgdir, 0, PGSIZE);

    if (P2V(PHYSTOP) > (void*)DEVSPACE)
        panic("PHYSTOP too high");

    for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
        mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                  (uint)k->phys_start, k->perm);
    return pgdir;
}
```

ILLINOIS TECH

College of Computing

xv6 paging structure initialization

```
// Allocate page tables and physical memory to grow process
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
    char *mem;
    uint a;

    if(newsz >= KERNBASE)
        return 0;

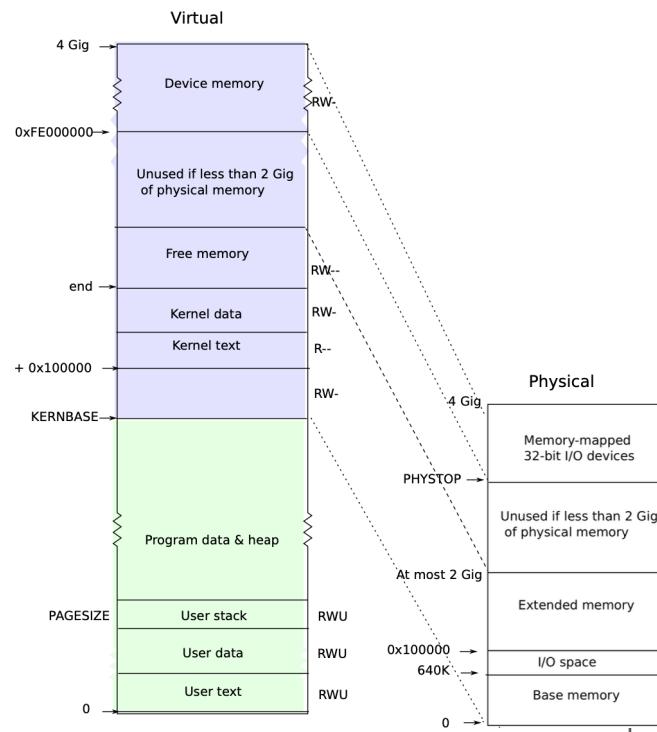
    if(newsz < oldsz)
        return oldsz;

    a = PGROUNDUP(oldsz);
    for(; a < newsz; a += PGSIZE){
        mem = kalloc();
        memset(mem, 0, PGSIZE);
        mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U);
    }
    return newsz;
}
```

ILLINOIS TECH

College of Computing

xv6 paging structure initialization



ILLINOIS TECH

College of Computing

Beyond 32-bit address spaces

Paging Mode	PG in CR0	PAE in CR4	LME in IA32_EFER	Lin.-Addr. Width	Phys.-Addr. Width ¹	Page Sizes	Supports Execute-Disable?	Supports PCIDs and protection keys?
None	0	N/A	N/A	32	32	N/A	No	No
32-bit	1	0	0 ²	32	Up to 40 ³	4 KB 4 MB ⁴	No	No
PAE	1	1	0	32	Up to 52	4 KB 2 MB	Yes ⁵	No
4-level	1	1	1	48	Up to 52	4 KB 2 MB 1 GB ⁶	Yes ⁵	Yes ⁷

aka x86-64, per original AMD specification

ILLINOIS TECH

College of Computing

x86-64 (aka IA-32e) modes

- Long mode: 48-bit virtual addresses (256TB virtual address spaces)
 - 4-levels of paging structures
 - All but two segment registers are forced to a flat model, and no segment limit checking is performed
 - FS & GS segments can contain non-zero bases (useful for OS)
- Compatibility mode allows for 32-bit code to run unaltered
- Intel has started implementing 5-level paging to support 57-bit virtual addresses (as of Ice Lake)

Ice Lake example

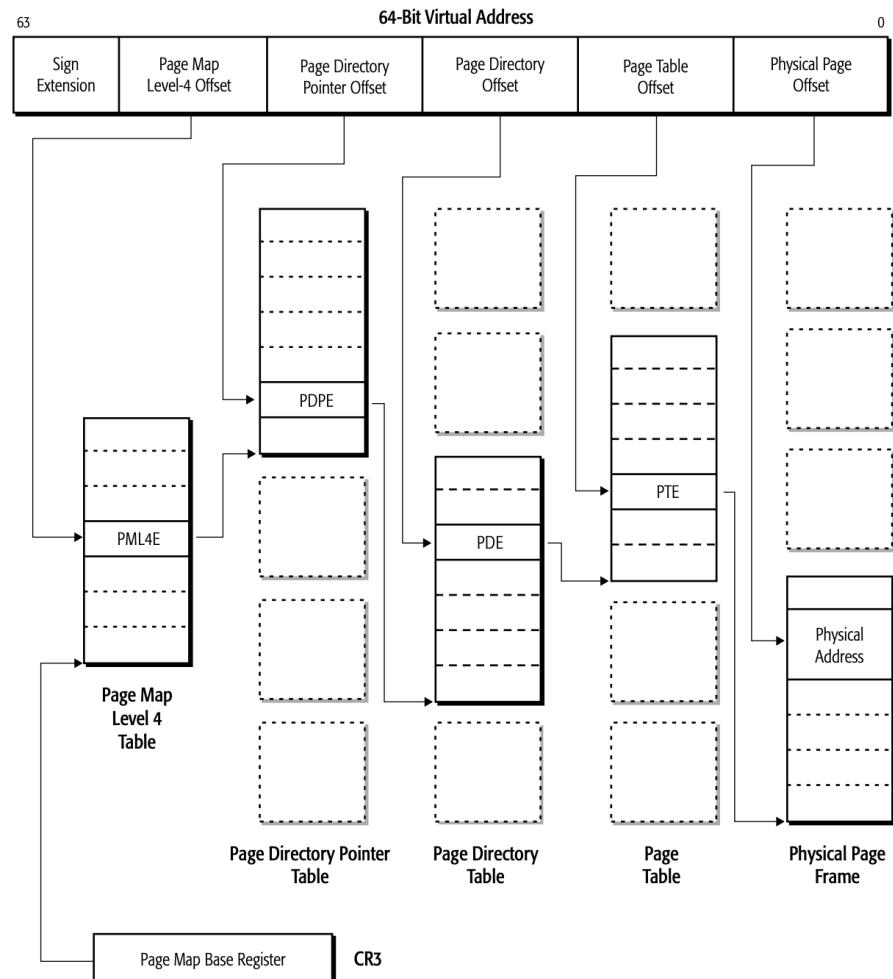
```
processor : 63
vendor_id : GenuineIntel
cpu family : 6
model : 106
model name : Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz
stepping : 6
microcode : 0xd0003a5
cpu MHz : 2900.000
cache size : 24576 KB
physical id : 1
siblings : 32
core id : 15
cpu cores : 16
apicid : 95
initial apicid : 95
fpu : yes
fpu_exception : yes
cpuid level : 27
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
od nopl xtopology nonstop_tsc cpuid aperfmpf perf_pni pclmulqdq dtes6
rand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3 invpcid_singl
t_a avx512f avx512dq rdseed adx smap avx512ifma clflushopt clwb ir
vd dtherm ida arat pln pts avx512vbmi umip pkru ospke avx512_ybm2
bugs : spectre_v1 spectre_v2 spec_store_bypass swapgs m
bogomips : 5823.62
clflush size : 64
cache_alignment : 64
address sizes : 46 bits physical, 57 bits virtual
power management:
```

ILLINOIS TECH

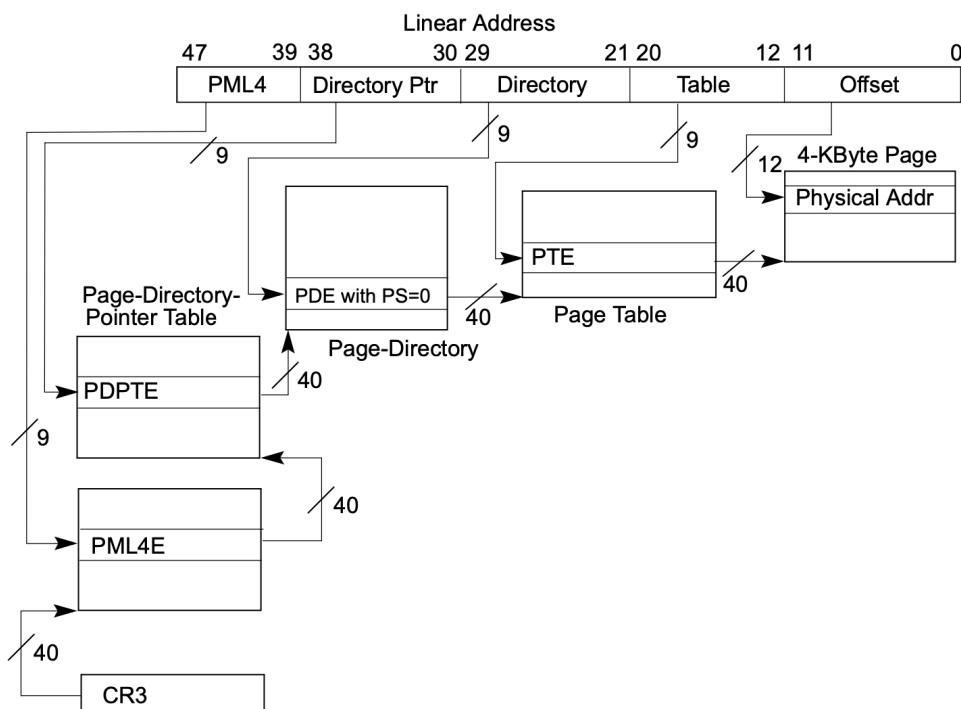
College of Computing

Long mode paging

- 48-bit virtual addresses with 4 levels of paging
- Depending on paging structure entries, supports 4KB, 2MB, 1GB page sizes



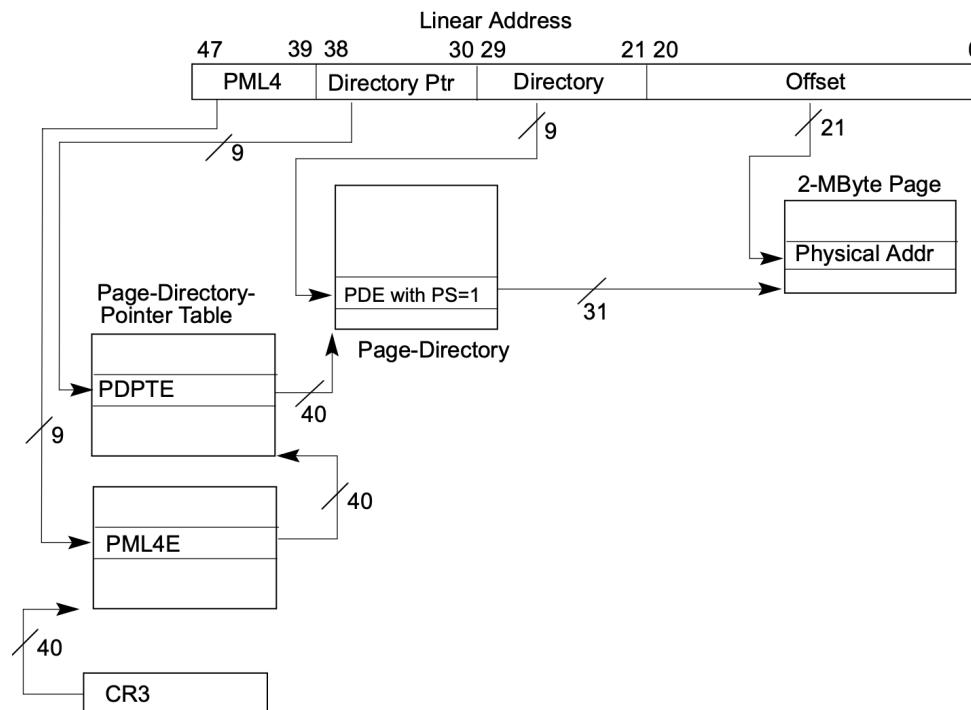
Long mode 4KB paging



ILLINOIS TECH

College of Computing

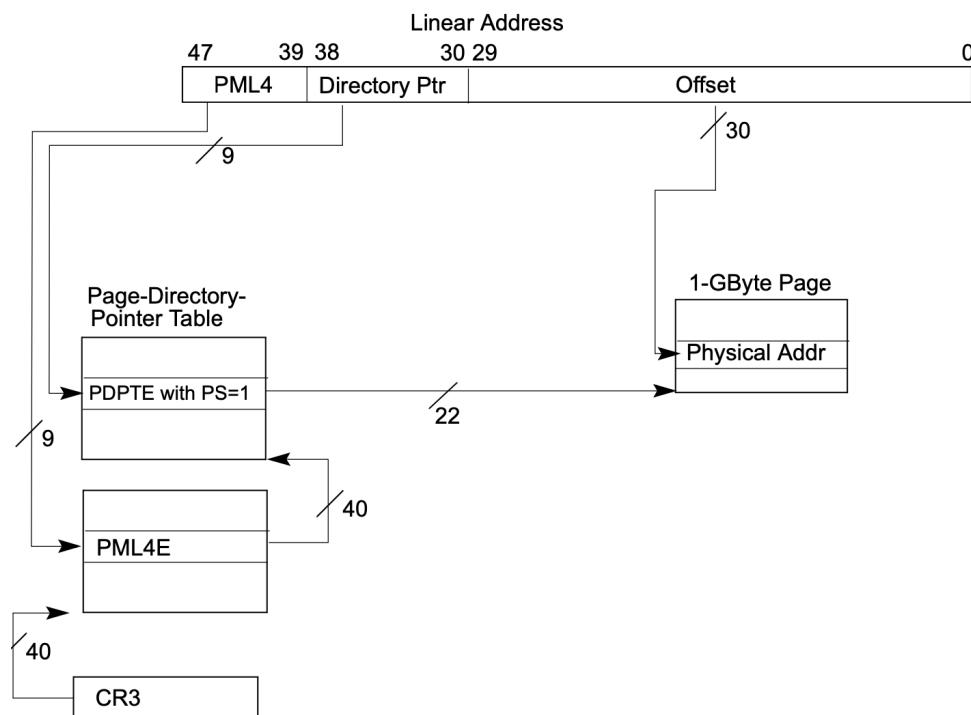
Long mode 2MB paging



ILLINOIS TECH

College of Computing

Long mode 1GB paging



ILLINOIS TECH

College of Computing

Access control and metadata

- User/Supervisor and Read/Write flags in paging structure entries can be used to guard access
- If U/S flag = 0 (supervisor), can only access page if CPL = 0
- Accessed and Dirty flags are also useful for kernel swapping policies

66665555555555	M ¹	M-1	33322222222222	11111111111111	P P C W D T	Ign.	
3210987654321			210987654321098765432109876543210				CR3
Reserved ²			Address of PML4 table			Ignored	
X D 3	Ignored	Rsvd.	Address of page-directory-pointer table			Ign. R S v d n	I g n A P P U R D T /S W 1
Ignored							0
X D	Prot. Key ⁴	Ignored	Rsvd.	Address of 1GB page frame	Reserved	P A T I g n G 1 D A P P U R D T /S W 1	PDPTPE: 1GB page
X D	Ignored	Rsvd.	Address of page directory			Ign. Q I g n A P P U R D T /S W 1	PDPTPE: page directory
Ignored							0
X D	Prot. Key ⁴	Ignored	Rsvd.	Address of 2MB page frame	Reserved	P A T I g n G 1 D A P P U R D T /S W 1	PDE: 2MB page
X D	Ignored	Rsvd.	Address of page table			Ign. Q I g n A P P U R D T /S W 1	PDE: page table
Ignored							0
X D	Prot. Key ⁴	Ignored	Rsvd.	Address of 4KB page frame	Ign. G A D A P P U R D T /S W 1	PTE: 4KB page	
Ignored							0

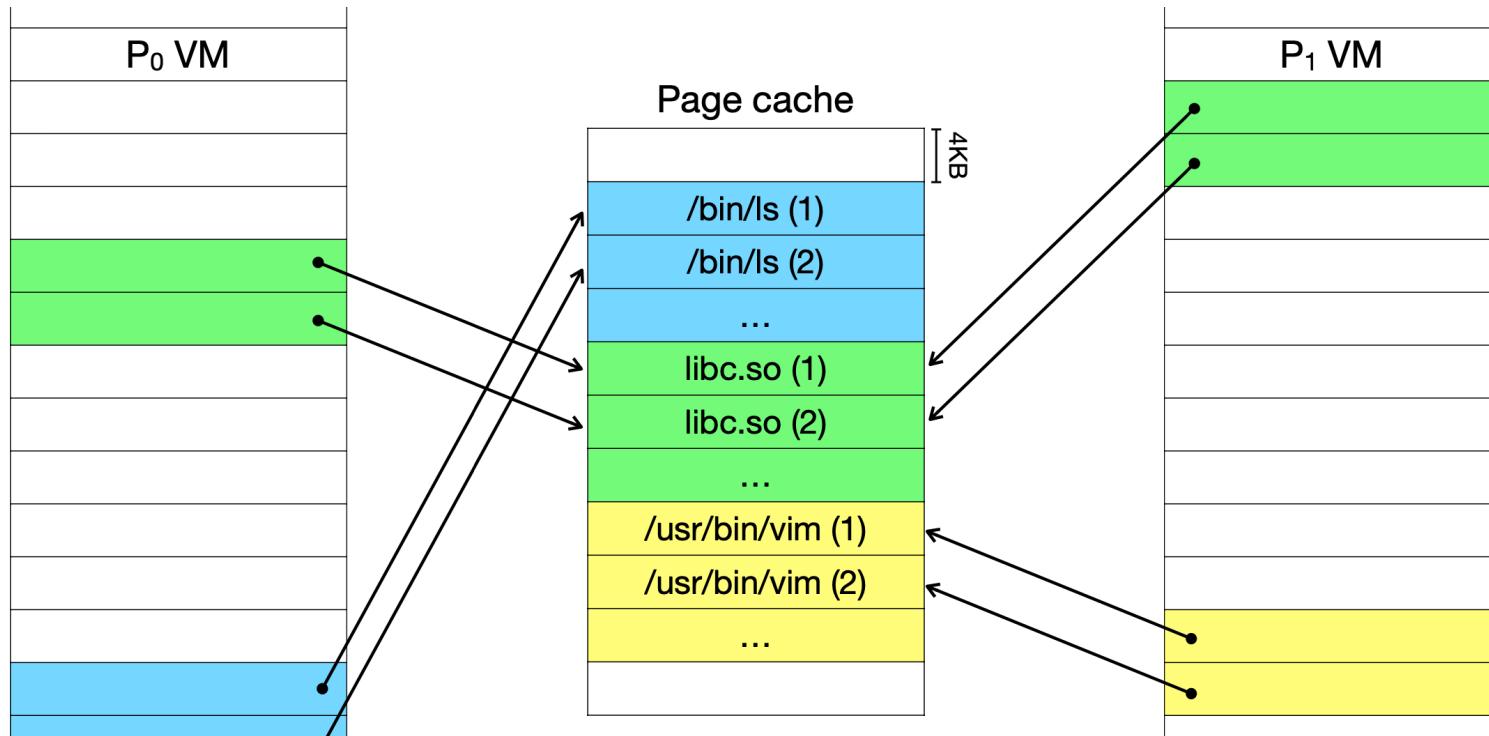
Linux Virtual Memory features

- Page cache and Sharing
- Swap cache
- Copy-on-write optimization
- Page allocation: buddy system
- Kernel internal memory management: slab allocator

Page cache and Sharing

- When executing a program (or loading shared libraries, etc.), the source file is not immediately loaded, but rather linked into the process's virtual address space
- Page faults cause data to be loaded, a page at a time
- All file data loaded this way have entries in the page cache, which the kernel consults before going to disk
 - If multiple processes access the same files, the kernel can share cached pages between them (potentially at different virtual addresses)
 - Dirty bit needed to ensure that page isn't modified

Page cache and Sharing



ILLINOIS TECH

College of Computing

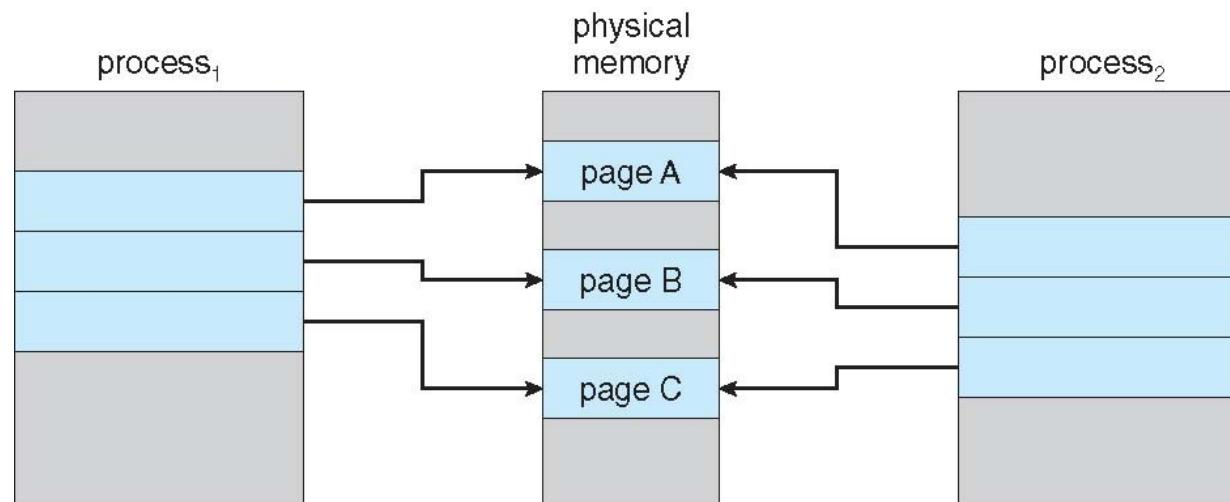
Swap cache

- Dirty pages are swapped out (to save their contents) when low on memory
 - Unmodified pages can just be loaded from the page cache!
- Swap cache keeps track of pages that have been written to swap
 - If a page was previously swapped out and wasn't modified after being swapped back in, can simply discard it
- Helpful optimization for when system is heavily swapping (thrashing)

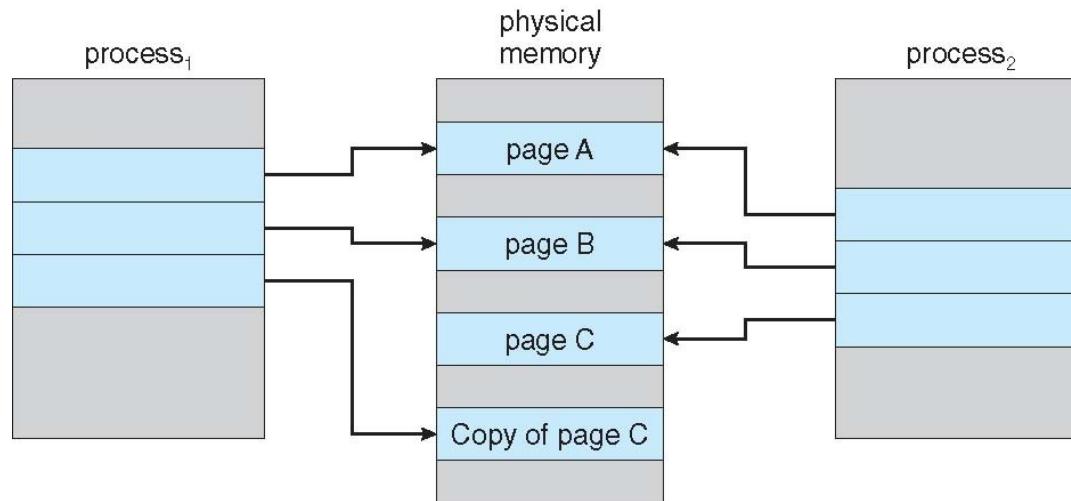
Copy-on-write (COW)

- “Clone” operation is quite common (e.g., used when fork-ing a process)
- But if carried out literally — duplicating entire memory image — is incredibly expensive (and likely unnecessary)
- At clone time, no data is actually copied; simply replicate paging structures and mark pages as read-only
 - Page faults that occur on write accesses trigger copy operation

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



ILLINOIS TECH

College of Computing

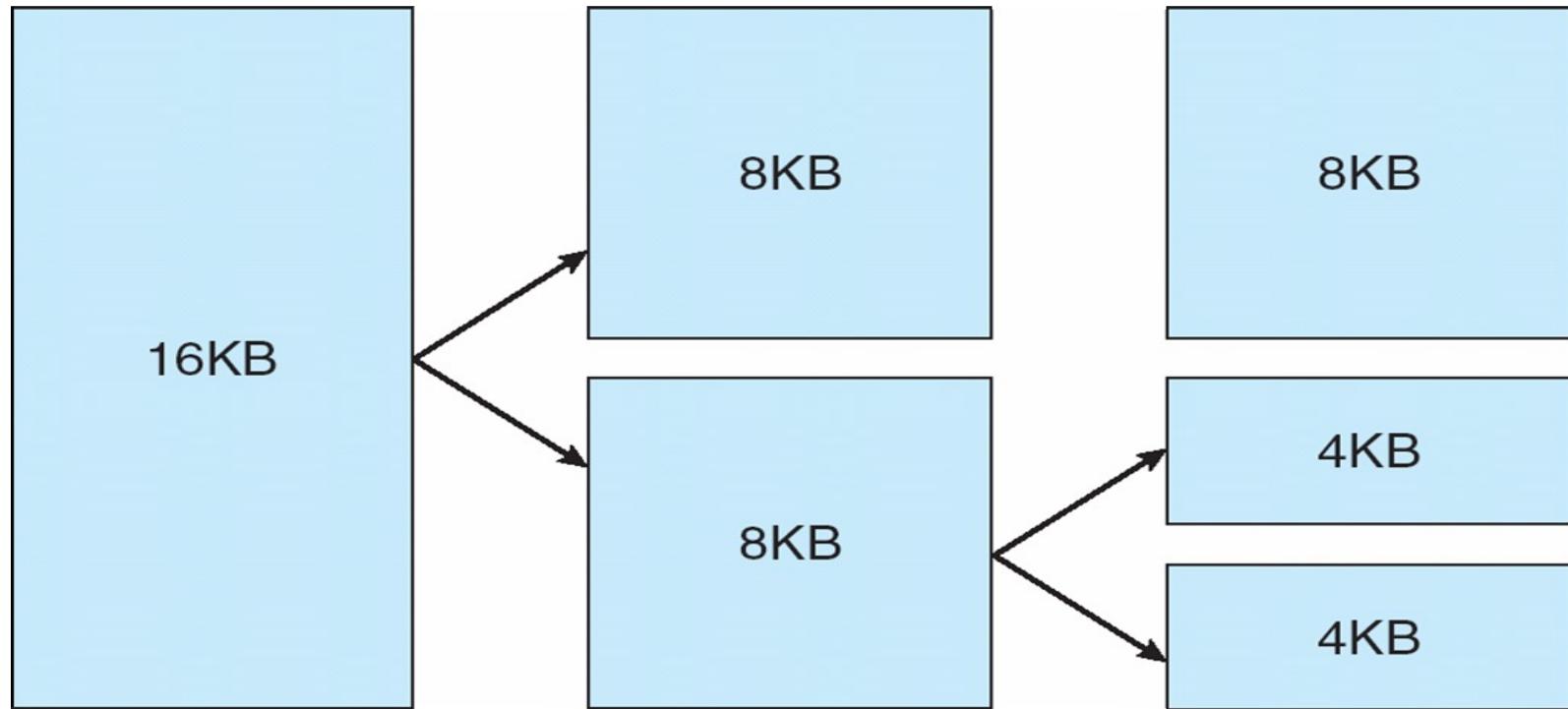
Page allocation

- Because pages are all the same size, theoretically we have no external fragmentation
 - Can allocate first free page we find and map it into any virtual address space using paging structures
- But recall: large blocks of contiguous pages can be mapped as a single huge page (e.g., 4KB vs. 4MB)
 - Can greatly improve TLB effectiveness!
 - Especially desirable given many levels of paging structures
 - Also needed for I/O device direct memory access (more on this later)

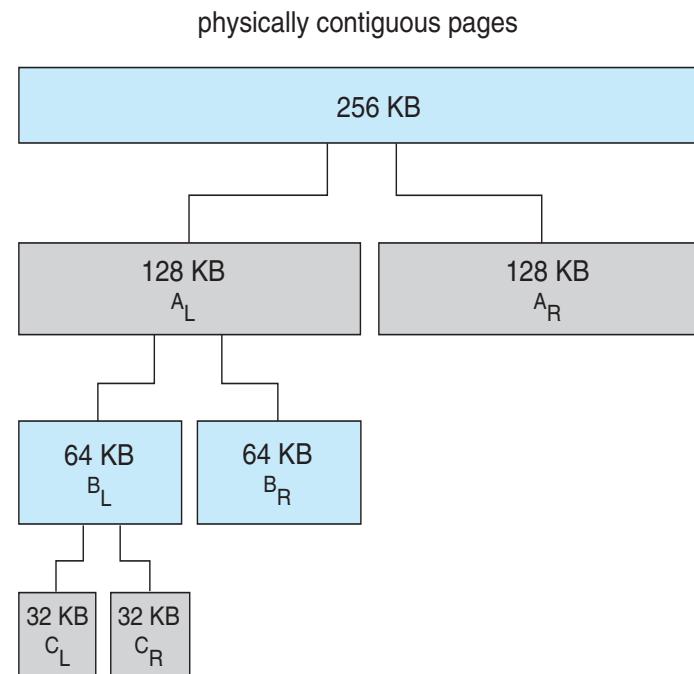
Buddy system allocator

- Linux uses a “buddy system” allocator to search for blocks of free pages
- Idea: maintain separate lists of free page blocks, with sizes = powers of 2
 - E.g., list #0 = 1 page blocks,
list #1 = 2 page blocks,
list #2 = 4 page blocks,
list #3 = 8 page blocks, etc.
- When allocating a block, keep splitting in half if possible
- When freeing a block, keep merging (doubling) if possible

Splitting of Memory in a Buddy Heap



Buddy System Allocator



ILLINOIS TECH

College of Computing

Buddy system pros & cons

Pros

- Fast allocation — search is easy
- Able to find contiguous blocks
- Good for huge pages
- Can simplify page table updates

Cons

- Small vs. Large blocks creates external fragmentation
- 2^n block sizes can result in significant internal fragmentation
- Compromise: speed vs. efficiency

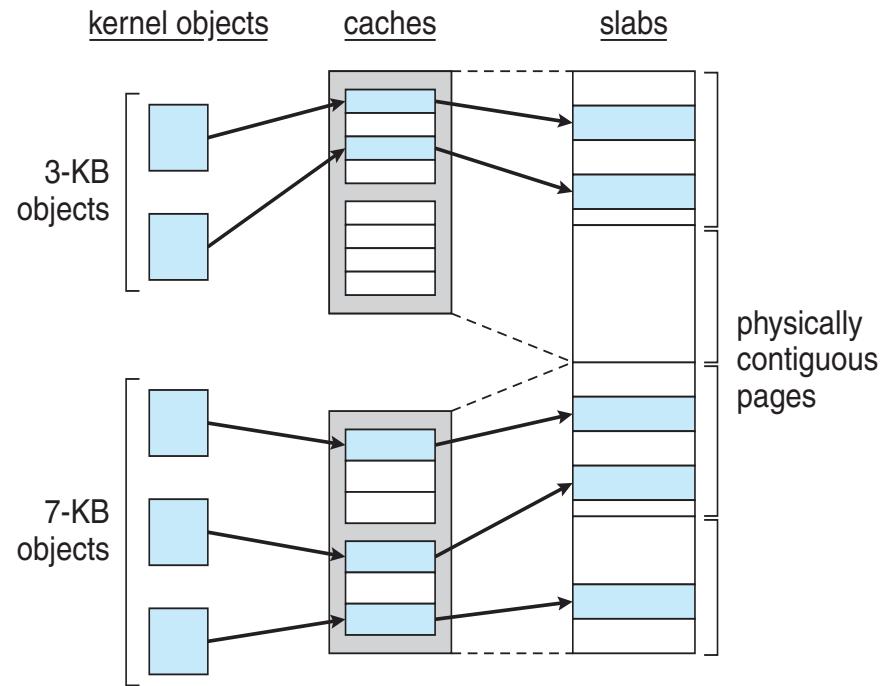
Kernel internal allocation

- Kernel frequently needs to free/allocate internal data structures
 - e.g., PCB entries, VM structures, file/inode structures
 - Fixed size, similarly initialized
- Buddy allocator is not ideal — too much internal fragmentation!
- Linux uses a slab allocator to allocate & free internal data structures

Slab allocator

- Built on top of the page buddy allocator
- Idea: allocate large blocks using buddy allocator, and carve them up into multiple data structure entries
 - Use the first one available, and leave partially initialized when freed
 - Effectively build dedicated caches for different data types
- Mitigates internal fragmentation due to buddy allocator

Slab Allocation



ILLINOIS TECH

College of Computing

Slab Allocator in Linux

- For example process descriptor is of type
`struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
 - Will use existing free `struct task_struct`
- Slab can be in three possible states
 - 1.Full – all used
 - 2.Empty – all free
 - 3.Partial – mix of free and used
- Upon request, slab allocator
 - 1.Uses free struct in partial slab
 - 2.If none, takes one from empty slab
 - 3.If no empty slab, create new empty

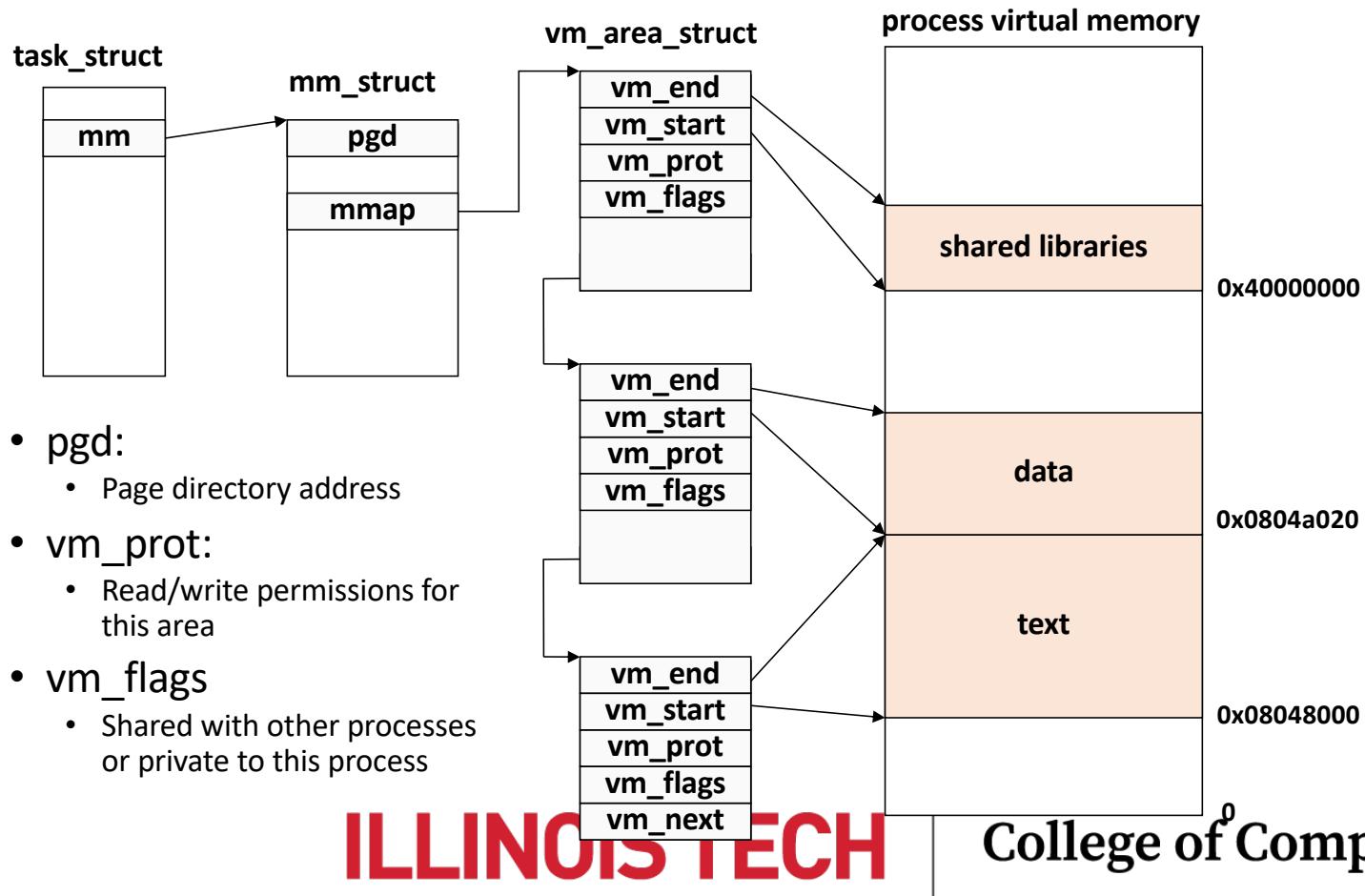
ILLINOIS TECH

College of Computing

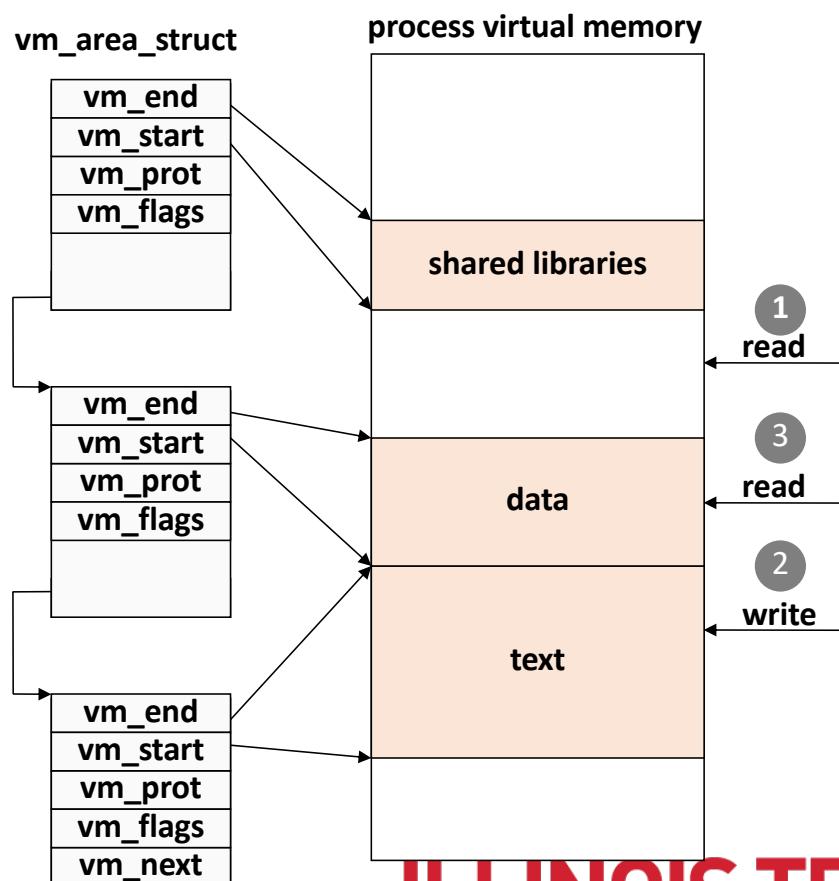
Slab Allocator in Linux (Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
 - SLOB for systems with limited memory
 - Simple List of Blocks – maintains 3 list objects for small, medium, large objects
 - SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure

Linux Organizes VM as Collection of “Areas”



Linux Page Fault Handling



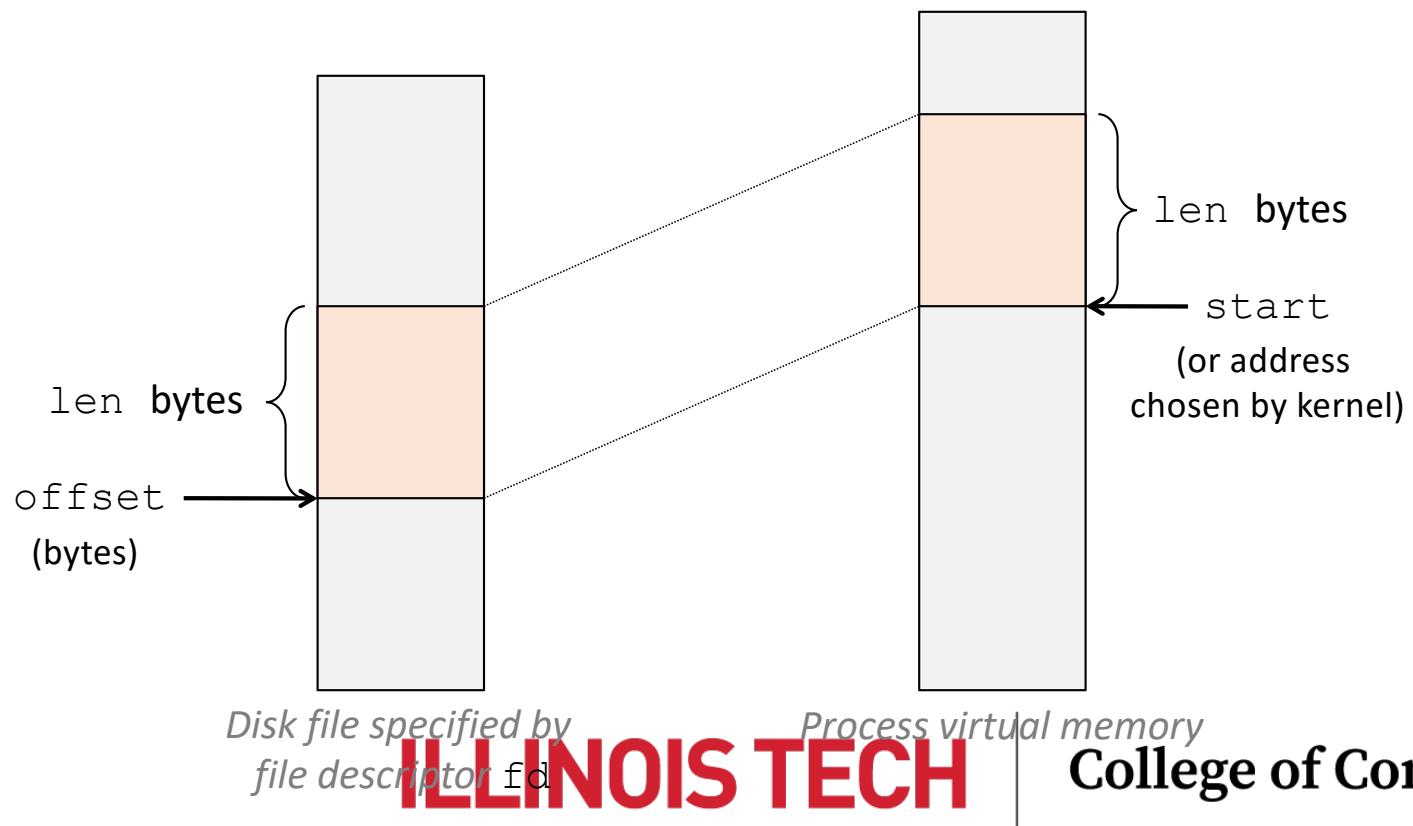
- Is the VA legal?
 - = Is it in an area defined by a `vm_area_struct`?
 - If not (#1), then signal segmentation violation
- Is the operation legal?
 - i.e., Can the process read/write this area?
 - If not (#2), then signal protection violation
- Otherwise
 - Valid address (#3): handle fault

Memory Mapping

- Creation of new VM *area* done via “memory mapping”
 - Create new vm_area_struct and page tables for area
- Area can be backed by (i.e., get its initial values from) :
 - Regular file on disk (e.g., an executable object file)
 - Initial page bytes come from a section of a file
 - Nothing (e.g., .bss)
 - First fault will allocate a physical page full of 0's (demand-zero)
 - Once the page is written to (dirtied), it is like any other page
- Dirty pages are swapped back and forth between a special swap file.
- *Key point:* no virtual pages are copied into physical memory until they are referenced!
 - Known as “demand paging”
 - Crucial for time and space efficiency

User-Level Memory Mapping

```
void *mmap(void *start, int len,  
           int prot, int flags, int fd, int offset)
```



*Disk file specified by
file descriptor fd*

ILLINOIS TECH

Process virtual memory

College of Computing

User-Level Memory Mapping

```
void *mmap(void *start, int len,  
           int prot, int flags, int fd, int offset)
```

- Map **len** bytes starting at offset **offset** of the file specified by file description **fd**, preferably at address **start**
 - **start**: may be 0 for “pick an address”
 - **prot**: PROT_READ, PROT_WRITE, ...
 - **flags**: MAP_PRIVATE, MAP_SHARED, ...
- Return a pointer to start of mapped area (may not be **start**)
- Example: fast file-copy
 - Useful for applications like Web servers that need to quickly copy files.
 - **mmap ()** allows file transfers without copying into user space.

mmap() Example: Fast File Copy

```
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/*
 * a program that uses mmap to copy
 * the file input.txt to stdout
 */
```

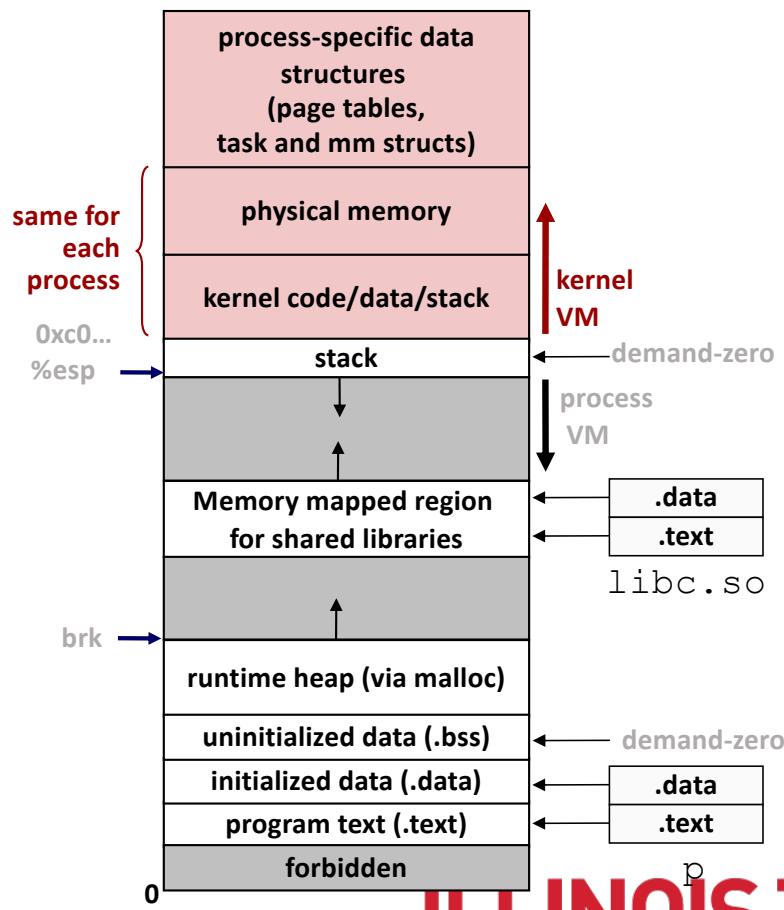
```
int main() {
    struct stat stat;
    int i, fd, size;
    char *bufp;

    /* open the file & get its size*/
    fd = open("./input.txt", O_RDONLY);
    fstat(fd, &stat);
    size = stat.st_size;

    /* map the file to a new VM area */
    bufp = mmap(0, size, PROT_READ,
               MAP_PRIVATE, fd, 0);

    /* write the VM area to stdout */
    write(1, bufp, size);
    exit(0);
}
```

Exec() Revisited



To run a new program p in the current process using `exec()`:

- Free `vm_area_struct`'s and page tables for old areas
- Create new `vm_area_struct`'s and page tables for new areas
 - Stack, BSS, data, text, shared libs.
 - Text and data backed by ELF executable object file
 - BSS and stack initialized to zero
- Set PC to entry point in `.text`
 - Linux will fault in code, data pages as needed

Fork() Revisited

- To create a new process using `fork()`:
 - Make copies of the old process's `mm_struct`, `vm_area_struct`'s, and page tables.
 - At this point the two processes share all of their pages.
 - How to get separate spaces without copying all the virtual pages from one space to another?
 - "Copy on Write" (COW) technique.
 - Copy-on-write
 - Mark PTE's of writeable areas as read-only
 - Writes by either process to these pages will cause page faults
 - Flag `vm_area_struct`'s for these areas as private "copy-on-write"
 - Fault handler recognizes copy-on-write, makes a copy of the page, and restores write permissions.
- Net result:
 - Copies are deferred until absolutely necessary (i.e., when one of the processes tries to modify a shared page).

ILLINOIS TECH

College of Computing

Memory System Summary

- L1/L2 Memory Cache
 - Purely a speed-up technique
 - Behavior invisible to application programmer and (mostly) OS
 - Implemented totally in hardware
- Virtual Memory
 - Supports many OS-related functions
 - Process creation, task switching, protection
 - Software
 - Allocates/shares physical memory among processes
 - Maintains high-level tables tracking memory type, source, sharing
 - Handles exceptions, fills in hardware-defined mapping tables
 - Hardware
 - Translates virtual addresses via mapping tables, enforcing permissions
 - Accelerates mapping via translation cache (TLB)

Questions?

ILLINOIS TECH

College of Computing

Midterm Review

Ben Lenard

blenard@iit.edu

ILLINOIS TECH

College of Computing

Agenda

- Midterm Review
 - These topics will be on the test but not the exact questions
 - **Do not give short answers when I ask you to explain something; more than 1 sentence.**

ILLINOIS TECH

College of Computing

Midterm

The Midterm will be online from 6:25pm to 7:45pm October 16th 2023
online - I gave you 5 extra minutes for opening it etc.

Join Zoom Meeting for questions

[https://iit-
edu.zoom.us/j/82470358553?pwd=eEI1SHcrWHArQnkzZkw4VlBDMnV
kZz09](https://iit-edu.zoom.us/j/82470358553?pwd=eEI1SHcrWHArQnkzZkw4VlBDMnVkZz09)

ILLINOIS TECH

College of Computing

Virtual memory

- **Virtual memory** – separation of user logical memory from physical memory
 - Only part of the program needs to be in memory for execution
 - Logical address space can therefore be much larger than physical address space
 - Allows address spaces to be shared by several processes
 - Allows for more efficient process creation
 - More programs running concurrently
 - Less I/O needed to load or swap processes

Virtual memory (Cont.)

- **Virtual address space** – logical view of how process is stored in memory
 - Usually start at address 0, contiguous addresses until end of space
 - Meanwhile, physical memory organized in page frames
 - MMU must map logical to physical
- Virtual memory can be implemented via:
 - Demand paging
 - Demand segmentation

Virtual Memory Intuition

Idea: OS keeps unreferenced pages on disk

- Slower, cheaper backing store than memory

Process can run when not all pages are loaded into main memory

OS and hardware cooperate to provide illusion of large disk as fast as main memory

- Same behavior as if all of address space in main memory
- Hopefully have similar performance

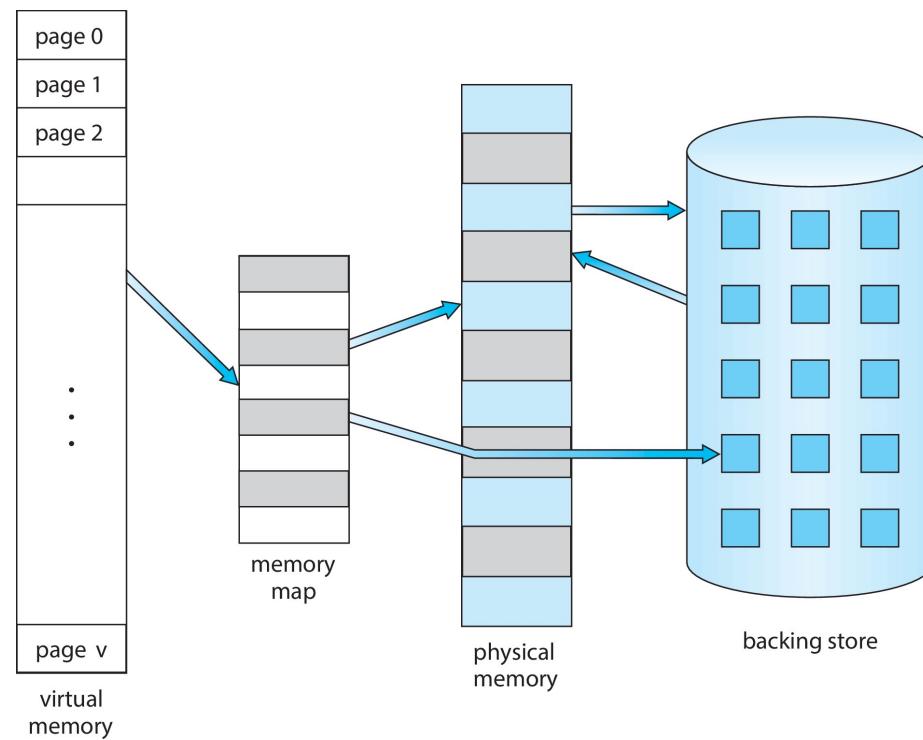
Requirements:

- OS must have **mechanism** to identify location of each page in address space → in memory or on disk
- OS must have **policy** for determining which pages live in memory and which on disk

ILLINOIS TECH

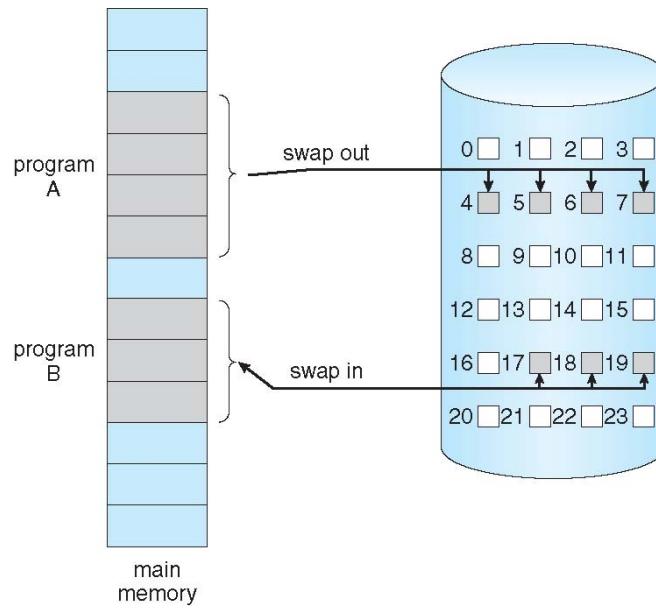
College of Computing

Virtual Memory That is Larger Than Physical Memory



Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
 - Less I/O needed, no unnecessary I/O
 - Less memory needed
 - Faster response
 - More users
- Similar to paging system with swapping (diagram on right)



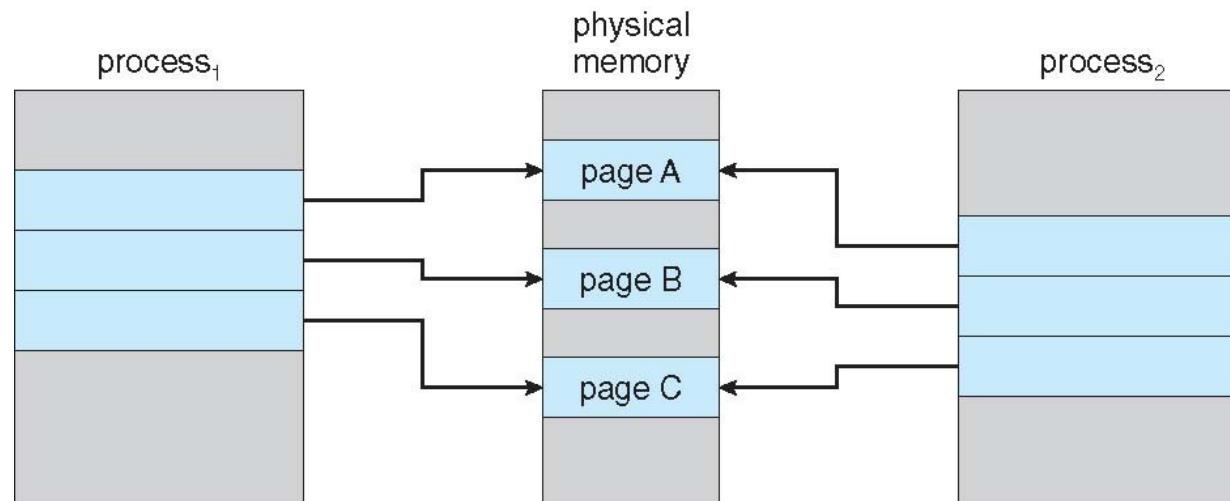
ILLINOIS TECH

College of Computing

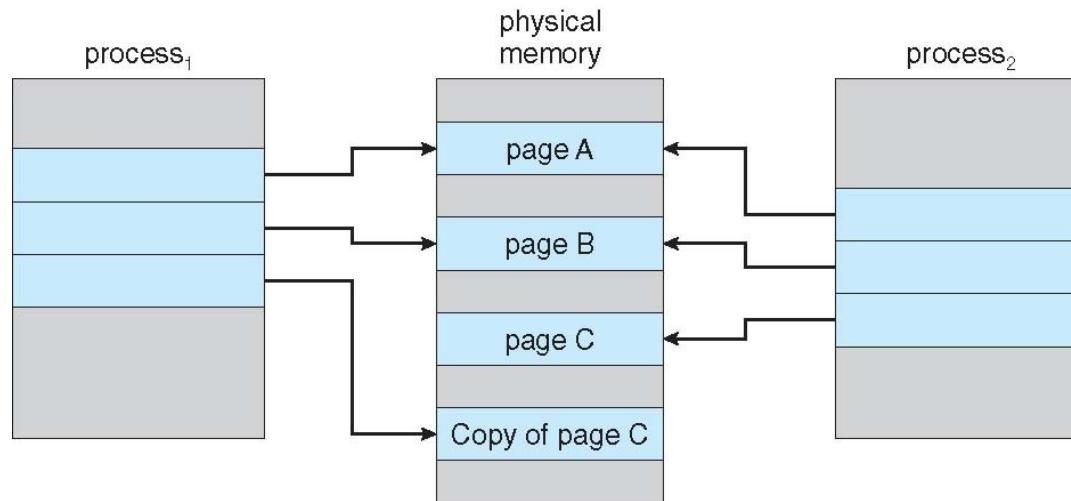
Copy-on-write (COW)

- “Clone” operation is quite common (e.g., used when fork-ing a process)
- But if carried out literally — duplicating entire memory image — is incredibly expensive (and likely unnecessary)
- At clone time, no data is actually copied; simply replicate paging structures and mark pages as read-only
 - Page faults that occur on write accesses trigger copy operation

Before Process 1 Modifies Page C



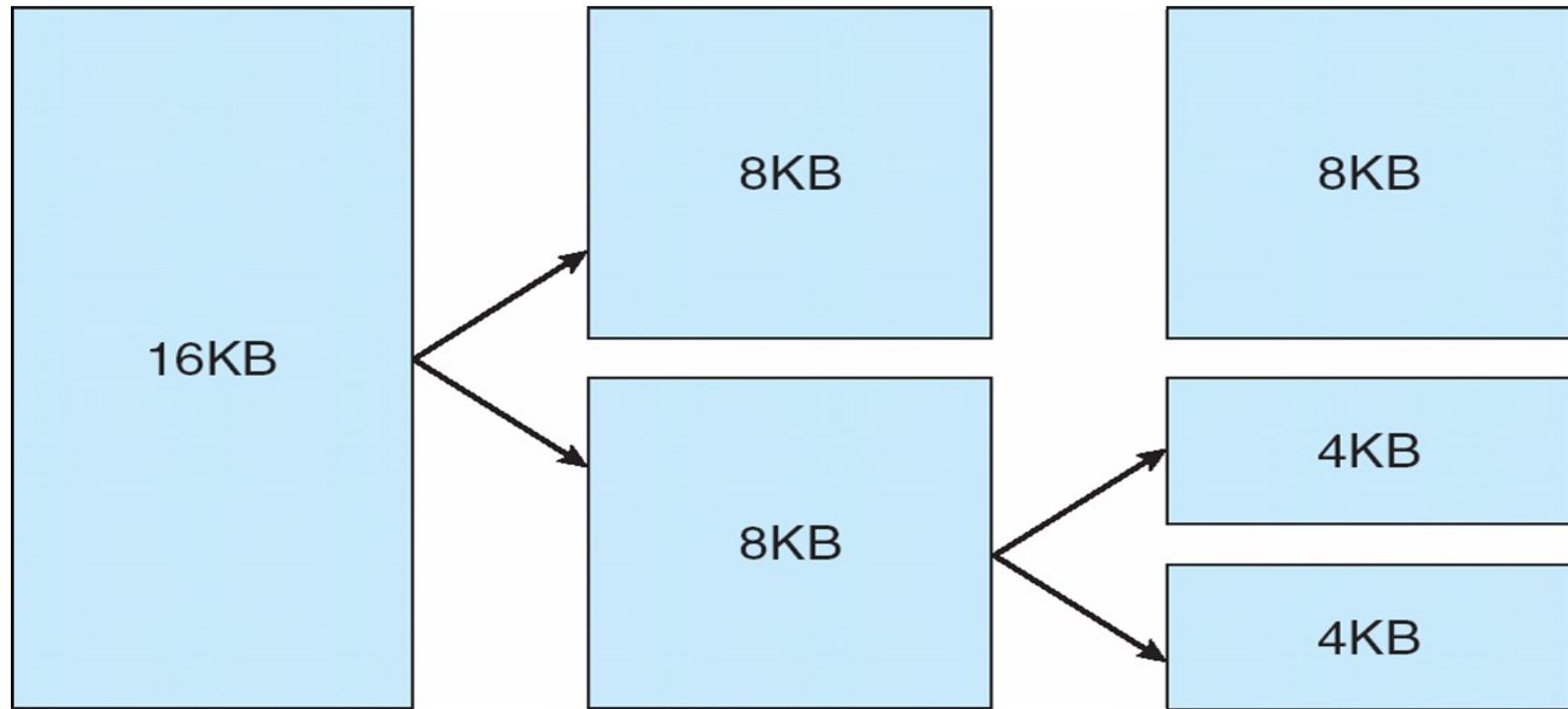
After Process 1 Modifies Page C



ILLINOIS TECH

College of Computing

Splitting of Memory in a Buddy Heap



Buddy system pros & cons

Pros

- Fast allocation — search is easy
- Able to find contiguous blocks
- Good for huge pages
- Can simplify page table updates

Cons

- Small vs. Large blocks creates external fragmentation
- 2^n block sizes can result in significant internal fragmentation
- Compromise: speed vs. efficiency

Where Are Pagetables Stored?

How big is a typical page table?

- assume **32-bit** address space
- assume 4 KB pages
- assume 4 byte entries

Final answer: $2^{(32 - \log(4KB))} * 4 = 4 \text{ MB}$

- Page table size = Num entries * size of each entry
- Num entries = num virtual pages = $2^{\text{bits for vpn}}$
- Bits for vpn = 32 – number of bits for page offset
 $= 32 - \lg(4KB) = 32 - 12 = 20$
- Num entries = $2^{20} = 1 \text{ MB}$
- Page table size = Num entries * 4 bytes = 4 MB

Implication: Store each page table in memory

- Hardware finds page table base with register (e.g., CR3 on x86)

What happens on a context-switch?

- Change contents of page table base register to newly scheduled process
- Save old page table base register in PCB of descheduled process

Other PT info

What other info is in pagetable entries besides translation?

- valid bit
- protection bits
- present bit (needed later)
- reference bit (needed later)
- dirty bit (needed later)

Pagetable entries are just bits stored in memory

- Agreement between hw and OS about interpretation

Page table walk

- The page table is too large to fit into the MMU, so resides in memory
- Translating a VPN → PPN requires indexing into the page table (known as a page table walk)
 - Performed by MMU
 - Page table is managed by the kernel for each process
 - Current process page table is selected by kernel on each context switch (e.g., by pointing a page table base register at it)

Swapping Motivation

OS goal: Support processes when not enough physical memory

- Single process with very large address space
- Multiple processes with combined address spaces

User code should be independent of amount of physical memory

- Correctness, if not performance

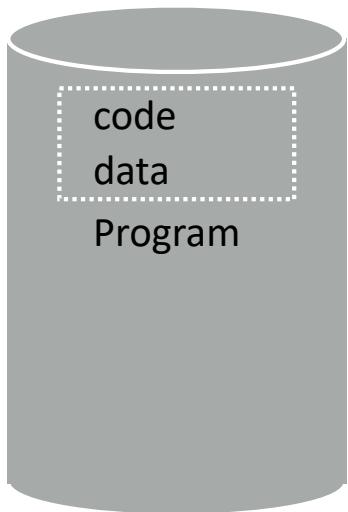
Virtual memory: OS provides illusion of more physical memory

Why does this work?

- Relies on key properties of user processes (workload) and machine architecture (hardware)

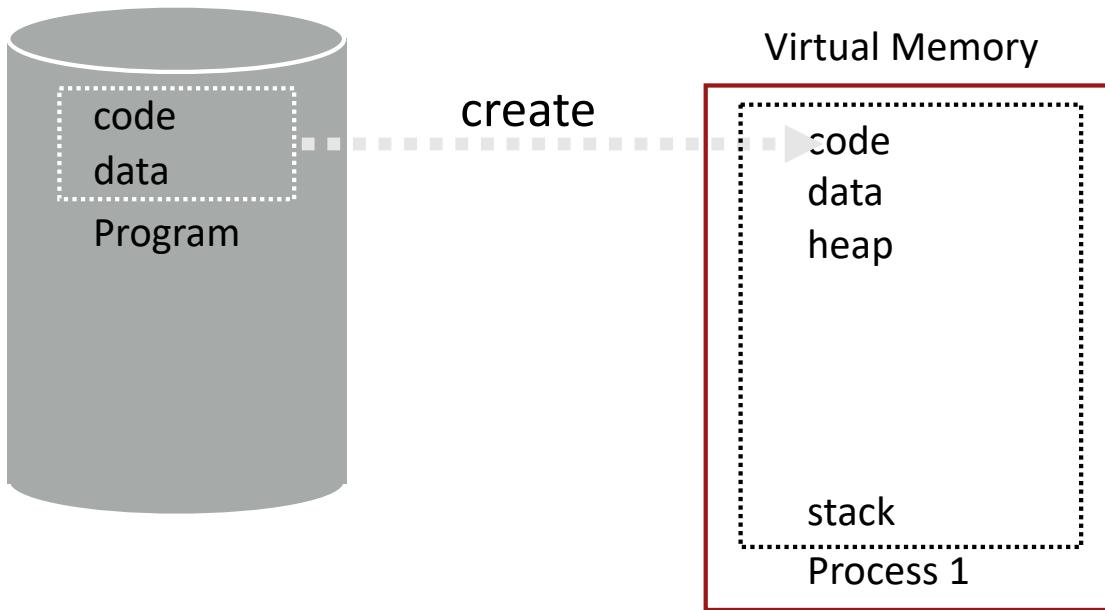
ILLINOIS TECH

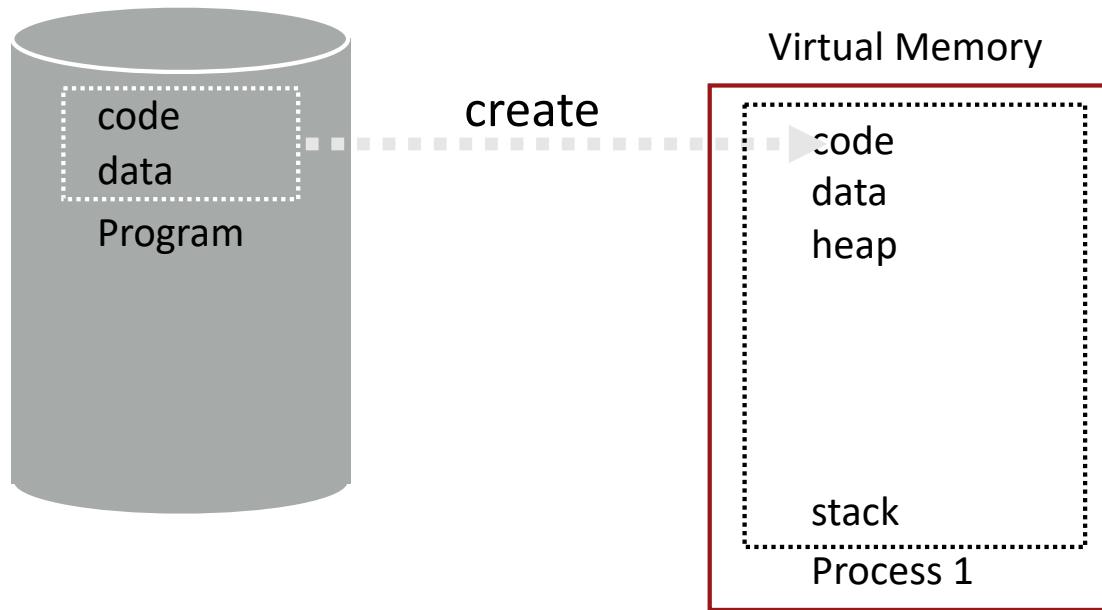
College of Computing



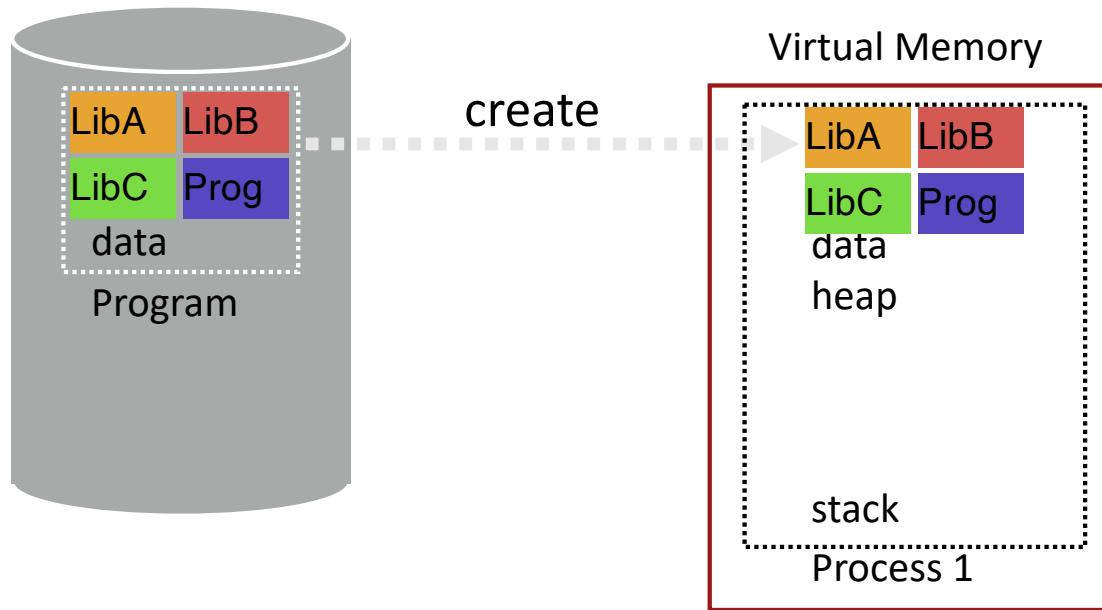
Virtual Memory





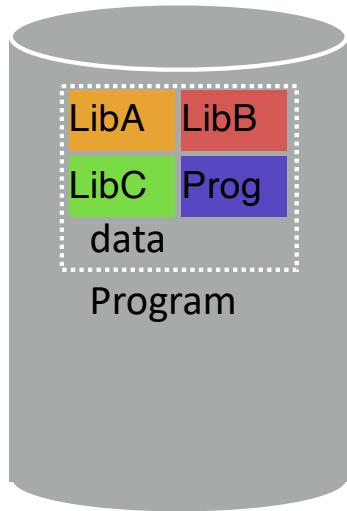


what's in code?



many large libraries, some
of which are rarely/never used

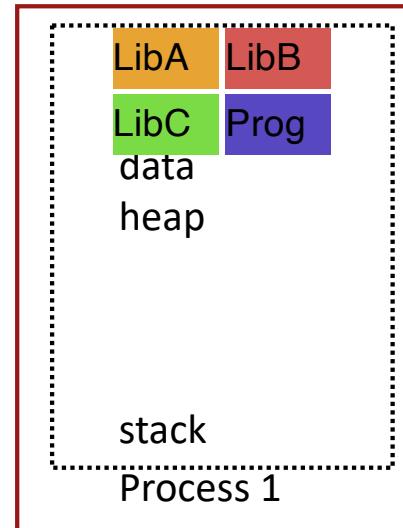
How to avoid wasting physical pages to back rarely used virtual pages?

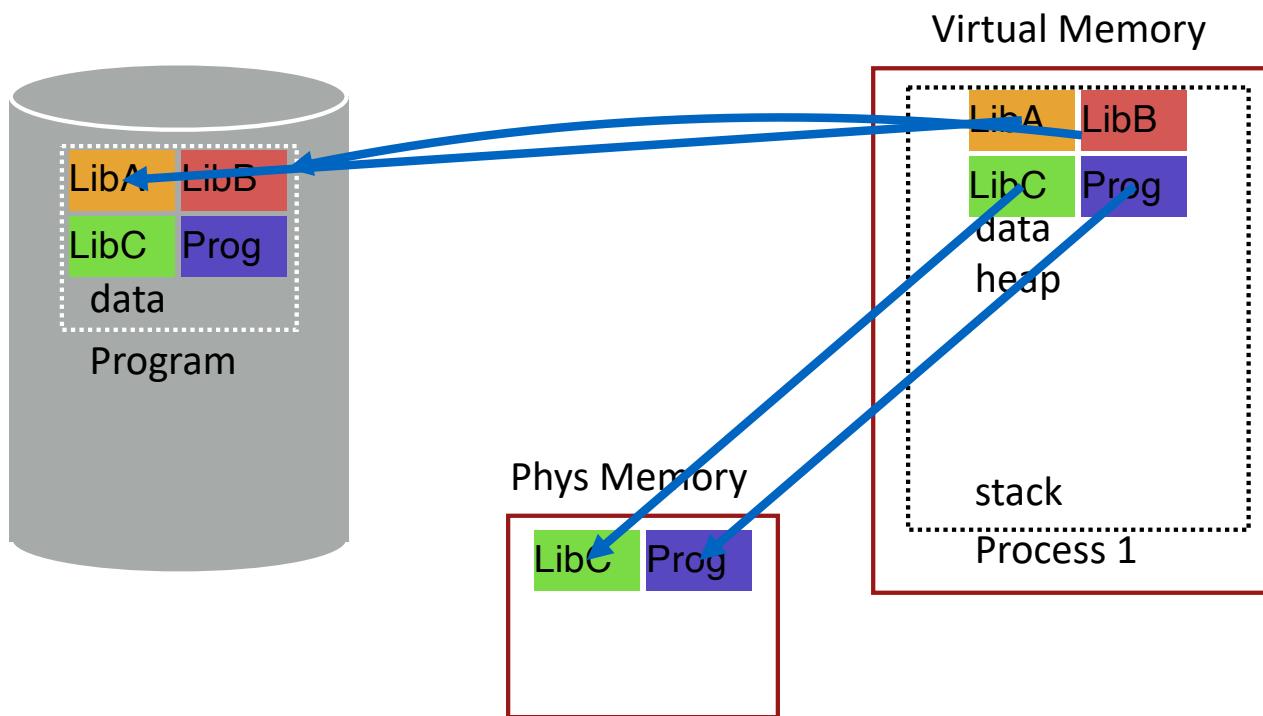


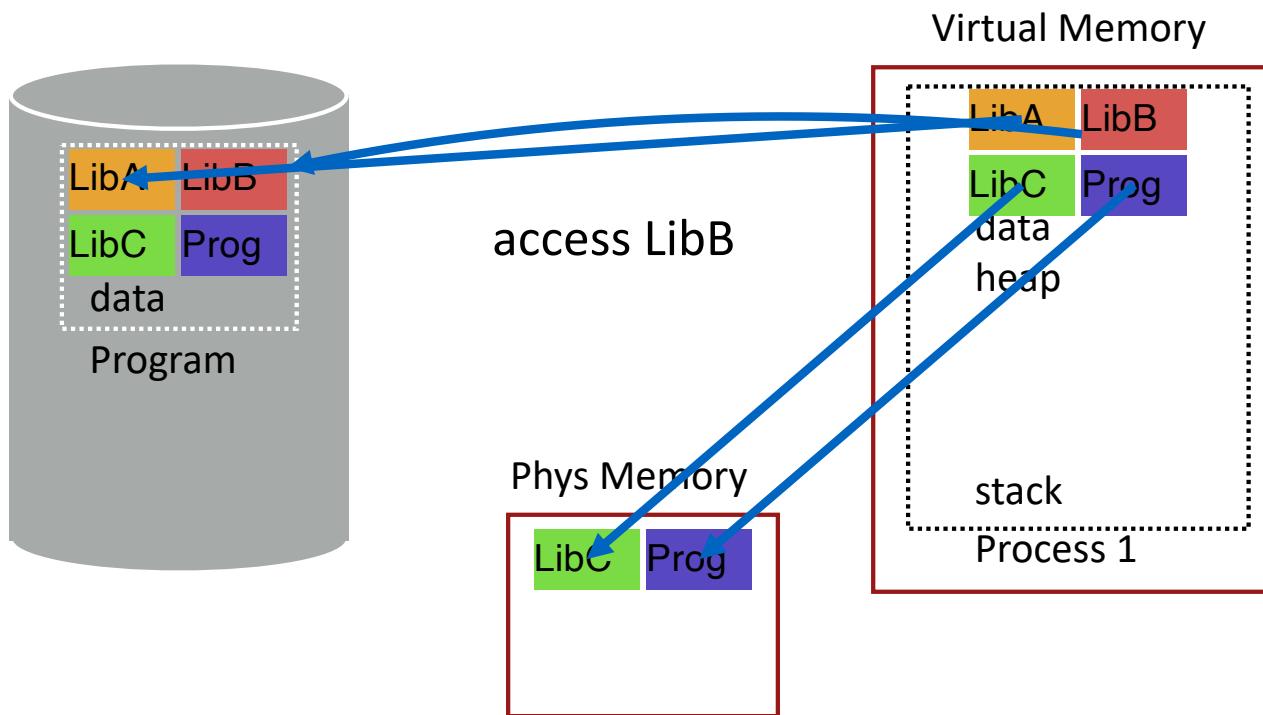
Phys Memory

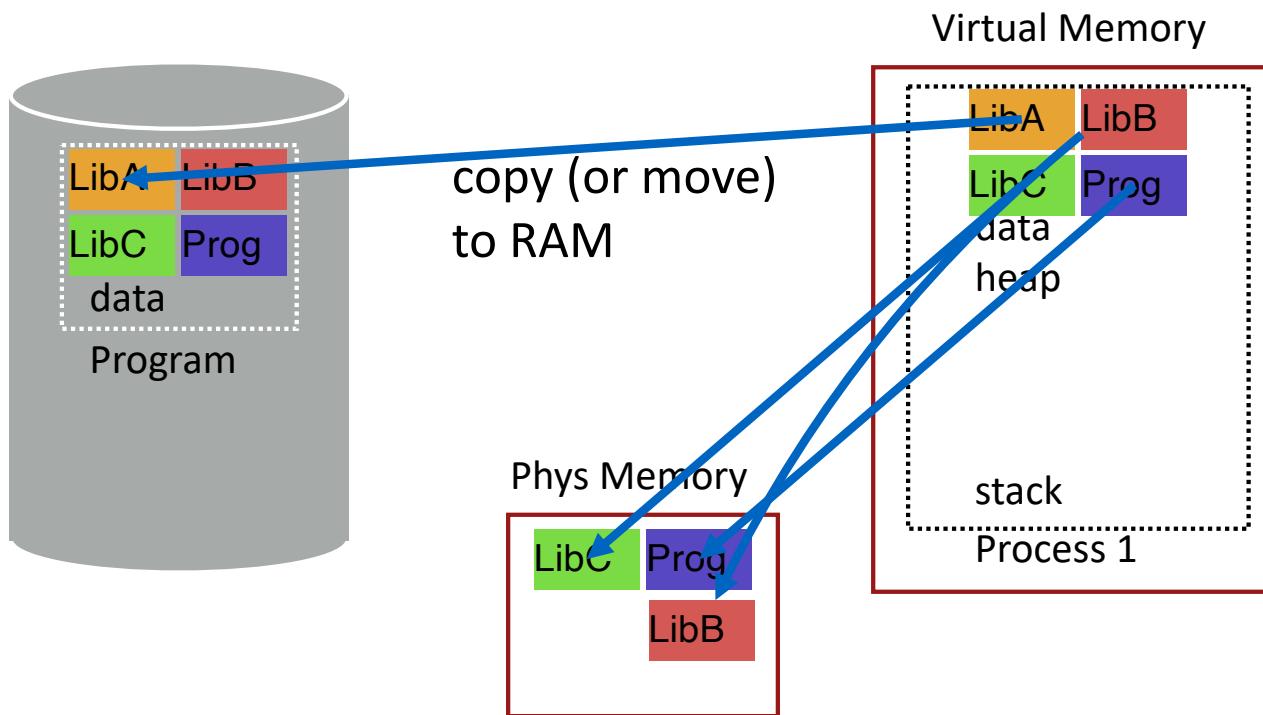


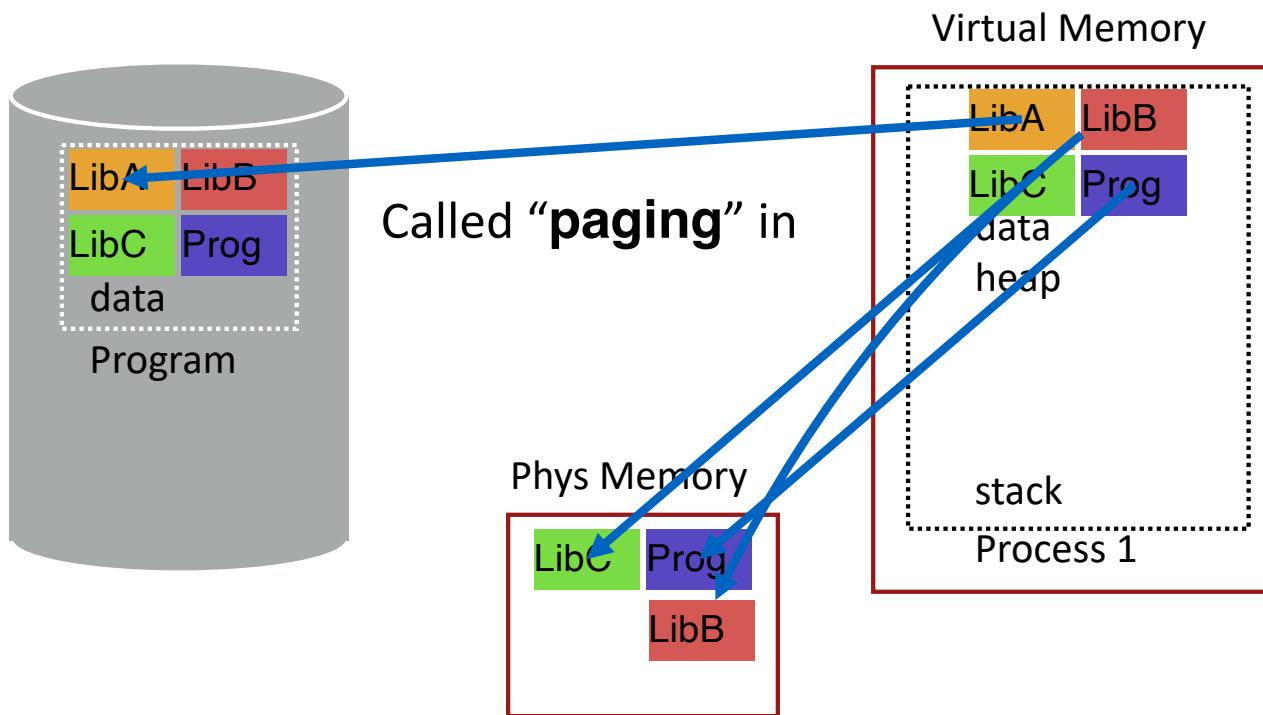
Virtual Memory











Locality of Reference

Leverage **locality of reference** within processes

- **Spatial:** reference memory addresses **near** previously referenced addresses
- **Temporal:** reference memory addresses that have referenced in the past
- Processes spend majority of time in small portion of code
 - Estimate: 90% of time in 10% of code

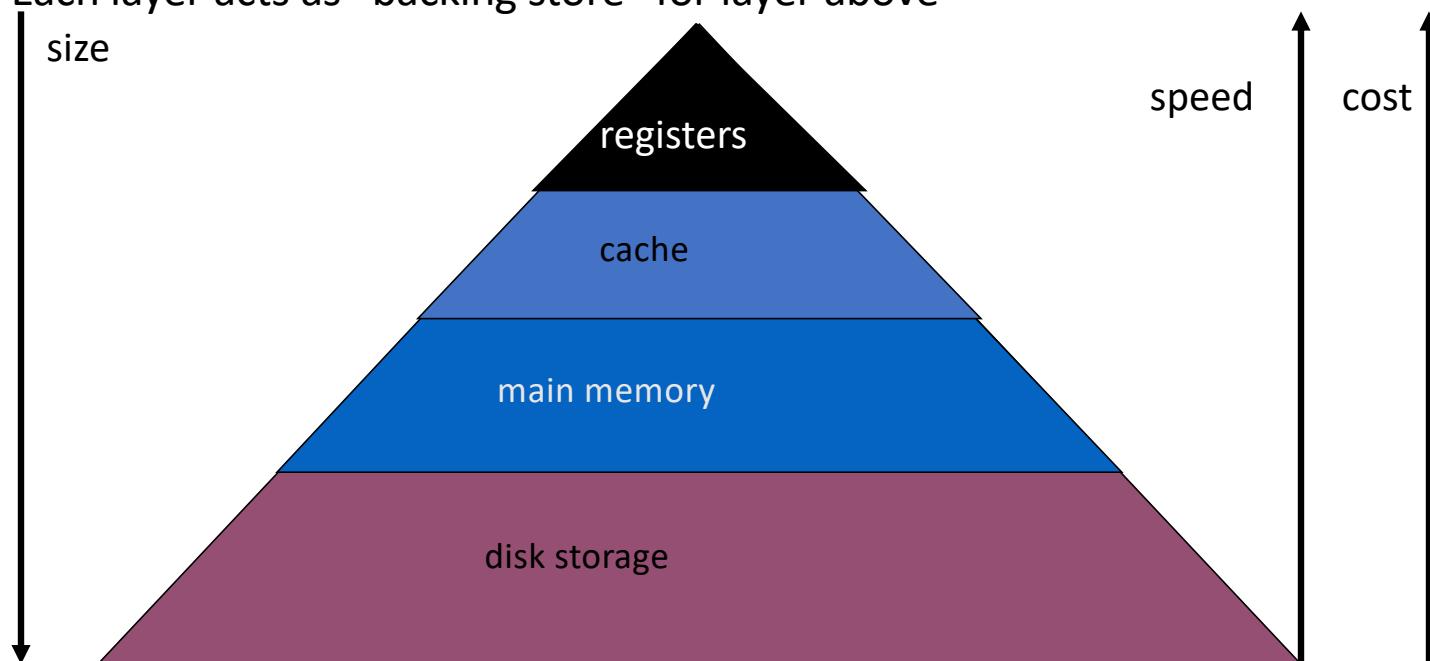
Implication:

- Process only uses small amount of address space at any moment
- Only small amount of address space must be resident in physical memory

Memory Hierarchy

Leverage **memory hierarchy** of machine architecture

Each layer acts as “backing store” for layer above



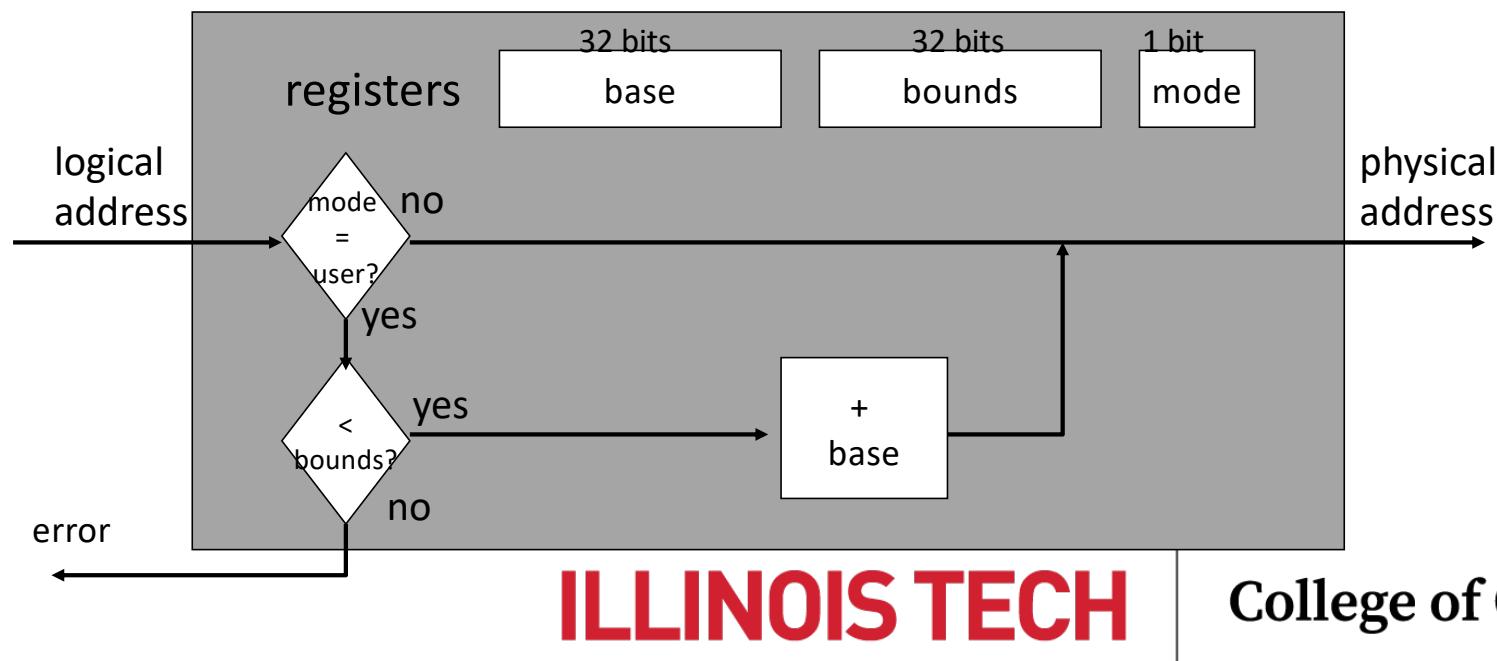
ILLINOIS TECH

College of Computing

Implementation of BASE+BOUNDS

Translation on every memory access of user process

- MMU compares logical address to bounds register
 - if logical address is greater, then generate error
- MMU adds base register to logical address to form physical address



ILLINOIS TECH

College of Computing

Base address

- Kernel maintains base address of each process in PCB
 - Load into base (address) register in MMU on each context switch
 - Relocation = register access + addition
- Problem: protection not guaranteed!

Base + Limit registers

- Incorporate a limit register to enforce memory protection
- Assertion failure triggers fault (software exception) and loads kernel

Managing Processes with Base and Bounds

Context-switch

- Add base and bounds registers to PCB
- Steps
 - Change to privileged mode
 - Save base and bounds registers of old process
 - Load base and bounds registers of new process
 - Change to user mode and jump to new process

What if don't change base and bounds registers when switch?

Protection requirement

- User process cannot change base and bounds registers
- User process cannot change to privileged mode

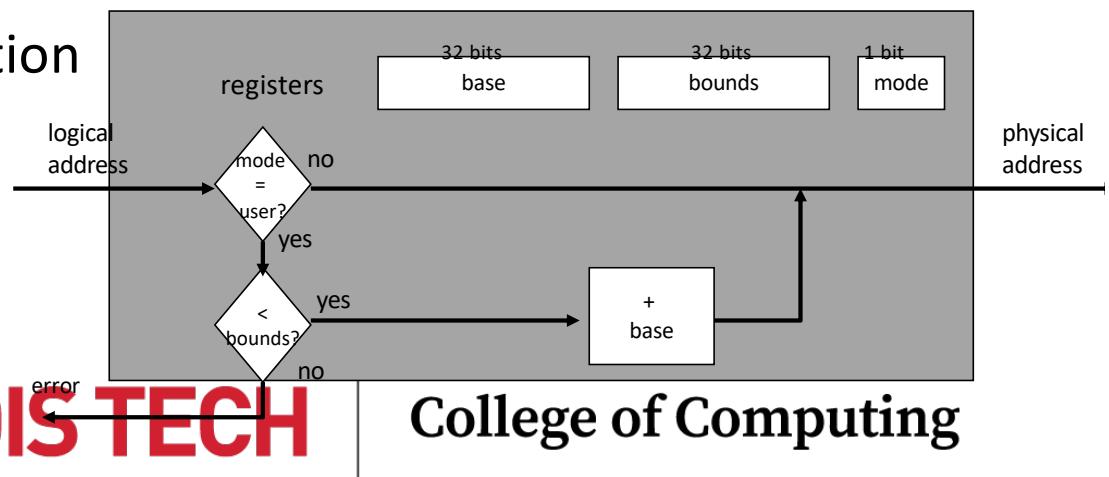
ILLINOIS TECH

College of Computing

Base and Bounds Advantages

Advantages

- Provides protection (both read and write) across address spaces
- Supports dynamic relocation
 - Can place process at different locations initially and also move address spaces
- Simple, inexpensive implementation
 - Few registers, little logic in MMU
- Fast
 - Add and compare in parallel



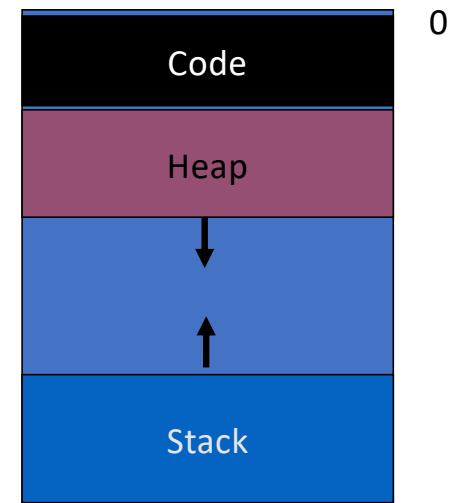
ILLINOIS TECH

College of Computing

Base and Bounds DISADVANTAGES

Disadvantages

- Each process must be allocated contiguously in physical memory
 - Must allocate memory that may not be used by process
- No partial sharing: Cannot share limited parts of address space



ILLINOIS TECH

College of Computing $_{2^{n-1}}$

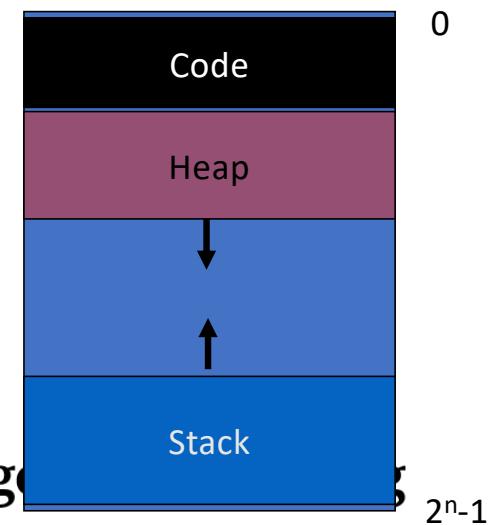
Segmentation

Divide address space into logical segments

- Each segment corresponds to logical entity in address space
 - code, stack, heap

Each segment can independently:

- be placed separately in physical memory
- grow and shrink
- be protected (separate read/write/execute protection bits)



Segmentation (Cont.)

- Partition virtual address space into multiple disjoint segments
 - Individually map onto physical memory with separate base/limit registers
 - Address space info stored in PCB and restored on context switch - Requires that memory requests are for segmented addresses
- Consist of segment selector and offset into segment
- Alternatively: segment can be implied by instruction (e.g., PC always refers to code segment)

Segmented Addressing

Process now specifies segment and offset within segment

How does process designate a particular segment?

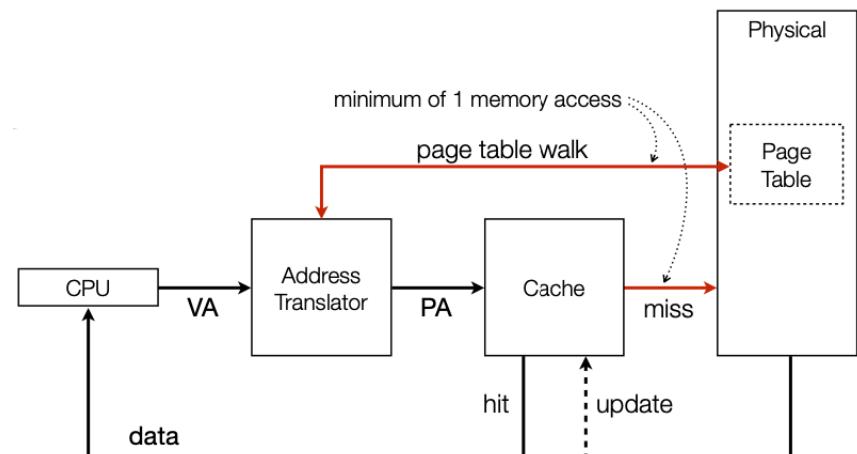
- Use part of logical address
 - Top bits of logical address select segment
 - Low bits of logical address select offset within segment

What if small address space, not enough bits?

- Implicitly by type of memory reference
- Special registers

Page table translations are slow!

- Most modern caching systems are physically addressed, so we cannot avoid translation before cache lookup
- I.e., each VA access requires up to two memory accesses!

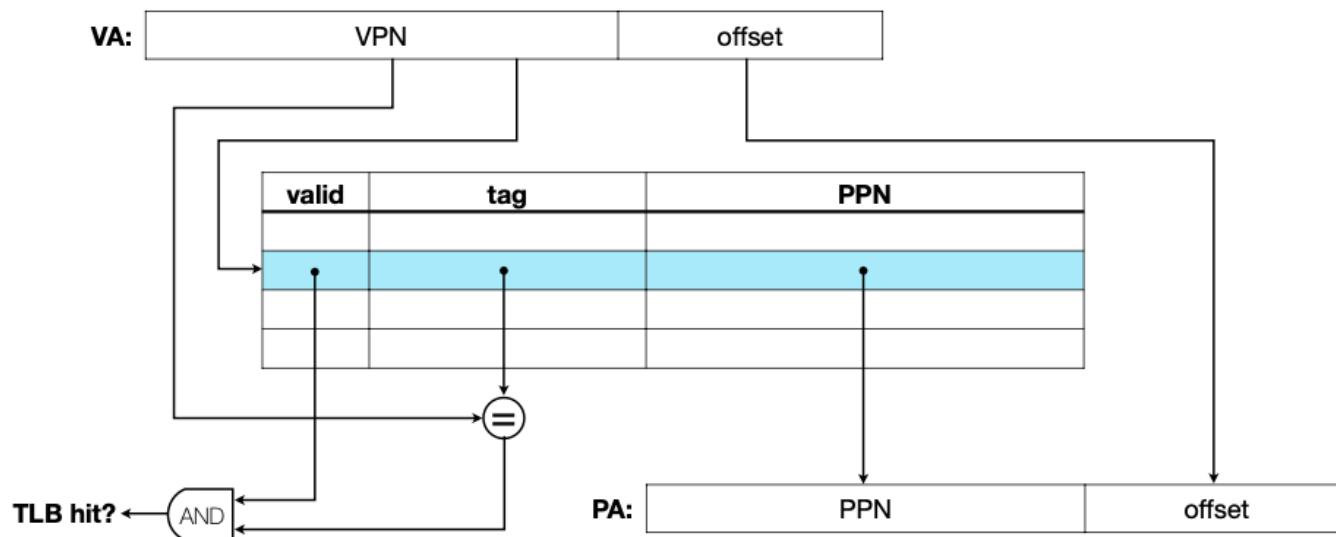


ILLINOIS TECH

College of Computing

Translation Lookaside Buffer (TLB)

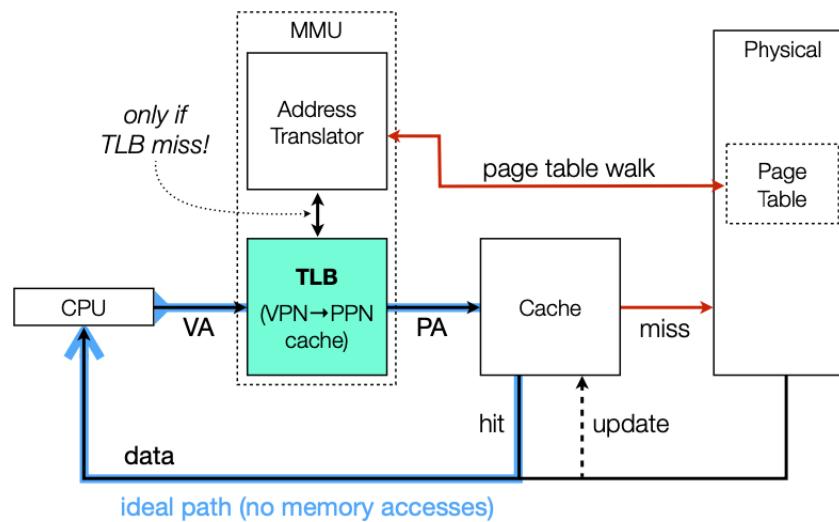
- Solution: dedicated cache for VPN → PPN translations
 - Page table walk only performed on TLB miss



ILLINOIS TECH

College of Computing

TLB / Cache / PT interaction



ILLINOIS TECH

College of Computing

TLB issues

- TLB mappings are process specific — requires flush on context switch
 - Some architectures store address space identifier per cache line
- TLB only caches a few thousand mappings, at most
 - vs. orders of magnitude more per process, potentially!
 - Effectiveness of TLB can be “tuned” by adjusting number of pages (larger page size = smaller number of pages)
 - Downside to large pages?

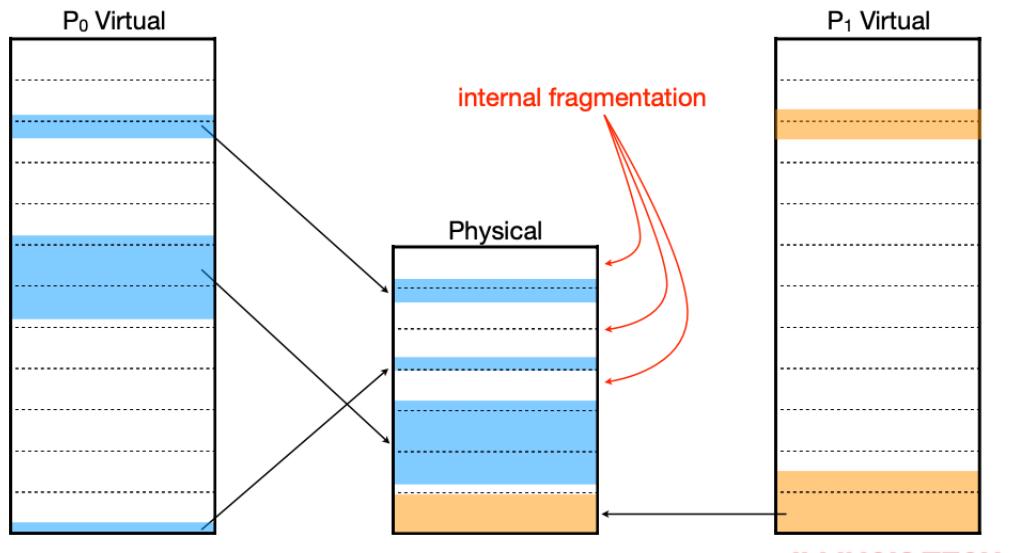
Internal fragmentation

- Large pages result in coarser mapping granularity
 - I.e., larger “chunks” carved out of physical memory at a time
 - May lower utilization, if large portions of pages are not used — known as internal fragmentation
- Must balance TLB effectiveness against memory utilization
- Depends on the application and workload

Internal fragmentation

- Large pages result in coarser mapping granularity
 - I.e., larger “chunks” carved out of physical memory at a time
 - May lower utilization, if large portions of pages are not used — known as internal fragmentation
- Must balance TLB effectiveness against memory utilization
- Depends on the application and workload

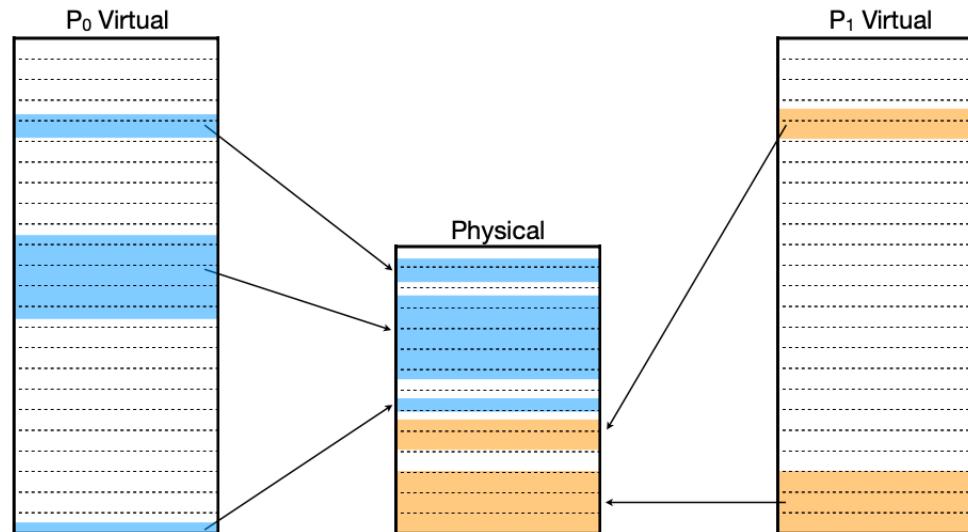
E.g., large(-ish) pages



ILLINOIS TECH

College of Computing

E.g., small(-ish) pages



ILLINOIS TECH

College of Computing

Address space = a lie!

- If processes are simultaneously accessing physical memory ... - Not all text sections can begin at 0x400000
 - Not all data sections can begin at 0x600000
 - Not all heaps can begin at 0x18f0000
 - Not all stacks can begin at 0x7fff00000000
- Uniform process address spaces are an illusion created by the kernel
 - To simplify program loading and execution (among other reasons)

Hardware Support for Dynamic Relocation

Two operating modes

- Privileged (protected, kernel) mode: OS runs
 - When enter OS (trap, system calls, interrupts, exceptions)
 - Allows certain instructions to be executed
 - Can manipulate contents of MMU
 - **Allows OS to access all of physical memory**
- User mode: User processes run
 - Perform translation of logical address to physical address

Minimal MMU contains **base register** for translation

- base: start location for address space

Scheduling: Proportional Share

- proportional-share scheduler or fair-share scheduler.
- Proportional-share is based around a simple concept: instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain percentage of CPU time.

Lottery Scheduling

- Simple Idea, at a given interval, hold a lottery to determine which process should get to run next; processes that should run more often should be given more chances to win the lottery.
- How can we design a scheduler to share the CPU in a proportional manner?
- What are the key mechanisms for doing so?
- How effective are they?

Tickets Represent Your Share

- Tickets represent your share
- Consider 10 people each one gets 1 ticket, verses 2,3,4,5 for one person
- Let's say we add timeslices, or cpu time
- Lottery Scheduling is probabilistic not deterministic
- It uses randomness

Tickets Represent Your Share (cont.)

- A gets 0 – 74
- B gets 75 - 99

63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43 0 49 12

Here is the resulting schedule:

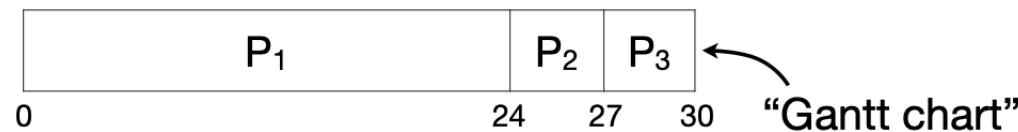
A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
B		B							B		B								

ILLINOIS TECH

College of Computing

First come first served (FCFS)

Process	Arrival Time	Burst Time
P ₁	0	24
P ₂	0	3
P ₃	0	3



Wait times: P₁ = 0, P₂ = 24, P₃ = 27

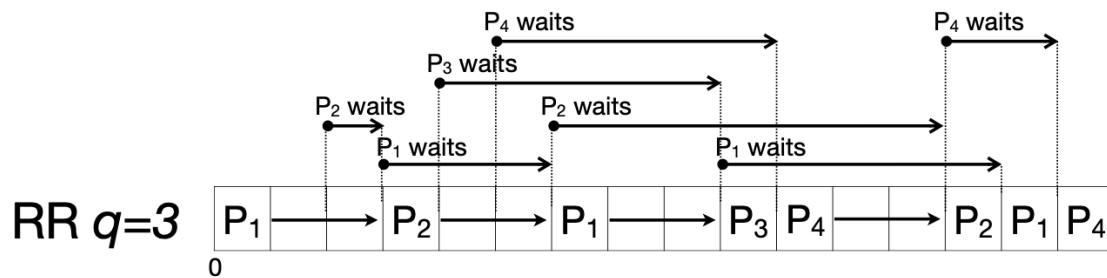
Average: (0 + 24 + 27) / 3 = 17

Round Robin (RR)

- The “fairest” of them all
- Uses a FIFO queue:
 - Each job runs for a maximum fixed time quantum q
 - If unfinished, re-enter queue at the tail end
- Given time quantum q and n jobs:
 - max wait time (per cycle) = $q \cdot (n - 1)$
 - each job receives $1/n$ timeshare
- RR is also used for other applications such as load balancing and dns

RR (cont.)

Process	Arrival Time	Burst Time
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4



Wait times: P₁ = 8, P₂ = 8, P₃ = 5, P₄ = 7

Average: $(8 + 8 + 5 + 7) / 4 = 7$

RR (cont.)

Process	Arrival Time	Burst Time
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4

	Avg. Turnaround	Avg. Wait Time
RR $q=7$	8.75	4.75
RR $q=4$	9	5
RR $q=3$	11	7
RR $q=1$	9.75	5.75

(FCFS)

ILLINOIS TECH

College of Computing

RR (cont.)

<i>Process</i>	<i>Arrival Time</i>	<i>Burst Time</i>
P ₁	0	7
P ₂	2	4
P ₃	4	1
P ₄	5	4

	Throughput	Utilization
RR q=7	0.25	1.0
RR q=4	0.25	1.0
RR q=3	0.25	1.0
RR q=1	0.25	1.0

ILLINOIS TECH

College of Computing

Greedy algorithms

- SJF/PSJF are greedy algorithms
 - i.e., they select the best choice at the moment (“local maximum”)
- Greedy algorithms don’t always produce globally maximal results
 - e.g., naive hill-climbing algorithm (only take a step if it brings me to higher ground) doesn’t always find the tallest peak!
- Are SJF/PSJF optimal?

Is It Optimal?

- Consider 4 jobs with burst lengths t_0, t_1, t_2, t_3 that just became ready
- What is the average wait time if scheduled in the order given?
 - $= (3 \cdot t_0 + 2 \cdot t_1 + t_2) / 4$
 - Weighted average — clearly minimized by running shortest jobs first!
- SJF/PSJF are provably optimal with respect to average wait time
 - But at what cost?
 - Potential CPU starvation! (e.g., longer jobs keep getting put off)

No way to tell the future

- We've been assuming that job/burst lengths are known in advance
- May be possible in rare circumstances (e.g., repeated jobs, job profiling), but unlikely in practice
- Common approach: predict future burst lengths based on past behavior
 - Simple moving average (sliding window of past values)
 - Exponentially weighted moving average (EMA)

Preemptive SJF (PSJF)

- aka “Shortest Time-to-Completion First” (STCF)
- aka “Shortest Remaining-Time First” (SRTF)
- May preempt running job to schedule a different (ready) job

Some scheduling metrics

- Turnaround time
- Wait time
- Response time
- Throughput
- Utilization

Definition

- Scheduling: policies & mechanisms used to allocate limited resources to some set of entities
- Initial focus: resource & entities = CPU & processes (aka jobs) - other possibilities:
 - resources: memory, I/O bus/devices
 - entities: threads, users, groups
- schedulers for the above may exist in an OS (and must play nice with each other)!

Policy

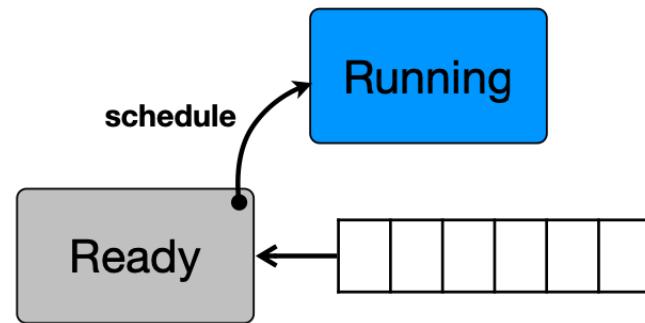
- high-level “what”
- scheduling disciplines
 - e.g., FCFS, SJF, RR, etc.
- driven by a variety of potentially conflicting goals
 - e.g., performance and fairness

Mechanism

- low-level “how”
- combination of HW/SW
 - e.g., clock interrupt, high precision timer, PCB
- scattered throughout kernel codebase

Ready queue isn't always FIFO

- convenient to envision a ready queue (though not necessarily FIFO!)
- the scheduling policy decides which job to select from the set of ready (runnable) jobs to run next



Privilege Escalation

- As it sounds, when a user needs privileges greater than what they have.
- In other words, corner case handling
- Consider sudo, do you need sudo all the time?

ILLINOIS TECH

College of Computing

Aspects Of The Access Control

- Subjects - A subject is the entity that wants to perform the access, perhaps a user or a process.
- Objects - An object is the thing the subject wants to access, perhaps a file or a device.
- Access - Access is some particular mode of dealing with the object, such as reading it or writing it.
- We sometimes refer to the process of determining if a particular subject is allowed to perform a particular form of access on a particular object as authorization.

Security Goals at a high level

- Confidentiality – Only showing data to the people/apps with the correct permissions – this is different then chmod 777
 - Integrity – Ensure something is correct and not altered
 - Availability – Ensuring apps/people have access to the data.
-
- These areas are covered on the CISSP exam

ILLINOIS TECH

College of Computing

Non-Repudiation

- All about the proof
- Proof that someone received a message or file and cannot say they never received it.
- This will come back with service accounts

Designing Secure Systems

1. **Economy of mechanism** – KISS
2. **Fail-safe defaults** – Default to security, not insecurity. Similar to the credit card networks, default is decline
3. **Complete mediation** – This is a security term meaning that you should check if an action to be performed meets security policies every single time the action is taken. Often ignored.
4. **Open design** – Assume your adversary knows every detail of your design.
5. **Separation of privilege** – Require separate parties or credentials to perform critical actions. Similar to SOX in banking? Why do we have SOX?
6. **Least privilege** – Smaller attack vectors. Similar to why we have userspace.
7. **Least common mechanism** – For different users or processes, use separate data structures or mechanisms to handle them. Aka don't share. We'll talk about containers and switch root.
8. **Acceptability** – If your users won't use it, your system is worthless. Aka the sales pitch

ILLINOIS TECH

College of Computing

Ulimits – Linux – A user

```
$ ulimit -a
core file size      (blocks, -c) 0
data seg size       (kbytes, -d) unlimited
scheduling priority (-e) 0
file size           (blocks, -f) unlimited
pending signals     (-i) 384666
max locked memory   (kbytes, -l) 64
max memory size     (kbytes, -m) unlimited
open files          (-n) 1024
pipe size           (512 bytes, -p) 8
POSIX message queues (bytes, -q) 819200
real-time priority   (-r) 0
stack size           (kbytes, -s) 8192
cpu time             (seconds, -t) unlimited
max user processes    (-u) 4096
virtual memory        (kbytes, -v) unlimited
file locks            (-x) unlimited
```

ILLINOIS TECH

College of Computing

Ulimits - root

```
ulimit -a
core file size      (blocks, -c) 0
data seg size       (kbytes, -d) unlimited
scheduling priority (-e) 0
file size           (blocks, -f) unlimited
pending signals     (-i) 384666
max locked memory   (kbytes, -l) 64
max memory size     (kbytes, -m) unlimited
open files          (-n) 1024
pipe size           (512 bytes, -p) 8
POSIX message queues (bytes, -q) 819200
real-time priority   (-r) 0
stack size           (kbytes, -s) 8192
cpu time             (seconds, -t) unlimited
max user processes   (-u) 384666
virtual memory        (kbytes, -v) unlimited
file locks            (-x) unlimited
```

ILLINOIS TECH

College of Computing

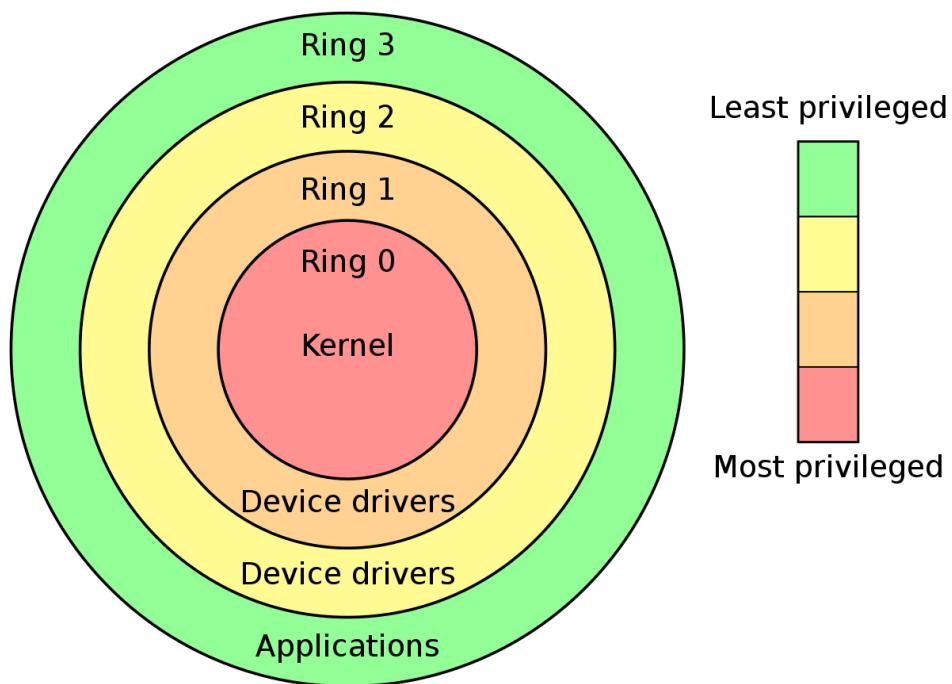
Limited Direct Execution

- Must prevent user from:
 - Accessing arbitrary memory addresses
 - Executing “dangerous” instructions
 - Access to I/O directly
 - System registers

Remember Kernel vs User Mode?

- Privileged instructions can only be executed in kernel mode
 - (what happens when user attempts to run?)
- On x86: ring/CPL flag in Code Selector register, 4 modes but 2 are commonly used: 0 = kernel, 3 = user
- After system boot, OS switches to user mode before delegating control to process

Protection Ring



ILLINOIS TECH

College of Computing

Restricted Operation

- What if a process wishes to perform some kind of restricted operation such as ...
 - Issuing an I/O request to a disk
 - Gaining access to more system resources such as CPU or memory
- Solution: Using protected control transfer (processor has to support it)
 - User mode: Applications do not have full access to hardware resources.
 - Kernel mode: The OS has access to the full resources of the machine

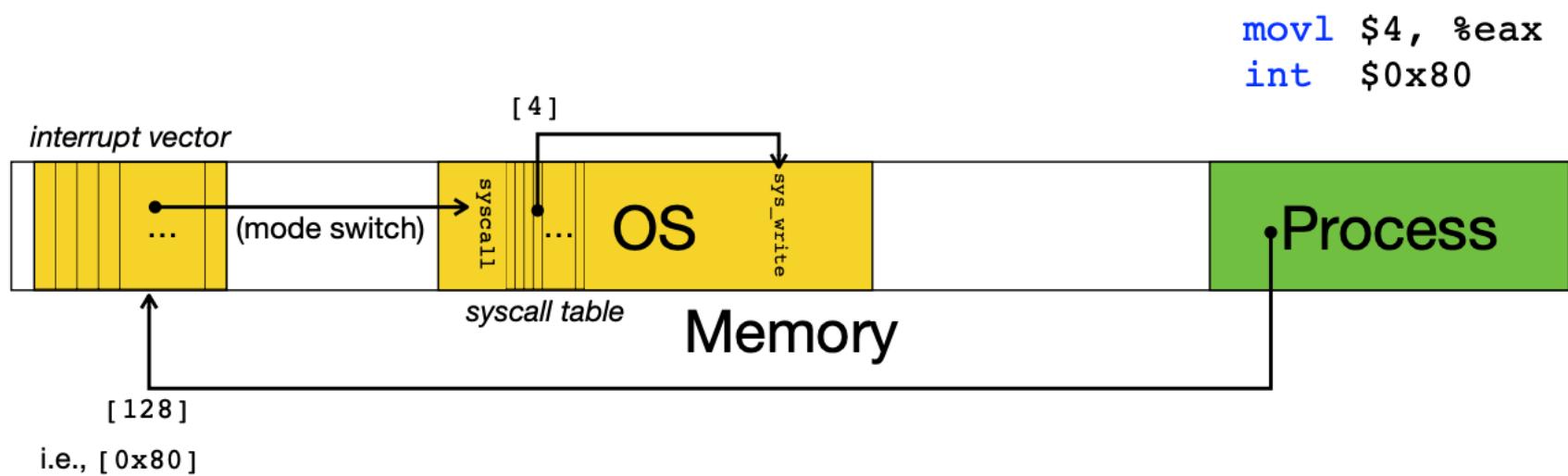
System Calls

- When user needs to perform I/O, invoke kernel-mode OS functions via system calls:
 - Accessing the file system
 - Creating and destroying processes
 - Communicating with other processes
 - Allocating more memory
- Looks like a regular function call, but isn't!

System Calls (Cont.)

char *str = "hello world";	movl len, %edx
int len = strlen(str);	movl str, %ecx
write(1, str, len);	movl \$1, %ebx
	movl \$4, %eax # syscall num
	int \$0x80 # trap instr

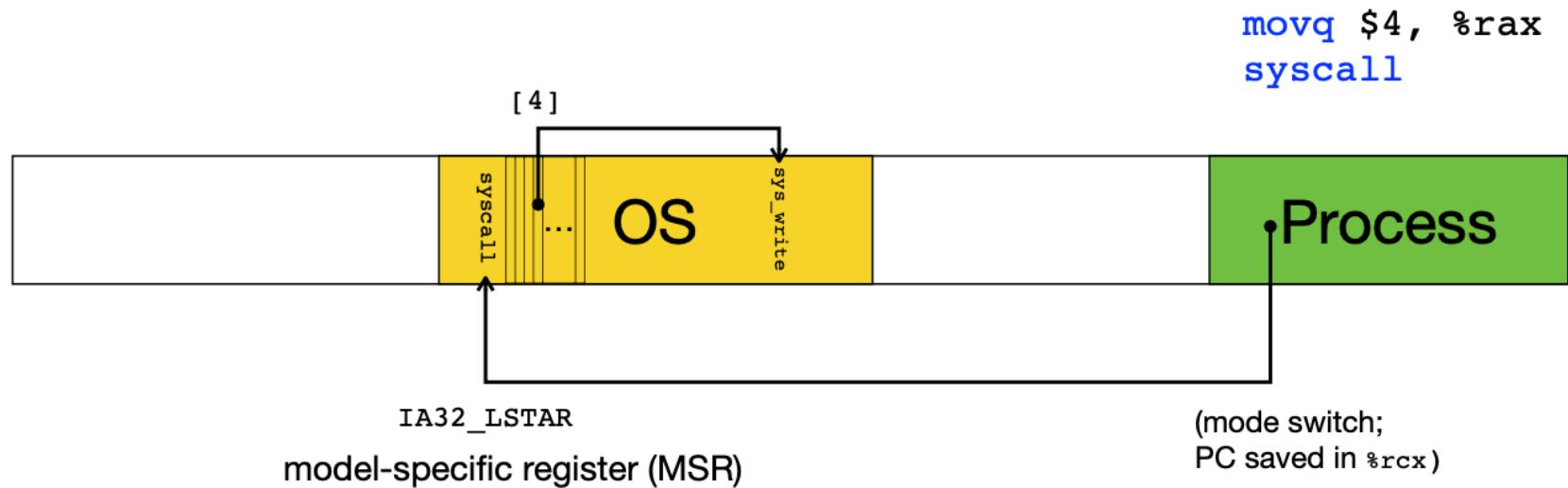
Trap Mechanism (x86)



ILLINOIS TECH

College of Computing

Trap Mechanism (x86-64) syscall added



ILLINOIS TECH

College of Computing

SYSTEM CALLS / TRAPS

- Defensive programming is key
- Why?
- Each call has its own number

ILLINOIS TECH

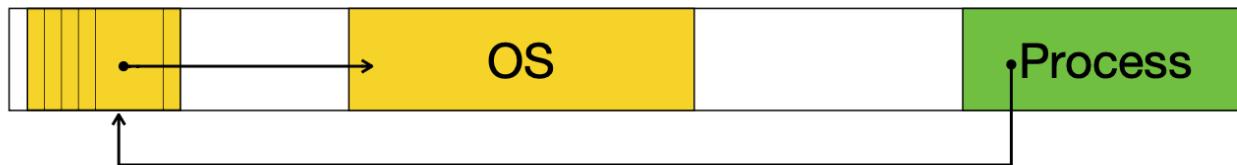
College of Computing

General Interrupt Mechanism

- IDTR (base address register) — populated by privileged lidtr instruction
 - 0-31 reserved for CPU-generated
 - 32-255 software configurable (for sw/hw interrupts) – not all are from User Mode

What to do?

- Problem: when transitioning to OS code, process state may be lost (e.g., PC, SP, etc.)
- Should save in case we return to process after servicing trap

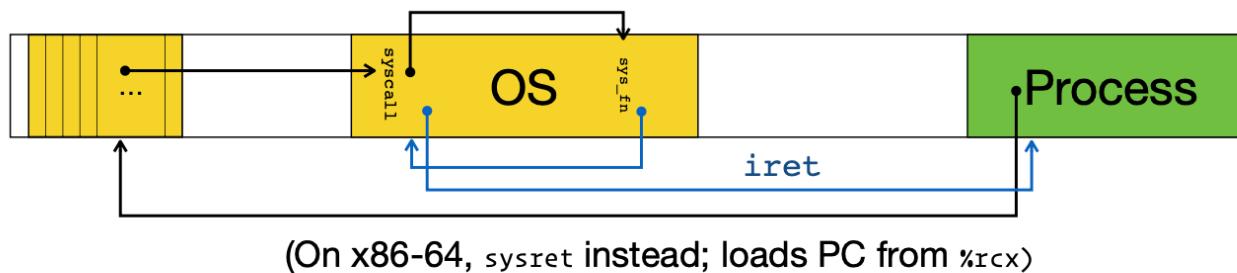


Saving Process State

- Hardware automatically saves current context during trap
- Where?
 - On kernel stack — automatically activated on mode switch
- Every process has its own separate kernel stack and it is used to keep track of kernel state (e.g., while handling I/O)

Restoring Process State

- “return from trap” instruction: iret — pops and restores trap frame and returns to process in user mode



Do we always immediately return to trapping process?

- Nope:
 - Process may be blocked (due to I/O request)
 - Scheduling decision

System Call (Cont.)

- Trap instruction
 - Jump into the kernel (how to tell where?)
 - Raise (the processor) privilege level to kernel mode
- Return-from-trap instruction
 - Return into the calling user program
 - Reduce (the processor) privilege level back to user mode

System Call (Cont.)

OS @ boot (kernel mode)	Hardware	
initialize trap table	remember address of ... syscall handler	
OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC return-from -trap	restore regs from kernel stack move to user mode jump to main	Run main() ... Call system trap into OS

ILLINOIS TECH

College of Computing

System Call (Cont.)

OS @ run (kernel mode)	Hardware	Program (user mode)
<i>(Cont.)</i>		
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall return-from-trap	restore regs from kernel stack move to user mode jump to PC after trap	
		... return from main trap (via exit())
Free memory of process Remove from process list		

ILLINOIS TECH

College of Computing

Switching Between Processes

- How can the OS regain control of the CPU so that it can switch between processes?
 - A cooperative Approach: Wait for system calls
 - A Non-Cooperative Approach: The OS takes control

A cooperative Approach: Wait for system calls

- Processes periodically give up the CPU by making system calls such as yield.
 - The OS decides to run some other task.
 - Application also transfer control to the OS when they do something illegal. i.e. Divide by zero
 - Try to access memory that it shouldn't be able to access
- Early versions of the Macintosh OS, The old Xerox Alto system

What can go wrong?

- A process gets stuck in an infinite loop = Reboot the machine

A Non-Cooperative Approach: OS Takes Control

- A timer interrupt
- During the boot sequence, the OS starts the timer (hardware).
- The timer raises an interrupt every so many milliseconds. (hardware)
- When the interrupt is raised :
 - The currently running process is halted.
 - Save enough of the state of the program.
 - A pre-configured interrupt handler in the OS runs.

What does this fix?

- The OS can jump back into execution.

ILLINOIS TECH

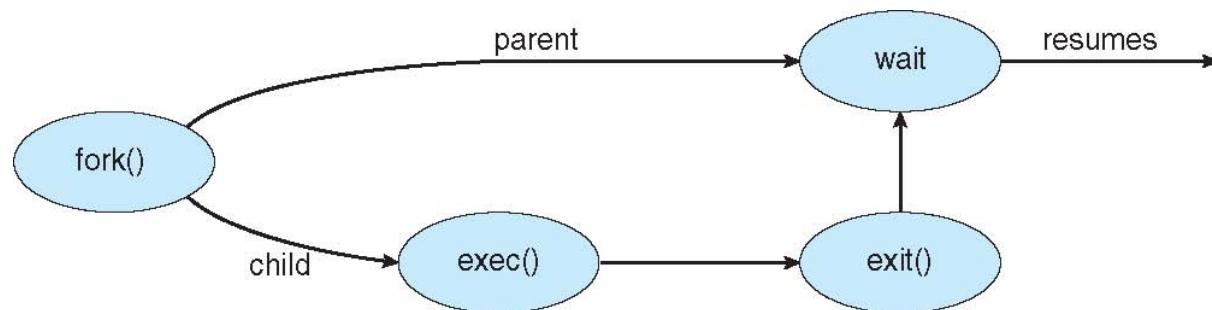
College of Computing

Context Switch

- A piece of assembly code
 - Save a few register values for the current process onto its kernel stack ¢ General purpose registers
 - kernel stack pointer
- Restore a few for the soon-to-be-executing process from its kernel stack
- Switch to the kernel stack for the soon-to-be-executing process

Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing options
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution options
 - Parent and children execute concurrently
 - Parent waits until children terminate



- Address space
 - Child duplicate of parent
 - Child has a program loaded into it
- UNIX examples
 - fork() system call creates new process
 - exec() system call used after a fork() to replace the process' memory space with a new program

Process Creation (Cont.)

ILLINOIS TECH

College of Computing

Metadata about a process

- Metadata examples:
 - PID, GID, UID
 - Allotted CPU time
 - Virtual → Physical memory mapping
 - Pending I/O operations

OS Data Structures

- Critical function of OS is to maintain data structures for keeping track of and managing all current processes
- Layout of many structures are dictated by hardware
 - e.g., VM structures, interrupt stack frame

When do processes end?

- Conditions that terminate processes can be
 - Voluntary
 - Involuntary
- Voluntary
 - Normal exit
 - Error exit
- Involuntary
 - Fatal error (only sort of involuntary)
 - Killed by another process

ILLINOIS TECH

College of Computing

Process Termination

- Process executes last statement and then asks the operating system to delete it using the exit() system call.
 - Returns status data from child to parent (via wait())
 - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the abort() system call. Some reasons for doing so:
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

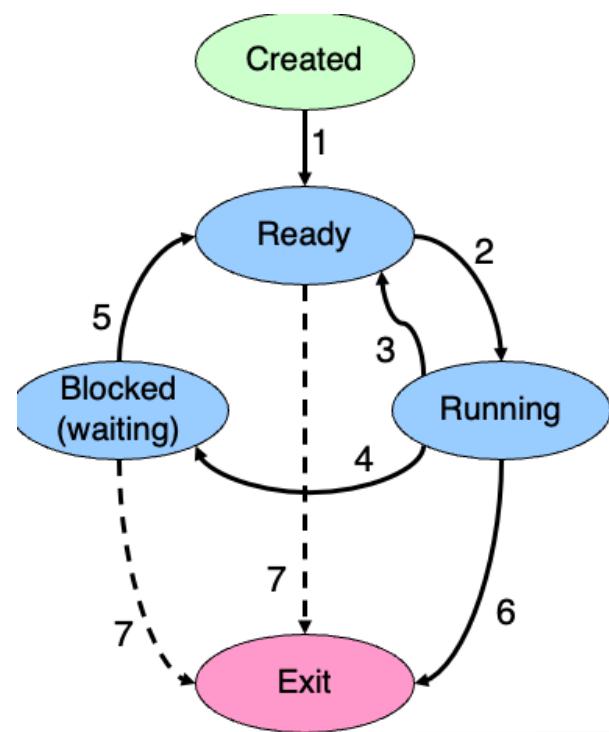
Process Termination (cont.)

- Some operating systems do not allow a child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
 - cascading termination. All children, grandchildren, etc. are terminated.
 - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process `pid = wait(&status);`
- If no parent waiting (did not invoke `wait()`) process is a zombie
- If parent terminated without invoking `wait`, process is an orphan

Process hierarchies

- Parent creates a child process
 - Child processes can create their own children
- Forms a hierarchy
 - UNIX calls this a “process group”
 - If a process exits, its children are “inherited” by the exiting process’s parent

Process states



Process in one of 5 states:

- Created
- Ready
- Running
- Blocked
- Exit

Transitions between states:

- 1 - Process enters ready queue
- 2 - Scheduler picks this process
- 3 - Scheduler picks a different process
- 4 - Process waits for event (such as I/O)
- 5 - Event occurs
- 6 - Process exits
- 7 - Process ended by another process

Process Control Block (PCB)

- Aggregate per-process data entry is referred to as the Process Control Block (PCB)
- Implementation likely consists of many disparate structures

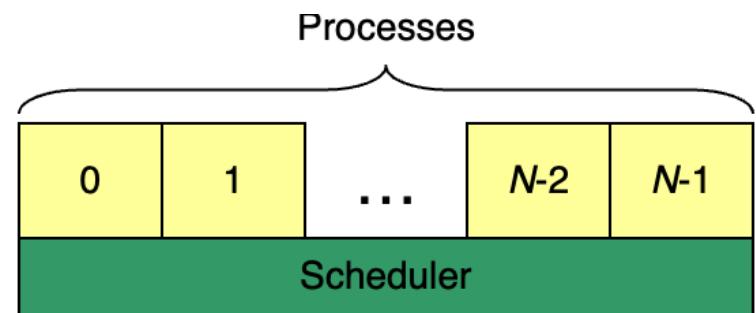
Part of the PCB

```
struct proc {  
    uint sz;  
    pde_t* pgdir;  
    char *kstack;  
    enum procstate state;  
    int pid;  
    later → struct proc *parent;  
    later → struct trapframe *tf;  
    struct context *context;  
    → void *chan;  
    int killed;  
    struct file *ofile[NFILE];  
    struct inode *cwd;  
    char name[16];  
};
```

→ Size of process memory
→ Page directory pointer for process
→ Kernel stack pointer
→ Files opened
→ Current working directory
→ Executable name

Processes in the OS

- Two “layers” for processes
- Lowest layer of process-structured OS handles interrupts, scheduling
- Above that layer are sequential processes
 - Processes tracked in the process table
 - Each process has a process table entry

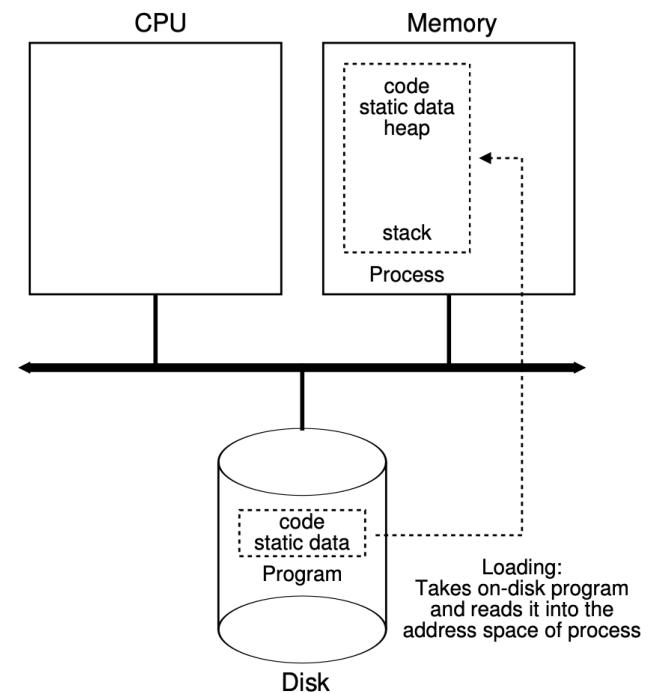


What's in a process table entry?

- Process management
 - Registers
 - Program counter
 - CPU status word
 - Stack pointer
 - Process state
- File management
 - Root directory
 - Working (current) directory...
- Memory management
 - Pointers to text, data, stack
 - Pointer to page table

Potatoes vs Potatoes

- Program vs process
- Database vs instance
- Program != process
- Database != instance



ILLINOIS TECH

College of Computing

Context Switches

- Multitasking via virtualization relies on seamlessly switching contexts between processes on hardware
 - Requires frequently saving/loading state to/from PCB
- At any point may have multiple processes ready to run
 - How does the scheduler pick the next process?

What happens on a trap/interrupt?

- Hardware saves program counter (on stack or in a special register)
- Hardware loads new PC, identifies interrupt
- Assembly language routine saves registers
- Assembly language routine sets up stack
- Assembly language calls C to run service routine
- Service routine calls scheduler
- Scheduler selects a process to run next (might be the one interrupted...)
- Assembly language routine loads PC & registers for the selected process

Scheduler

- Scheduler triggered to run when timer interrupt occurs or when running process is blocked on I/O
- Scheduler picks another process from the ready queue
- Performs a context switch

Why schedule processes?

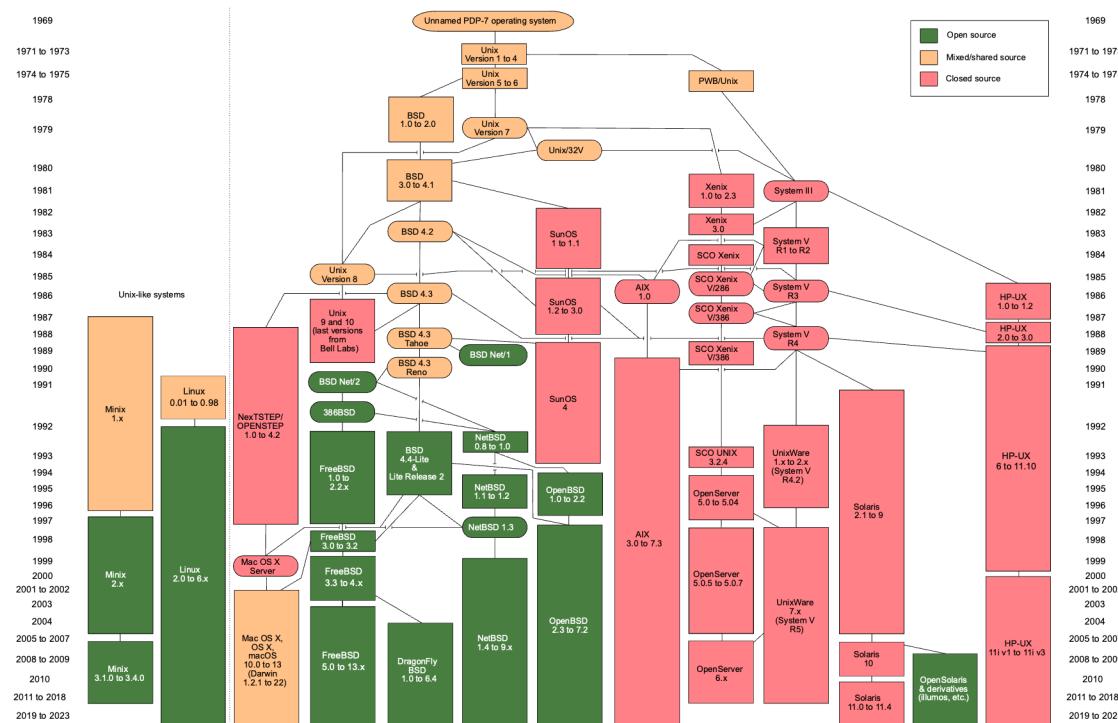
- Bursts of CPU usage alternate with periods of I/O wait
- Some processes are CPU-bound: they don't make many I/O requests
- Other processes are I/O-bound and make many kernel requests

When are processes scheduled?

- At the time they enter the system
 - Common in batch systems
 - Two types of batch scheduling
 - Submission of a new job causes the scheduler to run
 - Scheduling only done when a job voluntarily gives up the CPU (i.e., while waiting for an I/O request)
- At relatively fixed intervals (clock interrupts)
 - Necessary for interactive systems
 - May also be used for batch systems
 - Scheduling algorithms at each interrupt, and picks the next process from the pool of “ready” processes

UNIX & AIX vs Linux

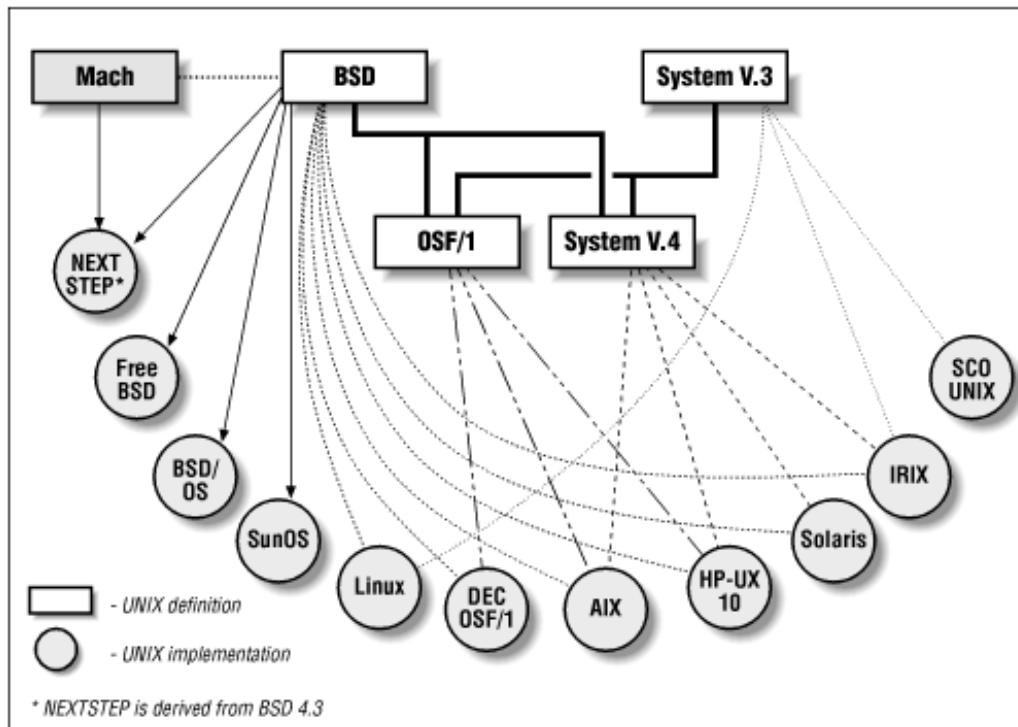
- UNIX – Commercial software, HPUX, AIX, Solaris, etc.
 - UNiplexed Information Computing System - Unix is not an acronym; it is a pun on "Multics". Multics is a large multi-user operating system that was being developed at Bell Labs shortly before Unix was created in the early '70s. Brian Kernighan is credited with the name.
 - AIX is developed by IBM and stands for Advanced Interactive eXecutive
 - Proprietary kernels
- Linux – Opensource with sometimes commercial support
 - Redhat, OpenSUSE, Fedora, Rocky, etc
 - Same kernel with tweaks



ILLINOIS TECH

College of Computing

Simple



ILLINOIS TECH

College of Computing

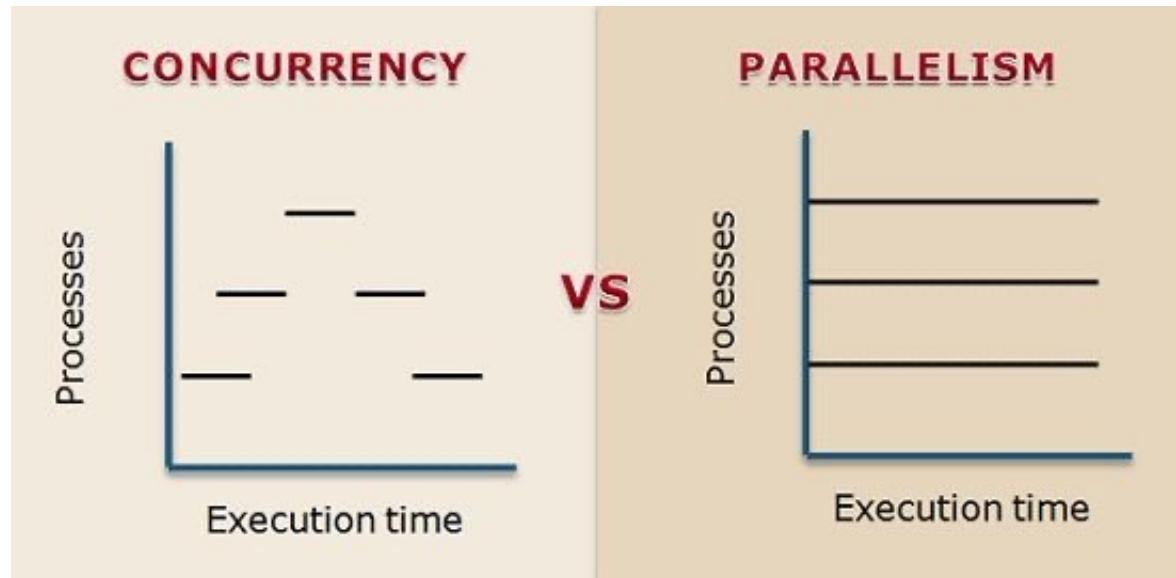
Concurrency

- Concurrency presents its own challenges and techniques for dealing with them
- Concurrent processes must be isolated from each other
- Nondeterministic execution and access to resources creates race conditions within the OS and between processes
- Dealing with these issues requires special tools and techniques
- Applies to Databases as well

ILLINOIS TECH

College of Computing

Difference Between Concurrency and Parallelism



ILLINOIS TECH

College of Computing

Persistence

- CPU and Memory state are volatile*
- I/O devices provide support for persistent storage as well as issues:
 - How to namespace persistent data?
 - What OS APIs are needed for accessing persistent data?
 - How to efficiently manage and access data on slow devices? (I/O scheduling and queueing)
 - If processes crash when updating persistent store, how to guarantee consistency?