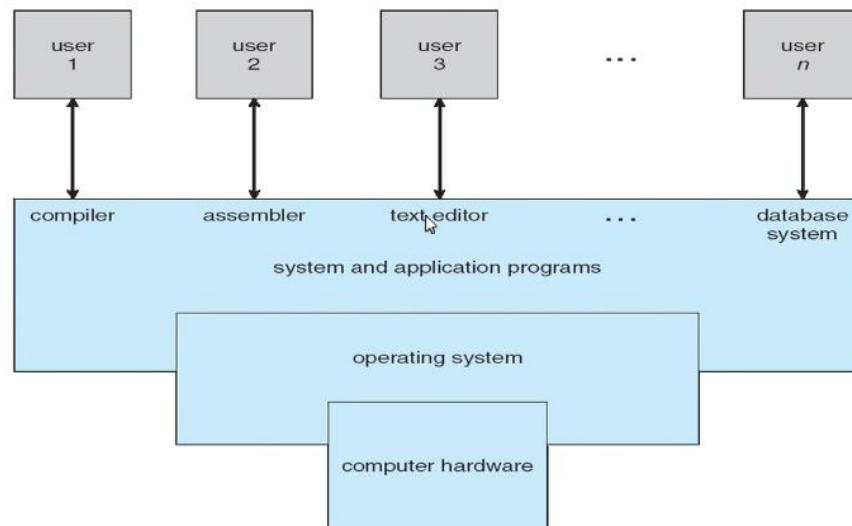


Lecture Video 01-1 (week1 – 1/10/22): Operating Systems

- Dr. Gou has worked in the car industry and has developed vehicle to vehicle networking protocols and communication systems
- DOWNLOAD XV6 (see syllabus) so we can program in C
- - First exam is on first topic which is CPU management:
- process and thread
- synchronization mechanisms
- scheduling strategies
- deadlock detection/avoidance
- (exam will be mid february)
- An operating system is just a program
 - It manages computer hardware resources, especially the CPU
 - Allows many tasks to execute at once, but there will always be the “convenience v conflict” tradeoff conflict.
 - Convenience as in allowing several users to print to a machine for example.
 - But is this the most efficient algorithm /use of resources to set up for an operating system to implement?
 - Another example: GUI interfaces are very convenient and user friendly, but they also use a lot of CPU resources – thus they are less efficient than using for example a CLI (command line interface) to execute tasks; this is because the graphics must be handled by the CPU when you implement a GUI.
- Four components of a computer system

Four Components of a Computer System



- - Use assembly language to write programs directly on top of hardware.
 - Operating system allows for encapsulation to make writing programs easier.

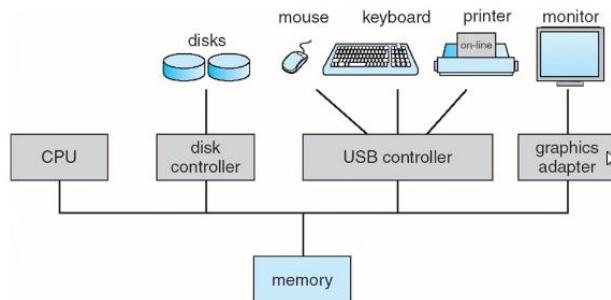
- Also operating system manages hardware resources

Computer System Structure

- Hardware – provides basic computing resources
 - CPU, memory, I/O devices
 - Operating system
 - Controls and coordinates use of hardware among various applications and users
 - Application programs – define the ways in which the system resources are used to solve the computing problems of the users
 - Word processors, compilers, web browsers, database systems, video games
 - Users
 - People, machines, other computers
-
- So in essence, hardware provides the physical means of a computer, the operating system program controls and authoritatively manages those physical means (resources), application programs use (talk to) the operating system to define different processes of how the system resources should be used to complete a task, and then users use applications to direct them to do specific tasks within the scope of the application's capabilities/programming.

Computer System Organization

- Computer-system operation
 - One or more CPUs, device controllers connect through common bus providing access to shared memory
 - Concurrent execution of CPUs and devices competing for memory cycles



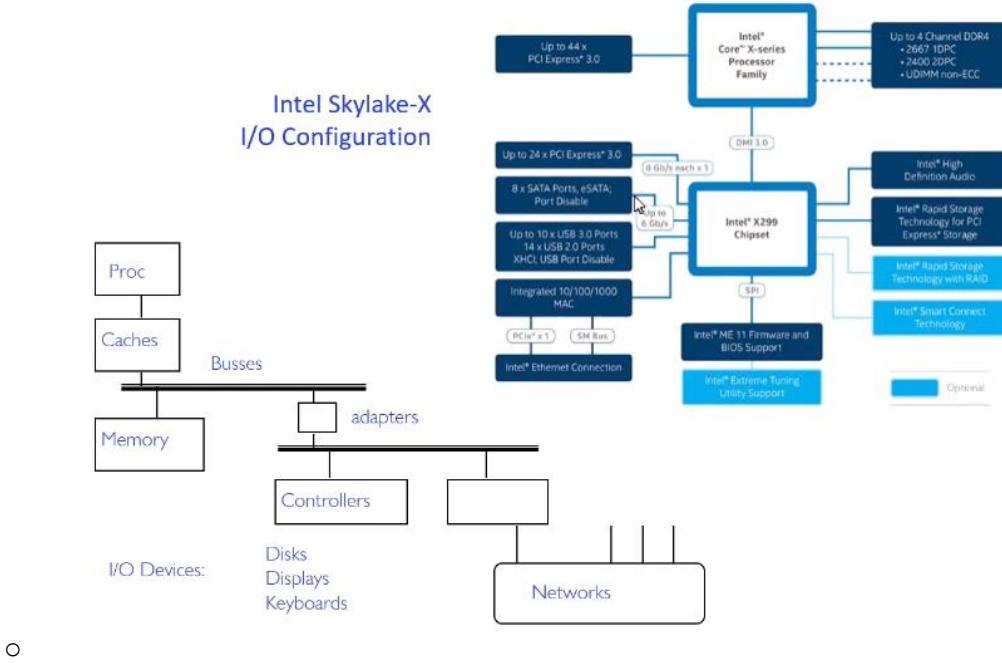
-
- Above, the diagram shows a single bus, but often each of those components are attached via separate busses that have various speeds depending on the connection and architecture design

- i/o devices often have buffers where data and instructions are written to the buffer via the devices and the CPU
 - devices start to work after the CPU sends the proper instruction to the buffer
 - CPU is so much faster than IO devices it will continue to complete other tasks instead of waiting for data from the IO devices; when IO device is finished writing its data to the buffer, it will send an interrupt to the CPU so that the CPU knows that data in the buffer is ready

Computer-System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an [interrupt](#)
 - Remember at the hardware level, there is great complexity!

HW Functionality comes with great complexity!



And Range of Timescales

CPU 5GHz

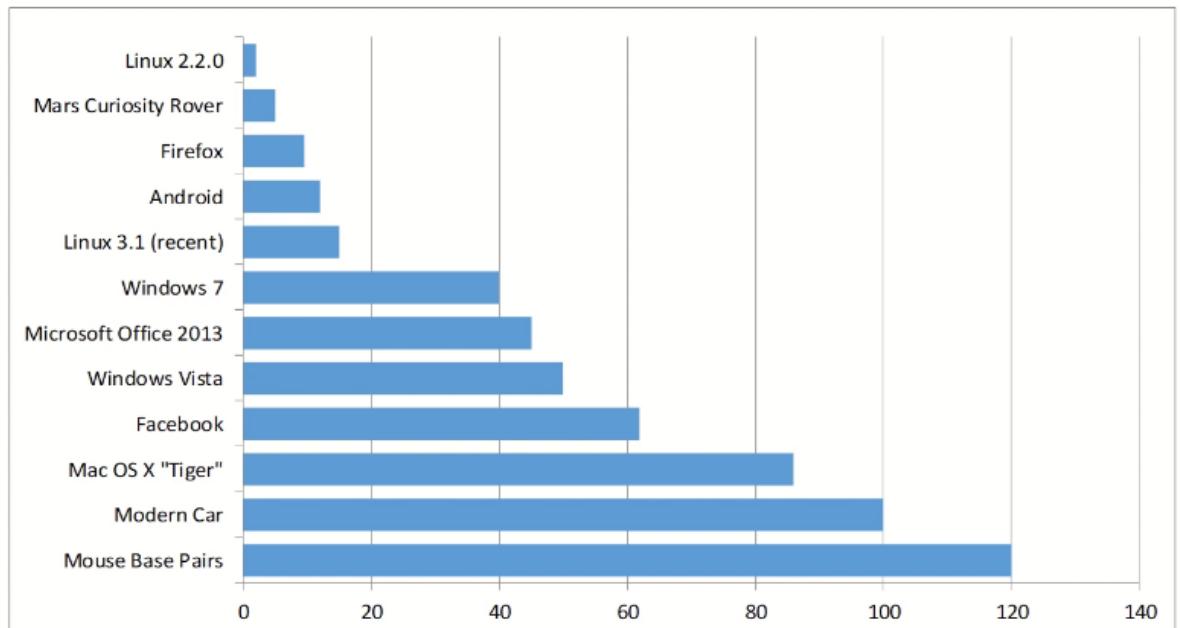
0.2 ns

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

- L1 cache is closest to CPU (in terms of orientation on the motherboard)

- Notice main memory is 500x slower than CPU (that's why we have the different layers of cache, and usually cache comes in 3 layers, each subsequent layer has a slower speed and it's orientation is further from the CPU)
- Notice that disk space memory (or flash memory as well) have even slower speeds
- Cache is a bridge: has more memory capacity than the CPU (but not as much as main memory of course) but runs faster than the main memory (but not as fast as the CPU)
- Disk seek is so slow because of the time it takes for the read and write head to move to the right track on the disk
- IO devices are slower than the main memory (because they use a buffer that is slower)
- Increasing software complexity (requires more lines of code as software becomes more complex (advanced))

Increasing Software Complexity



Millions of Lines of Code

(source <https://informationisbeautiful.net/visualizations/million-lines-of-code/>) ↴

- Notice how complex of a software program Facebook is! over 60million lines of code!
- Notice too the modern car has 100 million lines of code!!!
 - Many different ICUs to control brakes, auto drive, engine, cooling systems...windows, etc
- How do we tame complexity???

How do we tame complexity?

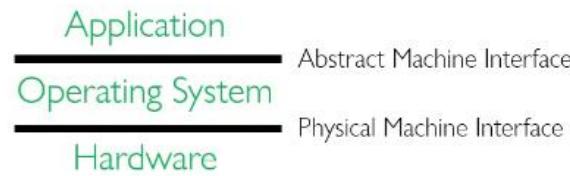
- Every piece of computer hardware different
 - Different CPU
 - x86, PowerPC, ColdFire, ARM, MIPS
 - Different amounts of memory, disk, ...
 - Different types of devices
 - Mice, Keyboards, Sensors, Cameras, Fingerprint readers
 - Different networking environment
 - Cable, DSL, Wireless, Firewalls, ...
- - Smart phones often use ARM architecture
 - X86 for PCs, and PowerPC was used in apple computers
 - Cable, DSL, Wireless, Firewalls, ...
- Questions:
 - Does a programmer need to write a single program that performs many independent activities?
 - Does every program have to be altered for every piece of hardware?
 - Does a faulty program crash everything?
 - Does every program have access to all hardware?
- - The answer to all questions above is: certainly not. Those only cause complexity to be more difficult to tame; we tame complexity of programs by compartmentalization: modular approach – many small programs that only control a small function and completes a task, and then works with other programs that do their own job as well; together, the sum of all of these modular tasks give one or many more complex programs higher up the chain: process if abstraction.

OS Abstracts the Underlying Hardware



- Processor → Thread
- Memory → Address Space
- Disks, SSDs, ... → Files
- Networks → Sockets
- Machines → Processes

- OS as an *Illusionist*:
 - Remove software/hardware quirks (*fight complexity*)
 - Optimize for convenience, utilization, reliability, ... (*help the programmer*)
- For any OS area (e.g. file systems, virtual memory, networking, scheduling):
 - What hardware interface to handle? (physical reality)
 - What's software interface to provide? (nicer abstraction)
- ○ The operating system is a perfect example: it abstracts the underlying hardware so that it is easier to control/write programs for (so then you write programs on top of the OS instead of on top of hardware level every time – such as MASM, which takes many more lines of code and complexity).
 - Then you can use application software to write other programs that do not have to worry about the various types of hardware of different machines – move up the ladder of abstraction!
- Notice the equivalencies; Processor corresponds to threads/processes, memory corresponds to an address space, or a network to a socket or a disk to a file; it is a single use of the system resource without sharing with other users.
- Above, each arrow points to how the OS abstracts the underlying hardware.
 - Notice that applications on a computer use Sockets, also think about the OSI model layer 4 (transport layer):
 - A port is a logical construct assigned to network processes so that they can be identified within the system. A socket is a combination of port and IP address. An incoming packet has a port number which is used to identify the process that needs to consume the packet.
 - So instead of controlling mac address or even IP address layer information, applications can use the abstraction of a socket (TCP or UDP sockets); an application does not actually need to know how the packet is sent or received; so here is an abstraction of the network layer, provided by the OS.



- Notice opening files for reading and writing (think about file explorer on windows computers); this is a very abstracted processes versus calling blocks of memory by their actual address space (as you would in MASM for example).
- When you run a program, you start a “process”.
 - Gives the illusion that a program owns the entire machine

What is an Operating System

- Most Likely:
 - Memory Management
 - I/O Management
 - CPU Scheduling
 - Communications
 - Multitasking/multiprogramming
- - We will discuss each part above (except for communication, which is covered in CIS 427 – Network class) throughout this course.
 - We will also discuss File Systems, Multimedia support (audio and video), user interface, and internet browsers (some argue internet browser should not be considered as part of OS).

Operating System Definition (Cont.)

- No universally accepted definition
- “Everything a vendor ships when you order an operating system” is good approximation
 - But varies wildly
- “The one program running at all times on the computer” is the **kernel**.
 - Everything else is either a system program (ships with the operating system) or an application program
-

Lecture Video 01-2 (week1 – 1/10/22): Operating Systems

- How OS protect processes (running programs) from each other and the OS itself

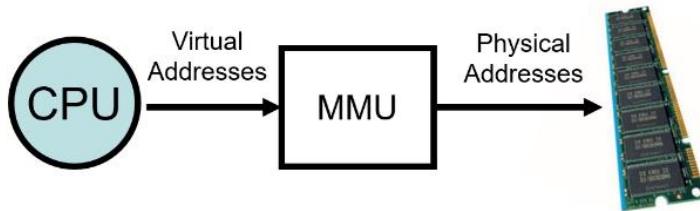
Example: Protecting Processes from Each Other

- Problem: Run multiple applications in such a way that they are protected from one another
- Goal:
 - Keep User Programs from Crashing OS
 - Keep User Programs from Crashing each other
- **Simple Policy:**
 - Programs are not allowed to read/write memory of other Programs or of Operating System
- **Mechanisms:**
 - Address Translation
 - Dual Mode Operation
-

- Address space: think about how programs are loaded from memory and placed on a stack (a block of memory dedicated to that particular program (process)).
- Stacks (address space) do not overlap with other stacks, so that each process has its own address space.
- However, stack is virtual address space, so they are still sharing the same physical memory space; thus there is a need for address translation – mapping the virtual blocks of allocated memory to a reserved physical memory location in an actual physical memory location: so in essence then, this still allows programs not to overlap and have their own space to work in to prevent crashing.

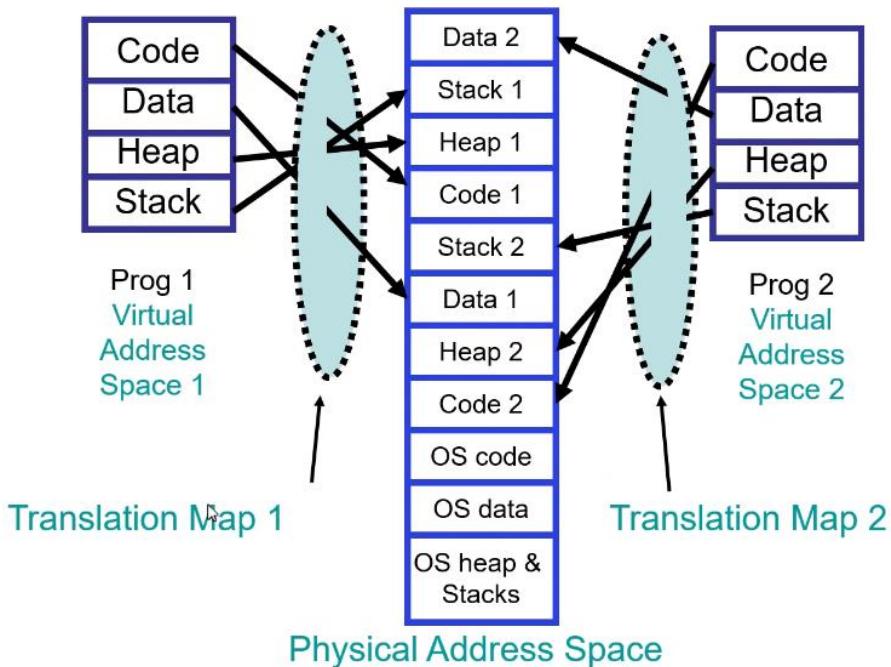
Address Translation

- Address Space
 - A group of memory addresses usable by something
 - Each program (process) and kernel has potentially different address spaces.
- Address Translation:
 - Translate from Virtual Addresses (emitted by CPU) into Physical Addresses (of memory)
 - Mapping often performed in Hardware by Memory Management Unit (MMU)



- MMU is controlled by OS

Example of Address Translation

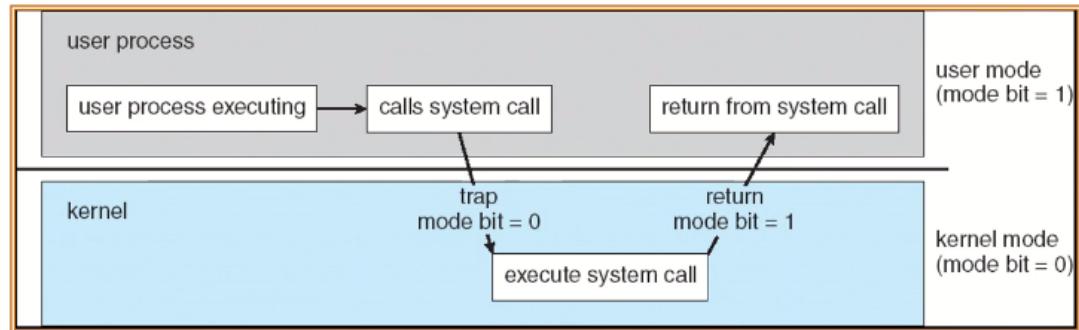


- Remember function calls go on the stack, and dynamically allocated memory is placed on the heap.
- Each process has the above 4 segments (normally) → code (instructions), data (that code will do work on), heap, stack.
- Notice above, in the actual physical memory, the 4 segments of a process are not necessarily in contiguous memory locations as the virtual memory displays.
- But notice that both processes shown above and the OS do not overlap; the translation map is (done by) the MMU.
- So the user program only deals with virtual address; to actual read and write to physical memory, it must go through the MMU; as long as the MMU is implemented properly, then you can prevent that one program crashes another, or the OS.
- Dual mode operation:

Dual Mode Operation

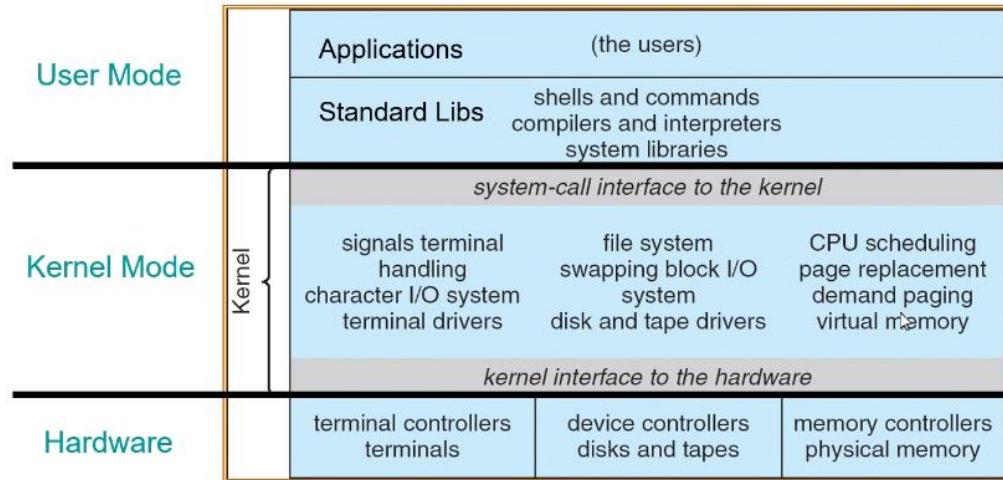
x 86

- **Hardware** provides at least two modes:
 - “Kernel” mode (or “supervisor” or “protected”)
 - “User” mode: Normal programs executed
- Some instructions/ops prohibited in user mode:
 - Example: cannot modify page tables in user mode
 - Attempt to modify \Rightarrow Exception generated
- Transitions from user mode to kernel mode:
 - System Calls, Interrupts, Other exceptions



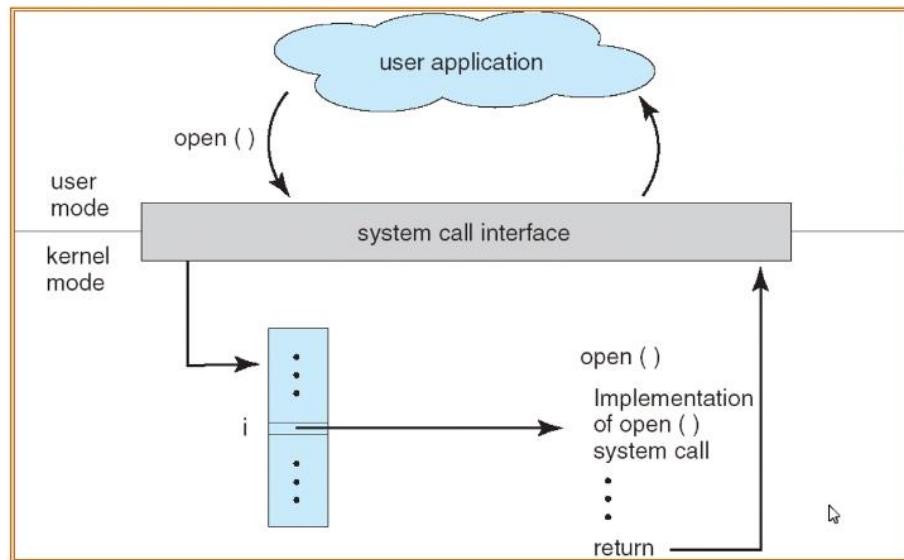
- In Kernel mode, you can control all operations, including MMUs, all IO devices, etc. – full control of OS basically.
- In User mode, you must ask the OS to do something – which it is programmed to decide if and how you are able to do some operation (like read or write).
 - You can only run the regular instructions.
 - Anytime you need to deal with any IO or instruction-required privileges, you must ask the OS – because OS is the program that access the kernel mode.
- X86 have 4 modes: 0 to 3 \rightarrow 0 is user mode, 3 is kernel mode; 1 and 2 are in between level of privileges not quite as broad as mode 3.
- You cannot change MMU tables (mapping tables) in user mode.
- For example: Opening a file may look like a normal function call, but it is not and must do a system call since it does not have the correct privileges and must go through the kernel.
- When a system call is made, hardware mode bit is changed from user mode to kernel mode in order to track through the kernel to execute a process that requires kernel privileges.

UNIX System Structure



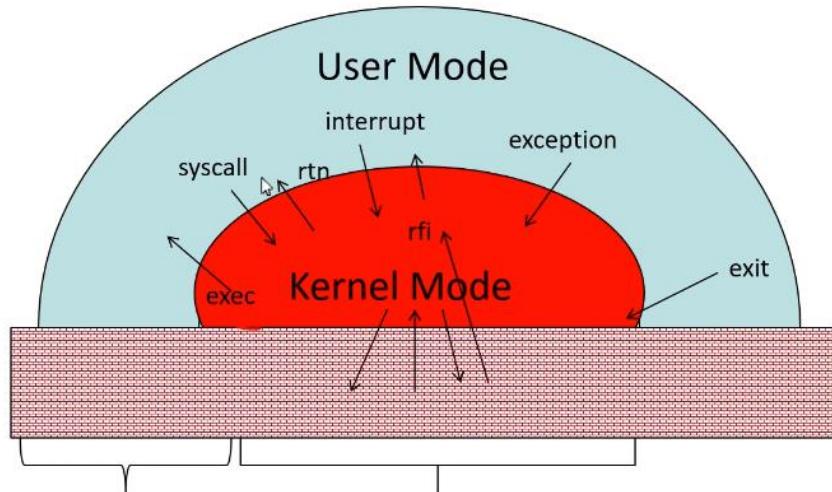
-
- In kernel mode you have access to all hardware directly.

System Calls (What is the API)



-
- Every system call has a unique number (a reference number), so that the OS/kernel knows to which system call code to execute.
- More details about switching from user to kernel mode

User/Kernel (Privileged) Mode

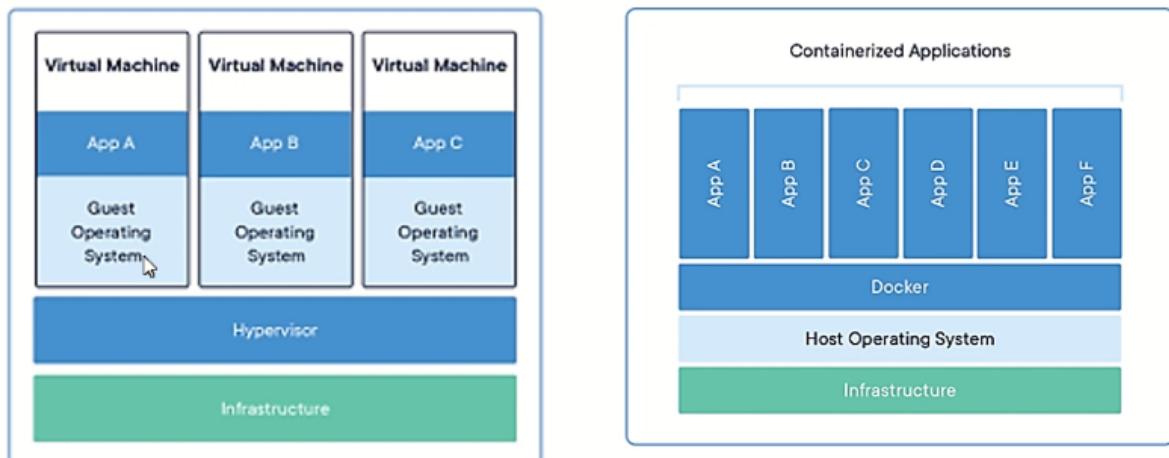


- Limited HW access Full HW access
- .exec will load file name and prompt a system call so that code for a program can be loaded from disk space into main memory, and create address space for the process (virtual mapping to the physical memory) so that program can be run in user mode.
- Program returns to kernel when it is terminated.
- Ways to get to kernel: exec, exit (start and exit a program), syscall during runtime of a program (which after system call finishes returns to user mode), and through system hardware interrupts, or through exceptions (which will often result in termination of a program since the exception generated is usually because of a program /system error that is caused, so it reaches kernel, only so that kernel will then cause program to terminate).
- 3 types of mode transfer

3 types of Mode Transfer

- Syscall
 - Process requests a system service, e.g., exit
 - Like a function call, but “outside” the process
 - Does not have the address of the system function to call
 - Like a Remote Procedure Call (RPC) – for later
 - Marshall the syscall id and args in registers and exec syscall
- Interrupt
 - External asynchronous event triggers context switch
 - e. g., Timer, I/O device
 - Independent of user process
- Trap or Exception
 - Internal synchronous event in process triggers context switch
 - e.g., Protection violation (segmentation fault), Divide by zero,
...
- All 3 are an UNPROGRAMMED CONTROL TRANSFER
 - Where does it go?
- Virtualization

Virtualization: Execution Environments for Systems



Additional layers of protection and isolation can help further manage complexity



- Virtual machines can run on the same physical machine and run different guest operating systems all on one physical machine (this can add isolation/a layer of protection to help manage the complexity of a system).
- Containers also offer some isolation among different applications:
 - Places all the required libraries into a container so that you do not have to depend on a lot of system configurations.
 - Container does not require guest OS, but still a different name space.
 - Docker is a very popular container management system.
- History of computers and operating systems
- Computers used to have inefficient use of hardware resources (idle time) and lack of interaction with users, and no protections – and one user at a time; and programs ran to completion one by one with no interrupts (batch programs)

Lecture Video 01-3 (week1 – 1/10/22): Operating Systems

History Phase 1 (1948—1970) Hardware Expensive, Humans Cheap

- When computers cost millions of \$'s, optimize for more efficient use of the hardware!
 - Lack of interaction between user and computer
- User at console: one user at a time
- Batch monitor: load program,  run, print
- Optimize to better use hardware
 - When user thinking at console, computer idle \Rightarrow BAD!
 - Feed computer batches and make users wait
- No protection: what if batch program has bug?
- In the 60 and 70s we improved: it's always best to maximize use of CPU – the most expensive component of a computer system.

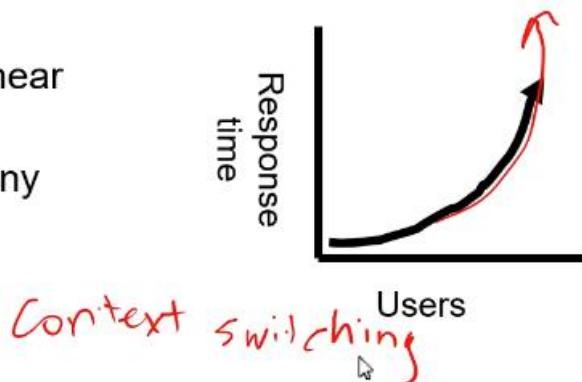
History Phase 1½ (late 60s/early 70s)

- **Data channels, Interrupts:** overlap I/O and compute
 - DMA – Direct Memory Access for I/O devices
 - I/O can be completed asynchronously
- **Multiprogramming:** several programs run simultaneously
 - Small jobs not delayed by large jobs
 - More overlap between I/O and CPU
 - Need memory protection between programs and/or OS
- **Complexity gets out of hand:**
 - Multics: announced in 1963, ran in 1969
 - 1777 people “contributed to Multics” (30-40 core dev)
 - Turing award lecture from Fernando Corbató (key researcher): “On building systems that will fail”
 - OS 360: released with 1000 known bugs (APARs)
 - “Anomalous Program Activity Report”
- **OS finally becomes an important science:**
 - How to deal with complexity???
 - UNIX based on Multics, but vastly simplified
- UNIX is considered the most successful OS
 - Key researches from Multics (which ultimately failed but brought a lot of great ideas) were key to the contribution and development of UNIX (they were also involved in the UNIX project – they created and started UNIX).

History Phase 2 (1970 – 1985)

Hardware Cheaper, Humans Expensive

- Computers available for tens of thousands of dollars instead of millions
- OS Technology maturing/stabilizing
- *Interactive timesharing:*
 - Use cheap terminals (~\$1000) to let multiple users interact with the system at the same time
 - Sacrifice CPU time to get better response time
 - Users do debugging, editing, and email online
- **Problem: Thrashing**
 - Performance vary non-linear response with load
 - Thrashing caused by many factors including
 - Swapping, queueing
- - Context switching: switch from one program to another.
 - Swapping: switching from one user to another.
 - Queueing: holding tasks in a queue as user request interrupts are prioritized first.
 - All of the above are expensive operations, and not to mention the interruptions sacrifice CPU utility and efficiency.



History Phase 3 (1981—)

Hardware Very Cheap, Humans Very Expensive

- Computer costs \$1K, Programmer costs \$100K/year
 - If you can make someone 1% more efficient by giving them a computer, it's worth it!
 - Use computers to make people more efficient
- Personal computing:
 - Computers cheap, so give everyone a PC
- Limited Hardware Resources Initially:
 - OS becomes a subroutine library
 - One application at a time (MSDOS, CP/M, ...)
- Eventually PCs become powerful:
 - OS regains all the complexity of a “big” OS
 - multiprogramming, memory protection, etc (NT,OS/2)
- - Initially, personal PC did not contain all of the latest OS technologies; so they had the old fashioned OS which essentially only acted as a subroutine library (it simply loaded and terminated programs from a library one at a time for the user).
- Graphical User Interfaces

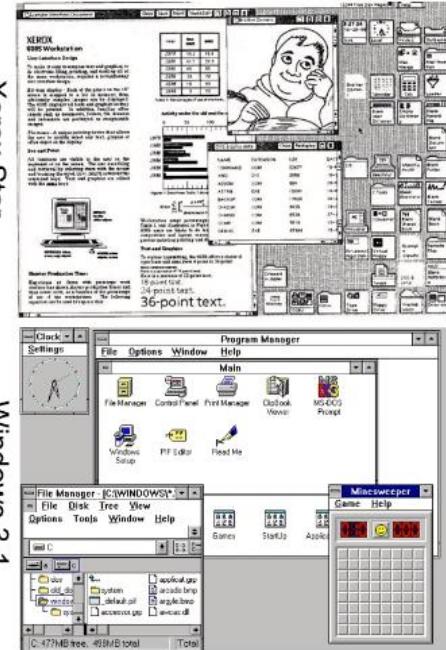
History Phase 3 (con't)

Graphical User Interfaces

Parc

- Xerox Star: 1981
 - Originally a research project (Alto)
 - First “mice”, “windows”
- Apple Lisa/Macintosh: 1984
 - “Look and Feel” suit 1988
- Microsoft Windows:
 - Win 1.0 (1985)
 - Win 3.1 (1990)
 - Win 95 (1995)
 - Win NT (1993)
 - Win 2000 (2000)
 - Win XP (2001)
 - Win Vista (2007)
 - Win 7 (2009)
 - Win 8 (2012)

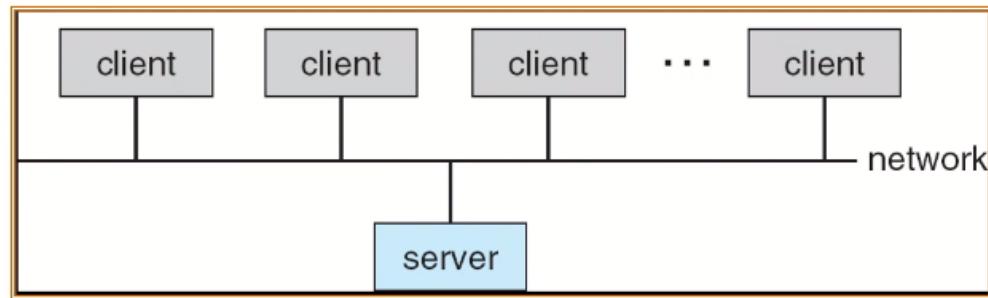
} Single Level
} HAL/Protection
} No HAL/
Full Prot



- Apple tried to sue Microsoft for having the same “look and feel”.
- Lawsuit failed; thus Microsoft continued to flourish (otherwise we may not have Windows today).
- First three versions of Windows had no Kernel/protections.
- GUI allowed normal users to use the computer without any training – they didn’t have to learn how to use the CLI – of course the tradeoff is CPU power; GUI is very expensive to process for the CPU (until GPU enters the scene, with the specific purpose of greatly reducing the load on the CPU); but it is more user friendly and allows more users to use the machine with less training.

History Phase 4 (1988—): Distributed Systems

- Networking (Local Area Networking)
 - Different machines share resources
 - Printers, File Servers, Web Servers
 - Client – Server Model
 - Services
 - Computing
 - File Storage
- Ethernet



History Phase 4 (1988—): Internet

- Developed by the research community
 - Based on open standard: Internet Protocol
 - Internet Engineering Task Force (IETF)
- Technical basis for many other types of networks
 - Intranet: enterprise IP network
- Services Provided by the Internet
 - Shared access to computing resources: telnet (1970's)
 - Shared access to data/files: FTP, NFS, AFS (1980's)
 - Communication medium over which people interact
 - email (1980's), on-line chat rooms, instant messaging (1990's)
 - audio, video (1990's, early 00's)
 - Medium for information dissemination
 - USENET (1980's)
 - WWW (1990's)
 - Audio, video (late 90's, early 00's) – replacing phone, TV?
 - File sharing (late 90's, early 00's)
- - Internet research started by department of defense funding through an organization called ARPA in the 70's.

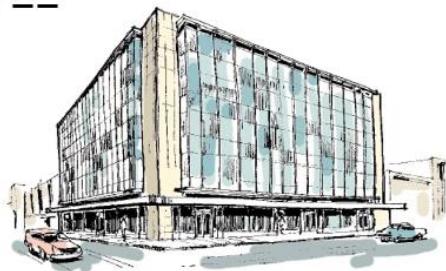
ARPA 70's

History Phase 5 (1995—): Mobile Systems

- Ubiquitous Mobile Devices
 - Laptops, PDAs, phones
 - Small, portable, and inexpensive
 - Recently twice as many smart phones as PDAs
 - Many computers/person!
 - Limited capabilities (memory, CPU, power, etc...)
- Wireless/Wide Area Networking
 - Leveraging the infrastructure
 - Huge distributed pool of resources extend devices
 - Traditional computers split into pieces. Wireless keyboards/mice, CPU distributed, storage remote
- Peer-to-peer systems
 - Many devices with equal responsibilities work together
 - Components of “Operating System” spread across globe
- - Newest peer-to-peer system: Block chain technology.
- Datacenters

Datacenter is the Computer

- Google *program* == Web search, Gmail,...
- Google *computer* ==



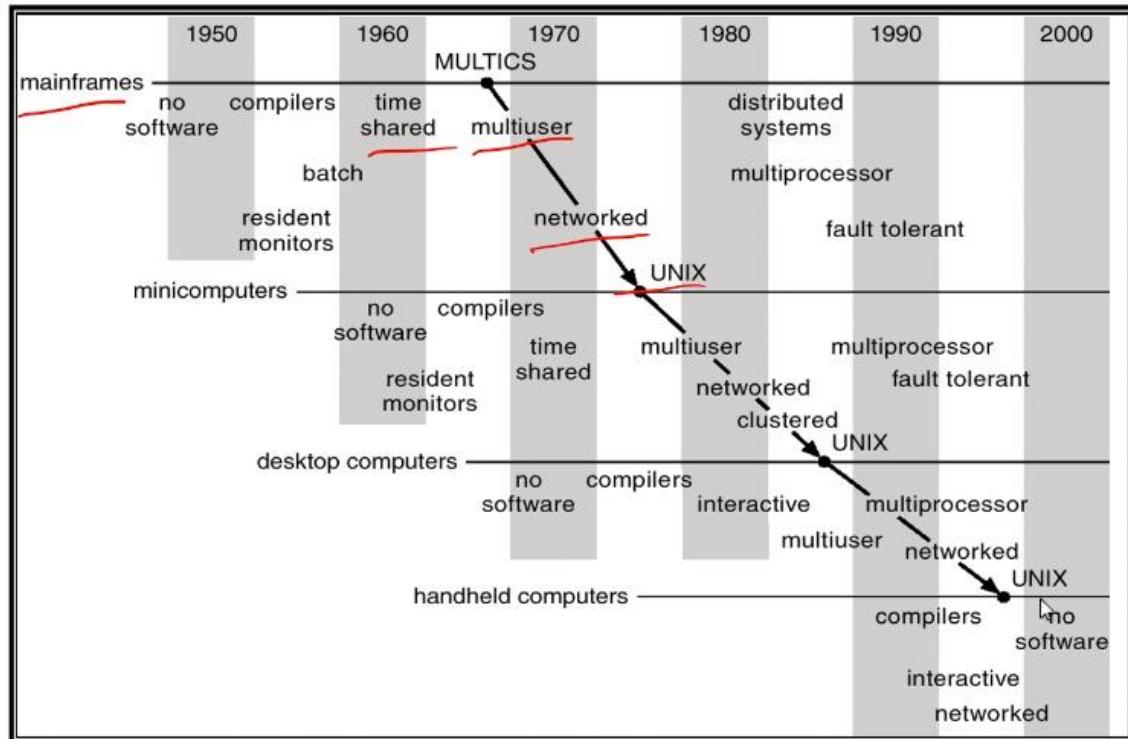
- Hundreds of thousands of computers, networking, storage
- Warehouse-sized facilities and workloads

•

OS Archaeology

- Because of the cost of developing an OS from scratch, most modern OSes have a long lineage:
- Multics → AT&T Unix → BSD Unix → Ultrix, SunOS, NetBSD,...
- Mach (micro-kernel) + BSD → NextStep → XNU → Apple OS X, iPhone iOS
- MINIX → Linux → Android OS, Chrome OS, RedHat, Ubuntu, Fedora, Debian, Suse,...
- CP/M → QDOS → MS-DOS → Windows 3.1 → NT → 95 → 98 → 2000 → XP → Vista → 7 → 8 → 10 →
-

Migration of Operating-System Concepts and Features



History of OS: Summary

- Change is continuous and OSs should adapt
 - Not: look how stupid batch processing was
 - But: Made sense at the time
- Situation today is much like the late 60s
 - Small OS: 100K lines
 - Large OS: 10M lines (5M for the browser!)
 - 100-1000 people-years
- Complexity still reigns
 - NT developed (early to late 90's): Never worked well
 - Windows 2000/XP: Very successful
 - Windows Vista (aka "Longhorn") delayed many times
 - Finally released in January 2007
 - Windows 10 (2015 to today) as an "operating system as a service"
- What we will talk about in this class:
 - Operating Systems Components
(What are the pieces of the OS)
 - Process Management ✓
 - Main-Memory Management
 - I/O System management
 - File Management
 - Networking
 - User Interfaces
 - Last two concepts above will not be covered in this class

Operating System Services

(What things does the OS do?)

- Services that (more-or-less) map onto components
 - Program execution
 - How do you execute concurrent sequences of instructions?
 - I/O operations
 - Standardized interfaces to extremely diverse devices
 - File system manipulation
 - How do you read/write/preserve files?
 - Looming concern: How do you even find files???
 - Communications
 - Networking protocols/Interface with CyberSpace?
- Cross-cutting capabilities
 - Error detection & recovery
 - Resource allocation
 - Accounting
 - Protection
- ---

 - You can abstract your I/O devices as files
 - (for example, your keyboard can be abstracted into a file – a buffer file, your hard drive can be abstracted as a file).

Implementation Issues (How is the OS implemented?)

- Policy vs. Mechanism
 - Policy: **What** do you want to do?
 - Mechanism: **How** are you going to do it?
 - Should be separated, since both change
- Algorithms used
 - Linear, Tree-based, Log Structured, etc...
- Event models used
 - threads vs event loops
- Backward compatibility issues
- System generation/configuration
 - How to make generic OS fit on specific hardware
- OS system principles

OS Systems Principles

- OS as illusionist:
 - Make hardware limitations go away
 - Provide illusion of dedicated machine with infinite memory and infinite processors
- OS as government:
 - Protect users from each other
 - Allocate resources efficiently and fairly
- OS as complex system:
 - Constant tension between simplicity and functionality or performance
- OS as history teacher
 - Learn from past
 - Adapt as hardware tradeoffs change
- OS Summary:

OS Summary

- Operating systems provide a virtual machine abstraction to handle diverse hardware
- Operating systems coordinate resources and protect users from each other
- Operating systems simplify application development by providing standard services
- Operating systems can provide an array of fault containment, fault tolerance, and fault recovery
- CIS 450 /ECE478 combines things from many other areas of computer science –
 - Languages, data structures, hardware, and algorithms
- Because this course combines many disciplines across computer science; most students consider this the most difficult course.

Lecture Video 02-1 (week2 – 1/17/22): Processes and Threads

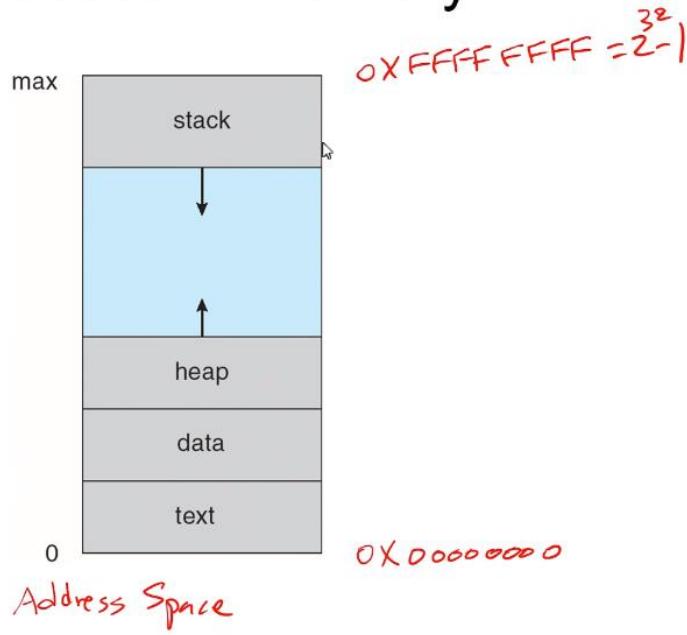
- What is a process?

What is a process

- Informally: program in execution
- Process state
 - ~~–~~ The program code, also called text section
 - Registers (including PC, SP)
 - Stack containing temporary data
 - ~~–~~ Data section containing global variables
 - Heap containing memory dynamically allocated during run time *malloc, free* *new, delete object*
 - Open file tables
- **Key concept:** processes are separated: no process can directly affect the state of another process.
- - If you run a program multiple times, you can start multiple processes.
 - The program is STATIC, the process is DYNAMIC; when you run a program, you start a process.
 - When the program terminates → the process ends (terminates).
 - Text section = instructions (the code).
 - Registers (all registers including PC and SP) are used to save/keep the process state when an operating system is switching between processes.
 - **Physical Registers** (not the value stored in the register) are shared between processes, and they need to be saved in order to do these switches.
 - (PC = program counter; SP = stack pointers)
 - PC points to next instruction (in the text section/aka code) that the process needs to run.
 - SP points to top (or bottom) of the stack of a running process.
 - Of course local variables are stored in the stack, and so are return addresses; every time a function is called, stack will grow as a space for local parameters/variables/return address is made; as functions return, stack will pop out these local variables and local values and shrink.
 - Data section contains global variables and static data
 - Static data example: string variable → “hello world”; the string literal is stored in the data section.

- Remember, before a program can be compiled and then executed, all static and global variables must be defined so that it can go into the data section of a program; program contains the instruction code (functions) and the static and global variables → that's why the program is STATIC, and the process is DYNAMIC → processes can elicit dynamic memory allocation (into the HEAP) during runtime, and call functions to create local variables (on the STACK) that can be changed during run time as well; all instruction (functions) and static variables and global variables (global var can have its values changed but the location of this variables data is stored in the data section) remain static during runtime (except global variables, which remain static in the sense that it is not destroyed until the program/process terminates → unlike local function variables).
- So, only DATA SECTION and TEXT SECTION are inside the program.
- When you start a process, it will load the text section (functions == instructions) and the data section from the program.
- Heap → malloc (memory allocation) and free (free up (return) memory back to the system).
 - In c++, new operator is same as “malloc”; delete[] is same as “free”
- Open file tables are process states.
- Remember concept of reducing errors: operating system prevents programs from crashing each other; so processes (all the different states == blocks of memory of a process are separated into their own address space; each process has its own sections for text, stack, data, heap, and file tables).
- Heap and stack grow in opposite directions and will eventually meet and then collide (that's when a program is essentially run out of dynamic memory allocated for that specific process and then the program can crash; think about how in CIS 200 we were given an assignment to grow a recursive function so large to cause the program to crash on purpose because it ran out of dynamic address space for the process).

Process in Memory



- Examples of processes:

Examples of Processes

- Shell: creates a process to execute command
 - > ls foo
(shell creates process that executes “ls”)
- When you execute a program you have just compiled, the OS generates a process to run the program.
- Your WWW browser is a process.
- - ls command in Linux = “list” (list all files and directories inside of a given folder/path level).

- ls is one of the programs in the shell system; so when you call ls, shell runs that program (starts a process that executes the program).
 - a process must be started to run a program.
- each window in a web browser is a separate process that the OS is quickly switching between as you navigate through each window and as websites are loaded and ran (as well as other background processes or other applications running (which are also all processes that the OS is switching between very quickly to “simultaneously” run them all)).
- Program v Process

Process vs. Program

- Program: static *text + data*
- Process: dynamic
- Example:
 - A user opens two browser windows: same program, different processes;
 - ○ So remember: program is static (data and text), process is dynamic (heap, stack, open file tables, maintain state of registers).
 - Notice, for the same program, you can create multiple processes; it is essentially like calling the same function multiple times → the function code is the same, but each function call is a “process” that gets its own local variables and address space that can modify local variables only for their own respective local instance (address space) of the function called; once the local function finishes, then it is destroyed (dynamic) → but the function code always remains (static) so that it can be called again and a new local function instance (process) can be started.
- Uniprogramming vs Multiprogramming
 - Uni Programming = MS-DOS (Microsoft Disk Operating System) (batch processing: run one program at a time at any point in the system).

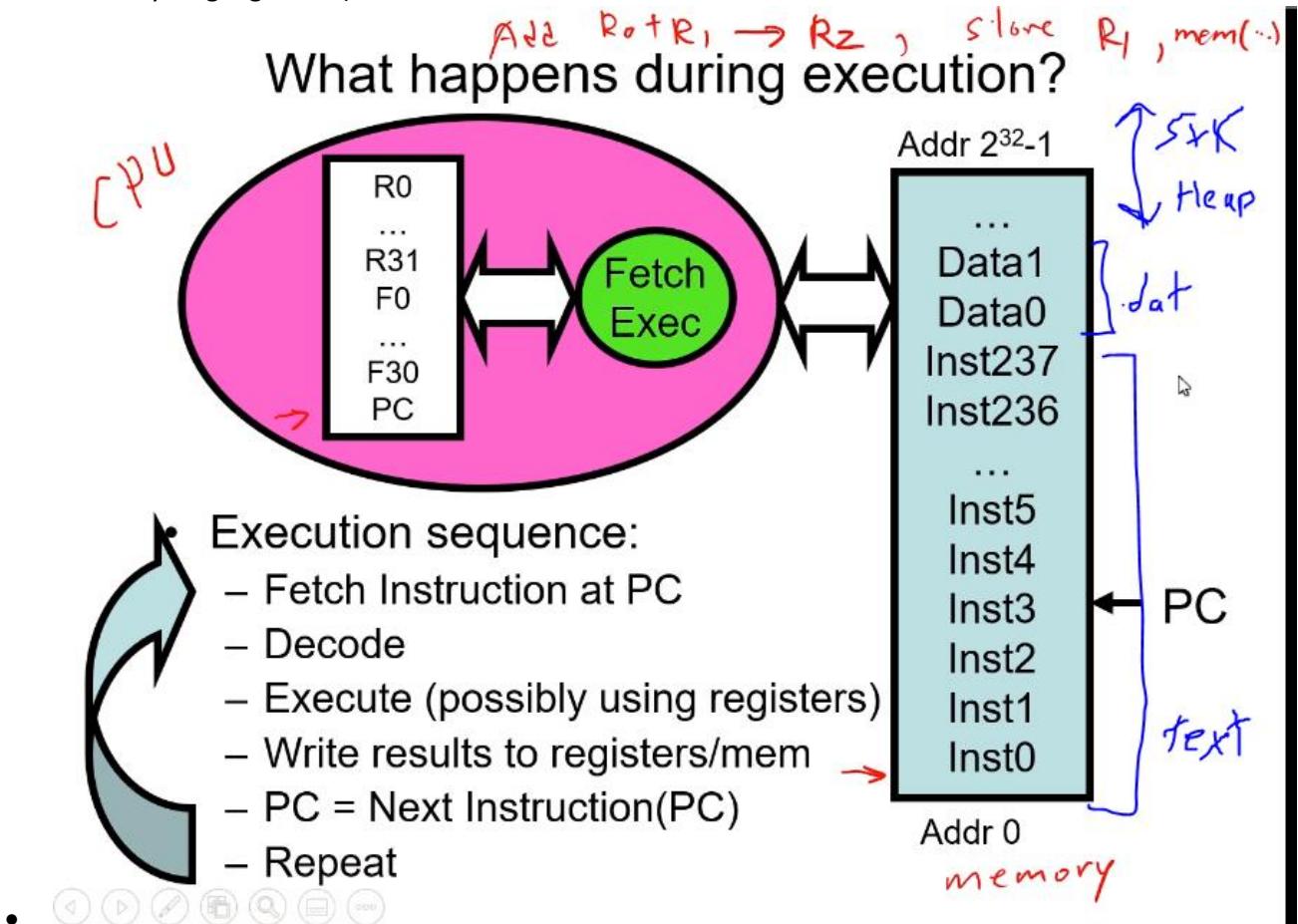
Uniprogramming vs Multiprogramming

- Uniprogramming
 - Only one process exists at any point
 - Makes designing OS easier *batch*
 - User can't do two things at once (MS-DOS)
 - Multiprogramming *single processor*
 - Multiple processes exist (only one runs at a time)
 - Requires protection, scheduling, etc
- ○ Also, since you only have one processor, you need to do scheduling for multiprogramming, so it knows how and when to switch between processes
 - Of course with multiple CPU cores (multiprocessors), then you can have parallel programming (parallelism) where several processes can in fact be executed or partially executed at the same time (parts of a process to be executed can be treated like packets in an IP network, where packets can be sent through multiple different routers/paths to reach the destination (load balancing); it is similar with parallel programming with multiple CPU cores → this reduces idle time and improves processing speed if multiple CPU cores can take care of any part of a process that does not have to wait on any additional information/steps/clock cycles; multiple processes can be processed within one clock cycle when you have multiprocessing (multiple CPU cores); for example, if two processes need to fetch the next instruction, fetching does not require any additional waiting; you only need one clock tick for this; thus with multiple CPU cores, both processes can be allowed to fetch their next instruction at the same time; but with so many processes running, even with multiple cores, you still need scheduling of process tasks and determining how executions between processes will be split up among the CPU cores (think about the CPU project I did in CIS-200 - first attempt)
- The Basic Problem of Concurrency

The Basic Problem of Concurrency

- The basic problem of concurrency involves resources:
 - Hardware: single CPU, single DRAM, single I/O devices
 - Multiprogramming: users think they have exclusive access to shared resources
- OS has to coordinate all activity
 - Multiple users, I/O interrupts, ...
 - How can it keep all these things straight?
- Basic Idea: Use Virtual Machine abstraction
 - Decompose hard problem into simpler ones
 - Abstract the notion of an executing program
 - Then, worry about multiplexing these abstract machines
- - Concurrency is the same as multiprogramming – multiple process instances exist, but only one process is executed at a time – and the OS rapidly switches between those processes as to which one should be on the CPU at a given moment – thus it can seem simultaneous – it gives the illusion that the CPU is executing more than one process at a time by interleaving (inserting between) several processes, and it gives a process the illusion that it is the only active process in the queue to be executed by the CPU as if the CPU is solely dedicated to that one process; however, there are some basic problems with concurrency
 - For example, with multiprogramming/concurrency, you may have a setup where OS puts process 1 on the CPU, then stops after one cycle, and places process 2 on the CPU, then 3...then back to process 1, and so on; there are all sorts of patterns that can be determined for use depending on the size of a process and how much various processes cost (complexity); remember, this is all still done even when multiple cores are introduced – there is still concurrency among each individual CPU; thus to simplify: a single CPU is (typically) designed to process only one task at a time – so even if you have multiple cores, each individual core can in reality only execute one process at a time – and then quickly switch to another processes and begin/continue execution; thus having more cores (multiprocessing) will improve speed of a computer by parallelism, but each core is concurrently (one after another) executing multiple processes one at a time.
 - Multiprogramming is obviously more complex to implement, but the benefits are very good, especially when implemented with multiprocessing (multiple CPUs)
 - The implementation of the virtual machine abstraction is what allows each process to believe it has exclusive access to the CPU and the rest of the computer hardware (memory, IO devices, etc..)

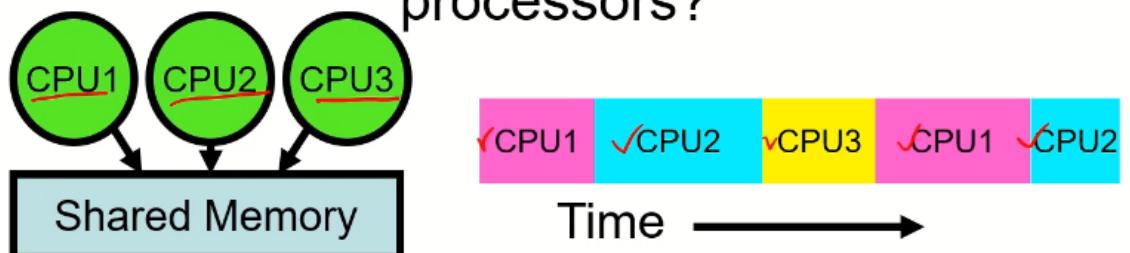
- Multiplex = interleave (spread out; insert between); the operating system is what deals with interleaving the hardware resources to create the exclusive process illusion (by implementing interrupts, etc..)
 - CIS 310 concepts review (see CIS 310 notes on Fetch-Decompile-Execute cycle; also see Irvin assembly language book):



- PC = Program Counter; increments to the address of where the next instruction is stored
 - Remember, in a 32-bit RISC (reduced instruction set architecture: where every instruction is the same size) address system, PC needs to be incremented by 4 (bytes) since each instruction takes up 32 bits == 4 bytes (every 32 bits (4bytes), a new instruction begins); note that in RISC architectures, memory is byte addressable (every 8 bits has a unique address); thus the registers (including the PC register) are pointing to bytes whenever it stores a number; so (using hexadecimal digits) if PC = 0x0000, then it is pointing to the 0 byte; if PC = 0x0001, then it is pointing to byte 1; thus for 4-byte instructions, in order to increment from one instruction to the next: PC = 0x0000; PC+4 → PC = 0x0004 (the address of the next instruction).
 - The MIPS architecture uses a byte-addressable instruction memory unit. MIPS is a RISC computer, and that means that **all the instructions are the same length: 32-bits**. Every cycle, therefore, the PC needs to be incremented by 4 (32 bits = 4 bytes).

- Remember PC can be directed to point to another set of instructions when it encounters a JUMP
- Then the data (instruction) stored at the location of where the PC is pointing to is decoded (acquire what instruction is stored at location PC)
- Then once the instruction is known (and placed into the registers), it is executed (on whatever is in the registers/in a given memory location to be a part of the execution and the result is (depending on the instruction set of a given architecture) stored in another register or memory location automatically as part of the execution)
- Then cycle repeats; so for example, instruction 0 could be add R1+ R2 → R1 (ADD registers 1 and 2, result is stored in R1), then instruction 1 could be to MOV R1 MEM(XXXX) → move the contents of R1 into a memory location
- How can we give the illusion of multiple processors?

How can we give the illusion of multiple processors?



- Assume a single processor. How do we provide the illusion of multiple processors?
 - Multiplex in time!
- Each virtual “CPU” needs a structure to hold:
 - Program Counter (PC), Stack Pointer (SP)
 - Registers (Integer, Floating point, others...?)
- How switch from one CPU to the next?
 - Save PC, SP, and registers in current state block
 - Load PC, SP, and registers from new state block
- What triggers switch?
 - Timer, voluntary yield, I/O, other things
- Preemptive
 - OS runs one process for a while, then takes the CPU away from that process and lets another process run.
- Multiplex time: Use virtual CPUs; where it is really one CPU quickly switching between processes, giving the illusion that each process has its own CPU
- But to do this, we need a way to “freeze” or “hold” a virtual CPU (pause a process currently being executed by the actual CPU) so that we can switch between processes
 - For each paused process, we need to remember the value of the PC (program counter / instruction pointer), the SP (stack pointer), and all register values of the CPU (we need to store these values in a virtual CPU, so that when the real

- CPU goes back to finish running a process, it can copy the values stored in the virtual CPU (a memory location) that stored the memory of the paused process
- So the virtual CPU takes on two forms then: it can act as a memory location for the frozen CPU state of a given process, and it can act as a singly dedicated CPU contiguously executing all instructions of a single process (batch processing).
- So switching from one virtual CPU to the next is really the real CPU going from one process to another – by first downloading the hold structure data (“frozen” state data == the last information in all CPU registers before pausing/holding the process, which was copied into a hold structure) of a CPU of a given process and then continuing where it left off at on running that process.
- The number of dedicated hold structures (state blocks) that a CPU has will determine how the max number of processes it can multiprogram/multiplex at any given time.
- NOTE: ALL DATA (heap, global, local stack data) OF A PROCESS REMAINS IN MAIN MEMORY! This does NOT need to be saved in the PCB, otherwise you would need really large PCB data structures (logically, or physically on the CPU).
- Preemptive is important for OS to be able to do; otherwise, a program with a bug in it will create a process with an infinite loop, or really long, slow processes will run for a long time and greatly slow down overall performance of a computer even if CPU is being maximally used; computer performance for multiprogramming will suffer.
 - Can cause entire system to hang, or crash if OS doesn't have ability to be preemptive
- Process States

Process State

- As a process executes, it changes **state**
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur
 - **ready**: The process is waiting to be assigned to a processor
 - **terminated**: The process has finished execution
- When a process goes into the terminated queue, it is in the “zombie state”; it is already a dead process, but it has not released all of its resources yet; it takes time/the OS will take time to free all of the memory and delete all the controls structures before it completely removes a process from the system (remember too it has to handle exit codes that the process sent with the exit function; exit(0) of course means normal

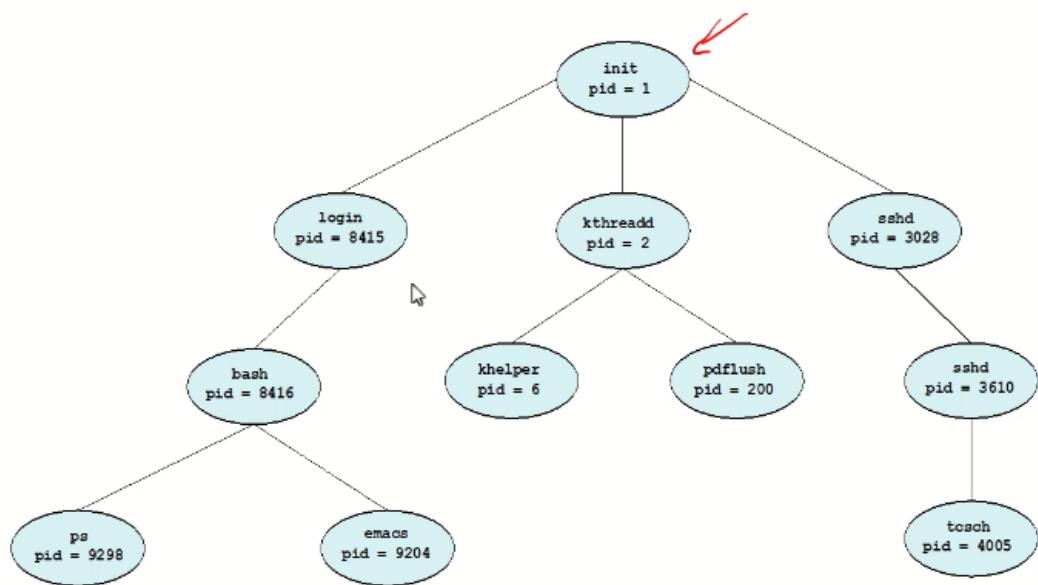
termination – no issues; other integers means usually that something went wrong and so program terminated on a bad note or something to that effect – OS can then take that return value and do something with it while it also frees up memory as it finally ends the process)

- Process Creation

Process Creation

- Parent process create 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
- ○ All processes stem from the root process (init), which has no parent

A Tree of Processes in Linux



o

More on Processes

- To start a process
 - Load code and data (from a file) *program*
 - Initialize PC to start of code
- Registers and Stack used while process runs
 - Both under compiler control
- Files can be opened, closed, read, etc

o

Creating a Process(Cont.)

- Must somehow specify code, data, files, stack, registers
- UNIX examples

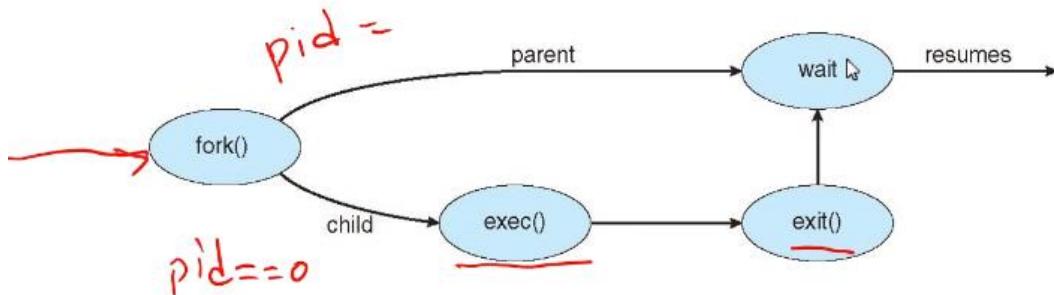
- `fork()` system call creates new process

- Semantics of Unix `fork()` are that the child process gets a complete copy of the parent memory and I/O state
 - (returns 0 to indicate child process)
- Originally very expensive
- Much less expensive with "copy on write"

- `exec()` system call used after a `fork()` to replace the process' memory space with a new program



$\text{pid} = \text{fork}();$



- - Copy on write is essentially letting the child point to the same memory location as the parent so that the operation and memory space used is not as expensive; you only make separate copies if the parent or child needs to be updated and thus their content will be different – so at this point then the child and parent do need their own memory space
 - Often child process will not continue to run what the parent was running, but will run a new program
 - Many times with the `fork()` operation, the parent has to wait for the child to exit to resume its process
 - P id of child process is often set to 0 so we know which is parent and which is the child

C Program Forking Separate Process

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

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
    }

    return 0;
}
```

○

How many new processes are created in the program below?

```

int main(void)
{
    for (int i = 0; i < 3; i++) {
        pid_t pid = fork();
        if (pid > 0)
            printf("Process id: %d\n", pid);
    }
}

```

The diagram illustrates the creation of processes. The parent process creates three children at iteration i=1. Each of these children then creates two more children at iteration i=2, resulting in a total of seven children. Finally, each of those seven children creates one more child at iteration i=3, resulting in a total of eight processes.

17

- essentially, program executables are just being passed into the currently running process to create new processes; this will allow new processes to work with the same data as the parent, starting at the execution location after the fork() call, but perform different functions (load a new program using exec() function, using parent's address space or copy of it if necessary (parent's data)) or perform the same function as the parent if needed (thus, that is why it is a different program/process that stems from the parent and all of its data); once any changes to data are made by the child, then the copy-on-write bit is turned off and the kernel makes a separate copy of the parent and child; essentially, they can no longer share memory if the child process is not a "read-only" process – it can make new data and do manipulation, as long as it makes its own copies of parent data and use parent data to assign/change values, but the value of the data variables from the parent itself cannot be changed by the child in order to maintain copy on write.
- Note: fork() can be called to create a child process; but often the parent is also a child too, even if that means it is a child of the OS, since the OS is the main process that is in charge of starting, running, and terminating all other processes.

What will the program print?

```

int main(void)
{
    char** argv = (char**) malloc(3*sizeof(char*));
    argv[0] = "/bin/ls";
    argv[1] = ".";
    argv[2] = NULL;
    for (int i = 0; i < 10; i++) {
        printf("%d\n", i);
        if (i == 3)
            execv("/bin/ls", argv);
    }
}

```

```

[jinhua@login2 examples]$ ./forks
Process id: 1770
Process id: 1771
Process id: 1772
[jinhua@login2 examples]$ Process id: 1773
Process id: 1775
Process id: 1776
Process id: 1777

[jinhua@login2 examples]$ ./forks
Process id: 1787
Process id: 1788
Process id: 1789
[jinhua@login2 examples]$ Process id: 1790
Process id: 1792
Process id: 1794
Process id: 1796

[jinhua@login2 examples]$ ./forks
Process id: 1802
Process id: 1803
Process id: 1806
[jinhua@login2 examples]$ Process id: 1807
Process id: 1808
Process id: 1809
Process id: 1810

[jinhua@login2 examples]$ clear
clear: Command not found.
[jinhua@login2 examples]$ cl

```

- Execv is different from fork in that it starts a brand new process that is not a copy of its parent
- }
- process termination

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

Conditions which terminate processes

1. Normal exit (voluntary) exit(0)
2. Error exit (voluntary) exit(1)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)
kill -9 pid

20

- Error exits: try to open file, or try to set up a network connection
- Fatal error: sometimes your program has fatal errors such as divide by 0, or program caused a segmentation fork
- Kill: provide process ID to kill that process using the kill exit value
- Context switch

Context Switch

- Switch from running one process to running another process
- Solution: save and restore hardware state on a context switch.
- Save the state in **Process Control Block (PCB)**
- What is in PCB?
- - A PCB is the entire state of a process – all information related to that state (running, waiting, etc...PC counter...etc..):
 - Process Control Block:

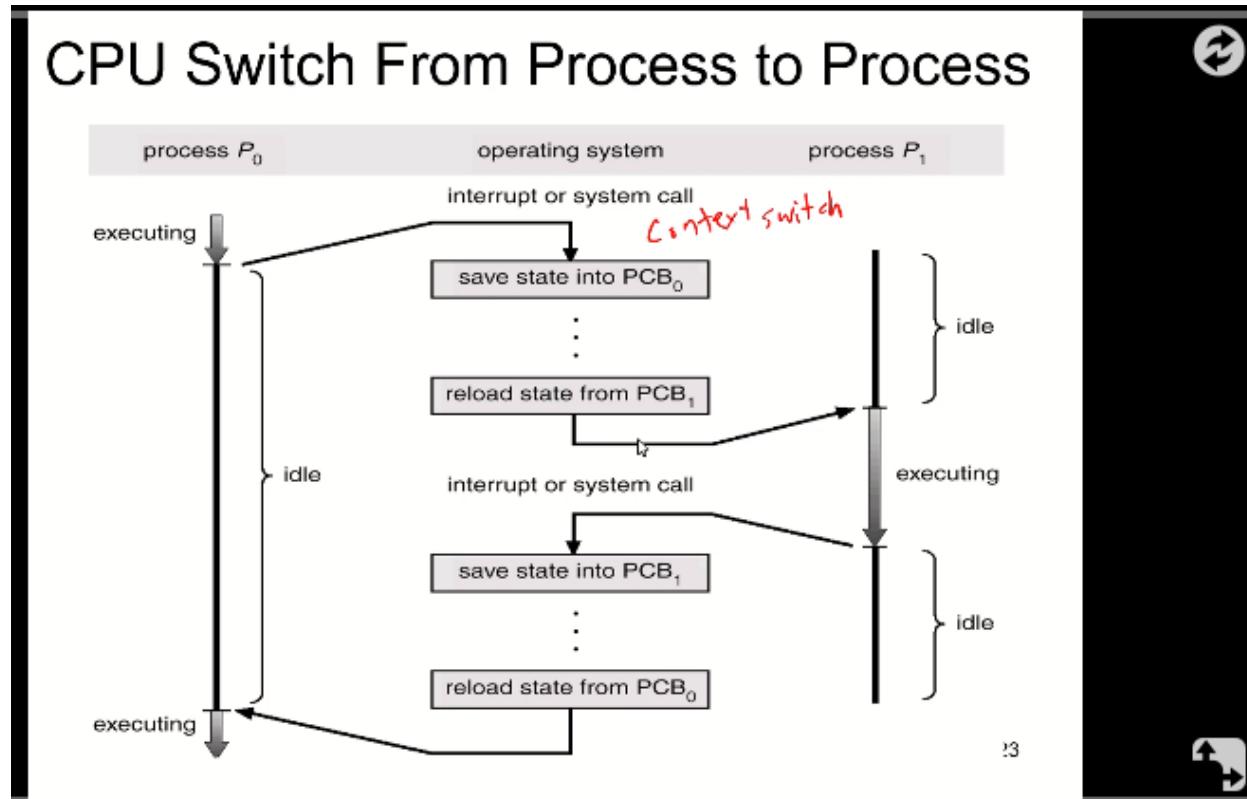
Process Control Block (PCB)

Information associated with each process

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information – priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

process state
process number
program counter
registers
memory limits
list of open files
• • •

○



- Switching from one process (CPU state) to another
 - Depending on how complex the CPU is, it may be expensive; if a CPU has more registers/pointers...there will be more CPU state instructions (larger PCB blocks) to save in order to do the context switch.
 - Big cost without doing any real useful work since during the switch, no process makes any progress toward completion; so there is a lot of overhead; thus we want to make sure we limit/plan accordingly how often we context switch (switch from one virtual CPU context, to another == pausing executing one process, to start/continue executing another)
 - Context switching is a balancing act: balance the responsiveness of processes by allowing context switching, but not switching too often so as to not have too much overhead and overall poorer performance; remember context switching is already inefficient (generally speaking) in terms of raw maximum CPU utilization on executing processes

Lecture Video 02-2 (week2 – 1/17/22): Processes and Threads

- Decomposing a Process

Decomposing a Process

- **Process:** everything needed to run a program
- **Consists of:**
 - Thread(s)
 - Address space
- - In earlier days, each process had a single sequence of executions (thread) but in modern days we also allow concurrencies within the same process; you can have multiple sequential streams of executions (threads) at the same time within the same process: multithreaded process
 - The threads are like “ip packets”, where each thread can be sent to multiple CPUs to be worked on simultaneously (or concurrently using 1 processor) when using multiple cores; threads can only be worked on in that way as long as they don’t depend on any other threads or they are not waiting for another thread which must be done before it to be executed
 - Thus, multiple threads from a single process can be worked on by several CPUs, just like data being sent over the network, where the data is broken up and sent by packets, and each packet can be sent via different routes and through different routers.
 - Thus, a process can be completed much faster and out of sequence, and of course different threads of a process can be paused
 - Of course, address space provide protection from other processes
 - Threads

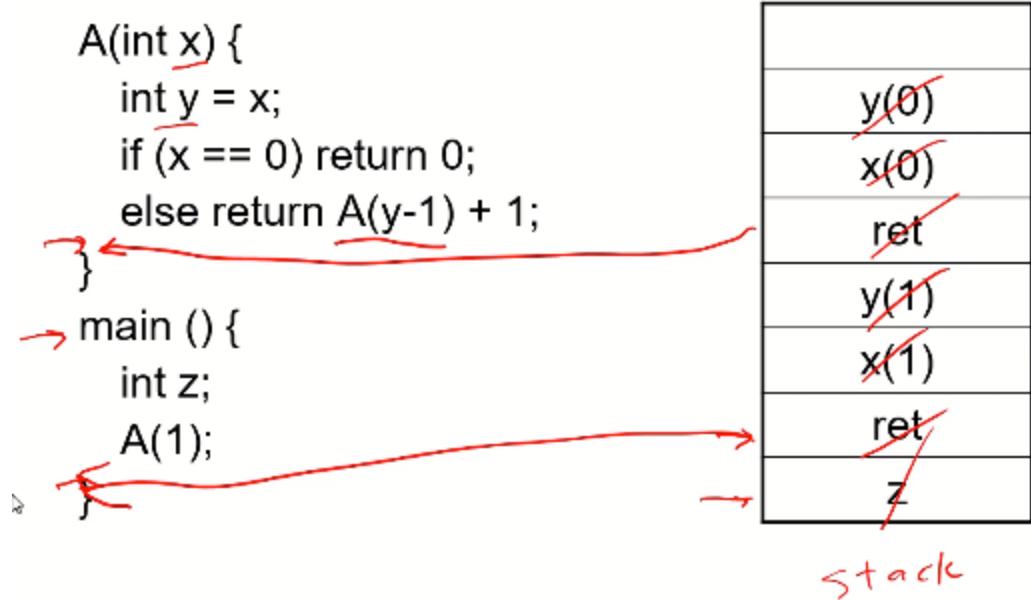
Thread

- Sequential stream of execution
- More concretely
 - Program counter (PC)
 - Register set (PC, SP, \dots)
 - Stack
- Sometime called lightweight process
 - Sometimes we call a process a heavy weight process, and a thread a lightweight process

Address Space

- Consists of
 - Code
 - Data
 - Open files
- Address space can have > 1 thread
 - threads share code, data, files
 - threads have separate stacks, register set
- - need to separate the stacks and register set for each thread because they are using the same hardware (CPU..etc)
- Run-time Stack

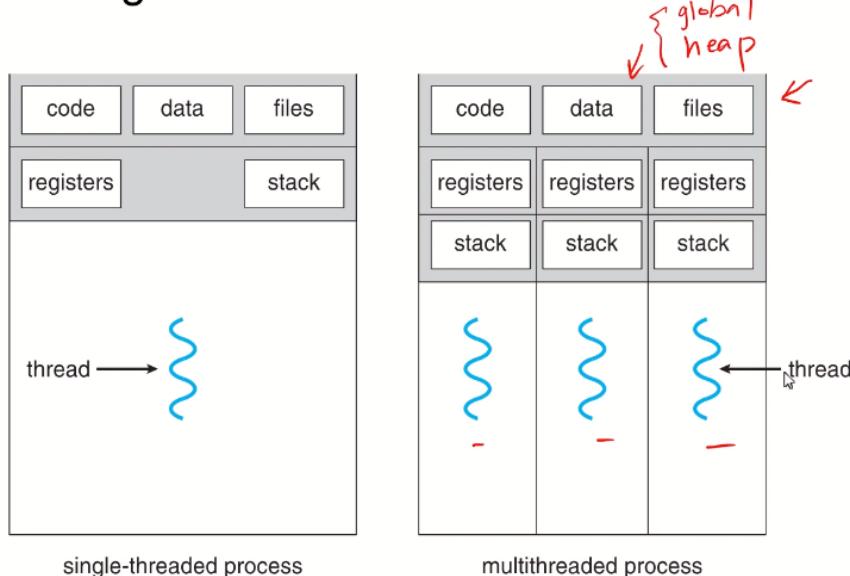
Run-time Stack



27

- ○ Remember, all local variables of all functions are stored in the run time stack, as well as the return address (as in the example above: ret == return address) of each of the local functions
- Single and Multithreaded Processes

Single and Multithreaded Processes



- ○ The advantage of a multithreaded process having each thread share the same code, data, and open files is that there is good communication between threads that are

working on/apart of the same process. This would be a bad thing to do (generally speaking) between processes as you can cause crashes – hence that's why for each process has its own data segments with its own address space; but for threads apart of the same process, it is advantageous and in fact is a key component to allowing for efficient and effective multithreading – although we are giving up protections between threads, it is necessary.

- This can lead to one thread potentially damaging another thread and causing a crash, and if one thread is damaged it could have a ripple effect on other threads since they share code, data (both heap and global var), and file segments.
- It is the programmer's responsibility to make sure thread collisions do not occur, not the OS (the OS is not programmed/tailored so much to protecting against thread collision errors)
- OS designed to be responsible to protect between processes; not between threads within the same process

Threads, Address Spaces

- Threads provide concurrency
 - Each thread is sequential, but can have > 1 thread
- Address spaces provide protection
 - Don't want my error to clobber your data

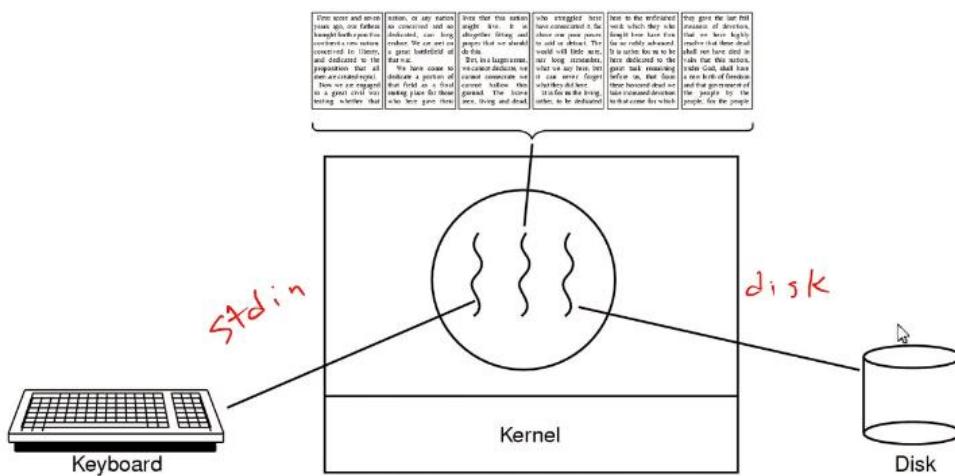
Modern OS's generally have threads, address spaces

○

- Note above: address space provide protection refers only to OS providing protection between processes, and also between each thread having their own address space for the register and stack data segments only – but OS does not provide protection between threads for all segments as it does for between processes; it does not protect between threads for the data, heap, and open files segments

- Remember concurrently is not referring to two threads (and of course processes too) being executed simultaneously, but rather “existing at the same time (existing simultaneously)” – and of course we know through multiprogramming they can be executed one at a time, with the CPU being able to switch between multiple processes while they are not yet run to full completion
 - Of course, threads provide concurrency between processes too; with one CPU, the CPU can switch between processes by executing a thread of one process and then freezing the process to switch to another thread of another process, or even a thread from the same process, which comes in handy when you have multiple CPU cores working together
 - There will be less errors and the program can be simpler if you write a program with process threads that focus on a single task
 - You don’t have to worry about multiplexing/interleaving multiple tasks together in a single execution – this will also provide better holds/freezes when CPU moves to do another process

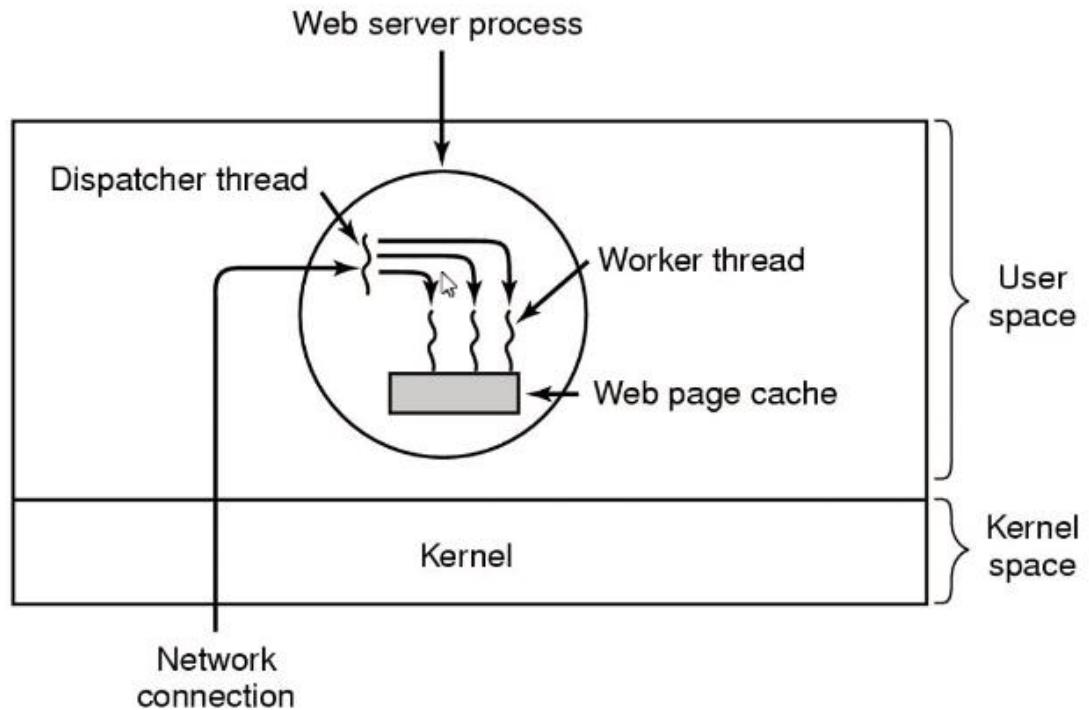
Thread Usage (1)



A word processor with three threads

- Above, you could have a program (process) where the keyboard input is in one thread, output to screen is in another thread, and read/write to disk is also another thread

Thread Usage (2)



A multithreaded Web server

- - Above, several clients could be asking for data or some have some request; thus in a multithreaded web server, each client is treated as a thread (basically networking concepts); the data that each connection (client/thread) has is that they all share the same web cache so that each client doesn't have to independently read from the disk – it's more efficient to just load the data into a cache so that several users (threads) can access that same very fast memory
 - Also the program for dealing with multiple clients for a webserver will be much easier
- Benefits of multithreaded programming

Benefits of multithreaded programming

- Responsiveness
 - Ex: a multithreaded web browser could still allow user interaction in one thread while an image is being loaded in another thread
- Resource sharing
 - Threads share the same address space
- Economy
 - More economical to create and context switch threads
- Utilization of multiprocessor architectures
- Simplify Programming
 - Having a separate thread for each activity.
Programmer does not have to deal with the complexity of interleaving multiple activities on the same thread.

32

-
- Also when you do a context switch, there is less data to save when you have a multithreaded process
- Multiprocessor architectures: modern computers have multiple CPUs, and each CPU has multiple cores! Thus if you have 2 CPUs, and each CPU has 4 cores, then you have the equivalent of 8 processors that can each be individually used in the multiprogramming and multithreading implementations – and then coordinated with each other (multiprocessing) so that you can actually have both simultaneous and concurrent processing (parallelism = processes being run at the same time through multiple processors); also multithreading is what allows multiprocessing to be done most efficiently – so that each processor is not bound to processing only 1 process at a time; but they can do a coordinated effort to most efficiently execute multiple processes at the same time; with multiple processors, you don't have to sacrifice as much CPU utilization through only doing multiplexing/concurrent/multiprogramming in order to make a system be able to execute multiple active processes – because now at any given moment several processors can work on available and ready-to-execute threads...
- Remember: interleave == insert between
 - Being able to separate tasks by thread allows for reduced complexity in programming and for reduced wait time of concurrent processing – having multiple tasks on one thread means that the task have to be done sequentially, even if they do not have to depend on each other, which means a processor

could only work on that one thread at a time, thus for the entire process to be completed it would take longer and have increased wait time.

- Classification (Uniprogramming v Multiprogramming, with or without threading)

Classification

# threads Per AS: # of addrs: spaces:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (Geoworks, VxWorks, JavaOS, etc) JavaOS, Pilot(PC)	Mach, OS/2, Linux, Win 95?, Mac OS X, Win NT to XP, Solaris, HP-UX

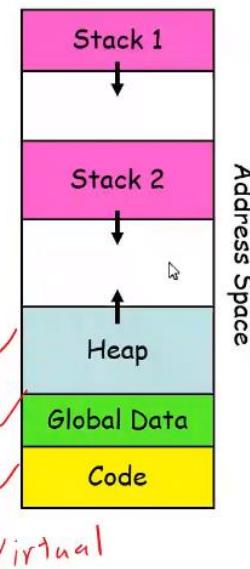
- Real operating systems have either
 - One or many address spaces
 - One or many threads per address space

- Above, MSDOS supported only one processes at a time, and one thread per process
- Some embedded systems allowed for only one process at a time, but multiple threads (so that during IO waits of a given thread could be paused and another thread could be executed if possible, and if you had multiple processors (multiprocessing), you could also execute multiple threads of process in parallel)
- Traditional UNIX allowed for multiprogramming programming but did not have multithreading (only supported a single thread per process)
- Most of todays system support both multithreading and multiprogramming, as well as multiprocessing

- Memory Footprint of Two-Thread example

Memory Footprint of Two-Thread Example

- This process has
 - Two sets of CPU registers
 - Two sets of Stacks
- Questions:
 - How do we position stacks relative to each other?
 - What maximum size should we choose for the stacks?
 - What happens if threads violate this?
 - How might you catch violations?✓

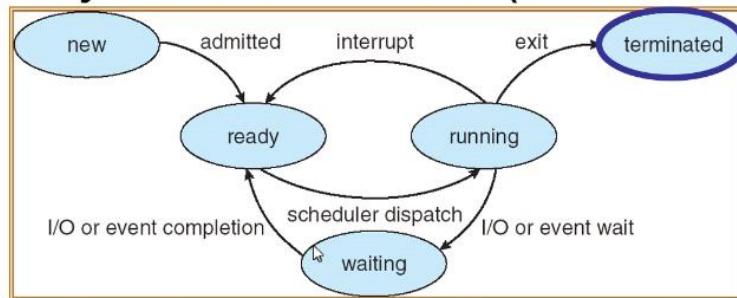


-
- Above, the address space shown is a virtual memory space and can be adjusted to be larger or smaller (depending on the actual physical memory space and hardware architecture)
- One big challenge is figuring out most efficient positioning of the stacks of each thread, so that stacks do not collide in memory space with each other or with the heap – but also trying to be efficient/not waste space so that we don't dedicate too much space for a thread; this is where all of the algorithms come in in terms of CPU scheduling, memory allocation algorithms, etc...analyzing time and space complexities...etc..
- Think: it makes sense that each thread has its own stack and registers, since those are dynamic, but the global data (dynamic in the sense that it is destroyed only at program termination and can be seen by all functions in the program), and the instruction code are static, and even though the heap is not static because it grows and shrinks, it is still static in the sense that there is max allocation limit given to the heap for each program
- Ways to deal with thread space violations: free up unused space (deallocate memory), or move the stack to another location (very expensive)
- Per Thread State
 - In Multiprogramming, the PCB (process control block) is what is used to freeze a process (create a virtual CPU to hold the state of the actual CPU when it holds/pauses a process)
 - But for Multithreading, we have the TCB (Thread control Block); it serves the exact same purpose as the PCB, only instead we are freezing/copying/holding/pausing a thread state instead of a CPU state

Per Thread State

- Each Thread has a *Thread Control Block* (TCB)
 - Execution State: CPU registers, program counter, (PC) pointer to stack (SP)
 - Scheduling info: State (more later), priority, CPU time
 - Accounting Info
 - Various Pointers (for implementing scheduling queues)
 - Pointer to enclosing process? (PCB)?
 - Etc (add stuff as you find a need)
- OS Keeps track of TCBs in protected memory
 - In Array, or Linked List, or ...
 - OS keeps track of TCBs in protected memory (OS memory space is protected memory)
- Life Cycle of a Thread (or Process)

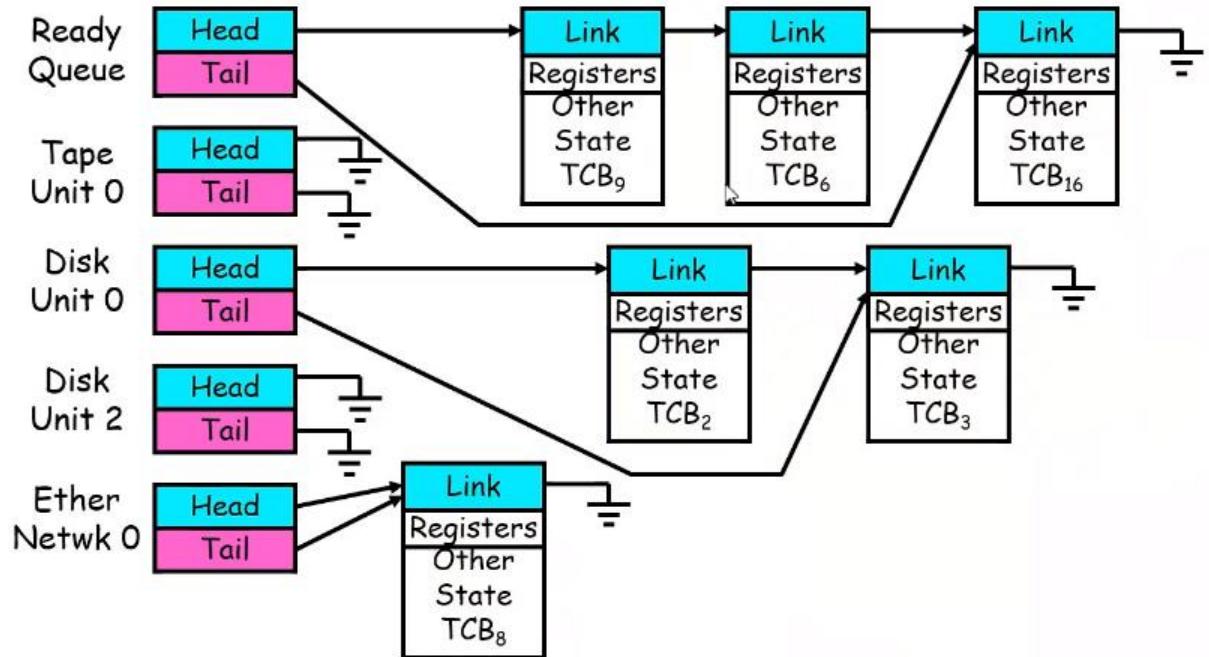
Lifecycle of a Thread (or Process)



- As a thread executes, it changes state:
 - **new**: The thread is being created
 - **ready**: The thread is waiting to run
 - **running**: Instructions are being executed
 - **waiting**: Thread waiting for some event to occur
 - **terminated**: The thread has finished execution
- “Active” threads are represented by their TCBs
 - TCBs organized into queues based on their state

Ready Queue And Various I/O Device Queues

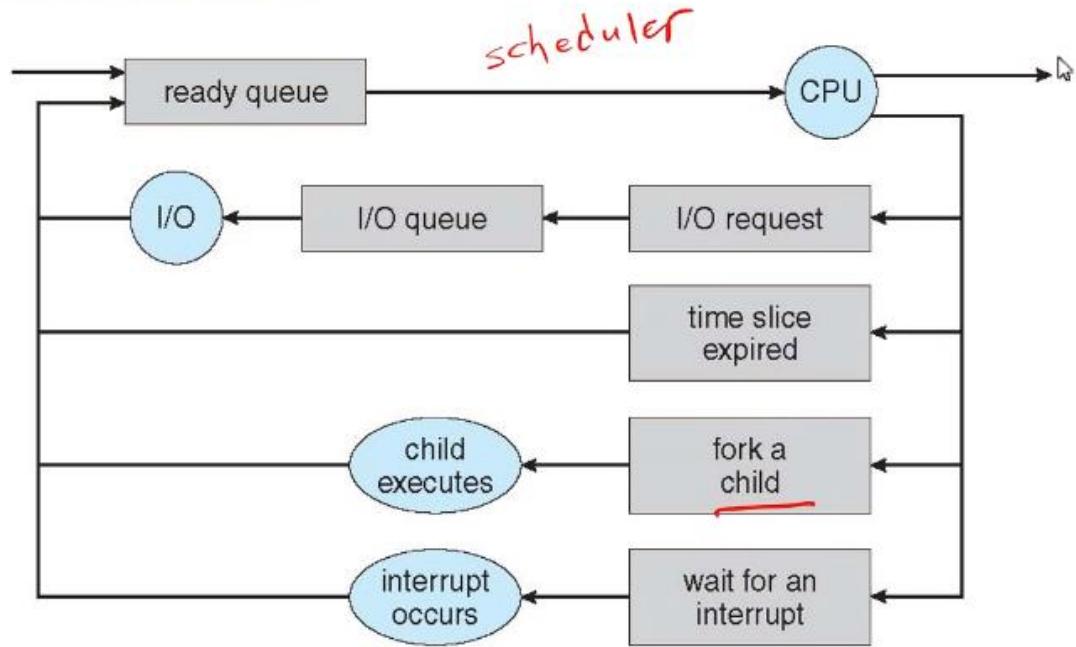
- Thread not running \Rightarrow TCB is in some scheduler queue
 - Separate queue for each device/signal/condition
 - Each queue can have a different scheduler policy



- Above, the ready state may have multiple queues for various tasks (like if a thread wants to read from a disk, or another for if a thread is a network process...etc.)
 - Each of those queues can be organized by some algorithm to determine which queue the scheduler (OS) will pull from

Representation of Thread Scheduling

- Queueing diagram represents queues, resources, flows



- Note: normally during a fork, the parent continues to execute and the child thread or process will move to the ready queue
- Scheduler

CPU Scheduler

- Decides which thread to run
 - From ready list only
- Choose from some algorithms
- Tries to provide fairness, good performance, quality of service (real time), or some other metric (depends on system goals)
-

Scheduler Regaining Control

- Internal Events (voluntary)
 - ex. Disk request, wait for another thread
- External Events (involuntary)
 - Ex. Timer
- Scheduler chooses new thread
 - Now must switch from old thread to new thread
- - When a process is running, OS is not in control
 - Above are examples of when a process or thread within a process (lightweight process) yields control or some external parameters set by the OS or another program forces the OS to regain control so that the scheduler can run again (scheduler runs as part of the OS)
 - Remember when scheduler selects a new thread from the ready queue, the old thread is frozen into its TCB (thread control block) and the new thread is reloaded from its state control block into the CPU cache and registers

Lecture Video 03-1 (week3 – 1/24/22): Critical Section Problem

- Kernel versus User-Mode threads
 - System calls are more expensive than regular function calls
 - So saving and changing states is expensive (the need to cross into kernel mode)
 - User threads: all scheduling can be done in user mode (can still save state of CPU and load another state from a control block into CPU) – no need to cross into the kernel
 - Even lighter than a lightweight thread
 - Scheduler Activation bit will allow the kernel to know that a thread is being split into multiple user threads

Kernel versus User-Mode threads

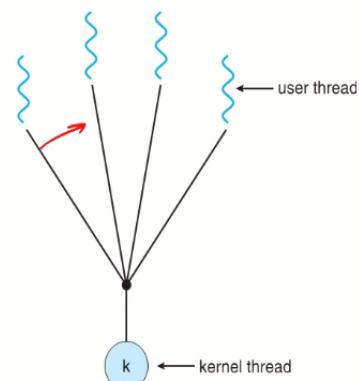
- We have been talking about Kernel threads
 - Native threads supported directly by the kernel
 - Every thread can run or block independently
 - One process may have several threads waiting on different things
- Downside of kernel threads: a bit expensive
 - Need to make a crossing into kernel mode to schedule
- Even lighter weight option: User Threads
 - User program provides scheduler and thread package
 - May have several user threads per kernel thread
 - User threads may be scheduled non-preemptively relative to each other (only switch on yield())
 - Cheap
- Downside of user threads:
 - When one thread blocks on I/O, all threads block
 - Kernel cannot adjust scheduling among all threads
 - Option: Scheduler Activations
 - Have kernel inform user level when thread blocks...
- Multithreading Models
 -

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many
 -
 - Many to one
 - Can switch from one user thread to another user thread without intervention of the kernel; it can be done in user mode
 - Notice that if a thread (kernel thread) is blocked, all user threads from that kernel thread will be also blocked

Many-to-One

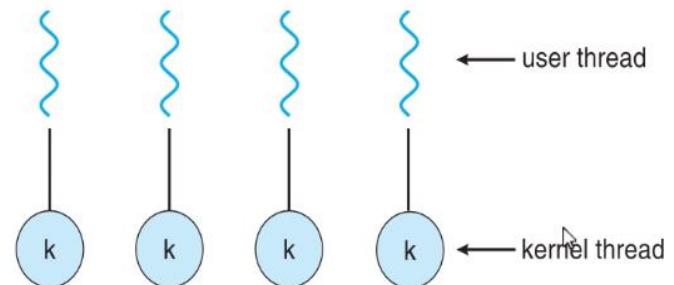
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads
-
- One to one



- Each thread can be scheduled to run and block independently; but because of this it is more expensive (more overhead) since switching a thread requires access to Kernel mode
- Can utilize multicore architectures
- One-to-one is essentially the model we were learning about when we were learning when we first learned about threading in Lecture 2 (week2)

One-to-One

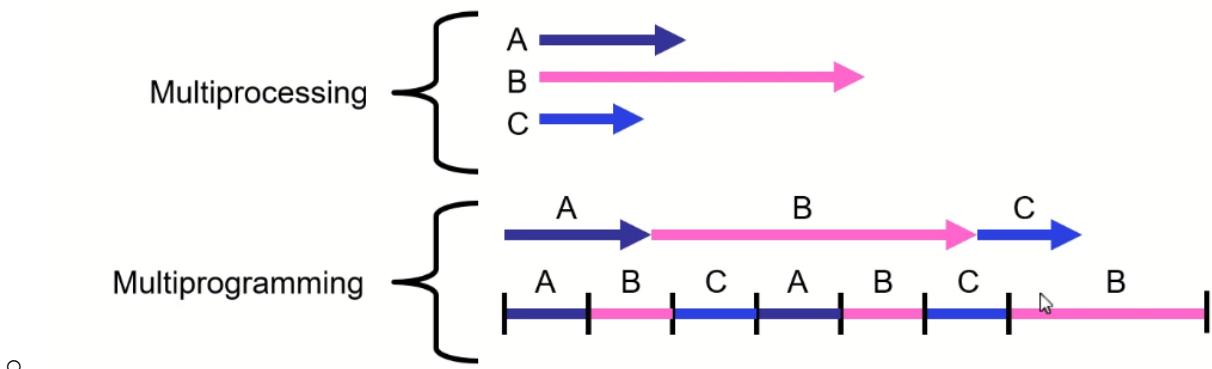
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
 - Windows ✓
 - Linux ✓
 - Solaris 9 and later



-
- Many to Many
 - Allows for flexibility – can use lightweight scheduling (change thread via user mode), can still prevent blocking of all threads by using the more expensive kernel-level thread change; it is essentially a combination of the one-to-one and many-to-one models
- Multiprocessing vs Multiprogramming
 - Multiprocessing: **parallel** processing of threads or other processes via multicore architectures (multiple CPUs)
 - Multiprogramming: **concurrent** processing of multiple threads or processes on the same CPU through interleaving (pausing one process and continuing to run another process); and scheduler is free to run threads in any order
 - User programs don't have any say over when a thread of its program will be scheduled – OS/kernel handles the scheduling

Multiprocessing vs Multiprogramming

- Remember Definitions:
 - Multiprocessing = Multiple CPUs *parallel*
 - Multiprogramming = Multiple Jobs or Processes *concurrency*
 - Multithreading = Multiple threads per Process
- What does it mean to run two threads “concurrently”?
 - Scheduler is free to run threads in any order and interleaving: FIFO, Random, ...
 - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks



- - Notice above, A B and C for multiprocessing is referring to 3 different processors processing different jobs simultaneously
 - Multiprogramming shows two different algorithms of multiplexing; the top shows a scheduler can use a single CPU to switch between tasks and run them concurrently to completion; the bottom shows how a scheduler can use a single CPU to partially complete tasks concurrently using time blocks before switching to the next task until eventually all tasks are completed.
 - We always want to make sure programs give the correct output regardless of how they are scheduled
- Correctness for systems with concurrent threads
 - There's now way to test all cases for correctness
 - Solution: use independent threads
 - Switch() function (context switch) should work; then order of scheduling doesn't matter; in general, all cases are correct for a given concurrent processing algorithm
 - For cooperating threads, they have a shared state between threads so its easier to communicate, however, because of this you can have non-deterministic, non-reproducible results because even though all threads will be updated to the most current state since the global variables and heap are shared, depending on which thread is processed first could cause some bugs which are often intermittent; these are called Heisenbugs, and they can be tolerated as long as a program works perfectly most of the time; when you try to debug it, it can be difficult; you may need to run the program

100s or thousands of times to find the instance where a group of threads were scheduled and processed in such an order that the shared state between threads caused the issue. These kinds of bugs are a “nightmare” and almost impossible to detect it, and once you detect it you cannot reproduce the same bug; so solution is to design our program to avoid this kind of program.

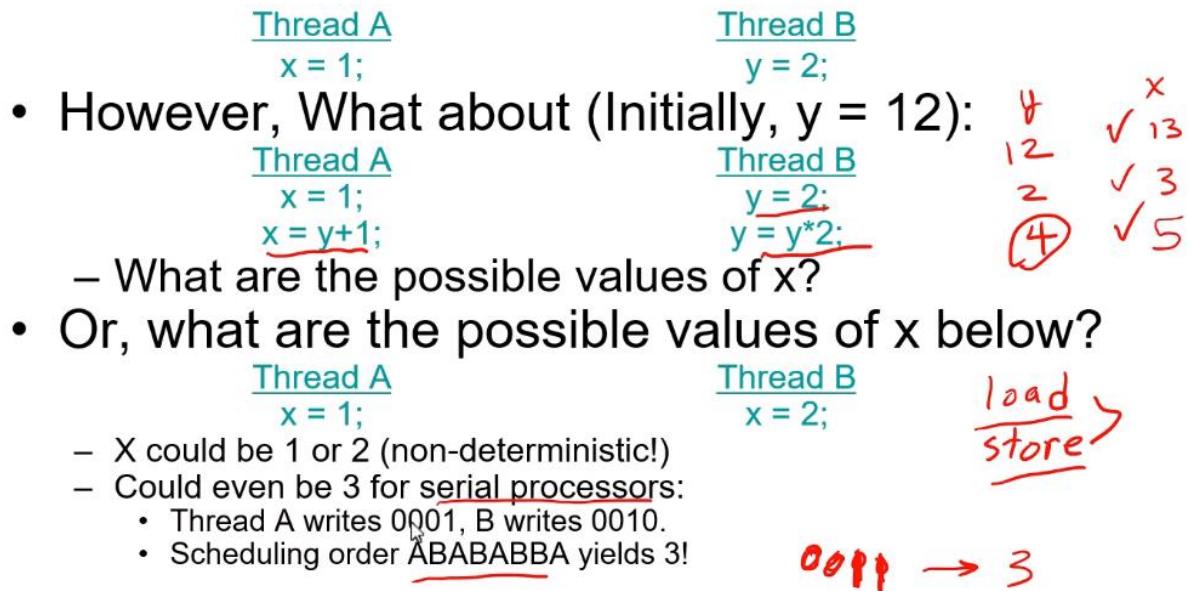
Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
 - Can you test for this?
 - How can you know if your program works?
- Independent Threads:
 - No state shared – separate address space
 - Deterministic \Rightarrow Input state determines results
 - Reproducible \Rightarrow Can recreate Starting Conditions, I/O
 - Scheduling order doesn't matter (if switch() works!!!)
- Cooperating Threads:
 - Same address space, shared state between multiple threads
 - Non-deterministic *global & heap*
 - Non-reproducible
- Non-deterministic and Non-reproducible means that bugs can be intermittent
 - Sometimes called “Heisenbugs”
- - Problem is at the lowest level
 - Remember: context switching can happen at any instruction at the machine level code (assembly language); thus, in the example, $x=1$ could be executed, and then the thread could switch to B and execute one or both statements, and then context could switch back to thread A and finish execution; etc.
 - Notice in the example, thread A shares space and depends on data from thread B, but thread B operates independently according to all code statements even though they share the same space; thus, the final value of X can vary depending on the order of thread scheduling, but thread B contains code where no matter what ever all instructions are executed, the final value of Y will be the same.
 - Notice in the next example where thread A and thread B update X, you have a similar situation where they share space and include code where the final value of X when all threads are completed depends on the order of which thread was processed.

- Load and store commands of CPUs are atomic (not divisible to bit-by-bit operations in terms of scheduling) in modern systems; but in older serial processors you could get more radical bugs as demonstrated below

Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:



- - Example of Concurrent Program
 - Suppose you are trying to do threads (deposit 1000 and deposit 20) at the same time, with no synchronization or concurrency of controls (but the program can still be ran concurrently among threads – and remember, concurrency occurs at the machine-level assembly language code) – then the results could be 120 or 1100 and not the logically correct (and still possible) value of 1120; this is because if both threads acquire the value of “balance” at the same time, and the threads are not sharing the same space, then they will each update “balance” with possible old information (not taking into account the updated value from the other thread that is running at the same time and updated “balance” with a new value)
 - In the example below, it demonstrates (at the assembly level language) a possible order of events (scheduling) in processing both threads concurrently, which can cause the issue
 - In this example, the only way to ensure no deposit is lost, is to run thread 1 to completion, and then run thread 2 to completion, or vice versa – or to share the same address space so the value (the correct state of the necessary variable) will be updated for the other thread; interleaving the threads using concurrency without shared space for this example causes a deposit to be lost.

Example of Concurrent Program

```
int balance = 100;
```

```
void deposit (int amount) {  
    balance += amount;  
}
```

```
int main () {
```

```
    threadCreate(deposit, 1000);
```

```
    threadCreate(deposit, 20);
```

```
    waitForAllDone(); /*make sure all children finish*/
```

```
    printf ("The balance is %d", balance);
```

```
}
```

What are the possible output?

- More Concurrent Programming: Linked lists (head is shared)
 - Using same parameters as the last example except this time the variable that causes the issue is shared, if insert and remove are called and ran concurrently, then there are multiple scenarios where the linked list can be broken or an invalid address space will be accessed (depending on which of the remove or insert lines were executed first, according to the scheduling). In this scenario of concurrent programming for this program, since "head" is shared, when it is updated by one of the threads, it will negatively impact the other thread using the shared "head" variable; Also, for this particular example, even if "head" were not a shared variable, you could potentially have the same results or some other negative effects (for example, trying to do a remove for of a head that was recently updated to a new head, thus breaking the linked list). So either way, results will vary depending on the scheduling of the insert and remove threads
 - Called Race Condition for shared variables between threads
- Race Condition

Race Condition

- This kind of bug, which only occurs under certain timing conditions, is called a ***race condition***.
- Output depends on ordering of thread execution
- More concretely:
 - (1) two or more threads access a shared variable with no synchronization, and
 - (2) at least one of the threads writes to the variable



12

-
- This is a big problem; we need a solution where the result is always consistent and reproducible no matter how threads are scheduled.
-

Lecture Video 03-2 (week3 – 1/24/22): Critical Section Problem

- Critical Section (solution to concurrency issues)
 - No concurrency inside this critical section of code; must be executed one thread at a time
 - Only insert and only delete types of code (from linked lists) can still be necessary to form a critical section for, since multiple insert threads alone could cause bugs, and so can several threads of delete code being called concurrently can cause bugs as well; so it is not just a matter of the insert and delete of a linked list together being a critical section (although that is true too), but even isolated instance of delete and insert and other situation having a similar category of an issue can cause bugs and need to be marked as a critical section of code as well.
 - If a set or sets of code satisfy a race condition, then that code or set of code needs to be considered a critical section.

Critical Section

- Section of code that:
 - Must be executed by one thread at a time
 - If more than one thread executes at a time, have a race condition
 - Ex: linked list from before
 - Insert/Delete code forms a critical section
 - What about just the Insert or Delete code?
- Critical Section (CS) Problem

Critical Section (CS) Problem

- Provide entry and exit routines
 - All threads must call entry before executing CS.
 - All threads must call exit after executing CS
 - Thread must not leave entry routine until it's safe.
- Structure of threads for Critical Section Problem
 - Critical sections of code must call entry() function first, which is a "gate" to check if a thread for the critical section of code is allowed to be created or if it must wait (only one thread at a time for critical code, thus, if there is already an instance of a related thread that is a part of the critical section at the time of entry() call, then the calling thread

- must wait for the current critical thread to be completed); it must do the same before exiting the function by calling exit() function.
- The while loop ensures that at any given moment, only one instance of a given thread (that contains a critical section) is running in the environment at a time.
 - This while loop essentially eliminates concurrency for this part of code (the critical code), while allowing all other noncritical code threads to continue to be ran concurrently (allows for several threads of the same non-critical code)
 - If a shared variable is involved, usually we can consider it critical code

Structure of threads for Critical Section Problem

- Threads do the following

```
While (1) {  
    call entry();  
    critical section ←  
    call exit();  
    do other stuff;  
}
```

- Properties of Critical Section Solution (4 main properties must be satisfied, usually)

Properties of Critical Section Solution

- **Mutual Exclusion:** at most one thread is executing CS.
- **Absence of Deadlock:** two or more threads trying to get into CS => at least one succeeds.
- **Absence of Unnecessary Delay:** if only one thread trying to get into CS, it succeeds
- **Bounded Waiting:** thread eventually gets into CS.
 - Critical-Section Handling in OS
 - For preemptive: need to deal with critical section of code
 - For non-preemptive: no need to worry about race conditions from critical code sections because all threads must be completed by the kernel and must exit kernel mode, or be blocked, or voluntarily yields to the CPU – at which all data will be updated properly before another thread can start; no need to worry about race conditions of critical sections of code

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non- preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
 - Essentially free of race conditions in kernel mode
- Software Solution for Critical Section Problem
 - Atomic instructions: cannot be interrupted (ensuring a computer is reliable in reading and writing); most computer architecture make load and store operations as atomic instructions (such as x86, etc..)
 - However if an operation involves reading or writing multiple data (such as many load and store instruction as a group) it is not considered atomic; just when a single load or store operation is being done it cannot be interrupted – you can

only interrupt between a completed load and store operation but not as one is currently being processed (on the current clock cycle..etc..)

- Int turn = value of 0 means thread 0 can get in (enter processing queue as a thread), and a value of 1 means thread 1 can enter processing queue as a thread.

Software Solution for Critical Section Problem

- Two thread solution
 - Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted
 - The two threads share two variables:
 - int turn;
 - Boolean flag[2]
 - The variable turn indicates whose turn it is to enter the critical section
 - The flag array is used to indicate if a thread is ready to enter the critical section. flag[i] = true implies that thread_i is ready!
- Critical Section solution attempt #1 (2 thread solution)

Critical Section Solution Attempt #1 (2 thread solution)

Initially, turn == 0

id = 0, 1

```
entry (id) {
    while (turn != id); /* if not my turn, spin */
}
```

0, 1, 0, 1, 0, 1 . . .

```
exit(id) {
    turn = 1 - id; /* other thread's turn*/
}
```

- Above, notice this two-thread solution implements mutual exclusiveness – only one thread can get in at a time and it depends on if the other thread is already in.
- Some limitations/problems with the above solution: threads 0 and 1 must always alternate at trying to become materialized; when thread 1 finishes for example, the above code sets it so that thread 0 can enter. But what if a new thread 1 needs to enter again to be executed after the other thread 1 finished? It couldn't; it has to wait for a thread 0 to be processed; and vice versa applies to the 0 thread, thus, consecutive threads of the same type is not allowed with the above solution. Also, if id is set initially (for example) to 1, if a thread 1 never enters, then no thread 0 will ever be able to enter for the same reasons as stated above. So, waiting is not bounded, and Absence of Unnecessary Delay are thus not satisfied for the four main principles that CS code solution should contain (but this example does have the mutually exclusive and absence of deadlock characteristics).
- Critical Section solution attempt #2 (2 thread solution)
 - Problem: violates absence of deadlock principle; if thread 1 sets flag to true, and then there is a context switch and thread 2 sets its flag to true, and then you switch back the context to thread 1, then both threads will keep waiting for the other to set its flag to false, and so you would have an infinite loop due to a deadlock in some scenarios for this code. Notice below for attempt 2, the shared variable "turn" is not used.

Critical Section Solution Attempt #2 (2 thread solution)

Initially, flag[0] == flags[1] == false

```
entry (id) {  
    flag [id] = true;      /* I want to go in */  
    while (flag[1-id]);  /* proceed if other not trying */  
}  
  
exit(id) {  
    flag[id] = false;     /* I am out */  
}
```

- o Critical Section solution attempt #3 (2 thread solution)

Critical Section Solution Attempt #3 (2 thread solution)

Initially, flag[0] == flags[1] == false, turn

```
entry (id) {  
    flag [id] = true;      /* I want to go in */  
    turn = 1 - id;  
    while (flag[1-id] && turn == 1-id); /* proceed if  
                                         other not trying */  
}  
exit(id) {  
    flag[id] = false;     /* I am out */  
}
```

- Deadlock situation from attempt #2 is solved; there is no deadlock issue when both threads are trying to enter at the same time, since after each thread sets turn = 1- id, depending on the context switch of who executes that line of code first, both flags will be true however the turn variable will be set for only one of the threads, thus there will be no deadlock.
- Notice for all three attempts of the 2-thread solution, all satisfied the condition "At most only 1 thread is executing in the CS at once" – mutual exclusive property; and notice that for all two-thread solution attempts, it only allows for one instance of each thread to try to be executed at a time; thus you cannot have a 1 and a 1, or a 0 and a 0 thread trying to simultaneously enter the CS code; you must finish 1, then you can submit for another thread 1, and so

on....but you can have a thread 1 and thread 0 exist in terms of trying to enter the critical section at any give instance.

- This solution works: it is called the Peterson's Solution.
- The "turn" variable is only useful for when both threads are trying to get in; otherwise, the flag variable is only used to determine if a thread can get into the CS; this satisfies the unbounded waiting property since first we check the flag – if it is false we enter, and then if flag is true then we check the turn variable (AND statements, like all execution, is checked from left to right and returns false immediately if the first statement is false)
- So, the solution works, but it is only for two threads.

Satisfying the 4 properties

- Mutual exclusion
 - turn must be 0 or 1 => only one thread can be in CS
- Absence of deadlock
 - turn must be 0 or 1 => one thread will be allowed in
- Absence of unnecessary delay
 - only one thread trying to get into CS => flag[other] is false => will get in
- Bounded Waiting
 - spinning thread will not modify turn
 - thread trying to go back in will set turn equal to spinning thread
- Notice, for bounded waiting, if a thread is spinning, it will stop when the other thread exits and sets its flag to false; and even if another thread of the same type tries to enter the CS after one exited the CS, when it reaches the "turn" variable it will set it equal to the spinning thread turn, so that the spinning thread will get priority and in fact not spin forever and eventually enter the CS.
- Critical Section Problem – multiple threads solution

Critical Section Problem multiple threads solutions

- Bakery algorithm
- It's a mess
 - Trying to prove it correct is a headache
- - Bakery Algorithm very complex and messy, but it is nonetheless a software solution that can handle multiple threads; however, it is not very often used because of its complexity.

Lecture Video 03-3 (week3 – 1/24/22): Critical Section Problem

- Hardware Support for CS solutions (the previous few slides were a software solution)
 - Easier solution is for developers to include it in the hardware design.
 - So far we usually have it as load and store instructions are atomic. But one solution is to make a Read/Modify/Write instructions (essentially a load, modify, and store instruction set) a single atomic instruction. Thus by doing this, you cannot interrupt between load and store instructions for some tasks even after a load finishes or a store finishes; rather a load, modify, store instructions will all be treated as one atomic instruction that must complete the full sequence without interruption (that sequence is essentially an indivisible critical section of instruction code)

Hardware Support

- Provide instruction that is:
 - Atomic
 - Fairly easy for hardware designer to implement
 - Read/Modify/Write
 - Atomically read value from memory, modify it in some way, write it back to memory
 - Use to develop simpler critical section solution for any number of threads.
- • Atomic Operation

Atomic Operation

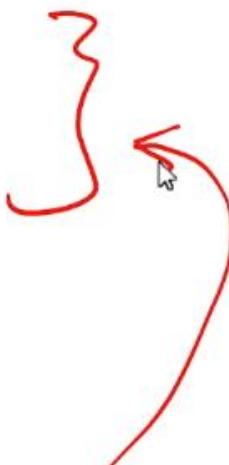
- An operation that, once started, runs to completion
- Indivisible
- Ex: loads and stores
 - Meaning: if thread A stores “1” into variable x and thread B stores “2” into variable x about the same time, result is either “1” or “2”.
 - - Serial processors where load and store are not atomic can result in a “3” being written to x. But for atomic operations, only 1 or 2 could be written in the above scenario.

- Test and Set (atomic instruction)
 - After the test and set function runs (as a single instruction, aka atomically), and since target is a reference (thus it is shared), all other calls to the function will have their target set to true.

Test-and-Set

- Many machines have it

```
bool TestAndSet(bool &target) {  
    bool b = target;  
    target = true;  
    return b;  
}
```



- **Executes atomically**

- Truly hardware-level instructions
- CS solution with Test-and-Set

CS solution with Test-and-Set

Initially, $s == \text{false}$;

```
✓ entry () {
    bool spin;
    spin = TestAndSet(s);
    while (spin)           busy waiting
        spin = TestAndSet(s);
}
```

```
✗ exit() {
    s = false;
}
```

- Notice this hardware solution is much simpler and can handle any number of threads, not just two like the Peterson Solution.
- However, there is one problem: the spinning thread will waste CPU cycles when it keeps calling the Test And Set function every loop (since test and set function is atomic, it will need several CPU cycles to read, modify, and write), which is not good; produces a lot of overhead for threads that are just waiting: called “busy waiting”. So, the solution works and satisfies the 4 principles for CS code solutions, but it does present CPU and resource inefficiencies that need a solutions.
 - We will discuss a solution later that does uses the hardware solution and avoids busy waiting
- Basic idea with atomic instructions
 - When multiple threads are waiting, as soon as one thread exits, the next thread that is switched into the CPU will immediately enter the CS and reset the shared variable to “true” so that all other spinning threads will continue to spin

Basic Idea With Atomic Instructions

- Each thread has a local flag
- One variable shared by all threads
- Use the atomic instruction with flag, shared variable
 - On a change, one thread to go in
 - Other threads will not see this change
- When done with CS, set shared variable back to initial state.
 - Other Atomic Instructions – swap instruction
 - Swap value of two memory location in a single instruction (atomic)
 - Same idea as test and set function: read and reset a value
 - This is one of the hw problems: write a CS entry and exit function using the atomic swap instruction instead of the test and set instruction

Other Atomic Instructions

The definition of the Swap instruction

```
void Swap(bool &a, bool &b) {  
    bool temp = a;  
    a = b;  
    b = temp;  
}
```

- Problems with busy-waiting CS solution

- Priority Inversion problem: High priority cannot enter the CS because there is a thread already in the CS – namely the low priority thread – which the low priority thread cannot exit() because it does not have a CPU to be processed on because the CPU is blocking it because there is a higher priority thread that needs execution.
 - Thus, they have circular waiting: a deadlock
- Solution is to block when waiting for CS; remember blocking refers to “go to waiting state”; remember a thread or process can be in three states; running, waiting, and ready. Thus, if a spinning thread instead stops spinning and goes to waiting (is blocked), then it can solve two problems: free up the CPU from busy-waiting (wasting cycles calling the test and set atomic instruction), and it can free up CPU to allow higher priority threads to enter the CS and be processed (thus solving the deadlock issue).

Problems with busy-waiting CS solution

- Complicated
- Inefficient
 - Consumes CPU cycles while spinning
- Priority inversion problem
 - Low priority thread in CS, high priority thread spinning can end up causing deadlock

Solution: block when waiting for CS

waiting

○

⋮

Lecture Video 04-1 (week4 – 1/31/22): Locks and Semaphores

- Remember we limit concurrency inside the critical sections of code; only one thread can be executed at a time between a group of threads apart of the critical sections of code.
 - Related threads cannot run concurrently or in parallel; they must be executed one by one (so as to mitigate the race condition).
- Atomic Instruction: Swap

Test And Set (s)

Atomic Instruction: Swap

The definition of the Swap instruction

```
spin=true      s
void Swap(bool &a, bool &b) {
    bool temp = a;
    a = b;
    b = temp;
}
```

- Similar to test and set; but here you let the first variable (a) be the spin variable to say whether or not a thread should spin (because there is already a thread in the CS), and let variable b represent the set variable so that we know that the given thread type that calls the function is ready and waiting to enter the CS. Any other CS thread that tries or is waiting to enter will have to worry about the “spinning” and “set” value that is set to true since a thread is in the CS. Of course once a thread exits the CS, it will set the “set” variable to false to allow a new thread into the CS. Additionally, if any thread tries to enter the CS and there is already a thread attempting to enter the CS (although this implementation does not include this): it will go back to the waiting state – this will save CPU resources so that you don’t have many instances of CS threads busy-waiting by constantly calling the swap function to see if it can enter the CS. To save even more CPU space, you could actually have all threads that encounter spinning=true and set=true when they call the swap function to leave the busy-waiting state immediately and go back to the waiting state until the dispatcher calls them again (reloads their state back onto the CPU for execution).
- Mutual Exclusion Swap
 - Notice the value of swap is always set to true by default at the start of the function, and that the spin variable is a local variable for each thread trying to enter the CS, while s is a shared variable among CS threads attempting to enter the CS; the only way the spin variable changes is if the s variable is eventually set to false; thus, if s is true when the

swap function is called, all subsequent swaps will still leave the spinning value set to true as well as the s variable will be still set to true since it will be a true-for-true swap. Essentially, if s is true when the swap function is reached, the thread will automatically spin since spin is a **local variable** and thus even between states of other threads, the value of spin cannot be changed until the local function calls the swap function again which can potentially change the local variable spin depending on if s has been changed by another thread's state; if s is false, the thread will automatically be allowed to enter without any issue of switching states between the swap function, since if s is false then it will be set to true at the end of the atomic swap function and then no other threads will be allowed to enter the CS (and the inner swap function under the while-loop will not execute since while(false) will cause an exit from that loop – and thus leaving s as true; and between loading other thread states, remember, the spin variable cannot suddenly be changed to “true” before the while(spin) line is executed since it is a local variable; thus, the entry function will run to completion meaning that the thread will be allowed to enter). The reason this is so simple and effective is because of the atomic swap function.

- Notice too, since spin is set to true by default and is a local variable that causes the spinning, the value of s after the any swap call will always be set to “true”; when spin is eventually set to false, notice, it will not enter the while loop and thus it will not perform a swap and set s to false; it only enters the while loop when spin=true and thus will continue to set s=true during the swap function even if when any swap is called s=false or true → this will always still set s=true, but spin= false or true. S can only be set to false by the exit() function.

Mutual Exclusion with Swap

Initially, $s == \text{false}$;

```
entry () {
    bool spin = true;
    Swap(spin, s);
    while (spin)
        Swap(spin, s);
}
```

\longleftrightarrow $\text{spin} = \text{TestAndSet}(s)$

\nearrow \searrow busy waiting

```
exit() {
    s = false;
}
```

- So, although I discussed how any thread caught spinning can go back to waiting state to save CPU resources and prevent loops; this particular mutual exclusion with swap function does not implement the busy-waiting/deadlock mitigation measures; it only solves the critical section problem. Remember the above solution for the CS problem can handle any number of threads (and of multiple different types).
- Mutex Locks
 - Essentially the same as the mutual exclusion function using atomic swap instruction:

Mutex Locks

- OS designers build software tools to solve critical section problem. Simplest is mutex lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
 - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
 - Usually implemented via hardware atomic instructions such as TestAndSet, swap, etc.
- But this solution requires **busy waiting**
 - This lock therefore called a **spinlock**

○

Mutual Exclusion with Swap

Initially, $s == \text{false}$;

```

entry () {
    ← Acquire()
    bool spin = true;
    Swap(spin, s);
    while (spin)
        Swap(spin, s);
    } ← release()
    exit() {
        s = false;
    }
  
```

$\leftarrow \text{Acquire}()$

$\leftarrow \text{Swap}(\text{spin}, s)$

$\leftarrow \text{spin} = \text{TestAndSet}(s)$

$\leftarrow \text{busy waiting}$

- Naïve use of Interrupt Enable/Disable
 - Basic idea: allow threads trying to enter the CS to disable and reenable the Interrupt ability; this essentially treats the entry function as an atomic function since a thread trying to enter the CS will be allowed to run the entire entry function without interrupts (context switching from one thread to another; which can interrupt non-atomic instructions being executed inside the entry function).
 - Remember, threads that have already entered the CS can be interrupted without issues because that is the whole purpose of the CS solution; so that no other related threads will be being executed during context switches (concurrency) that can cause a conflict; thus related threads cannot be executed concurrently; only non-related threads can be executed concurrently (through context switching).
 - Remember the OS (or OS dispatcher) is the scheduler of threads and processes

- As a side note: usually network processes and completion of I/O take a high priority; so, OS will often prioritize CPU to stop what it is doing and process network or completed I/O instructions (for example, someone presses return on their keyboard to send their written input).
 - Notice then: when someone PC is running slow, one excellent thing to do would possibly be to put the device in airplane mode or shut down all internet ports temporarily while I run PC diagnostics that don't require network connection (some installations, virus scans, etc..).
- Notice this solution of disabling interrupts will only work for uniprocessors (single CPU/1 core) – it will not work for multiprocessors (multiple CPUs / CPU cores).

Naïve use of Interrupt Enable/Disable

How can we build multi-instruction atomic operations?

- Recall: dispatcher gets control in two ways.
 - Internal: Thread does something to relinquish the CPU
 - External: Interrupts cause dispatcher to take CPU
- On a uniprocessor, can avoid context-switching by:
 - Avoiding internal events (although virtual memory tricky)
 - Preventing external events by disabling interrupts

Syscall

Consequently, naïve implementation of locks:

```
LockAcquire { disable Ints; }  
LockRelease { enable Ints; }
```

- So you would place lock acquire at start of critical section entry function to disable interrupts, and then place lock release at end of critical section entry function to reenable interrupts. This essentially makes a thread that contains CS code an atomic instruction – and a potentially very long one which can be problematic. Remember, locks are an alternative CS-problem solution to the Swap atomic function and the Test-and-Set atomic function.
 - Problem: what if CS thread is long? System will be hung up on one thread; performance will suffer due to lack of concurrency (long pauses potentially if many CS threads consecutively are being executed, or if CS of one thread is very long).

- lots of pausing in the system will not make for a smooth interaction between the user and the machine and will affect prioritization due to the lack or lapse in concurrency. So this solution works and meets the 4 requirements of a CS problem solution, however at the cost of concurrency; remember we want a solution that does not affect the performance of the computer while still avoiding crashes due to CS threads (threads with race conditions).

Naïve use of Interrupt Enable/Disable: Problems

Can't let user do this! Consider following:

```
LockAcquire();  
While(TRUE) { ; } 
```

Real-Time system—no guarantees on timing!



What happens with I/O or other important events?

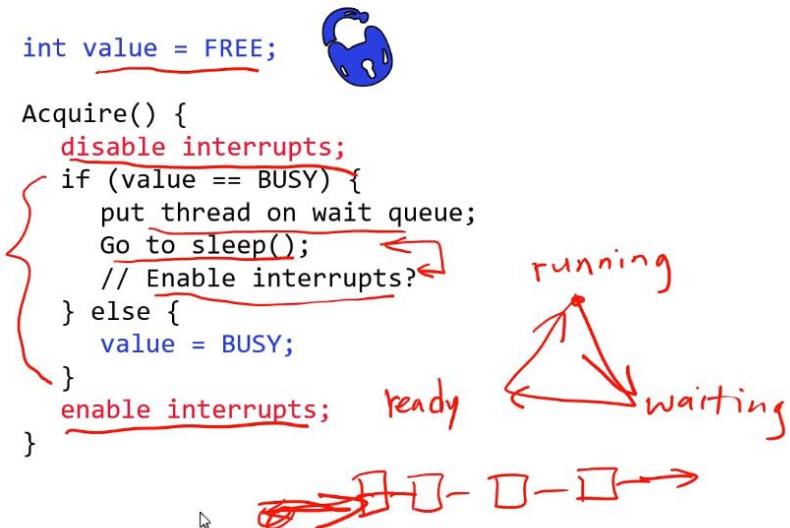
- Critical Sections might be arbitrarily long
- “Reactor about to meltdown. Help?”
-
- Better Implementation of Locks by disabling interrupts
 - Disable interrupts for a short time (using an acquire lock function);
 - Locks are basically the method used to immediately send threads to waiting state instead of busy-waiting (using up CPU resources) and prevents deadlocks; but it must be implemented carefully so that the system does not get paused or lack performance since disabling interrupts for a long period of time can be bad. Solution is to introduce a lock variable that acts as the lock instead, that allows only one thread at a time to acquire a lock while not keeping interrupts disabled; if the lock is “busy”, that means a thread will unsuccessfully acquire a lock and will immediately be sent to waiting. If lock is not busy, then the given thread will successfully acquire the lock.
 - The short interrupt disable time is negligible compared to if threads disabled interrupts for long periods and then reenabled them after their CS code completed– it is only a few 100 nano seconds more or less.
 - Threads waiting for a lock go into the FIFO lock queue
 - And since that thread is waiting for a lock and cannot be executed yet, we also set its thread state from running to waiting (send it to sleep).
 - So, once a thread finally acquires a lock from the lock queue, it will go from the waiting state to the ready state, and then from ready to running as the OS schedules it to run.
 - Note that Acquire and Release are multi-instruction atomic functions because their first instruction disables interrupts (no context switching when this function is being

executed on the CPU) so that one lock cannot be acquired by more than one thread at the same time (violates mutual exclusion requirement).

- Reenabling interrupts is essential when the atomic function returns, or else no other thread will be able to acquire a lock, and system interrupts disabled will cause the other previously mentioned performance issues.

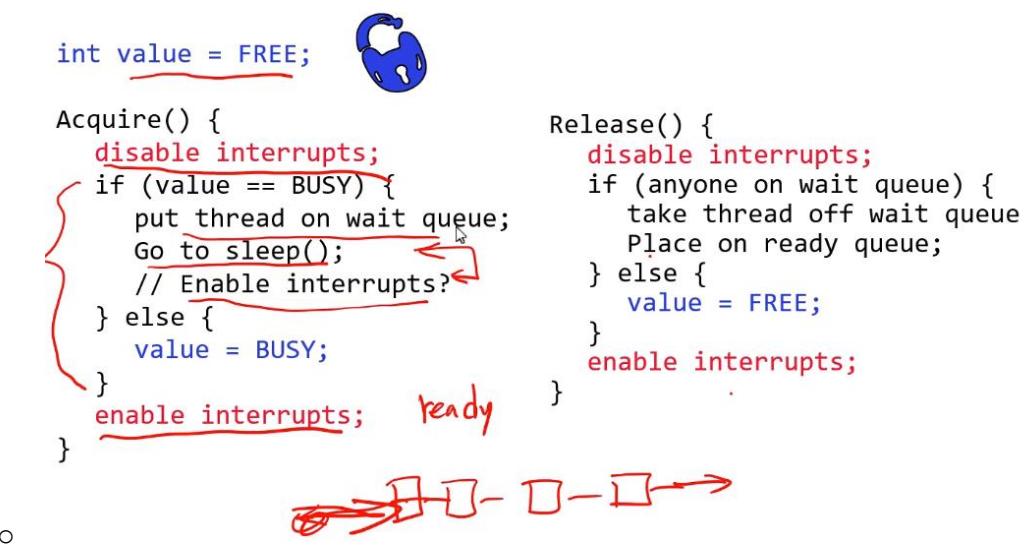
Better Implementation of Locks by Disabling Interrupts

Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable



Better Implementation of Locks by Disabling Interrupts

Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable



- Note: “maintain mutual exclusion” refers to the fact that no other functions can **concurrently** operate on the shared “value” variable that says whether a key is free or busy. Thus, these atomic functions contain critical section code which protects itself from race conditions since the function itself is atomic (by using disable and enable interrupts instructions). This means that these functions are their own solution to their CS problem (and thus satisfy the four requirements). However, it does this through “blunt force” because it takes advantage of the OS commands of interrupt in order to get exclusive, uninterrupted access to the CPU when it is being executed.
- New Lock implementation: Discussion

New Lock Implementation: Discussion

- Why do we need to disable interrupts at all?
 - Avoid interruption between checking and setting lock value
 - Otherwise two threads could think that they both have lock

```
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

**Critical
Section**

- Note: unlike previous solution, the critical section (inside **Acquire()**) is very short
 - User of lock can take as long as they like in their own critical section: doesn't impact global machine behavior
 - Critical interrupts taken in time!
- Thus we allow OS and other threads to resume normal activities (concurrent processing) since interrupt is reenabled after the small critical section code (sandwiched between disable and enable interrupts serving as a multi-instruction atomic operation to allow the CS to complete without any interruptions) completes. The “value” variable acts as the key to the “lock” that a thread can use so that it can execute without worrying about another CS interrupting and thus causing a race condition, and so that there will be no busy-waiting or deadlocks.
 - Only when a thread that is using the lock releases the lock can it be acquired by another thread. So you see, it is very similar to Test-and-Set and Swap solutions; and like those solutions it uses an atomic instruction, however this solution mitigates for all of the other problems that arise.

- Interrupt Re-enable in Going to Sleep
 - How do we send a thread to sleep and at the same time enable the interrupt?

Interrupt Re-enable in Going to Sleep

- What about re-enabling ints when going to sleep?

```

Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
    } else {
        value = BUSY;
    }
    enable interrupts;
}

```

Enable Position



- Before Putting thread on the wait queue?
 - Release can check the queue and not wake up thread
- - If you place enable interrupt before putting the thread on the waiting queue for a lock, then a context switch can happen such that a Release function takes place and then tries to place the next thread in the lock wait queue into the ready state, but the thread will not be in the wait queue because of where the interrupt may occur. Remember, once Release returns, it will not be called any more until another thread acquires a lock. The acquire function is also only called once per thread; thus once Acquire is finished, the thread will be placed in the waiting queue and sent to wait state indefinitely since it was supposed to be the next thread to acquire a lock and call a release but got interrupted in its acquisition of the lock. So essentially, new threads that can still use their 1-time call to the Acquire function will be able to get the (value=free) lock as long as they are not also interrupted, but the locks that already called acquire and got interrupted before being placed in the wait queue will never be executed and stay in the wait state (sleep).

Interrupt Re-enab~~x~~ in Going to Sleep

- What about re-enabling ints when going to sleep?

```

Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
    } else {
        value = BUSY;
    }
    enable interrupts;
}

```

Enable Position

- Before Putting thread on the wait queue?
 - Release can check the queue and not wake up thread
- After putting the thread on the wait queue
 - Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
 - Misses wakeup and still holds lock (deadlock!)
- - Placing the enable after placing thread on wait queue may cause this issue: an interrupt may occur and the Release function may run and place the thread in the wait queue in the ready state (and keep value=busy since there was a thread in the queue to receive the lock), but when the thread that was interrupted is placed back on CPU, then that thread will continue executing the next line, which is Go to sleep(), which will then cause that thread to go back to the wait state (go back to sleep) while it is holding a lock (value=busy), and then the acquire function exits: thus the next time any thread calls Acquire, including the function that has the lock but missed its wake-up by going back to sleep, no thread will be able to acquire a lock (value=busy), and the thread that has the lock will never use it to execute its critical code and call Release() so as to allow new threads to acquire the lock.
 -

Interrupt Re-enable in Going to Sleep

- What about re-enabling ints when going to sleep?

```

Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep(); ----->
    } else {
        value = BUSY;
    }
    enable interrupts;
}

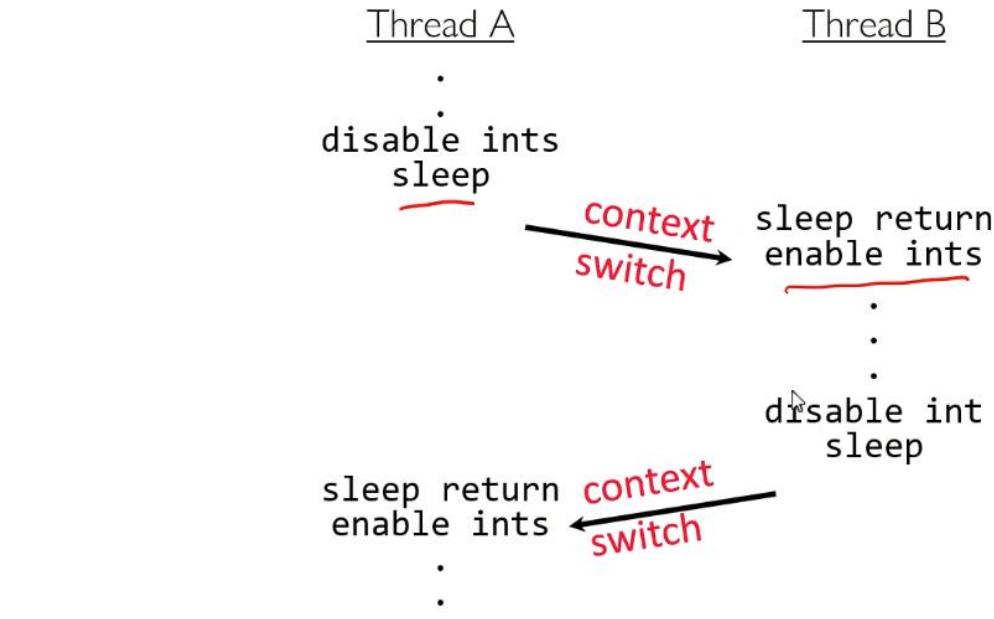
```

Enable Position

- Before Putting thread on the wait queue?
 - Release can check the queue and not wake up thread
- After putting the thread on the wait queue
 - Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
 - Misses wakeup and still holds lock (deadlock!)
- Want to put it after **sleep()**. But – how?
 - How to re-enable after Sleep()

How to Re-enable After Sleep()?

- In scheduler, since interrupts are disabled when you call sleep:
 - Responsibility of the next thread to re-enable ints
 - When the sleeping thread wakes up, returns to acquire and re-enables interrupts



- BIG IDEA: we need to reenable interrupts because when the thread goes to sleep, it will **voluntarily** give up the CPU to another CS thread (because going to sleep = go to wait state; thus there is nothing else to do for a given thread at the given moment). We don't want interrupts disabled in the system for the entire time that a thread is waiting to be processed. Thus, solution is placing it after sleep, but only because when the next thread is woken up from sleep (goes to ready state) it will return to acquire() function and finish execution and re-enable interrupts again. Remember, a thread will wake up when it acquires a lock, thus after it wakes it is safe for the thread to re-enable interrupts – even if it gets interrupted, its okay because that is the whole point of using this lock method so that CS threads are processed in such an order that other CS threads will not conflict with each other (only one CS thread at a time can have a lock and will run without any other CS threads changing CS variables that could create indeterminate results).

- Notice how it would be placed after the if statement and after the else statement; it is still necessary after the else statement because if the else executes, it means the lock is free and thus the thread can acquire the lock, stay in the ready state, skip the acquire lock queue, and of course re-enabling interrupts is critical. The re-enable interrupts is after the if statement so that whenever a thread goes to sleep[give up CPU] (because there is a acquire lock queue), the next thread that is first in line in the queue will be re-awakened[set

in ready state] and will re-enable interrupts. So the whole point of placing the enable after the if and after the else statements is to make sure that interrupts are enabled as much as possible – we want to minimize the amount of time interrupts are disabled. Remember, when interrupts are disabled, context switching can then only happen **voluntarily**.

- Better Locks using test and set (smaller atomic instruction)
 - no disabling of interrupts needed; better solution.
 - Guard integer essentially acts as the enable/disable interrupts; 0 is representative of interrupts being enabled; and 1 = interrupts disabled.
 - This CS solution works for multiprocessor systems because guard and value are in memory, and are shared variables among all processors sharing that same physical memory
 - We still do a busy-waiting, but only for a very short time only to check the lock values (as discussed before). But deadlock issue and long busy waiting issue is resolved.
 - Note: the busy waiting is not waiting for a thread to leave the CS (of the actual thread, not the CS after the while() statement in the Acquire or Release function), but it is just waiting for other waiting threads to check the flag values before heading into the rest of the function. So If multiple threads call Acquire() or Release(), they may have to busy wait (calling test and set function in a while loop) only for a small time while the other threads are allowed to switch into context to check the guard flag value and determine if they can proceed into the rest of the instructions.

Better Locks using test&set

- Can we build test&set locks without busy-waiting?
 - Can't entirely, but can minimize!
 - Idea: only busy-wait to atomically check lock value

```
int guard = 0;
int value = FREE;
```



```
Acquire() {
    // Short busy-wait time
    while (test&set(guard));
    if (value == BUSY) {
        put thread on wait queue;
        → go to sleep() & guard = 0;
    } else {
        value = BUSY;
        guard = 0;
    }
}

Release() {
    // Short busy-wait time
    while (test&set(guard));
    if anyone on wait queue {
        take thread off wait queue
        Place on ready queue;
    } else {
        value = FREE;
    }
    guard = 0;
}
```

- Note: sleep has to be sure to reset the guard variable
 - Why can't we do it just before or just after the sleep?
- Note: go to sleep() & guard=0 must be done atomically through the hardware, otherwise you can have the same issues as discussed before with the enable interrupt version of this code.
- Locks using Interrupts vs. test&set

Locks using Interrupts vs. test&set

Compare to “disable interrupt” solution



```

int value = FREE;

Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if (anyone on wait queue) {
        take thread off wait queue;
        Place on ready queue;
    } else {
        value = FREE;
    }
    enable interrupts;
}

```

Basically replace

- disable interrupts → while (test&set(guard));
- enable interrupts → guard = 0;

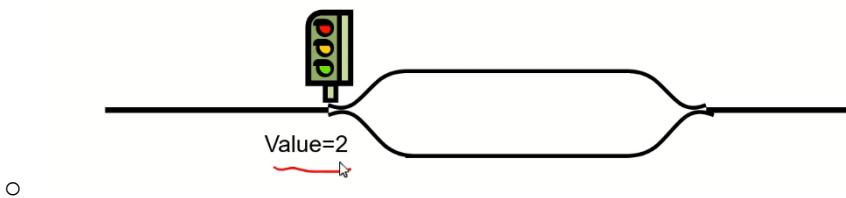
o
o

Lecture Video 04-2 (week4 – 1/31/22): Locks and Semaphores

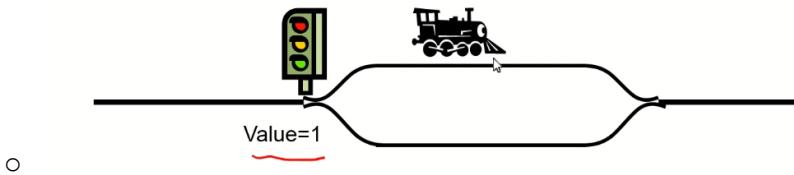
- Semaphores (Dijkstra)
 - Dijkstra is one of the most famous Computer Scientist, from the 1970s and 1980s; he is also the inventor of the SPF (shortest path first) algorithm; he is from Holland (he is Deutsch)
 - P and V in the Deutsch language are alphabetical and they refer to the first letter of two words for decrease and increase.
 - Integer value is equivalent to number of resources available; Semaphores manage the allocation of resources among the threads or processes

Semaphores (Dijkstra)

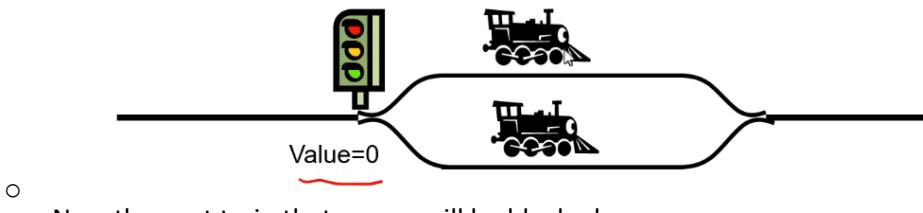
- Synchronization tool that does not require busy waiting.
- Semaphore is an object contains a (private) integer value and 2 operations.
 - P operation, also called Down or Wait
 - Acquire a resource
 - V operation, also called Up or Signal
 - Release a resource
- Semaphores are “resource counters”.
- - Semaphore from railway analogy
 - Semaphore from railway analogy
 - Here is a semaphore initialized to 2 for resource control:



- Semaphore from railway analogy
 - Here is a semaphore initialized to 2 for resource control:

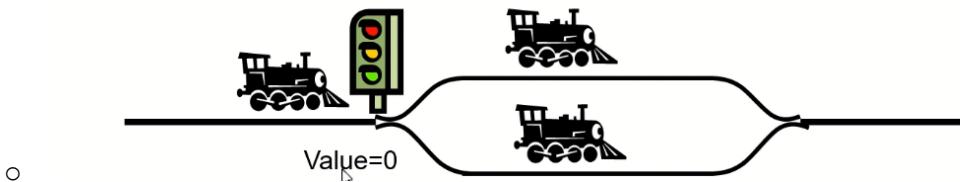


- - Semaphore from railway analogy
 - Here is a semaphore initialized to 2 for resource control:



- - Now the next train that comes will be blocked:

- Semaphore from railway analogy
 - Here is a semaphore initialized to 2 for resource control:



- When a train leaves, value increases, and then another train can enter and then decrease value again to reflect another train has entered and is using up a resource.
- Semaphore Implementation
 - So: every semaphore has a separate waiting queue (struct process *L;).
 - Remember, in the 70s and 80s there was no multithreading (parallel processing), so when you see the word "process" it is referring to a thread (lightweight process) and processes that could be in the waiting queue.

Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {
    int value;
    struct process *L;
} semaphore;
```

- Assume two simple operations:

- block(), suspends the process that invokes it.
- wakeup(P), changes the thread from the waiting state to the ready state.

- - Notice this is very similar logic to Locks as discussed earlier

- Implementation of Semaphores
 - Remember “value” is a private variable
 - Notice with this implementation there is no busy waiting
 - Block() = goToSleep() = go to waiting state = voluntarily give up CPU

Implementation

- Semaphore operations now defined as

P(S) wait(S):
S.value--;
if (S.value < 0) {
 add this process to S.L;
 block();
}

V(S) signal(S):
S.value++;
if (S.value <= 0) {
 remove a process P from S.L;
 wakeup(P);
}

- The wait and signal operations are atomic.

-
- Critical Sections with Semaphores
 - Semaphores easily solve the Critical Section problem
 - Semaphore mutex = 1: mutex refers to “mutual exclusion”; the name mutex and setting it equal to one is intended to show how only one thread can get into the critical section at a time; only one will succeed; so you see it is fulfilling a requirement of the 4 principles to solve the CS problem.
 - Remember “wait” is a function trying to get into the CS, and “signal” is when a thread leaves the CS and signals for the next thread to enter the CS. So semaphores can be set to other values other than 1 as discussed, but for mutual exclusion (which is necessary when solving the CS problem) it is set to 1 so that its implementation can be useful in solving the problem of CS code.
 - Semaphores can also be used for general synchronizations

Critical Sections with Semaphores

```
semaphore mutex = 1;  
entry()  
    wait(mutex);  
exit()  
    signal(mutex);
```

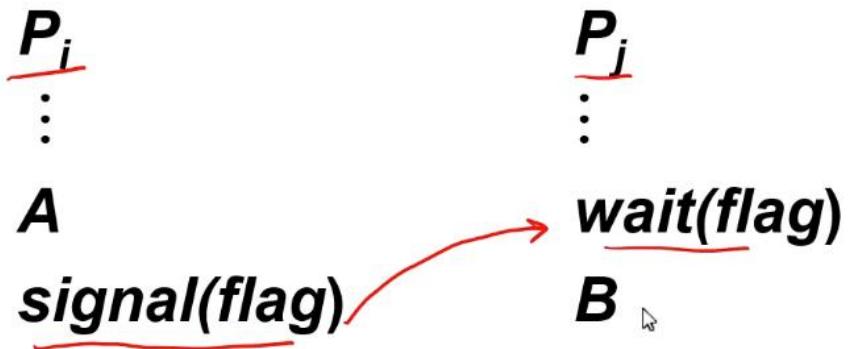
- For mutual exclusion, initialize semaphore to 1.

-
- Semaphore as a General Synchronization Tool
 - Most of the time, threads are independent and don't have CS code and can run in any order relative to all other threads and processes; Semaphores can also be used as a synchronization tool to process threads that are still related (cooperating or corroborating threads = subprocesses that are all related to a main process or main processes; could be a group of threads that contain CS code, or it could be a group of threads that do not contain CS) but that do not contain CS code (like a specific kind of shared variable that can affect an outcome depending on which thread does execution first); for example, it could be several threads stemming from a single process that can all be executed in any order, or a group of threads that still must be done in a certain order/sequence (but that do not contain any conflicting shared variables; however they may still need to wait on information from another thread, or wait for a related thread to update a shared value and thus go to a waiting state until that shared value is updated to the necessary value) – a semaphore can help with this synchronization process so that all related threads can complete and thus complete a full process. This synchronization is important, otherwise a main process may not be completed in a timely manner if its related threads do not all get scheduled properly or in good timing; goal is to make cooperating threads finish in a timely manner using efficient algorithms, i.e. Semaphores.
 - Note: synchronization may be necessary for threads related to one main process, or for threads between many related main processes.

- Notice if flag is set to 0; you can set signal() which will increment (similar to release) the flag by 1 into the part of a thread that must execute first (thread A), and place a wait() which blocks a thread before any of the other code executes in the thread that must execute its respective parts of the code secondly (thread B). By doing this, if the secondary thread B is loaded into CPU and reaches a line that must be executed AFTER another thread (thread A) executes some particular lines of its code, it will be blocked (give up CPU and go to wait state). The only way that thread can be woken up is by thread A executing its lines of code and reaching the signal() in order to release thread B and wake it up so that it can finish its execution. Notice: several of these signal() and wait() clauses can be placed in the same threads or multiple threads strategically so that the threads can all be synchronized even for very long threads that have dependencies scattered in various parts of its code; hence, Semaphores can be very useful in writing synchronization algorithms and implementing synchronization code when necessary.

Semaphore as a General Synchronization Tool

- Execute *B* in P_j only after *A* executed in P_i
- Use semaphore flag initialized to 0
- Code:



- - Bounded Buffer Problem (type of synchronization problem)
 - Mutual exclusion necessary = there is a race condition possible = critical section problem

Bounded Buffer Problem

- There is one Buffer object used to pass objects from producers to consumers. The problem is to allow concurrent access to the Buffer by producers and consumers, while ensuring that
 - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
 - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
 - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Examples: TCP buffers; applications read data from network buffers, which are buffers that are updated from data sent through IP traffic. OS must manage these data buffers (and other buffers) so that when application is reading and writing to these network buffers, the correct amount of data is in or out of the buffers before reading and writing and transmitting...etc..
 - So if an application is too fast, and the TCP buffers are full and still transmitting data or dealing with other traffic, that excess application can be blocked (queued, essentially).
 - Also if TCP data is being transmitted too fast and thus trying to send when the buffer is empty, then TCP threads can be blocked so that it will transmit AFTER buffer is full (for example, a control flag/bit must be set before a transmission TCP thread can be processed by the OS).
 - So most of the time, with the buffer problem, threads can be done independently, but only up to a certain bound
- Bounded Buffer (1 producer, and 1 consumer)
 - Can use semaphore in our solution
 - Two situations: sometimes the consumer has to wait on producer, and sometimes the producer has to wait on the consumer; kind of like real life; we want the supply chain to be balanced so there is no surplus, and so there is no shortage; both of which produce waiting.
 - Notice “empty” and “full” variables are interdependent; Producer has to wait when empty = 0 (no available slots to input data into), and consumer has to wait when full = n (no data in any of the slots for consumption) (n refers to number of empty slots)

available in a buffer n-sized buffer). For the most part, the Producer() and Consumer() threads can work concurrently and without any waiting; it's only when either one of them reaches their bound, do they have to wait. Notice too how only one at a time will reach a bound – either producer which reaches a bound when there are no slots available to input data and the consumer has a surplus of data to consume, or the consumer will reach a bound when it wants to consume data but there is no data in the buffers to consume.

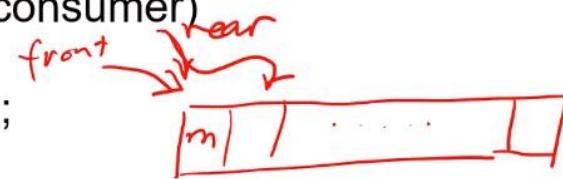
- Notice we can use mod operator so that there is wrap-around, since the front and rear pointers can work independently (i.e., producer can write data to a buffer at the next empty location, while consumer can acquire data from the next written buffer location, and they can just continue to work down the buffer list and wrap back around to the front, working at whichever spot has data in it and moving on as it is appropriate/sigaled to read or write data from next location in the buffer)

Bounded Buffer

(1 producer, 1 consumer)

char buf[n], int front = 0, rear = 0;
semaphore empty = n, full = 0;

Producer()
while (1) {
 produce message m;
wait(empty);
 buf[rear] = m;
 rear = rear + "1"; mod n
signal(full);
}



Consumer ()
while (1) {
wait(full);
 m = buf[front];
 front = front + 1;
signal(empty);
 consume m;
}

- Bounded Buffer (multiple producers and consumers)
 - Need mutually exclusive access to the buffers because we can have two producers running concurrently and overwrite each other's buffer data, and we could have a race condition where two concurrently running consumers both try to consume data from the same location: what if one consumer takes data from a buffer slot, and then a context switch happens right after and another thread tries to take data from that same location? Answer: race condition; CS problem. Solution: more semaphores.

Bounded Buffer

(multiple producers and consumers)

```
char buf[n], int front = 0, rear = 0;
semaphore empty = n, full = 0, mutex = 1;
```

```
Producer()
while (1) {
    produce message m;
    wait(empty);
    wait(mutex);
    buf[rear] = m; CS
    rear = rear "+" 1; CS
    signal(mutex);
    signal(full)
}
```

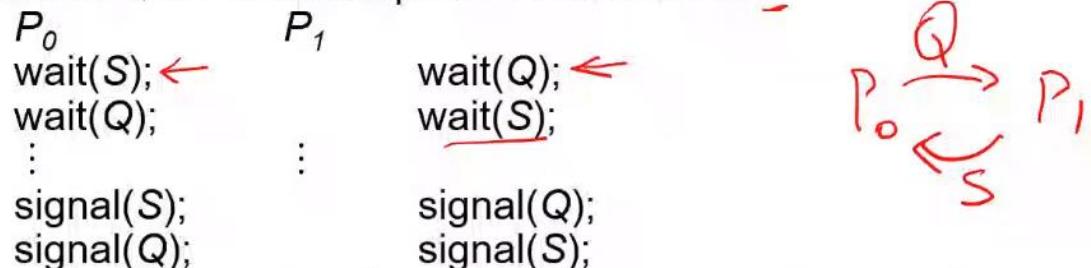
```
Consumer ()
while (1) {
    wait(full);
    wait(mutex);
    m = buf[front];
    front = front "+" 1; CS
    signal(mutex);
    signal(empty);
    consume m;
}
```

- ○ What if we switched wait(empty or full) and wait(mutex)?
 - You could end up deadlocked because consumer could get the mutex lock first when there is nothing in the buffer and thus wait, then producer will try to get a mutex lock but it won't be able to because it is waiting on the consumer to release the lock. So then consumer will be waiting on the producer, and the producer will be waiting on consumer = deadlock. Deadlocks can easily happen when using semaphores if not used properly.
- Deadlock and Starvation
 - Priority inversion: circular waiting here too because of the issue discussed earlier – low priority thread holding a lock but is waiting on a higher priority thread, which is waiting on the lock to be released.
 - Solve by priority-inheritance protocol; when priority inversion occurs, we can elevate the lower-priority thread to the same level as the higher priority thread.

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

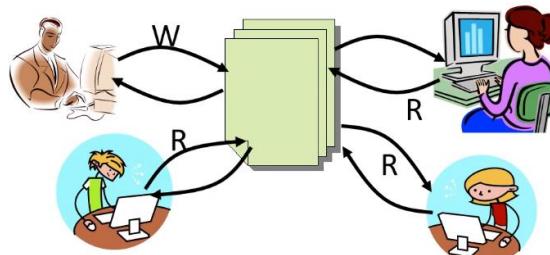
Let S and Q be two semaphores initialized to 1



- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
 - Solved via **priority-inheritance protocol**

- Readers-Writers Problem
 - Can only allow one writer at a time; mutual exclusion necessary because of race condition of having multiple writers updating the database at the same time.

Readers-Writers Problem



- Given a database
 - Can have multiple “readers” at a time
read only, *don’t ever modify database*
 - Only one “writer”
will read and modify database
- Readers-Writers Problem Semaphore Solution
 - Will only allow one writer in at a time

Readers-Writers Problem

Semaphore Solution

- Shared data

```
semaphore mutex = 1, wrt = 1;
int readcount = 0;
```

- Writer Process

```
→ writeEnter () {
    wait(wrt);
}
```

```
writeExit () {
    signal(wrt);
}
```

- • Reader Process

```
readEnter () {
    wait(mutex);
    readcount++;
    if (readcount == 1) first reader
    wait(wrt);
    signal(mutex);
}
```

first reader

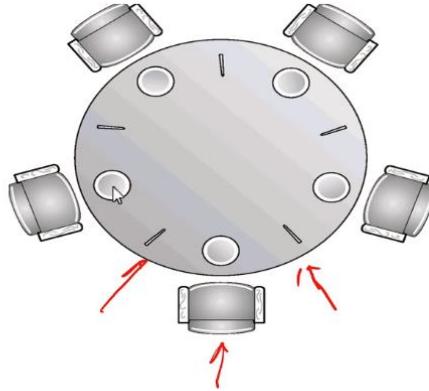
writer starvation

```
readExit() {
    wait(mutex);
    readcount--;
    if (readcount == 0) last reader
    signal(wrt);
    signal(mutex);
}
```

- Once a reader gets in, all following readers do not need to acquire a mutex lock, since multiple readers allowed at one time. Only when the first reader ($\text{readcount} == 1$) or any

- writer tries to enter the database do they need a writing lock. When the last reader leaves the database ($readcount==0$), they must be the one to release the writing lock: `signal(wrt)`. Remember only 1 writer is allowed at a time; and only readers and writers cannot be in the data base together; so a writer and readers are mutually exclusive.
- Notice this solution gives priority to readers, because readers can keep coming in and jumping ahead of writers; so this solution can cause writer starvation, which can be a serious problem.
 - We normally give priority to writers, not readers, since it is easier for readers to get updated information, such as scores to a football game; we want whoever is updating the score to get priority to write, and then allow readers to get back in and check score. Usually less problems with this implementation.
 - Dining-Philosophers Problem

Dining-Philosophers Problem



- Shared data

semaphore chopstick[5];

Initially all values are 1

- Remember, to eat with chopsticks, you need two; thus at this table if a person grabs a chopstick from there left and from there write to eat, that will leave there two neighbors with only up to one possible chopstick to eat with (because their other neighbor may have grabbed the other chopstick too!)
- The most that there can be is 2/5 people eating, since there are only 5 chopsticks; 1 chopstick will be left among the other 3 people.
- If one person takes two sticks, the next person that can get two must be at least one person away from the other person who took two.
- Resources = 5 = chopsticks; and we need to use an array of size 5 in order to mark the availability of a given chopstick position, with all positions initially set to 1 = true/available
- Each philosopher = a thread/process

- When a thread is done “eating” it can release its chopsticks (resources) to be used by another thread.
- Note: each thread only has access to its adjacent chopsticks, so a thread must check the chopstick array corresponding to the thread’s position.

Dining-Philosophers Problem

- Philosopher i :

```

do {
    wait(chopstick[i])           ↙
    wait(chopstick[(i+1) % 5])   ↙
    ...
    eat                         ↓
    ...
    signal(chopstick[i]);       ↙
    signal(chopstick[(i+1) % 5]); ↙
    ...
    think
    ...
} while (1);

```

- Above, eat == have acquired a lock on the two necessary and corresponding chopsticks (resources) (to the left == i , and to the right == $i+1$), think == release resources (chopsticks) for use by another thread and continue to execute other code where the resources (mutex lock) is not needed.
- The above solution works, but can get into a deadlock situation: what if each thread acquires a lock on 1 resource, or two or more adjacent threads take a resource that the other needs? Then some or all of threads will be in a circular waiting cycle for a chopstick = a deadlock.
- Dining-Philosophers Problem (Deadlock Free)
 - Solution: most philosophers will select in a rightward fashion: i and $i+1$ chopsticks, in that order (clockwise); but we can make it so that philosopher 0 (the else statement) will select chopsticks in a counterclockwise fashion (select 1, then 0, in that order = leftward). This will eliminate circular waiting scenario.

Dining-Philosophers Problem (Deadlock Free)

- Philosopher i :

```

do {
    if (i != 0) {
        wait(chopstick[i]); wait(chopstick[i+1]);
        ...           eat           ...
        signal(chopstick[i]); signal(chopstick[i+1]);
    } else {
        wait(chopstick[1]); wait(chopstick[0]);
        ...           eat           ...
        signal(chopstick[1]); signal(chopstick[0]);
    }
    ...
    think
    ...
}
```
- Problems with Semaphores
 - Can be complex; so it is best that we separate mutual exclusion and condition synchronization.
 - Solution: Monitors; supposed to be easier to use than semaphores.

Problems with Semaphores

- Used for 2 independent purposes
 - Mutual exclusion
 - Condition Synchronization
- Hard to get right
 - Small mistake easily leads to deadlock

May want to separate mutual exclusion,
condition synchronization

- Semaphores does both mutual exclusion and condition synchronization, but it often requires multiple semaphores working together.

Lecture Video 05-1 (week4 – 1/31/22): Monitors

- Another tool for thread synchronization
- Monitors (invented by Hoare)
 - Since only one thread at a time in a monitor, don't need to worry about race conditions inside the monitor.
 - Every condition variable has its own queue; wait = add to the queue, signal=remove one from the queue, broadcast = remove everyone from the queue.

Monitors (Hoare)

- Abstract Data Type
 - Consists of vars and procedures, like C++ class.
 - 3 key differences from a regular class:
 - Only one thread in monitor at a time (mutual exclusion is automatic);
 - Special type of variable allowed, called “condition variable”
 - 3 special ops allowed only on condition variables: wait, signal, broadcast
 - No public data allowed (must call methods to effect any change)



3

- Monitors Pseudo code
 - Remember, you don't have to worry about race conditions of a shared variable within the monitor.

Monitors

```
monitor monitor-name
{
    → shared variable declarations
    procedure body P1 (...) {
        ...
    }
    procedure body P2 (...) {
        ...
    }
    procedure body Pn (...) {
        ...
    }
    {
        initialization code
    }
}
```

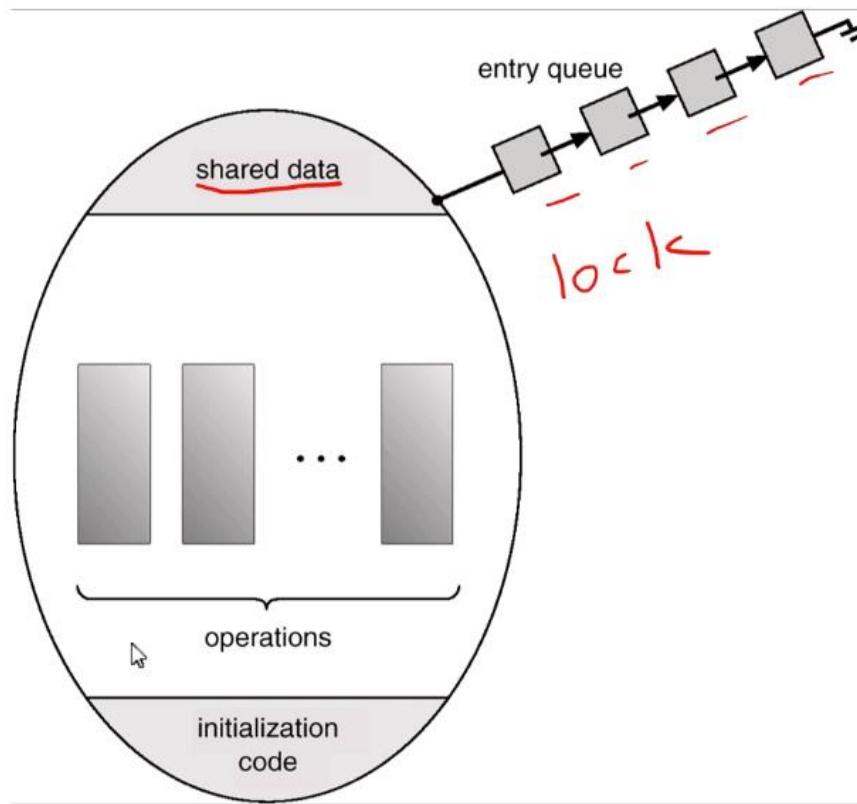
- Monitors (continued)

- Remember, each condition variable has its own separate queue; and you can create as many condition variables as you like (as shown in the slide example).
- Notice unlike semaphores, the condition variables in monitors do not have a state (so for example, cond.signal() will not do anything (such as change state to free, since there is no variable to tell the state of a monitor or condition variable) if there is no thread in the queue waiting to be woken up.

Monitors

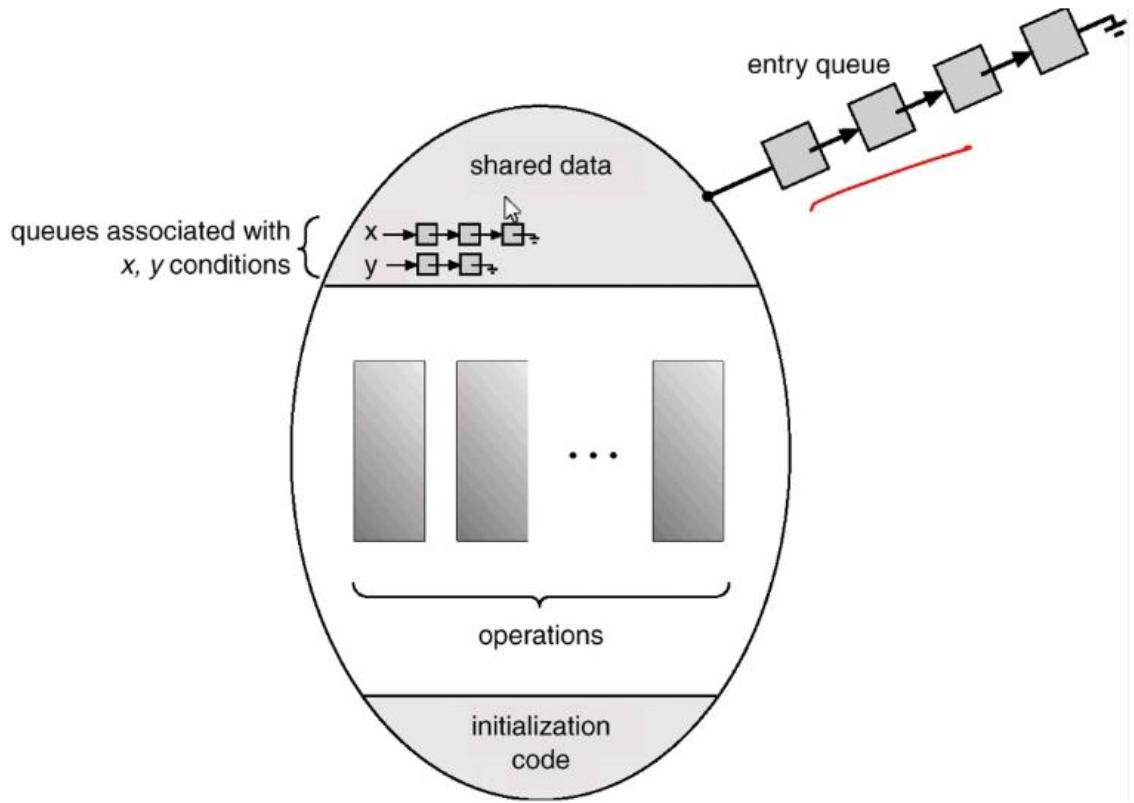
- To allow a process to wait within the monitor, a **condition** variable must be declared, as
condition x, y, cond;
- Given a condition variable “cond”
 - **cond.wait():**
 - thread is put on queue for “cond”, goes to sleep
 - **cond.signal():**
 - if queue for “cond” not empty, wakeup one thread
 - **cond.broadcast()**
 - wakeup all threads waiting on queue for “cond”
- Schematic View of a Monitor
 - Only one active thread can be inside the monitor at any time (mutual exclusion built in, as shown by the entry queue)

Schematic View of a Monitor



- - Monitor With Condition Variables
 - With condition variables, we have a queue for each condition variable as shown, in addition to the entry queue. The condition variable queue provides synchronization capabilities, and the entry queue provides mutual exclusion.

Monitor With Condition Variables



-
- Note, you can use a monitor without condition variables if no synchronization is needed. Similarly, you can use a monitor with no entry queue, solely for the purpose of synchronization.
- Semantics of Signal

Semantics of Signal

- Signal and Wait (Hoare)
 - signaler immediately gives up control
 - thread that was waiting executes
- **Signal and Continue (Java, Pthread, Mesa)**
 - signaler continues executing
 - thread that was waiting put on ready queue
 - when thread actually gets to run
 - State may have changed! **Use “while”, not “if”**

○

- Signal and Wait monitor is complicated and generally not used/supported in Operating System code and programming languages.
- Typically, Signal and Continue is used.
- Remember, there are three states for a thread or process: waiting (queue), ready (queue), and running(on a CPU).
 - Need to while loop to check state, because during context switches the signaled thread (thread told to wake up and put in read state) may have changed states by then, so we use a while loop to check the state of the thread every time the thread gets to run (it's a double check system to ensure a thread can go ahead and execute whenever it is placed on the CPU).
- Monitor Solution to Critical Section

Monitor Solution to Critical Section

- Just make the critical section a monitor routine!
-
- Readers/Writers Solution Using Monitors
 - Mutex = mutual exclusion; semaphores use mutex locks
 - Only use the monitor for giving permission to the reader or writer; cannot put the code for reading or for writing into the monitor.
 - So we only put the code for entry and exit into the monitor; the actual code for writing and reading is outside the monitor.

Readers/Writers Solution Using Monitors

- Similar idea to semaphore solution
 - Simpler, because don't worry about mutex
- When can't get into database, wait on appropriate condition variable
- When done with database, signal others

Note: can't just put code for "**reading database**" and code for "**writing database**" in the monitor (couldn't have >1 reader)

- Pseudo code of the Readers/Writers solution using a monitor (reader functions):
 -

Readers/Writers Monitor Solution

Monitor ReadersWriters {

```

int nr=0; nw=0;
cond *okToRead, *okToWrite;

→ readEnter() {
    while (nw > 0)                                //writer is in
        okToRead->Wait();                         //block
    nr++;                                         //mark that reader is in
}

→ readExit() {
    nr--;                                         //one reader is out
    if (nr == 0)
        okToWrite->Signal();           //hint to writers that maybe one can get in
}

```

11

- nr = number of readers, nw = number of writers in the database (or waiting to use the database)
- nw can be at most 1, nr= 1 or more (many readers)
- two condition variables: okToRead and okToWrite
- need the while loop (not an if statement which would only check once) in the entry function because it is possible that a thread will be woken up and need to go back to sleep immediately, as discussed earlier, because condition could change when the thread is finally scheduled to run; so you need to check nw before a thread goes to wait, and also after the thread wakes up.
- Remember reader only has to worry about if a writer is in the database, if there is a reader(s) in the database, another reader can enter.
- Pseudo code of the Readers/Writers solution using a monitor (writer functions):

Readers/Writers Monitor Solution

```

writeEnter() {
    ↗ while (nw > 0 || nr > 0)           //someone is in
        okToWrite->Wait();                //block
        nw++;                                //mark that writer is in
    }

writeExit() {
    nw--;                                //one writer is out

    //hint to writers that maybe one can get in
    ↗ okToWrite->Signal();               //hint to readers that maybe they can get in
    ↗ okToRead->Broadcast();             }
}

} // end monitor

```

- Remember, only one writer at a time; so writeEnter function must check to see if anyone is in the database (readers or writers).
- Notice all entry and exit functions of both the write and read functions share the condition variables (okToRead and okToWrite), and they also share the state variables nw and nr that tracks the number of readers or writers (stored the condition variables queues) in or trying to get into the database.
 - Have two condition variables because we need to separate the queue of the read and write conditions, since they have different conditions for entry.
- Notice priority: signal any potential writers first, then signal any potential readers.
 - Also we signal both writers and readers because we do not keep track of readers and writers in this particular implementation (nr and nw are not checked at this point when we signal); so we just signal both, and we make sure to send a broadcast signal to all readers since if there are no waiting writers, then all readers can enter.
 - And remember, all threads that are signaled=placed into ready queue will double check the nw or nr conditions when they are scheduled to run because the while() statement will be checked again to make sure conditions have not changed by the time it is taken off the ready queue and scheduled to run (state=running → on the CPU).

- Signal is used to make sure somebody wakes up sleeping threads, and it is okay if they wake up and shouldn't because they will check the condition every time they wake up and go back to sleep if necessary.
- Note, this implementation also favors the reader just as the semaphore solution; **priority goes to readers** because they only need to make sure there are no writers; however, for writers, every time they wake up and then get scheduled to run they must make sure there are no readers or writers – so as soon as any reader (or writer) enters the database or is waiting to enter, the writer then loses its spot until all readers have finished (and of course it must be the next writer in line to get into the database if there are writers in the queue).
 - Not ideal usually; we want to give priority to writers so that readers don't continue to read old data.
- Another Readers/Writers Solution Using Monitors (giving priority to writers for this solution)

Another Readers/Writers Solution Using Monitors

- Correctness Constraints:
 - Readers can access database when no writers
 - Writers can access database when no readers or writers
 - Only one thread manipulates state variables at a time
- State variables :
 - int nr: Number of active readers; initially = 0
 - int wr: Number of waiting readers; initially = 0
 - int nw: Number of active writers; initially = 0
 - int ww: Number of waiting writers; initially = 0
 - Condition okToRead = NIL
 - Condition okToWrite = NIL

○

- Notice, two new state variables added: ww (waiting writers) and wr (waiting readers). These will be used so that readers cannot skip ahead of waiting writers.

Readers/Writers Solution: Give Priority to Writers

```

Monitor ReadersWriters {
    int nr=0; nw=0; wr=0; ww=0;
    cond *okToRead, *okToWrite;

    readEnter() {
        while ((nw + ww) > 0) {           //Is it safe to read?
            wr++;                         //No, writers exist
            okToRead->Wait();             //sleep on cond var
            wr--;
        }
        nr++;                            //mark that reader is in
    }

    readExit() {
        nr--;                           //one reader is out
        if ((nr == 0) && (ww >0))      //No other active readers
            okToWrite->Signal();       //Wake up one writer
    }
}

```

- Notice if(nr==0) because we do not want to signal a writer to wake up if there are still active readers in the database; we cannot kick them off the database.

Readers/Writers Solution: Give Priority to Writers

```

writeEnter() {
    while (nw > 0 || nr > 0) {           //Is it safe to write?
        ww++;                            //No, active users exist
        okToWrite->Wait();              //block
        ww--;                            //No longer waiting
    }
    nw++;                             //mark that writer is in
}
writeExit() {
    nw--;                            //No longer active
    if (ww > 0) {                   //Give priority to writers
        okToWrite->Signal();        //Wake up one writer
    } else if (wr > 0) {            //Otherwise, wake reader
        okToRead->Broadcast();     //Wake all readers
    }
}
} // end monitor

```

- Notice: ww++ before writer goes to sleep; that way when any new (waiting) readers will not be allowed to skip ahead of waiting writers.
- Using semaphores, this solution is even more complex/almost impossible. Using monitors is much easier because the ability to create queues through conditions and the built in mutex.
- Semaphores and Monitors (and the likes) are the most difficult concepts/code to write in the OS.
- Monitor Solution to Dining Philosophers

Monitor Solution to Dining Philosophers

```

monitor DiningPhilosophers
{
    enum { THINKING, HUNGRY, EATING} state [5] ;
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
}

```

-
- Remember condition is that both left and right neighbor must not be using the chopstick for a given philosopher to use both chopsticks to get in and execute the associated code.
- We maintain a state variable for each philosopher so we can check chopstick availability
→ state[5]
 - Each philosopher can be in one of three different states: Thinking = don't need chopsticks, Hungry= waiting for chopstick(s), Eating = using chopsticks.
 - Each position in the state array is storing an enum type; so state[0] is an enum of the type declared, that can store an integer constant, and can be associated with the constant names provided in the enum declaration for integers 0, 1, and 2.
 - NOTE: state[5] is an array of size 5, of an enum (enumerator) type; remember enum types enumerate a set of constant integers associated to a name so that it is easier to read/write a program. So, THINKING = 0, HUNGRY =1, EATING=2, and they are all constants of type enum. Usually enums are given a name of its specific enumeration type declared, such as: enum PhilosopherStateEnum {THINKING, HUNGRY, EATING}, followed by a variable now or later of that type

- of enum; in this case, the variable state[5] is declared and of course it is an array of size 5.
- Note: the condition self[5] variable is the condition variable; it is an array because in this implementation each philosopher will technically have their own condition variable **solely for the purpose** of sending themselves to sleep (self[i].wait) or signaling themselves to wake up (self[i].signal).
 - But note: although each philosopher has their own condition variable and thus their own condition waiting queue (and thus only one thread – the philosopher themselves, can be in the waiting queue at max), whenever any philosopher puts themselves on the waiting queue of the condition queue and to sleep(place in waiting state queue), or wakes up, all philosophers will go to the waiting state (queue) or ready state (queue) respectively; So yes each thread in this case will go to its own separate condition variable waiting queue, but upon doing that they will also go to the wait state queue.
 - In the pickup function, you have to first declare you are hungry (state[i] = HUNGRY), then you have to test to see if both neighboring chopsticks are available → test(i)

Solution to Dining Philosophers (Cont.)

```

void test (int i) { (i-1)%5
    if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
        state[i] = EATING ;
        self[i].signal () ;
    }
}

initialization_code() {
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}

```

- - Above, we do $(i+4)\%5$ to guarantee you get a positive number, but conceptually we are really doing $(i-1)\%5$, in order to check left neighbor.
 - Putdown = done eating; then you check left and right neighbors; if they are hungry (waiting) then you can wake them up.
 - Pickup and Putdown are the two public procedures of the monitor

Solution to Dining Philosophers (Cont.)

- Each philosopher i invokes the operations `pickup()` and `putdown()` in the following sequence:

`DiningPhilosophers.pickup(i) ;`

EAT

`DiningPhilosophers.putdown(i) ;`

- No deadlock, but starvation is possible
 - ▪ Solution is not a first come first serve solution; it just depends on whenever both chopsticks are available for a given thread, that's why there can be starvation under the right circumstances of context switches and other eating philosophers. This is because each philosopher's condition variable waiting queue is separate, as each has its own condition variable.
 - Solution: need to some kind of implementation to place all philosophers in the same waiting queue inside the monitor so they can take turns to eat
- Implementing Monitors using Semaphores
 - We can use semaphores to implement the condition variables of monitors
 - Semaphore mutex = 1 is how we make sure monitor is mutually exclusive
 - Semaphore c = 0 → variable for each condition variable, c can be any positive integer; this is useful for thread synchronization.
 - nc = 0 (counter variable) → used to check number of waiting threads for each condition variable.

Implementing Monitors using Semaphores

- Shared vars:
 - semaphore mutex = 1(one per monitor)
 - semaphore c = 0; (one per condition var)
 - int nc = 0; (one per condition var)
- Monitor entry: wait(mutex);
- Monitor exit: signal(mutex);
- - Implementing Monitors using Semaphores
 - We pass mutex inside of cond.wait() call because we have to release the mutex lock for the entire monitor via signal(mutex) [semaphore signal call] before a thread goes to wait queue so we can allow other threads to get in, otherwise you get into a deadlock → a thread could go to waiting queue but also still hold the lock for the entire monitor.
When thread wakes up, then it acquires the lock again → wait(mutex) [semaphore wait call], meaning it is the only thread currently operational in the monitor (mutual exclusion).
 - If nc (number of waiting conditions on a given condition variable) > 0; decrement by 1, and signal [semaphore signal call] the next thread to acquire a lock and be operational in the monitor.

Implementing Monitors using Semaphores

- cond->Wait(mutex):
nc++;
signal(mutex);
wait(c);
wait(mutex);
- cond->Signal():
`if (nc > 0) {
 nc--;
 signal(c);
}`
- Difference between Monitors and Semaphores

Difference between Monitors and Semaphores

- Monitors enforce mutual exclusion
- Semaphore wait vs condition variable wait
 - semaphore wait blocks if value is 0,
 - condition variable wait always blocks
- Semaphore signal vs condition variable signal
 - semaphore signal either wakes up a thread or increments value
 - condition variable signal only has effect if a thread waiting
- Semaphores have “memory”

○

Lecture Video 06-1 (week5 – 2/7/22): JavaThread

- Java Threads

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
 - Extending Thread class
 - Implementing the Runnable interface
- Extend Example
 -

Extend Example

```
public class Main extends Thread {  
    public static void main(String[] args) {  
        Main thread = new Main();  
        thread.start();  
        System.out.println("This code is outside of the thread");  
    }  
    public void run() {  
        System.out.println("This code is running in a thread");  
    }  
}
```

- Implement Example
 -

Implement Example

```

interface
public class Main implements Runnable {
    public static void main(String[] args) {
        Main obj = new Main();
        Thread thread = new Thread(obj);
        thread.start();
        System.out.println("This code is outside of the thread");
    }
    public void run() {
        System.out.println("This code is running in a thread");
    }
}

```

- More popular approach because the extend method has limitations: if a class is created that extends another class already, that class cannot be used as extended class; it must be implemented via implement (Runnable).
- Java-style monitors

Java-style monitors

- Integrated into the class mechanism
 - Annotation “synchronized” can be applied to a member function
 - This function executes with implicit mutual exclusion
 - Wait, Signal, and Broadcast are called wait, notify, and notifyAll, respectively

- There are no semaphores in Java, but there are monitors.

- Java Support for Synchronization (synchronized methods)
 - All synchronized sections together form a critical section (race conditions), which is why the keyword “synchronize” automatically treats all code with that label inside of a class as critical sections that need to be ran mutually exclusive.
 - So synchronized keyword acts as a lock

Java Support for Synchronization

- Bank Account example:

```

class Account {
    private int balance;
    // object constructor
    public Account (int initialBalance) {
        balance = initialBalance;
    }
    → public synchronized int getBalance() {
        return balance;
    }
    → public synchronized void deposit(int amount) {
        balance += amount;
    }
}

```

Critical section

- Every object has an associated lock which gets automatically acquired and released on entry and exit from a synchronized method.
- - Java Support for Synchronization cont'd (synchronized statements)
 - Usually locks “this” object (this-> being the pointer to the current object), although you can pass in a reference to another object.
 - Synchronized(object) uses an object's lock to lock the associated statements {}
 - Any synchronized statements associated with the same object together form a critical section

Java Support for Synchronization (con't)

- Java also has *synchronized* statements:

```
this  
synchronized (object) {  
    ...  
}
```

- Since every Java object has an associated lock, this type of statement acquires and releases the object's lock on entry and exit of the body
- Works properly even with exceptions:

```
synchronized (object) {  
    ...  
    DoFoo();  
    ...  
}  
void DoFoo() {  
    throw errException;  
}
```

- Java Support for Synchronization cont'd

Java Support for Synchronization (con't 2)

- In addition to a lock, every object has a single condition variable associated with it
 - How to wait inside a synchronization method or block:

- void wait(long timeout); // Wait for timeout
- void wait(long timeout, int nanoseconds); //variant
- void wait();

- How to signal in a synchronized method or block:

- void notify(); // wakes up oldest waiter
- void notifyAll(); // like broadcast, wakes everyone

- Example:

```
// Java program to implement solution of producer  
// consumer problem.  
  
import java.util.LinkedList;  
  
public class Threadexample {  
    public static void main(String[] args)  
        throws InterruptedException  
    {  
        // Object of a class that has both produce()  
        // and consume() methods  
        final PC pc = new PC();  
  
        // Create producer thread  
        Thread t1 = new Thread(new Runnable() {  
            @Override  
            public void run()  
            {  
                try {  
                    pc.produce();  
                }  
                catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        });  
    }  
}
```

- o
- o Above, create a new thread called pc.

```
    public void run()
    {
        try {
            pc.produce();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

// Create consumer thread
Thread t2 = new Thread(new Runnable() {
    @Override
    public void run()
    {
        try {
            pc.consume();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

```

o // Start both threads

o Now, above (using both screenshots), create two **child** threads of the pc() thread: t1 and t2, pc.produce() and pc.consume(), respectively.

```
// Create consumer thread
Thread t2 = new Thread(new Runnable() {
    @Override
    public void run()
    {
        try {
            pc.consume();
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

// Start both threads
t1.start();
t2.start(); // Clicked here

// t1 finishes before t2
t1.join();
t2.join();

}

// This class has a list, producer (adds items to list
// and consumer (removes items).
public static class PC {
```

```

// Create a list shared by producer and consumer
// Size of list is 2.
LinkedList<Integer> list = new LinkedList<>();
int capacity = 2;

// Function called by producer thread
public void produce() throws InterruptedException
{
    int value = 0;
    while (true) {
        synchronized (this)
        {
            // producer thread waits while list
            // is full
            while (list.size() == capacity)
                wait();

            System.out.println("Producer produced-"
                               + value);

            // to insert the jobs in the list
            list.add(value++);

            // notifies the consumer thread that
            // now it can start consuming
            notify();
        }
    }
}

```

- The `wait()` call is the associated condition variable for the current particular object.
- Also, capacity is set to 2; meaning only a max of two items can be consumed or produced at one time (concurrently/interleaved); otherwise a consumer or producer must `wait()` (go to sleep) to consume or produce another item until `list.size() != capacity` (less than 2; elements 0 and 1 = list size of 2).
-

```
// producer thread waits while list
// is full
while (list.size() == capacity)
    wait();

System.out.println("Producer produced-"
    + value);

// to insert the jobs in the list
list.add(value++);

// notifies the consumer thread that
// now it can start consuming
notify();

// makes the working of program easier
// to understand
Thread.sleep(1000);
}

}

// Function called by consumer thread
public void consume() throws InterruptedException
{
    while (true) {
        synchronized (this)
```

- o To increase interleaving between the two threads, we allow the thread to sleep for 1000ms (1s) before it continues (since we have an infinite while loop for the produce() and consume() methods that make up the t1 and t2 threads running for this program where producers and consumers can indefinitely add and take away from a cache of goods as necessary).

```
// Function called by consumer thread
public void consume() throws InterruptedException
{
    while (true) {
        synchronized (this)
        {
            // consumer thread waits while list
            // is empty
            while (list.size() == 0)
                wait();

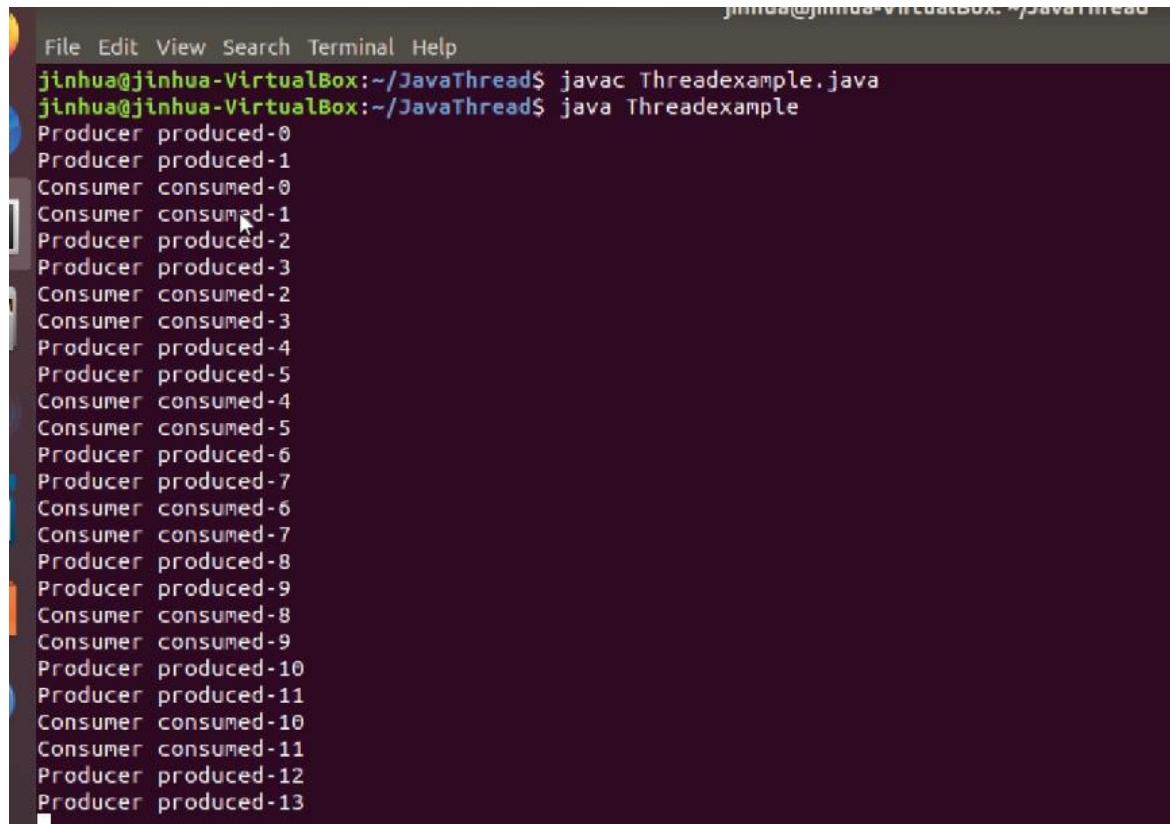
            // to retrieve the first job in the list
            int val = list.removeFirst();

            System.out.println("Consumer consumed-"
                               + val);

            // Wake up producer thread
            notify();

            // and sleep
            Thread.sleep(1000);
        }
    }
}
```

- o
- o Above, similar idea for the consume() function/method.
- o Running the program in java virtual machine:



A screenshot of a terminal window titled "jinhua@jinhua-VirtualBox:~/JavaThread\$". The window shows the execution of a Java program named "Threadexample.java". The output consists of alternating "Producer produced" and "Consumer consumed" messages, each followed by a number from 0 to 13. The producer starts at 0 and increments by 1, while the consumer starts at 0 and also increments by 1. The consumer message "Consumer consumed-1" is highlighted with a cursor.

```
jinhua@jinhua-VirtualBox:~/JavaThread$ javac Threadexample.java
jinhua@jinhua-VirtualBox:~/JavaThread$ java Threadexample
Producer produced-0
Producer produced-1
Consumer consumed-0
Consumer consumed-1
Producer produced-2
Producer produced-3
Consumer consumed-2
Consumer consumed-3
Producer produced-4
Producer produced-5
Consumer consumed-4
Consumer consumed-5
Producer produced-6
Producer produced-7
Consumer consumed-6
Consumer consumed-7
Producer produced-8
Producer produced-9
Consumer consumed-8
Consumer consumed-9
Producer produced-10
Producer produced-11
Consumer consumed-10
Consumer consumed-11
Producer produced-12
Producer produced-13
```

- Now for another example if we change capacity to 10:

```
jinhua@jinhua-VirtualBox:~/JavaThreads$ java Consumer  
jinhua@jinhua-VirtualBox:~/JavaThreads$ java Producer  
Producer produced-0  
Producer produced-1  
Producer produced-2  
Producer produced-3  
Producer produced-4  
Producer produced-5  
Producer produced-6  
Producer produced-7  
Producer produced-8  
Producer produced-9  
Consumer consumed-0  
Consumer consumed-1  
Consumer consumed-2  
Consumer consumed-3  
Consumer consumed-4  
Consumer consumed-5  
Consumer consumed-6  
Consumer consumed-7  
Consumer consumed-8  
Consumer consumed-9  
Producer produced-10  
Producer produced-11  
Producer produced-12  
Producer produced-13  
Producer produced-14
```

-
- Now for both examples, you could switch between consumers and producers, but the examples don't show that; but as long as the linked list of size capacity does not reach the capacity, then a consumer can consume and a producer can produce without having to go to sleep and letting the other thread run.
- The code will be posted on canvas so we can run it on our own.
-

Lecture Video 06-2 (week5 – 2/7/22): Pthread

- Review: Definition of Monitor
 - Use of a monitor is a programming paradigm (pattern) because: if you use a lock and a condition variable, you essentially are using a monitor, which is formed by implementing both of those tools into one cohesive function/data structure. Javathread does contain monitors, but Pthread does not; however it combines a lock and a condition variable, so

essentially it does incorporate a monitor because it incorporates the equivalent of a monitor.

Review: Definition of Monitor

- Semaphores are confusing because dual purpose:
 - Both mutual exclusion and scheduling constraints
 - Cleaner idea: Use **locks** for mutual exclusion and **condition variables** for scheduling constraints
- **Monitor**: a lock and zero or more condition variables for managing concurrent access to shared data
 - Use of Monitors is a programming paradigm pattern
- ✓ • **Lock**: provides mutual exclusion to shared data:
 - Always acquire before accessing shared data structure
 - Always release after finishing with shared data
- ✓ • **Condition Variable**: a queue of threads waiting for something inside a critical section
 - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
 - Contrast to semaphores: Can't wait inside critical section
 - Programming with Monitors
 - We use a reference (&lock) because when a thread waits, it releases the lock, but when it wakes up again, it must acquire that same lock.

Programming with Monitors

- Monitors represent the logic of the program
 - Wait if necessary
 - Signal when change something so any waiting threads can proceed
- Basic structure of monitor-based program:

```

lock
while (need to wait) {
    condvar.wait(&lock);
}
unlock
do something so no need to wait

lock
condvar.signal();
unlock
    
```

- What are Pthreads?

What are Pthreads?

- Pthread is a standardized thread programming interface specified by the IEEE POSIX (portable operating systems interface) in 1995.
- Pthreads are defined as a set of C language programming types and procedure calls, implemented with a **pthread.h** header/include file and a **thread library**.
- Why Pthread?

- So as we have done before, we create a new thread by duplicating the entire address space of the parent (into a separate address space; via fork()); but with Pthread, you can create new threads by basically just creating a stack inside the same address space for each thread.

Why Pthread ?

- To realize potential program performance gains.
- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
- All threads within a process share the same address space. Inter-thread communication is more efficient and in many cases, easier to use than inter-process communication.

○



- Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:
 - Overlapping CPU work with I/O
 - Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.
 - Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.

- - Above, the drawing shows how overlapping CPU work can occur with I/O
- Include files and libraries

Include files and libraries

- .h file

```
#include <pthread.h>
#include <semaphore.h> //for semaphore only
```

- Compile and Linking

```
$gcc foo.c -o foo -lpthread -lrt  
g++ (for semaphore)
```

- - Above, use gcc for c programming files, and g++ for c++ program files. -l means link, thus -lpthread means link pthread library
- The Pthreads API
 - (API = application programming interface = which is a set of *general*/definitions and protocols for building and integrating a particular application software (aka, a program that is written with a specific usage and processing objective). So for example, I could give the API for making a GPS/map application, which could implement many different kinds of functions from the API of your choice, but the overall goal is to provide navigation, and so no matter how the program is written using the API (which is just another program, that can be used in the implementation of another program), it must include some general/common characteristics in order to meet the particular end objective, which can be achieved by an API.

The Pthreads API

- **Thread management:** The first class of functions work directly on threads - creating, terminating, joining, etc.
- **Semaphores:** provide for create, destroy, wait, and post on semaphores.
lock → *signal*
- **Mutexes:** provide for creating, destroying, locking and unlocking mutexes.
- **Condition variables:** include functions to create, destroy, wait and signal based upon specified variable values.
 -
- Thread Creation
 - Attr = attributes of the thread, such as priority level.
 - Arg must be cast as a (`void*`) pointer in order to pass a parameter to the function to which you decide to point to; it needs to be void pointer since the parameter of the function called could take various types of object types, thus using void, you just have to make sure you are passing in the correct type of value so that the function being pointed to can actually handle what you passed to it when it dereferences the void ptr and treats it as the type that should be passed to its function.

Thread Creation

pthread_create (tid, attr, start_routine, arg)

- It returns the new thread ID via the *tid* argument.
- The *attr* parameter is used to set thread attributes, NULL for the default values.
- The *start_routine* is the C routine that the thread will execute once it is created.
- A single argument may be passed to *start_routine* via *arg*. It must be passed by reference as a pointer cast of type void.
- - Thread Termination and Join
 - Join function normally used by the parent thread to wait for a child to complete

Thread Termination and Join

pthread_exit (*value*) ;

- This Function is used by a thread to terminate. The return value is passed as a pointer.

pthread_join (tid, value_ptr);

Parent *wait for child*

- The pthread_join() subroutine blocks the calling thread until the specified *threadid* thread terminates.
- Return 0 on success, and negative on failure. The returned value is a pointer returned by reference. If you do not care about the return value, you can pass NULL for the second argument.
- - Example of a Pthread Program

- In the example, we have three threads: the main thread, and then two new threads created inside of the main thread; thus main thread is a parent, with two child threads, which gives the program a total of 3 threads.
- Notice each `pthread_create` function call passes in the function `PrintHello` (via the function pointer as a parameter) as the place where the execution of the given created thread should start its execution; essentially, `PrintHello` will serve as the code used for the created thread.
- In the example, no synchronization or controls are used, thus it is up to the CPU scheduler, thus the output could be variable: could be "thread 1 hello world" then "thread 0 hello world" or vice versa.
- We can try running this program to see the possible outputs and prove that it can be variable every time you run it.

Example Code - Pthread Creation and Termination

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

void *PrintHello(void * id)
{
    printf("Thread%d: Hello World!\n", id);
    pthread_exit(NULL);
}

→ int main (int argc, char *argv[])
{
    pthread_t thread0, thread1;
    → pthread_create(&thread0, NULL, PrintHello, (void *) 0);
    → pthread_create(&thread1, NULL, PrintHello, (void *) 1);
    pthread_exit(NULL);
}

• Thread Functions
```

main thread

Thread functions

```
sched_yield();  
pthread_self();  
pthread_equal (tid1,tid2);
```

- ✓ • **sched_yield()** suspends the execution of the current thread. This allows another thread immediate access.
- The **pthread_self()** routine returns the unique, system assigned thread ID of the calling thread.
- The **pthread_equal()** routine compares two thread IDs. If the two IDs are different 0 is returned, otherwise a non-zero value is returned.
 -
 - Mutex Variables
 - (in Pthread, it is the same as Lock)

lock

Mutex Variables

- For thread synchronization and protecting shared data when multiple writes occur.
 - A mutex variable acts like a "lock" protecting access to a shared data resource.
 - **Only one** thread can lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful. No other thread can own that mutex until the owning thread unlocks that mutex. Threads must "take turns" accessing protected data.
- Creating / Destroying Mutexes

Creating / Destroying Mutexes

`pthread_mutex_init (mutex, attr)`

`pthread_mutex_destroy (mutex)`

- Mutex variables must be declared with type `pthread_mutex_t`, and must be initialized before they can be used.
- `attr`, mutex object attributes, specified as `NULL` to accept defaults

- Locking / Unlocking Mutexes
 - The reason for trylock() is so that the calling function/thread doesn't get blocked (as with lock()) if they cannot acquire a lock; that way, if the calling function is able to do something else, it can proceed to stay in the running state. Of course if a function does need to wait for some other thread or for some CS code that must be done before it can do anything else, then it will just use the lock() function so that it can be blocked and go to sleep (wait).

Locking / Unlocking Mutexes

```
pthread_mutex_lock (mutex); ←  
pthread_mutex_trylock (mutex); ↗  
pthread_mutex_unlock (mutex);
```

- The pthread_mutex_lock() routine is used by a thread to acquire a lock on the specified *mutex* variable. The thread **blocks** if the mutex is already locked by another thread.
 - pthread_mutex_trylock() will attempt to lock a mutex. However, if the mutex is already locked, the routine will **return immediately** with a "busy" error code.
 - pthread_mutex_unlock() will **unlock** a mutex if called by the owning thread.
- Semaphores

Semaphores

- Semaphore are not defined in the POSIX.4a (pthread) specifications, but they are included in the POSIX.4 (realtime extension) specifications.

- .h

#include <semaphore.h>

- Semaphore descriptors are declared global

ex: sem_t mutex, full, empty;

- Routines of Semaphores
 - In the sem_init function, sp is passed as reference
 - init_value is an integer that initializes the value of the semaphore; if you want mutual exclusion, set it to 1; if you want condition synchronization it should be 0 depending on your problem (remember, if it is 0, then thread A can run and then do a signal, and only when it does a signal will thread B wake up, as thread B could be put to sleep at a certain point in its code and be forced to wait for a signal from thread A...etc...).
 - For pshared, if you want to do thread synchronization between threads in a single process, the value should be 0; but if you want to do thread synchronization between threads of different processes, the value should be nonzero.

Routines of Semaphore

`sem_t sp;`

- `sem_init(&sp, pshared, init_value)`
 - If `pshared` is nonzero, the semaphore can be shared between processes.
- `sem_destroy(&sp)`
- `sem_wait (&sp); //P operation, wait`
- `sem_trywait(&sp);`
- `sem_post (&sp); // V operation, signal`

- Condition Variables
 -

Condition Variables

- While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
- Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met. This can be very resource consuming since the thread would be continuously busy in this activity. A condition variable is a way to achieve the same goal without polling.
- A condition variable is always used in conjunction with a mutex lock.

○

Conditional Variable Routines

- **`pthread_cond_init (condition, attr)`**
 - **`pthread_cond_destroy (condition)`**
 - **`pthread_cond_wait (condition, mutex)`**
 - **`pthread_cond_signal (condition)`**
 - **`pthread_cond_broadcast (condition)`**
- - A representative sequence for using condition variables for thread synchronization

A representative sequence for using condition variables

Main Thread	
<ul style="list-style-type: none"> • Declare and initialize global data/variables • Declare and initialize a condition variable object • Declare and initialize an associated mutex • Create threads A and B to do work 	
Thread A <ul style="list-style-type: none"> • Do work up to the point where a certain condition must occur (such as "count" must reach a specified value). • Lock associated mutex and check value of a global variable • Call <code>pthread_cond_wait()</code> to perform a blocking wait for signal from Thread-B. It will automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B. • When signalled, wake up. Mutex is automatically and atomically locked. • Explicitly unlock mutex. • Continue 	Thread B <ul style="list-style-type: none"> • Do work • Lock associated mutex • Change the value of the global variable that Thread-A is waiting upon. • Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A. • Unlock mutex. • Continue <p style="text-align: right;">monitor</p>
Main Thread Join / Continue	

○

- If you implement using the above template, you are essentially implementing the monitor data structure/function.
- Example of thread synchronization using Semaphores and Pthreads
 - Same producer and consumer example, as with the Javathread example

```

edBuffer.c  condvar1.c
└── *
  └── * boundedBuffer.c
    *
    * A complete example of simple producer/consumer program. The Producer
    * and Consumer functions are executed as independent threads. They
    * share access to a single buffer, data. The producer deposits a sequence
    * of integers from 1 to numIters into the buffer. The Consumer fetches
    * these values and adds them. Two semaphores,empty and full are used to
    * ensure that the producer and consumer alternate access to the buffer.
    *
    * SOURCE: adapted from example code in "Multithreaded, Parallel, and
    *          Distributed Programming" by Gregory R. Andrews.
    */
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

#define SHARED 1

void *Producer (void *); // the two threads
void *Consumer (void *);

sem_t empty, full;      //global semaphores
int data;                // shared buffer, size = 1
int numIters;

// main() -- read command line and create threads

```

```
bddBuffer.c [ ] | condvar1.c [ ] | 
#include <stdlib.h>

#define SHARED 1

void *Producer (void *); // the two threads
void *Consumer (void *);

sem_t empty, full;          //global semaphores
int data;                   // shared buffer, size = 1
int numIters;

// main() -- read command line and create threads
int main(int argc, char *argv[]) {
    pthread_t pid, cid;

    sem_init(&empty, SHARED, 1); // sem empty = 1
    sem_init(&full, SHARED, 0); //sem full = 0

    if (argc < 2) {
        printf("Usage: boundedBuffer <Number of Iterations>\n");
        exit(0);
    }
    numIters = atoi(argv[1]);

    pthread_create(&pid, NULL, Producer, NULL);
    pthread_create(&cid, NULL, Consumer, NULL);
}
```

```

sem_t empty, full;           //global semaphores
int data;                   // shared buffer, size = 1
int numIters;

// main() -- read command line and create threads
int main(int argc, char *argv[]) {
    pthread_t pid, cid;

    sem_init(&empty, SHARED, 1);      // sem empty = 1
    sem_init(&full, SHARED, 0); //sem full = 0

    if (argc < 2) {
        printf("Usage: boundedBuffer <Number of Iterations>\n");
        exit(0);
    }
    numIters = atoi(argv[1]);

    pthread_create(&pid, NULL, Producer, NULL);
    pthread_create(&cid, NULL, Consumer, NULL);

    pthread_join(pid, NULL);
    pthread_join(cid, NULL);
    pthread_exit(0);
}

// deposit 1, ..., numIters into the data buffer

```

-
- See javathread example, as it is very similar to this.
- In this implementation, like before, we only have a single producer and single consumer
- Shared buffer is only a single cell: SHARED = 1
- Two child threads of the main thread created
- Join called for both child threads, thus the parent (the main thread) will wait for both of its child threads to exit
- We have created 2 semaphores, one initialized to 1, the other to 0.

```

        pthread_join(pid, NULL);
        pthread_join(cid, NULL);
        pthread_exit(0);
    }

    // deposit 1, ..., numIters into the data buffer
void *Producer(void *arg) {
    int produced;

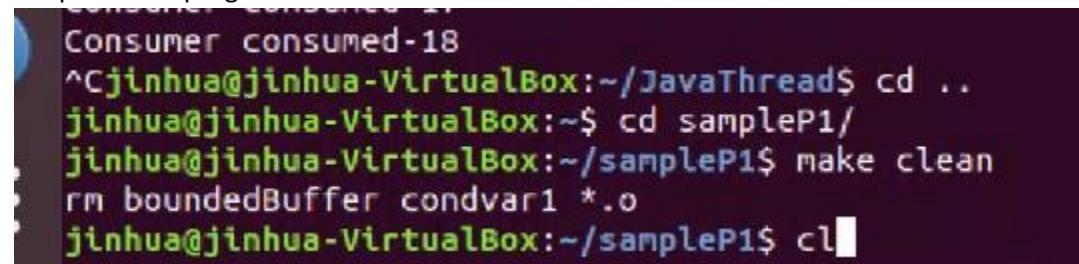
    for (produced = 0; produced < numIters; produced++) {
        sem_wait(&empty);
        data = produced;
        sem_post(&full);
    }
}

//fetch numIters items from the buffer and sum them
void *Consumer(void *arg) {
    int total = 0;
    int consumed;

    for (consumed = 0; consumed < numIters; consumed++) {
        sem_wait(&full);
        total = total + data;
        sem_post(&empty);
    }
}

```

- For producer, we wait for an empty cell before we produce and update the cell (data)
- For consumer, we wait for a full cell before trying to consume.
- Notice when each loop finishes for both threads, they wake each other up, using the corresponding semaphore.
- Output of the program:

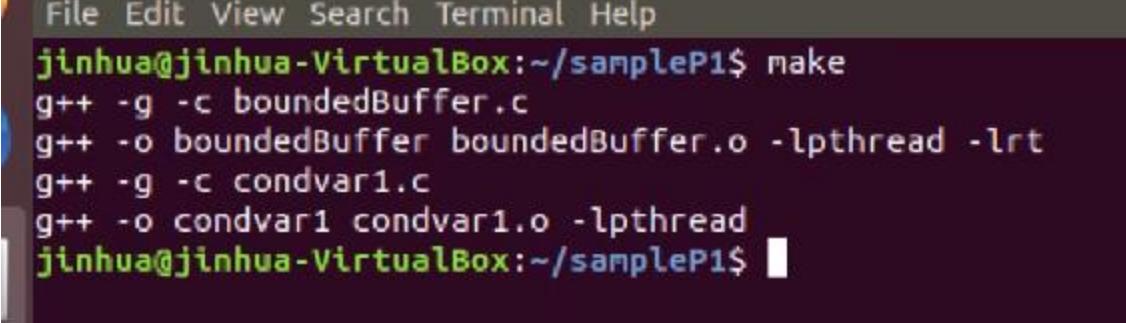


```

Consumer consumed-1
Consumer consumed-18
^Cjinhua@jinhua-VirtualBox:~/JavaThread$ cd ..
jinhua@jinhua-VirtualBox:~$ cd sampleP1/
jinhua@jinhua-VirtualBox:~/sampleP1$ make clean
rm boundedBuffer condvar1 *.o
jinhua@jinhua-VirtualBox:~/sampleP1$ cl

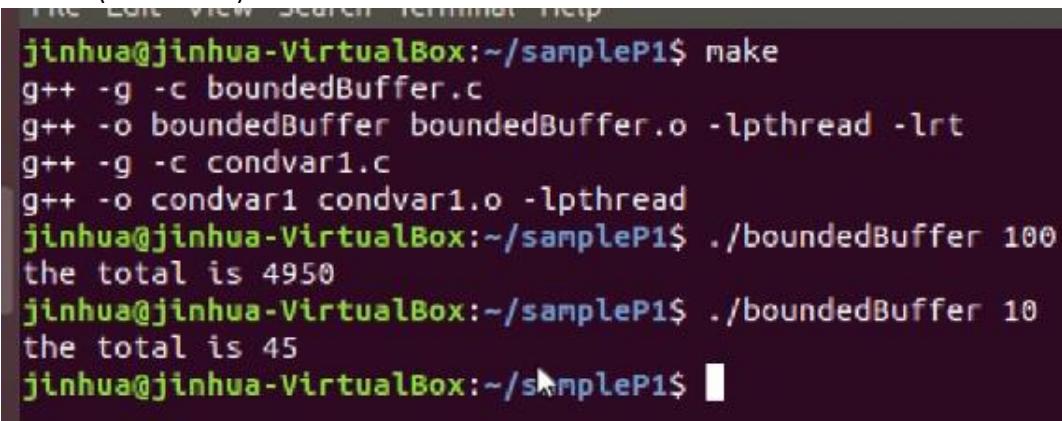
```

- - Side note: notice he ran “make clean” command; this must remove (rm) .o files in a given directory.



```
File Edit View Search Terminal Help
jinhua@jinhua-VirtualBox:~/sampleP1$ make
g++ -g -c boundedBuffer.c
g++ -o boundedBuffer boundedBuffer.o -lpthread -lrt
g++ -g -c condvar1.c
g++ -o condvar1 condvar1.o -lpthread
jinhua@jinhua-VirtualBox:~/sampleP1$
```

- - Notice the make command reads the includes/header files and automatically and appropriately compiles, links necessary libraries, and creates all necessary object files for all files in its directory, and from that it creates the .exe (executable) file.



```
File Edit View Search Terminal Help
jinhua@jinhua-VirtualBox:~/sampleP1$ make
g++ -g -c boundedBuffer.c
g++ -o boundedBuffer boundedBuffer.o -lpthread -lrt
g++ -g -c condvar1.c
g++ -o condvar1 condvar1.o -lpthread
jinhua@jinhua-VirtualBox:~/sampleP1$ ./boundedBuffer 100
the total is 4950
jinhua@jinhua-VirtualBox:~/sampleP1$ ./boundedBuffer 10
the total is 45
jinhua@jinhua-VirtualBox:~/sampleP1$
```

- - So above, no matter how they interleave, for this program, total will always be 45 for numInt = 10; this is because the program essentially is doing this:
 $0+1+2+3+4+5+6+7+8+9 = 45$. Number of buffers is 1, thus each consumer and producer has to share the buffer before doing any consumption or production in a buffer.

- Example of a monitor using Pthreads

```

/*
 * FILE: condvar1.c
 * DESCRIPTION:
 *   Example code for using Pthreads condition variables. The main thread
 *   creates three threads. Two of those threads increment a "count" variable,
 *   while the third thread watches the value of "count". When "count"
 *   reaches a predefined limit, the waiting thread is signaled by one of the
 *   incrementing threads.
 *
 * SOURCE: Adapted from example code in "Pthreads Programming", B. Nichols
 * et al. O'Reilly and Associates.
 * LAST REVISED: 02/11/2002 Blaise Barney
 ****

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

#define NUM_THREADS 3
#define TCOUNT 10
#define COUNT_LIMIT 12

int count = 0;
int thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

pthread_cond_t count_threshold_cv;

void *inc_count(void *idp)
{
    int j,i;
    double result=0.0;
    int *my_id;

    my_id = (int *) idp;

    for (i=0; i < TCOUNT; i++) {
        pthread_mutex_lock(&count_mutex);
        count++;

        //Check the value of count and signal waiting thread when condition is
        //reached. Note that this occurs while mutex is locked.
        if (count == COUNT_LIMIT) {
            pthread_cond_signal(&count_threshold_cv);
            printf("inc_count(): thread %d, count = %d Threshold reached.\n",
                   *my_id, count);
        }
        printf("inc_count(): thread %d, count = %d, unlocking mutex\n",
               *my_id, count);
        pthread_mutex_unlock(&count_mutex);

        /* Do some work so threads can alternate on mutex lock */
        for (j=0; j < 1000; j++)
}

```

```

dedBuffer.c condvar1.c
}

printf("inc_count(): thread %d, count = %d, unlocking mutex\n",
       *my_id, count);
pthread_mutex_unlock(&count_mutex);

/* Do some work so threads can alternate on mutex lock */
for (j=0; j < 1000; j++)
    result = result + (double)random();
    sched_yield();
}
pthread_exit(NULL);

void *watch_count(void *idp)
{
    int *my_id;

    my_id = (int *) idp;

    printf("Starting watch_count(): thread %d\n", *my_id);

    /*
    Lock mutex and wait for signal. Note that the pthread_cond_wait routine
    will automatically and atomically unlock mutex while it waits.
    Also, note that if COUNT_LIMIT is reached before this routine is run by
    the waiting thread, the loop will be skipped to prevent pthread_cond_wait
    from never returning.
    */
}

```

- ○ Above, the *do some work so threads can alternate* is just so we can simulate the fact that a thread might have other things to do after it releases a lock; thus they have a for-loop which iterates 1k times; and they also added in a sched_yield() just to also increase interleaving of threads, as part of code simulating how a programmer might use that function.

```

/*
Lock mutex and wait for signal. Note that the pthread_cond_wait routine
will automatically and atomically unlock mutex while it waits.
Also, note that if COUNT_LIMIT is reached before this routine is run by
the waiting thread, the loop will be skipped to prevent pthread_cond_wait
from never returning.
*/
pthread_mutex_lock(&count_mutex);
while (count < COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %d Condition signal received.\n", *my_id);
}
printf("watch_count(): count = %d\n", count);
pthread_mutex_unlock(&count_mutex);
pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    int i, rc;
    pthread_t threads[3];
    pthread_attr_t attr;

    /* Initialize mutex and condition variable objects */
    pthread_mutex_init(&count_mutex, NULL);
    pthread_cond_init (&count_threshold_cv, NULL);

    /* The waiting thread, the loop will be skipped to prevent pthread_cond_wait
     * from never returning.
     */
    pthread_mutex_lock(&count_mutex);
    while (count < COUNT_LIMIT) {
        pthread_cond_wait(&count_threshold_cv, &count_mutex);
        printf("watch_count(): thread %d Condition signal received.\n", *my_id);
    }
    printf("watch_count(): count = %d\n", count);
}

```

- Remember, as shown above, when the watch() thread (thread2) goes to sleep, based on a condition, you must pass in a reference to the lock, because it must release the lock before going to sleep, but then it also needs to re-acquire that same lock when it is signaled and woken up. Then finally, it will release the lock when it leaves the critical section (section of code that only allows up to 1 active thread).
-

```
Buffer.c | condvar1.c |
pthread_mutex_init(&count_mutex, NULL);
pthread_cond_init (&count_threshold_cv, NULL);

/*
For portability, explicitly create threads in a joinable state
*/
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
pthread_create(&threads[0], &attr, inc_count, (void *)&thread_ids[0]);
pthread_create(&threads[1], &attr, inc_count, (void *)&thread_ids[1]);
pthread_create(&threads[2], &attr, watch_count, (void *)&thread_ids[2]);

/* Wait for all threads to complete */
for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);           ←
}
printf ("Main(): Waited on %d threads. Done.\n", NUM_THREADS);

/* Clean up and exit */
pthread_attr_destroy(&attr);
pthread_mutex_destroy(&count_mutex);
pthread_cond_destroy(&count_threshold_cv);
pthread_exit (NULL);
}
```

```
jinhua@jinhua-VirtualBox:~/sampleP1$ ./condvar1
Starting watch_count(): thread 2
inc_count(): thread 1, count = 1
inc_count(): thread 0, count = 2
inc_count(): thread 1, count = 3
inc_count(): thread 0, count = 4
inc_count(): thread 1, count = 5
inc_count(): thread 0, count = 6
inc_count(): thread 1, count = 7
inc_count(): thread 0, count = 8
inc_count(): thread 1, count = 9
inc_count(): thread 0, count = 10
inc_count(): thread 1, count = 11
inc_count(): thread 0, count = 12
inc_count(): thread 0, count = 12 Threshold reached.
watch_count(): thread 2 Condition signal received.
inc_count(): thread 1, count = 13
inc_count(): thread 0, count = 14
inc_count(): thread 1, count = 15
inc_count(): thread 0, count = 16
inc_count(): thread 1, count = 17
inc_count(): thread 0, count = 18
inc_count(): thread 1, count = 19
inc_count(): thread 0, count = 20
Main(): Waited on 3 threads. Done.
jinhua@jinhua-VirtualBox:~/sampleP1$ █
```

-
- Notice, this program implements multiple threads. Notice the watch thread (thread2 = the third thread (threads 0 to 2 = 3 threads)) receives a signal from another thread allowing it to wake up.
- Both t1 and t0 (inc() threads) count to TCOUNT = 10, so total count will be 20.
- References/additional information/further reading on Pthreads

References

- **pthread libraries**

<https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html>

https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html

- POSIX thread programming

<http://www.llnl.gov/computing/tutorials/pthreads/>

- "Multithreaded, Parallel, and Distributed Programming" by Gregory R. Andrews
 -

•

Lecture Video 07-1 (week6 – 2/14/22): CPU Scheduling

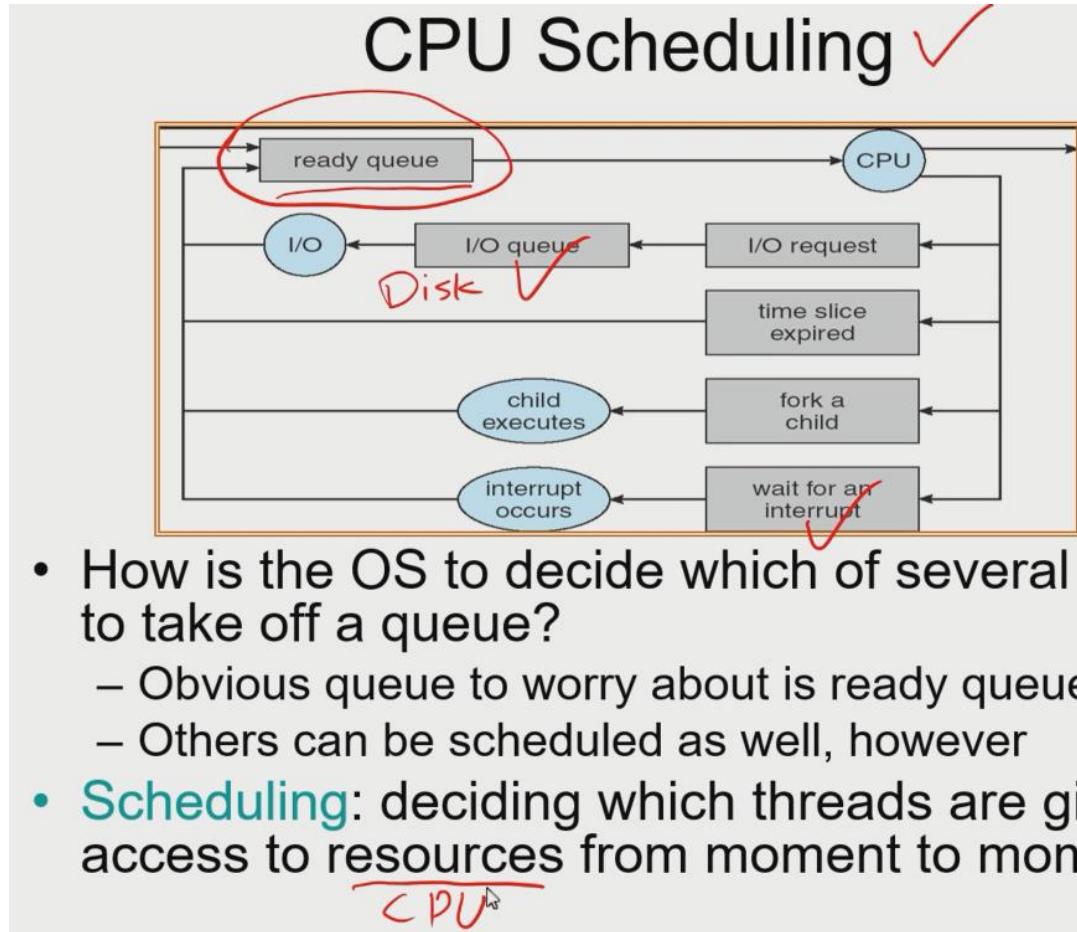
CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time Scheduling

Some slides and/or pictures are adapted from

- Operating System Concepts, 9th edition, by Silberschatz, Galvin, Gagne, John Wiley & Sons, 2013
- Lecture notes by Dr. Prof. John Kubiatowicz (Berkeley)

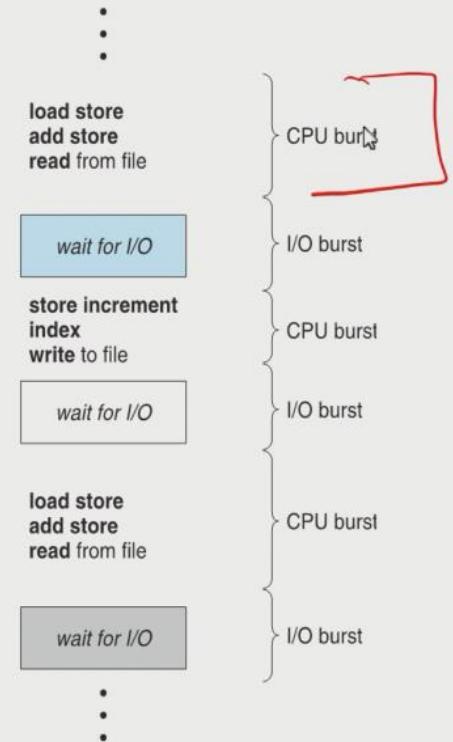
-
- CPU scheduling
 - There are other schedulers too (I/O scheduler, etc...) but we will focus on CPU scheduling



- Basic Concepts
 - Goal is maximum CPU utilization

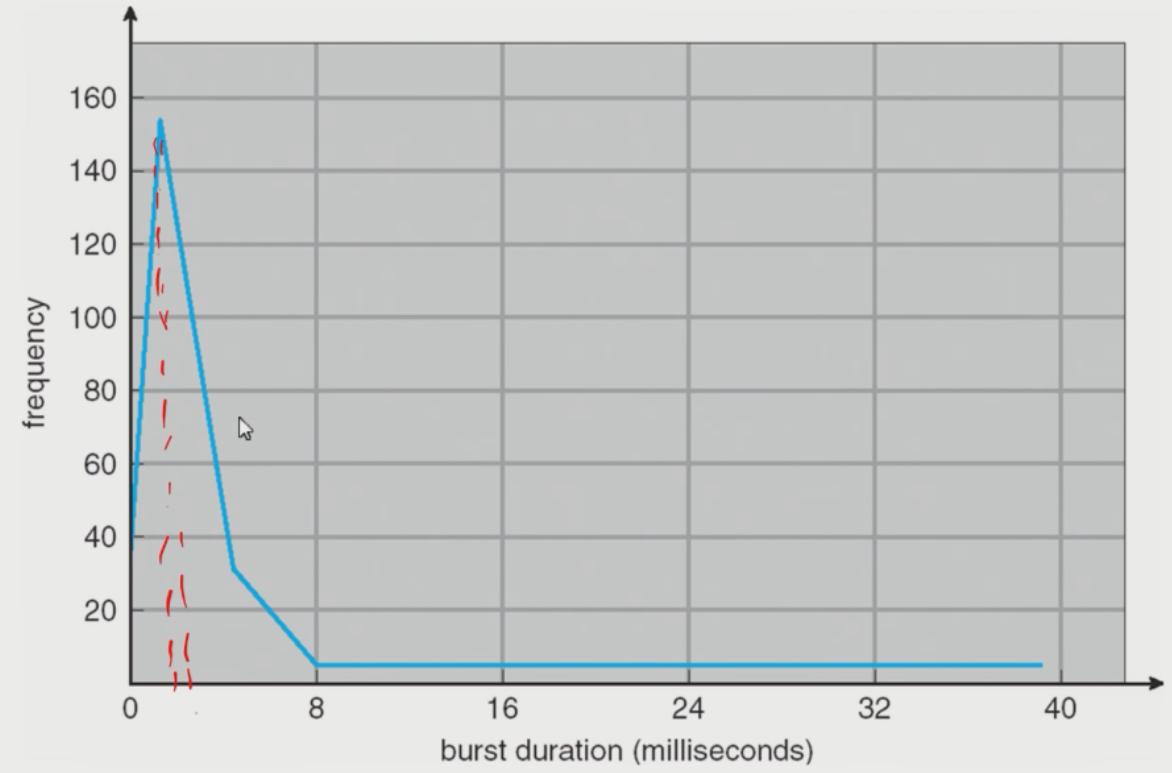
Basic Concepts

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



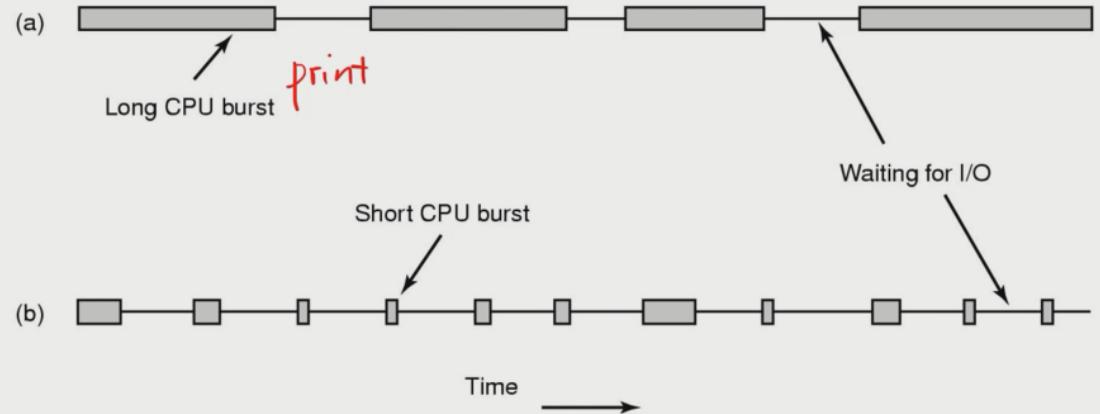
- - Histogram of CPU-burst Times
 - CPU bursts are usually for a very short duration
 - This is because most processes are dealing with the user and therefore are often interrupted
 - Also, most of the time, user does a lot of I/O tasks such as playing a video from the disk, typing input, etc; of course CPU is a lot faster than I/O, which often has to read from the disk. CPU takes about 1 nanosecond to execute an instruction, while I/O often requires 10ms (so 1 million times slower for the disk I/O).

Histogram of CPU-burst Times



- - CPU bound vs. I/O bound Process
 - Not all processes have long I/O; some processes require long CPU bursts and short I/O bursts
 - For example: a computer dedicated to scientific/mathematical calculations, such as calculating digits of PI, will mostly be processing, with small bursts of outputting results (for example), then continuing to do calculations. Such a process would be considered CPU-bound; but most processes are I/O bound.

CPU bound vs. I/O bound Process



- Bursts of CPU usage alternate with periods of I/O wait
 - a) a CPU-bound process
 - b) an I/O bound process
- CPU Scheduler

CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
 - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
 - ✓ 1. Switches from running to waiting state
 - 2. Switches from running to ready state
 - 3. Switches from waiting to ready
 - ✓ 4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
 - Consider access to shared data
 - Consider preemption while in kernel mode
 - Consider interrupts occurring during crucial OS activities
- Run -> wait: think about CS situations/shared data; when a thread needs to acquire a lock or has a synchronization condition that needs to be met, it will voluntarily go to sleep/wait.
- Run -> ready: consider I/O interrupts; a thread is still ready to run but it needs to stop temporarily to allow I/O
- Wait-> ready: thread is woken up by another thread, acquires/tries to acquire a lock...etc.
- Dispatcher
 - Context switching often involves a few thousand instructions (very expensive); affects/determines dispatch latency
 - Thus, we want to minimize dispatch latency when doing CPU scheduling

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
 - switching context TCB
 - switching to user mode
 - jumping to the proper location in the user program to restart that program PC
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

-
- Scheduling Policy Goals/Criteria
 - We want the computer to wait for user, not user waiting for computer; for example, the time it takes from when the user strikes a key, until the computer has a response to that input, such as showing the keystroke that a user pressed and displaying it on the screen in response to the user input.
 - Another example: for cars, the computer controller that controls the engine power input must meet a minimum response time requirement so that the car is safe to drive; if you press the gas, for instance, you need the car to respond quickly to the ratio of your input so that you can speed up appropriately and the mechanical capabilities will not be limited by the computer.
 - Another example: a streamed or local video might lag and not play smoothly because of a scheduling missed deadline.

Scheduling Policy Goals/Criteria

- **Minimize Response Time**

- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
 - Response time is what the user sees:
 - Time to echo a keystroke in editor
 - Time to compile a program
 - Real-time tasks: Must meet deadlines imposed by World

- Maximizing throughput often conflicts with minimizing response time; if you reduce context switching, then, for instance on threads that will produce output for the user to see/wants viewable results from they will potentially have to wait longer; on the other hand, if you when you give some priority to viewable/tangible user output, then you may sacrifice throughput performance. So it is a inversely proportional relationship ($1/x$, or you could think of it as a/b); as you improve efficiency of one, the other gets smaller. It is a balancing act.
 - Mathematically, perhaps think of it like this:
 - ($b = \text{computer performance}$; $a = \text{response time}$): a/b , where we are trying to minimize a , but maximize b .
 - Overall, our goal is usually to make min. response time primary and max. throughput as secondary.

Scheduling Policy Goals/Criteria (Cont.)

- **Maximize Throughput**
 - **Throughput** – # of processes that complete their execution per time unit
 - Throughput related to response time, but not identical:
 - Minimizing response time will lead to more context switching than if you only maximized throughput
 - Two parts to maximizing throughput
 - Minimize overhead (for example, context-switching)
 - Efficient use of resources (CPU, disk, memory, etc)
- We want to keep resources as busy as possible for efficient use of resources

Scheduling Policy Goals/Criteria (Cont.)

- **Fairness**
 - Share CPU among users in some equitable way
 - Fairness is not minimizing average response time:
 - Better average response time by making system less fair
- Also, for real time environments, meeting deadlines is another requirement.
- First Come, First Served (FCFS) Scheduling

First-Come, First-Served (FCFS) Scheduling

Process	^{CPU} Burst Time
P_1	24
P_2	3
P_3	3

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



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

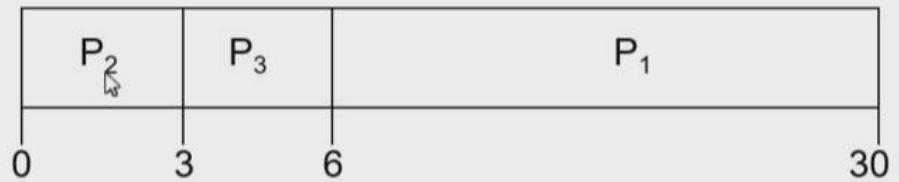
- Waiting time is referring to in the ready queue waiting for the CPU to schedule the thread.
- Average waiting time + average running time = average response time.
 - You cannot really reduce average running time; thus, reducing average response time is mainly about reducing *average wait time*.
- Convoy effect: think about a convoy; if you put the bigger heavier trucks in the back, it is not a big deal since they take longer to get going anyway. But if you place the big trucks in the front of the convoy, well then, when the convoy starts and stops it will have a much bigger impact on the rest of the convoy since they will be bounded/limited by the larger truck that takes so long to get going. So it is better to put them behind the lighter vehicles since they can get going faster and not hold up the bigger trucks or other trucks behind them.
- So, it is better to schedule short processes to run first.

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2, P_3, P_1.$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case.
- Convoy effect short process behind long process*
- Shortest Job First (SJF) Scheduling

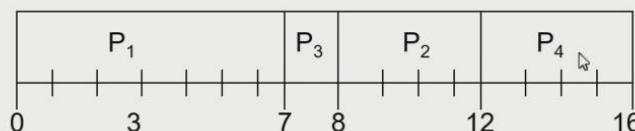
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two schemes:
 - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.
 - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as **the shortest-Restaining-Time-First (SRTF)**.
- SJF is optimal – gives minimum average waiting time for a given set of processes.
 - Most efficient scheme in terms of minimum average wait time for both non preemptive and preemptive schemes (for preemptive, using SRTF version of SJF)
- Example of Non-Preemptive SJF

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time	
P_1	0.0	7	
P_2	2.0	4	✓
P_3	4.0	1	✓
P_4	5.0	4	✓

- SJF (non-preemptive)



- Average waiting time = $(0 + 6 + 3 + 7)/4 = 4$

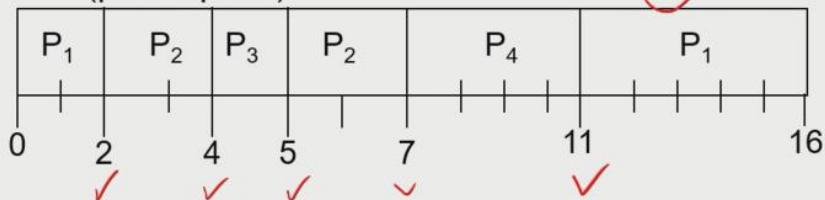
- P1 arrived at 0; wait time is 0 units.
- P2, P3, and P4 all arrived at different times, but they all had to go to ready queue and wait since P1 is still processing by the time they all finally arrived.
- P2 waits from time t = 2 to t= 8; 8-2 = 6 units of wait time.
- P3 waits from t=4 to t=7; 7-4 = 3 units of wait time.
- P4 waits from time t=5 to t=12; 12-5 = 7 units of wait time.
- $(0+6+3+7)/4 = 4$ units of wait time on average.
- For non-preemptive, we cannot stop a process even when a shorter process arrives; so process 1 had to be completed first or ran until the set max CPU burst is reached.
- Example of Preemptive SJF (using SRTF option)

Example of Preemptive SJF (SRTF)

predictive

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3 ✓	4.0	1
P_4	5.0	4

- SJF (preemptive)



more context + switching

- Average waiting time = $\underline{(9 + 1 + 0 + 2)/4 = 3}$

- - P1 arrives at time 0 so it starts to run,
 - but after 2 units of time, P2 arrives with a shorter remaining burst time than P1 (now at 5); so P2 runs for 2 units
 - until P3 arrives; now P2=2, P3=1; thus run P3. Then P4=4 arrives at t=5, thus the shortest remaining burst is P2=2; thus it runs until completion,
 - then P4=4 runs until completion
 - then P1=5 finally runs until completion.
- So notice average wait time is reduced using preemptive (SRTF) option of SJF; however this is at the cost (as we noted before) of **many more context switches**, which is expensive and thus reduces performance: this is the cost of having a reduced average wait time.

- Note that the assigned value for Burst time of a process is predicted using ***predictive*** algorithms (basically AI-like algorithms to predict patterns) that predict how long a process type will take based on past CPU burst time that previous processes of the same type used.
 - So in order to leverage assigning burst time to processes using any SJF scheme, you need to use a predictive algorithm; thus the SJF scheme is the optimal scheme only if you assumed perfect knowledge of the future (aka exact burst times that it will actually take for all processes); thus, in order to harness this optimal scheme, you need a strong predictive algorithms – and you can make this easier by writing programs that yield processes with preferably very predictable run times.
 - So SJF is a benchmark to strive for when writing wait-time algorithms; can use this optimal solution to compare how far away practical schemes are from that optimal solution (under ideal conditions, i.e.: if we knew exact run times of all processes all the time).
- Round Ronin (RR)
 - Widely used in most unix-based and unix-like systems, but also most current CPU scheduling uses some version of RR
 - Set time quantum value at a value between 10-100ms, because remember most CPU bursts are very small; thus, using 10-100ms range for time quantum should allow most small processes to run to completion without interruption for being more than >1 time quantum burst (which then if other smaller processes come in, it will be interrupted).
 -

Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Notice: if **average burst time= SUM_burst_allProcesses*(1/n)**, of all processes is $< q$, then we set that average as the *max* time that a process runs before *yielding* (if necessary); if the average burst time $\geq q$, then we must use q as the default max burst time all processes are given before yielding (again, if necessary; a process may require

less burst time than the currently set max). In doing this (tracking the current average), we can predict the optimal max time needed for all current processes in the ready queue to run to completion (to reduce context switching), with q acting as a bound and default value in case that the average of all processes is very large – that way we never allow smaller processes to wait for extended periods of time.



Demetrius Johnson <meech@umich.edu>
to Jinhua ▾

Okay,

I think I figured it out: so does CPU TIME refer to the total burst time of all processes in the ready queue?

Thus, $(1/n) * \text{CPU TIME} = \text{average burst time for all processes}$; and use that value (bounded by q ; $\text{average} < q$) to determine the max burst time each process will get on the CPU before yielding?

And then if the current burst time average needed by all ready processes is $\geq q$, then by default, use q as the current max CPU burst time for all processes before yielding?

Thanks,

Meech

...

--
Demetrius E Johnson



Jinhua Guo
to me ▾

You are over thinking.

By definition, quantum $q=20$, it means a process can run for 20 time units before it yields CPU. It does not matter how many processes in the system.

The burst time is how long it takes to complete the computation of a process, which is a property of a process and independent of the scheduling.

If the burst time of a process is longer than the quantum, the process must yield before it completes and thus it will need to be scheduled multiple rounds before completion.

-JG

...

- Example of RR with Time Quantum = 20
 - Average completion time = average processing time + average wait time

Example of RR with Time Quantum = 20

- Example:

<u>Process</u>	<u>Burst Time</u>
<u>P_1</u>	53
<u>P_2</u>	8
<u>P_3</u>	68
<u>P_4</u>	24

 - The Gantt chart is:

P₁ P₂ P₃ P₄ P₁ P₃ P₄ P₁ P₃ P₃

0 20 28 48 68 88 108 112 125 145 153

 - Waiting time for P₁ = $(68-20) + (112-88) = 72 + 24 = 96$
 - P₂ = $(20-0) = 20$
 - P₃ = $(28-0) + (88-48) + (125-108) = 85$
 - P₄ = $(48-0) + (108-68) = 80$
 - Average waiting time = $(72+20+85+80)/4 = 66\frac{1}{4}$
 - Average completion time = $(125+28+153+112)/4 = 104\frac{1}{2}$
- Thus, Round-Robin Pros and Cons:
 - Better for short jobs, Fair (+)
 - Context-switching time adds up for long jobs (-)
 - Round-Robin Discussion

Round-Robin Discussion

- How do you choose time slice?
 - What if too big?
 - Response time suffers
 - What if infinite (∞)?
 - Get back FCFS
 - What if time slice too small?
 - Throughput suffers!
- Actual choices of timeslice:
 - Initially, UNIX timeslice one second:
 - Worked ok when UNIX was used by one or two people.
 - What if three compilations going on? 3 seconds to echo each keystroke!
 - In practice, need to balance short-job performance and long-job throughput:
 - Typical time slice today is between ~~10ms – 100ms~~
 - Typical context-switching overhead is ~~0.1ms – 1ms~~
 - Roughly ~~1%~~ overhead due to context-switching

- Comparison between FCFS AND RR
 - In the example, for RR, average wait time is around 900s for each job since there is a context switch every 1 second; all jobs finish at around the same time and no job can finish early; this is much worst in this case since for FCFS, average wait time is only about 450s, and all jobs don't have to finish at the same time and all jobs run to completion; this is better when all jobs are the same length.
 - Also, for RR, every time there is a context switch, cache values are invalid; thus cache state must be cleared and reset, and thus you cannot take advantage of the faster cache memory and having better (and 0 chances in this case) chances of getting a cache hit.

Comparisons between FCFS and Round Robin

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example:

10 jobs, each take 100s of CPU time
 RR scheduler quantum of 1s
 All jobs start at the same time

- Completion Times:

Job #	FCFS	RR
1	100	991
2	200	992
...
9	900	999
10	1000	1000



- Both RR and FCFS finish at the same time
- Average response time is much worse under RR!
 - Bad when all jobs same length
- Also: ~~Cache state must be shared between all jobs with RR but can be devoted to each job with FCFS~~
 - Total time for RR longer even for zero-cost switch!
- This example is an extreme case, usually RR is better than FCFS, especially when you have a larger quantum value and more variable CPU time needed for each job.
 - For this example, quantum is very small; only 1% of the average CPU burst time needed for each job.
- Earlier Example with Different Time Quantum
 - Average of all averages of RR with choosing various quantum values gives a better average than the worst case FCFS, but still it is not better than the best case FCFS, since best case FCFS is the optimal scheduling scheme (jobs run to completion, scheduled SJF: shortest job first).

Earlier Example with Different Time Quantum

Best FCFS:

	P ₂ [8]	P ₄ [24]	P ₁ [53]	P ₃ [68]	
	0	8	32	85	153

Wait Time

	Quantum	P ₁	P ₂	P ₃	P ₄	Average
Best FCFS	32	0	85	8	32	31 $\frac{1}{4}$
Q = 1	84	22	85	57	62	
Q = 5	82	20	85	58	61 $\frac{1}{4}$	
Q = 8	80	8	85	56	57 $\frac{1}{4}$	
Q = 10	82	10	85	68	61 $\frac{1}{4}$	
Q = 20	72	20	85	88	66 $\frac{1}{4}$	
Worst FCFS	68	145	0	121	83 $\frac{1}{2}$	

Completion Time

	Quantum	P ₁	P ₂	P ₃	P ₄	Average
Best FCFS	85	8	153	32	69 $\frac{1}{2}$	
Q = 1	137	30	153	81	100 $\frac{1}{2}$	
Q = 5	135	28	153	82	99 $\frac{1}{2}$	
Q = 8	133	16	153	80	95 $\frac{1}{2}$	
Q = 10	135	18	153	92	99 $\frac{1}{2}$	
Q = 20	125	28	153	112	104 $\frac{1}{2}$	
Worst FCFS	121	153	68	145	121 $\frac{3}{4}$	20

- Notice, the shortest job (P2) has the greatest variation in wait time and completion time:

Earlier Example with Different Time Quantum

Best FCFS: 

	Quantum	P ₁	P ₂	P ₃	P ₄	Average
Wait Time	Best FCFS	32	0	85	8	$31\frac{1}{4}$
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	$61\frac{1}{4}$
	Q = 8	80	8	85	56	$57\frac{1}{4}$
	Q = 10	82	10	85	68	$61\frac{1}{4}$
	Q = 20	72	20	85	88	$66\frac{1}{4}$
	Worst FCFS	68	145	0	121	$83\frac{1}{2}$
Completion Time	Best FCFS	85	8	153	32	$69\frac{1}{2}$
	Q = 1	137	30	153	81	$100\frac{1}{2}$
	Q = 5	135	28	153	82	$99\frac{1}{2}$
	Q = 8	133	16	153	80	$95\frac{1}{2}$
	Q = 10	135	18	153	92	$99\frac{1}{2}$
	Q = 20	125	28	153	112	$104\frac{1}{2}$
	Worst FCFS	121	153	68	145	$121\frac{3}{4}$

- - Remember, completion time = wait time + processing time.
 - Also notice for the shortest job (P2), q values lie between best case and worst case of FCFS.
- Also notice, that for the larger/largest job (P3), whether it doesn't really make a difference as to whether you use RR (and any quantum value chosen) or FCFS since they all end up completing last anyways (except for worst case FCFS):

○ Earlier Example with Different Time Quantum

Best FCFS:		P ₂ [8]	P ₄ [24]	P ₁ [53]	P ₃ [68]	
		0	8	32	85	153
Wait Time	Quantum	P ₁	P ₂	P ₃	P ₄	Average
	Best FCFS	32	0	85	8	31½
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	61¼
	Q = 8	80	8	85	56	57¾
	Q = 10	82	10	85	68	61¼
	Q = 20	72	20	85	88	66¼
Completion Time	Best FCFS	85	8	153	32	69½
	Q = 1	137	30	153	81	100½
	Q = 5	135	28	153	82	99½
	Q = 8	133	16	153	80	95½
	Q = 10	135	18	153	92	99½
	Q = 20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

- Thus, smaller jobs are more sensitive to the quantum value used.
- Also note for this example we are still not considering/including cost of context switching.
- So, RR is kind of in between method.
-

Lecture Video 07-2 (week6 – 2/14/22): CPU Scheduling

- Priority Scheduling

✓
✓
✓

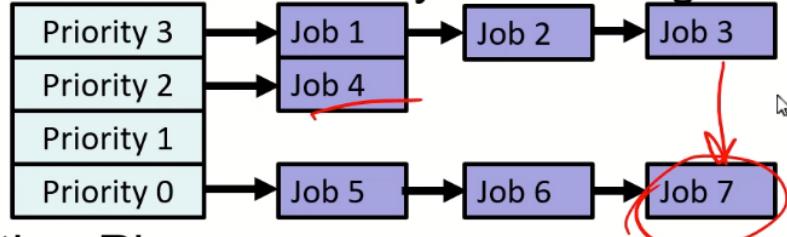
20

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority).
 - Preemptive
 - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- Problem = Starvation – low priority processes may never execute.
- Solution = Aging – as time progresses increase the priority of the process.
 - Handling Differences in Importance: Strict Priority Scheduling
 - RR = Round Robin

Handling Differences in Importance:

Strict Priority Scheduling



- **Execution Plan**

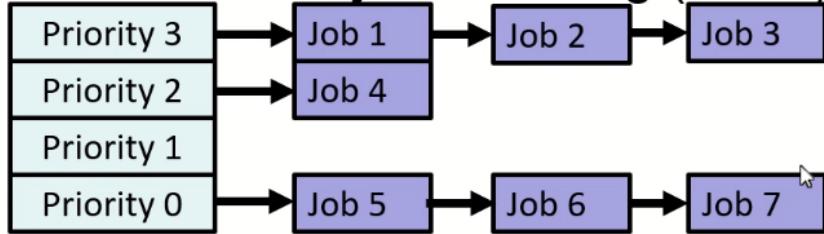
- Always execute highest-priority runnable jobs to completion
- Each queue can be processed in RR with some time-quantum

- **Problems:**

- Starvation:
 - Lower priority jobs don't get to run because higher priority jobs
- Deadlock: Priority Inversion
 - Not strictly a problem with priority scheduling, but happens when low priority task has lock needed by high-priority task
 - Usually involves third, intermediate priority task that keeps running even though high-priority task should be running

Handling Differences in Importance:

Strict Priority Scheduling (Cont.)



- **How to fix problems?**

- Dynamic priorities – adjust base-level priority up or down based on heuristics about interactivity, locking, burst behavior, etc...

- What Are Heuristics? A heuristic, or heuristic technique, is any approach to problem-solving that uses a practical method or various shortcuts in order to produce solutions that may not be optimal but are sufficient given a limited timeframe or deadline.

- Ex: if burst time needed for a given type of thread is found to be less than predicted, then adjust priority level and give the thread a higher priority.
- Scheduling Fairness
 - Multics = one of the first multiprogramming operating systems

Scheduling Fairness

- What about fairness?
 - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
 - long running jobs may never get CPU
 - In Multics, shut down machine, found 10-year-old job
 - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
 - Tradeoff: fairness gained by hurting avg response time!

Scheduling Fairness

- How to implement fairness?
 - Could give each queue some fraction of the CPU
 - What if one long-running job and 100 short-running ones? $\frac{10\%}{100\%}$
 - Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
 - Could increase priority of jobs that don't get service
 - What is done in some variants of UNIX
 - This is ad hoc—what rate should you increase priorities?
 - And, as system gets overloaded, no job gets CPU time, so everyone increases in priority \Rightarrow Interactive jobs suffer
 - Ad Hoc = Ad hoc is a word that originally comes from Latin and means "for this" or "for this situation." In current American English it is used to describe something that has been formed or used for a special and immediate purpose, without previous planning.
 - Ad Hoc = when necessary or needed.
"the group was constituted ad hoc"
 - Notice: above 10% for one long-running job versus 90% for 100 short running-jobs is unfair (unbalanced), since the 1 long job gets 10% CPU time, while all short jobs get an average of $.90/100 = 9\%$ CPU time per job; thus we are giving the 1 long job a higher priority than that of short jobs – which is not the objective.
- Lottery Scheduling (Program 4)
 - We will implement lottery scheduling program in project 4 after midterm exam
 - SRTF = shortest remaining time first

Lottery Scheduling



- Yet another alternative: Lottery Scheduling
 - Give each job some number of lottery tickets
 - On each time slice, randomly pick a winning ticket
 - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
 - To approximate SRTF, short running jobs get more, long running jobs get fewer
 - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
 - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

Lottery Scheduling Example

- Lottery Scheduling Example
 - Assume short jobs get 10 tickets, long jobs get 1 ticket

# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91% $\frac{10}{11}$	9% $\frac{1}{11}$
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9% $\frac{10}{101}$	0.99% $\frac{1}{101}$
1/10	50% $\frac{10}{20}$	5% $\frac{1}{20}$

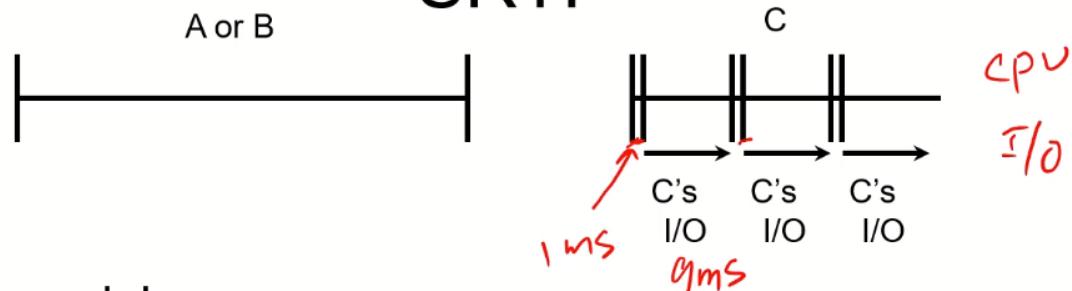
- What if too many short jobs to give reasonable response time?
 - If load average is 100, hard to make progress
 - One approach: log some user out

- Fractions shown in example is the fraction that represents the percentage given in each scenario; it is num tickets for a given job type (a single quantity, such as 10 for short, or 1 for long) divided by the total tickets in the system. Total tickets = (number of tickets for a single quantity of a short job)*(total number of short jobs in the system) + (number of tickets for a single quantity of a long job)*(total number of long jobs in the system).
- Notice in the examples, no matter how many jobs are in the system, the short jobs always get a higher priority (for lottery = average CPU time per job) than the long jobs while still giving long jobs a time to run; thus system is more balanced and fair but of course there are some costs to this method too in terms of tailored efficiency; this approach gives strong general efficiency.
- Log-out some users = kill some processes
 - Do this in the event when you cannot make progress and we are constantly context switching = overloads system
- Discussion of SJF/SRTF, FCFS, and RR

Discussion

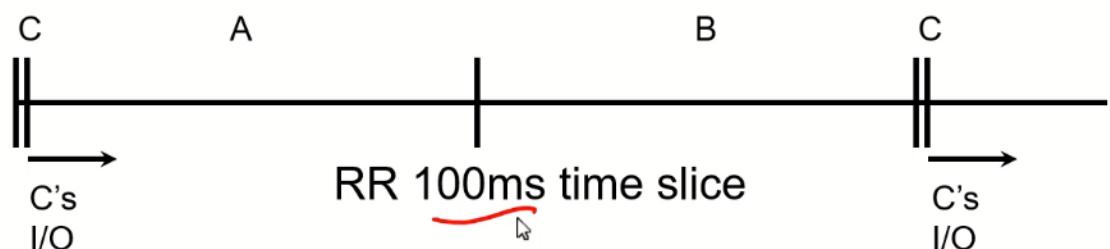
- SJF/SRTF are the best you can do at minimizing average response time
 - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
 - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS and RR
 - What if all jobs the same length?
 - SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
 - What if jobs have varying length?
 - SRTF (and RR): short jobs not stuck behind long ones
 - shortest remaining time first
- Example to illustrate benefits of SRTF (shortest remaining time first)
 - (CPU burst is 1 week for jobs A or B)
 - Bound = the ultimate determinate (of burst time in this case)
 - Remember: we not only want CPU to be maximally and optimally busy, but also all other devices such as I/O devices and disk usage; we want to minimize idle time.

Example to illustrate benefits of SRTF



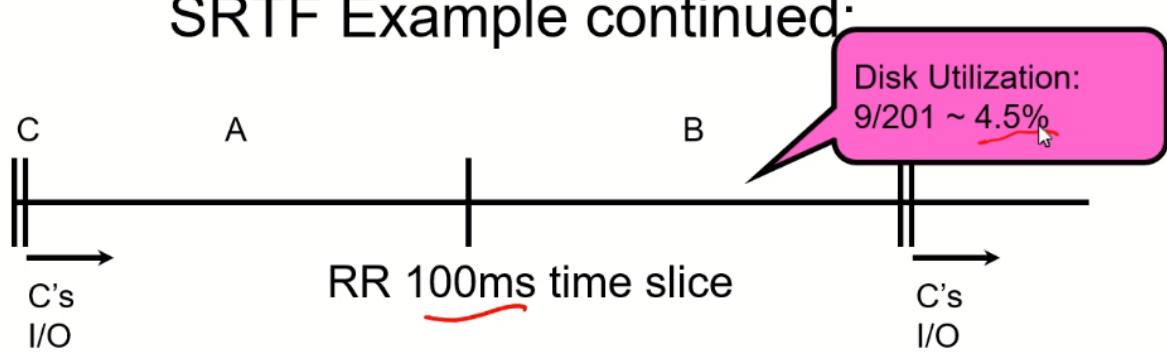
- Three jobs:
 - A,B: both CPU bound, run for week
 - C: I/O bound, loop 1ms CPU, 9ms disk I/O
 - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FIFO:
 - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
 - Easier to see with a timeline
- With FIFO, CPU would be busy 100% of the time, but I/O will be 0% utilized while A or B run.

SRTF Example continued:



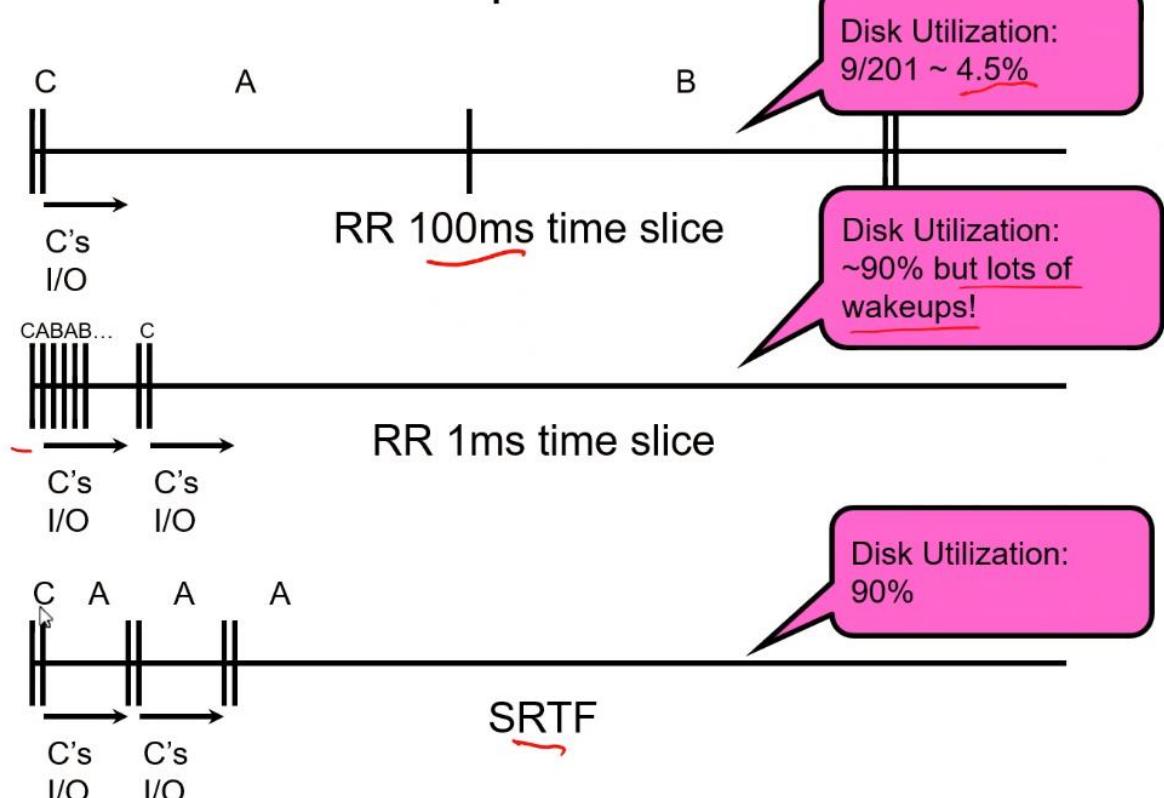
- Sequence (quantum q is set to 100ms): C runs for 1ms, I/O runs for 9ms, A runs for 100ms, B runs for 100ms...etc..

SRTF Example continued:



- RR where $q=1\text{ms}$ has 90% utilization but lots of context switching and the very long jobs A and B cannot really make enough progress. Notice how I/O overlaps the time when A and B are processing (of course it is concurrent with C, since job C is what prompts the 9ms I/O operation), allowing for both CPU and I/O to both be working simultaneously.
- For SRTF, notice how it accomplished what RR does by not making short jobs wait, but also by allowing longer periods of execution for the larger jobs, while simultaneously allowing I/O to run while the CPU is running and working on jobs A or B, only pausing briefly when it is waiting for thread C.
 - SRTF is optimal solution as it minimizes response time since most small jobs are mostly I/O bound (but as we see, B will not have any run time until after A fully completes). But in terms of CPU and I/O (such as disk) utilization and not making short jobs wait, it is the optimal solution.

SRTF Example continued:

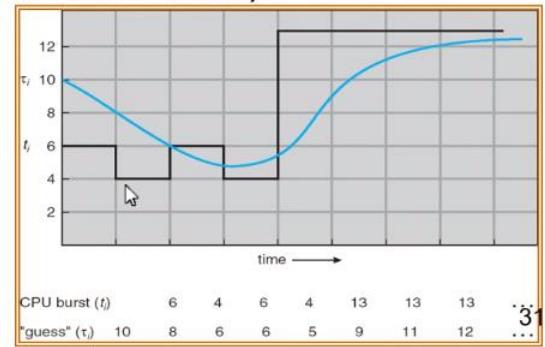


30

- Predicting the Length of the Next CPU Burst
 - For I/O bound processes: policy will usually predict to give it a high priority since CPU burst will likely be small; this will provide much faster response time and utilization as shown from the previous example as I/O and CPU can run in parallel.
 - Estimate the next burst as a function of previous bursts.
 - We use a moving average function often times so that we can do better and adapt for better future predictions based on more recent/relevant CPU burst times.
 - T_{n-1} is the new moving average from the exponential averaging function.

Predicting the Length of the Next CPU Burst

- **Adaptive:** Changing policy based on past behavior
 - CPU scheduling, in virtual memory, in file systems, etc
 - Works because programs have predictable behavior
 - If program was I/O bound in past, likely in future
 - If computer behavior were random, wouldn't help
- Example: SRTF with estimated burst length
 - Use an estimator function on previous bursts:
Let $t_{n-1}, t_{n-2}, t_{n-3}$, etc. be previous CPU burst lengths.
Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \dots)$
 - Function f could be one of many different time series estimation schemes (Kalman filters, etc)
 - For instance,
exponential averaging
 $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$
with $(0 < \alpha \leq 1)$



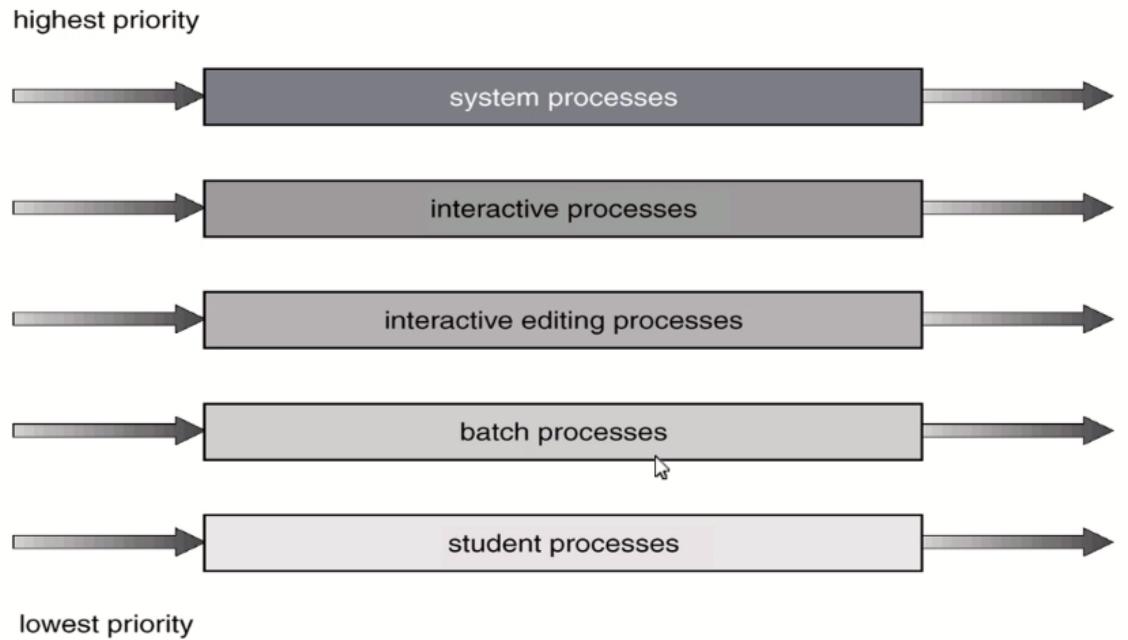
- - Multilevel Queue Scheduling
 - Partition: break up; split (not necessarily evenly); separate.
 - Foreground = user applications (I/O bound)
 - Background = OS/system applications (CPU bound)
 - In a computer, a batch job (associated with batch files) is a program that is assigned to the computer to run without further user interaction.
 - Examples of batch jobs in a PC are a printing request or an analysis of a Web site log. In larger commercial computers or servers, batch jobs are usually initiated by a system user.

Multilevel Queue

- Ready queue is partitioned into separate queues:
foreground (interactive) I/O
background (batch) CPU
- Each queue has its own scheduling algorithm,
foreground – RR
background – FCFS
- Scheduling must be done between the queues.
 - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
 - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e.,
80% to foreground in RR
20% to background in FCFS

o

Multilevel Queue Scheduling

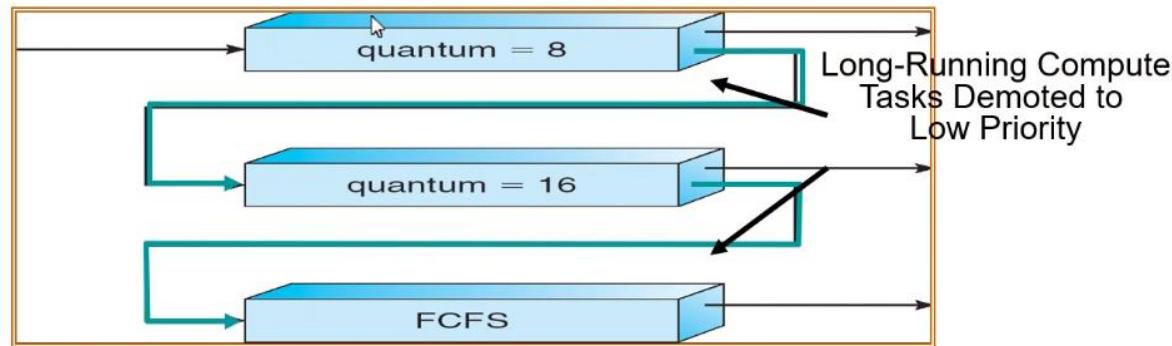


- Multilevel Feedback Queue
 - We can have some heuristics that we can move up and down the priority of a process to provide some feedback

Multilevel Feedback Queue

- Another method for exploiting past behavior
- A process can move between the various queues; aging can be implemented this way.
- Multilevel-feedback-queue scheduler defined by the following parameters:
 - number of queues
 - scheduling algorithms for each queue
 - method used to determine when to upgrade a process
 - method used to determine when to demote a process
 - method used to determine which queue a process will enter when that process needs service
- A time variable is assigned; when that time variable expires (goes past quantum of the given level), we make a process drop one level; if timeout doesn't expire before a job finishes, we jump to higher priority level for these processes.

Multilevel Feedback Queues



- Adjust each job's priority as follows (details vary)
 - Job starts in highest priority queue
 - If timeout expires, drop one level
 - If timeout doesn't expire, push up one level (or to top)
 - This method is widely used today in newer systems.
- Scheduling Details
 - For scheduling between queues: we can use fixed priority, Time slice, or lottery method.
 - **Brilliant statement from Dr. Guo** that not only applies to programming: "*For any kind of policy there can be abuse, if you know exactly how it works, so that you can take advantage of it*".
 - you can periodically place simple I/O statements such as `Printf()` solely for the purpose of causing the prediction and scheduling algorithm to assign your process a higher priority as it will appear that it is I/O bound when in reality it is a CPU bound process.

Scheduling Details

- Result approximates SRTF:
 - CPU bound jobs drop like a rock
 - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
 - **Fixed priority scheduling:**
 - serve all from highest priority, then next priority, etc.
 - **Time slice:**
 - each queue gets a certain amount of CPU time
 - e.g., 70% to highest, 20% next, 10% lowest
- **Countermeasure:** user action that can foil intent of the OS designer
 - For multilevel feedback, put in a bunch of *printf* meaningless I/O to keep job's priority high
 - Of course, if everyone did this, wouldn't work!
- Example of Othello program:
 - Playing against competitor, so key was to do computing at higher priority than competitors.
 - Put in printf's, ran much faster!

36

○

Lecture Video 07-3 (week6 – 2/14/22): CPU Scheduling

- Multiple-Processor Scheduling

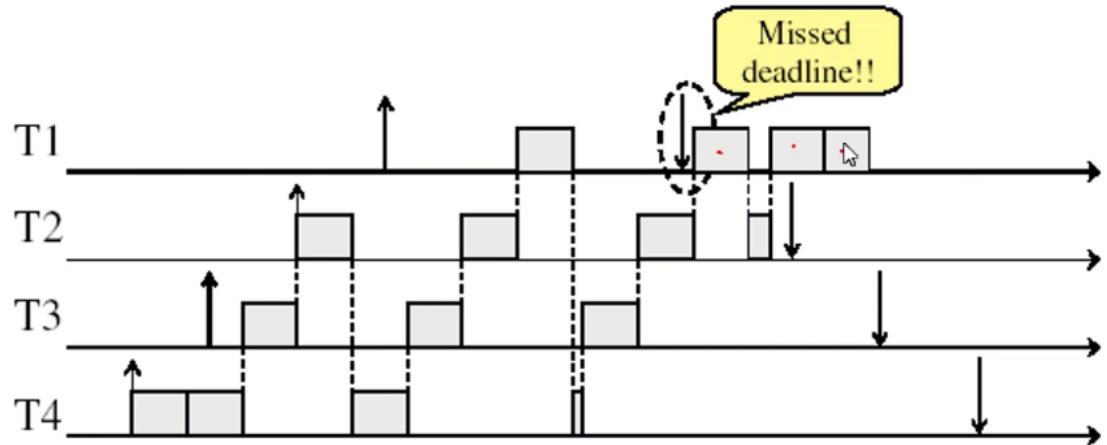
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
 - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
 - **soft affinity**
 - **hard affinity**
 - Variations including **processor sets**
- Real-Time Scheduling (RTS)

Real-Time Scheduling (RTS)

- Efficiency is important but **predictability** is essential:
 - We need to predict with confidence worst case response times for systems
 - In RTS, performance guarantees are:
 - Task- and/or class centric and often ensured a priori
 - In conventional systems, performance is:
 - System/throughput oriented with post-processing (... wait and see ...)
 - Real-time is about enforcing predictability, and does not equal fast computing!!!
- Hard Real-Time (RTOS) QNX
 - Attempt to meet all deadlines
 - must execute process in specific time, or reject it; known as resource reservation.
 - require different kind of OS.
- Soft Real-Time
 - Attempt to meet deadlines with high probability
 - Minimize miss ratio / maximize completion ratio (firm real-time)
 - Important for multimedia applications
- Predictability is essential: we want to ensure that the tasks finish before the deadline
- RTOS = Real time Operating system; needed to implement Hard Real-Time policy
 - Lenox OS is not a RTOS
 - Usually, RTOS is usually required for embedded systems that require real time implementations. Examples of embedded systems include controlling engine, controlling breaks, control of a nuclear reactor..etc.
 - QNX: RTOS used in cars often; developed by Blackberry
 - They use RTOS because their QNX is widely used even today (they used to focus on server OS, now they focus on RTOS in their QNX operating system) in real time systems, and they provide better security.
- For firm deadline (maximize completion ratio), an example could be outputting video at 30 fps == 1 frame is sent to screen every 33ms
- Example: Workload Characteristics

Example: Round-Robin Scheduling Doesn't Work

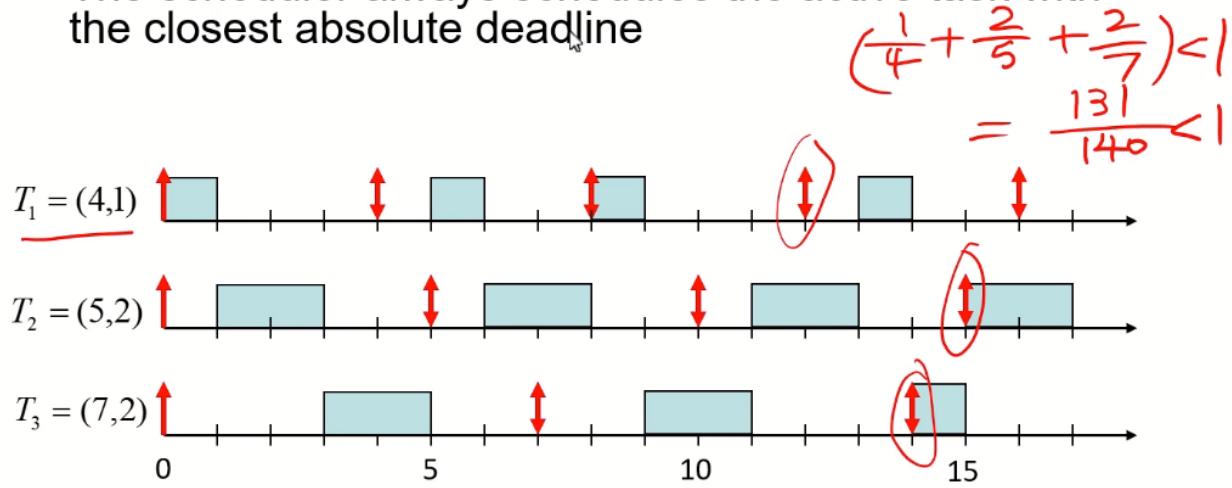


Time —————→

- Computation time is derived by studying and analyzing the source code and running the worst case scenario; C is set as worst-case scenario computation time (or could be based on previous run times.)
- In the example: up arrow indicates arrival time.
- Round-robin does not work since T1 missed its deadline, but only because it arrived later than all of the other tasks.
- Solution: earliest deadline first; if a thread has a higher process, it will be given a higher priority.
- Earliest Deadline First (EDF)

Earliest Deadline First (EDF)

- Tasks periodic with period P and computation C in each period: (P, C)
- Preemptive priority-based dynamic scheduling
- Each task is assigned a (current) priority based on how close the absolute deadline is
- The scheduler always schedules the active task with the closest absolute deadline



- In real-time/embedded systems they often have periodic tasks they need to attempt
- $P = \text{period} = \text{deadline}/\text{total time allotted before task will be killed}$
- $C = \text{computation time}$
- If all ratios add up to less than 1, then the tasks are schedulable (all tasks can complete before deadline)
-
- Scheduling in Real-Time Systems

Scheduling in Real-Time Systems

Schedulable real-time system

- Given
 - m periodic events
 - event i occurs within period P_i and requires C_i seconds
- Then the load can only be handled if

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

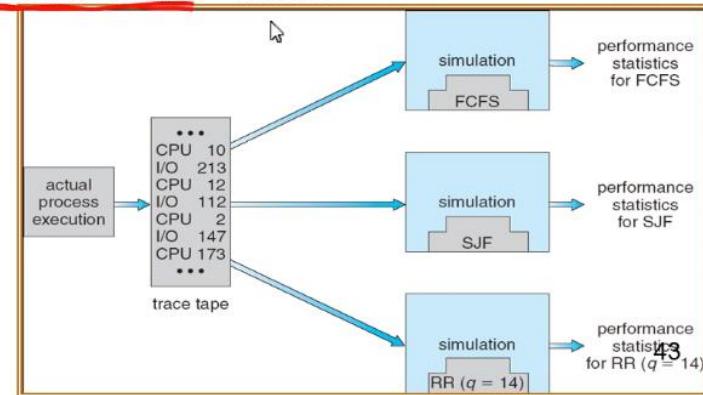
$= (\frac{1}{4} + \frac{2}{5} + \frac{2}{7}) < 1$

- How to Evaluate a Scheduling Algorithm?

How to Evaluate a Scheduling algorithm?

- + • Deterministic modeling
 - takes a predetermined workload and compute the performance of each algorithm for that workload
- + • Queueing models
 - Mathematical approach for handling stochastic workloads
- ✓ • Implementation/Simulation:
 - Build system which allows actual algorithms to be run against actual data. Most flexible/general.

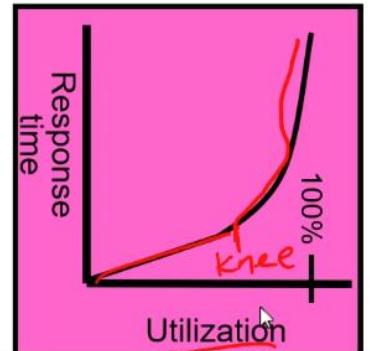
System Research



- Usually, implementation is the most practical evaluation method, and it can be enhanced or complimented with deterministic and queueing modeling.
- A Final Word on Scheduling

A Final Word On Scheduling

- When do the details of the scheduling policy and fairness really matter?
 - When there aren't enough resources to go around
- When should you simply buy a faster computer?
 - One approach: Buy it when it will pay for itself in improved response time
 - Assuming you're paying for worse response time in reduced productivity, customer angst, etc...
 - Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization $\Rightarrow 100\%$



- An interesting implication of this curve:
 - Most scheduling algorithms work fine in the “linear” load portion of the load curve, fail otherwise
 - Argues for buying a faster X when hit “knee” of curve
 - When CPU and other system resources are actually being heavily used (i.e., not a lot of idle time), that is when scheduling really makes the difference in terms of performance, fairness, response time, etc..
 - Also: you might upgrade your system hardware to faster system and buy a new computer at around 60% (esp. at 80%) resource utilization (at the “knee”), since then response time increases (meaning it takes longer for responses; thus performance slows down for the user) linearly initially, but then exponentially the closer we move to 100% utilization. That is why you can never 100% utilize your system resources and expect to actually be able to use it/that it will be efficient or even responsive.
 - After a certain point of high utilization, as shown by the curve at the “knee”, scheduling algorithm doesn’t matter and will fail and the machine will not function properly or with sufficient responsiveness and performance.
- Summary

Summary

- Scheduling: selecting a waiting process from the ready queue and allocating the CPU to it
- FCFS Scheduling:
 - Run threads to completion in order of submission
 - Pros: Simple
 - Cons: Short jobs get stuck behind long ones
- Round-Robin Scheduling:
 - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
 - Pros: Better for short jobs
 - Cons: Poor when jobs are same length
- Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):
 - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
 - Pros: Optimal (average response time)
 - Cons: Hard to predict future, Unfair
- Multi-Level Feedback Scheduling:
 - Multiple queues of different priorities
 - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF
 - Also don't forget about Lottery option.

Lecture Video 08-1 (week7 – 2/21/22): Deadlock

- Deadlock

Deadlock

- Definition of deadlock
- Condition for its occurrence
- Solutions for avoiding and breaking deadlock
 - Deadlock Prevention
 - Deadlock Avoidance
 - Deadlock Detection
 - Recovery from Deadlock
- Resources
 -

Resources

- Examples of computer resources
 - printers
 - tape drives
 - tables
- Processes need access to resources in reasonable order
- Suppose a process holds resource A and requests resource B
 - at the same time another process holds B and requests A
 - both are blocked and remain so
 - Resources could be hardware (hard drive..etc.) or software (locks, semaphores, etc.)
 - Preemptable resources: CPU has no ill effects since you can save CPU states of processes that are paused, and the same goes for Memory since you can save the memory state into the HDD so that memory resources can be freed up without disrupting paused memory apart of another process/processes using a swap function (RAM → HDD, then back to memory when memory is freed again HDD → RAM).

Resources (1)

- Deadlocks occur when ...
 - processes are granted exclusive access to devices
 - we refer to these devices generally as **resources**
- Preemptable resources
 - can be taken away from a process with no ill effects
 - e.g. CPU, Memory *Swap*
- Nonpreemptable resources
 - will cause the process to fail if taken away
 - e.g. Disk space, plotter, chunk of virtual address space
 - Mutual exclusion – the right to enter a critical section
 - Essentially, non preemptable means we have a race condition /critical section problem, which of course then means we have a resource, which of course means we need some mechanism to enter exclusively, use, and exit/release a resource → aka: Locks/Semaphores/Monitors.

Resources (2)

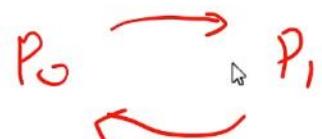
- Sequence of events required to use a resource
 1. request the resource
 2. use the resource
 3. release the resource
- Must wait if request is denied
 - requesting process may be blocked
 - may fail with error code
- Introduction to Deadlocks
 - “ALL of them (all processes that have some dependency relationship) must be waiting on each other” = deadlock situation / circular waiting.

Introduction to Deadlocks

- Formal Definition:
 - A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.
- Example
 - semaphores A and B, initialized to 1

P_0
① $\text{wait}(A);$
 $\text{wait}(B);$

P_1
② $\text{wait}(B)$
 $\text{wait}(A)$



- Starvation vs. Deadlock

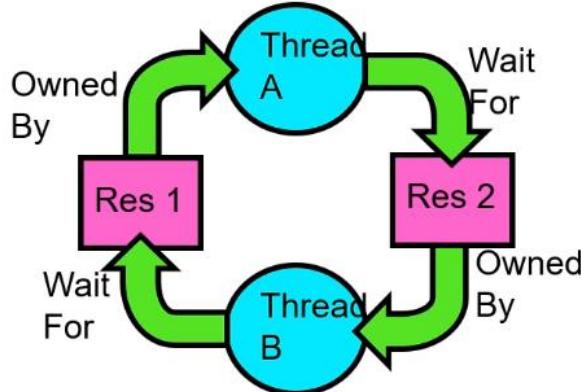
- Starvation is also called a “live lock” since eventually a starved process will run or possibly run; problem can eventually be resolved by itself.
- Deadlock = no chance a process/processes run; need external intervention.
 - For example: circular waiting situation; the circle of waiting/loop will last forever.

Starvation vs Deadlock

- Starvation vs. Deadlock
 - Starvation: thread waits indefinitely

*Live lock
SJF*

- Example, low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock: circular waiting for resources
 - Thread A owns Res 1 and is waiting for Res 2
 - Thread B owns Res 2 and is waiting for Res 1



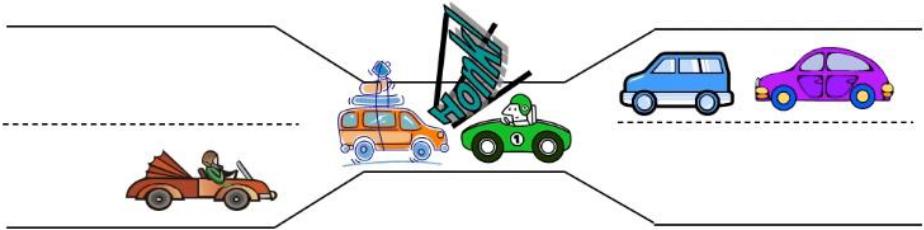
- Deadlock \Rightarrow Starvation but not vice versa
 - Starvation can end (but doesn't have to)
 - Deadlock can't end without external intervention

- The 4 Necessary Conditions for Deadlock
 - Similar to: Properties of Critical Section Solution (4 main properties must be satisfied, usually)

Necessary Conditions for Deadlock

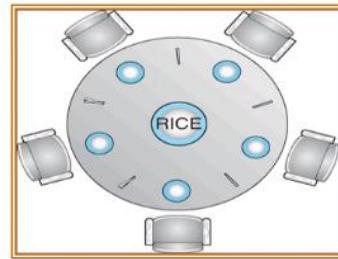
1. Mutual exclusion condition
 - . each resource assigned to 1 process or is available
2. Hold and wait condition
 - . process holding resources can request additional
3. No preemption condition
 - . previously granted resources cannot forcibly taken away
4. Circular wait condition
 - . must be a circular chain of 2 or more processes
 - . each is waiting for resource held by next member of the chain
 - Bridge Crossing Example

Bridge Crossing Example



- Each segment of road can be viewed as a resource
 - Car must own the segment under them
 - Must acquire segment that they are moving into
- For bridge: must acquire both halves
 - Traffic only in one direction at a time
 - Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
 - Several cars may have to be backed up
- Starvation is possible
 - East-going traffic really fast \Rightarrow no one goes west
 - The algorithm for project 3 can cause starvation; we give the preference to the car in the same direction as the direction of cars with the most cars in that direction.
 - Cars in the opposite direction will be starved.
- Dining Philosopher Problem
 - To prevent deadlock, we can try to attack any one of the 4 necessary conditions for deadlock; if we can break one of them, we can break the deadlock; for example: we can use preemption (make one of them give up a chopstick when there is a deadlock: everyone has a single chopstick).

Dining Philosophers Problem



- Five chopsticks/Five Philosophers (really cheap restaurant)
- What if all grab at same time?
 - Deadlock!
- How to fix deadlock?
 - Make one of them give up a chopstick (Hah!)
 - Eventually everyone will get chance to eat
- How to prevent deadlock?
 - Never let Philosophers take last chopstick if no hungry Philosophers has two chopsticks afterwards
 - Request both chopsticks at the same time
- Resource-Allocation Graph

Resource-Allocation Graph

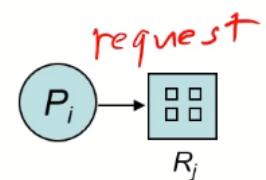
- Process



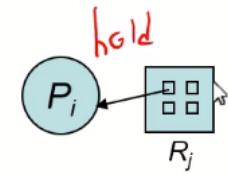
- Resource Type with 4 instances



- P_i requests instance of R_j

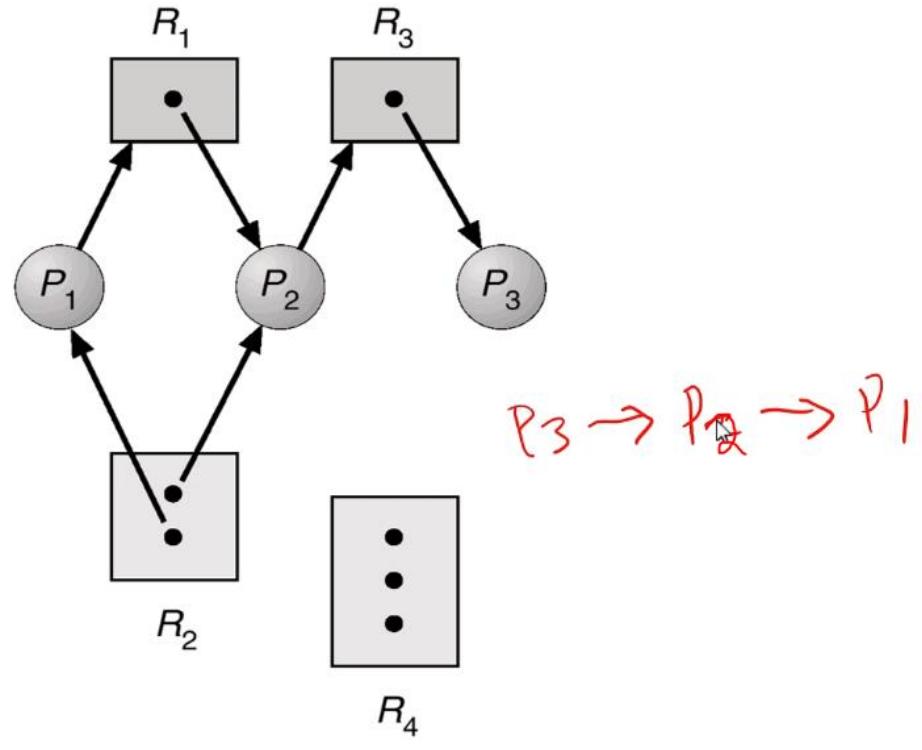


- P_i is holding an instance of R_j



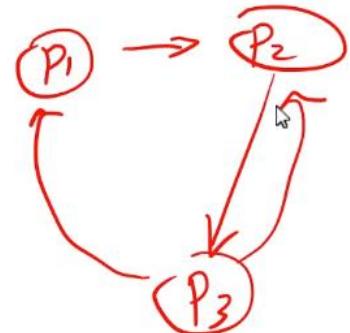
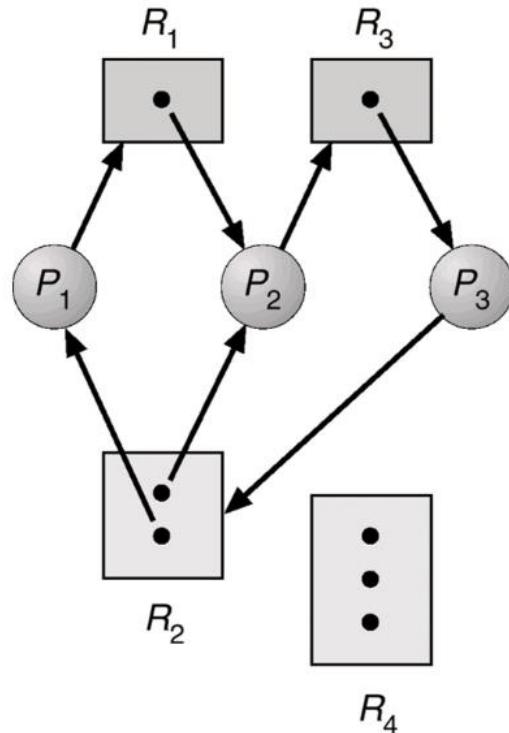
o

Example of a Resource Allocation Graph



- - Above: not a deadlock situation.

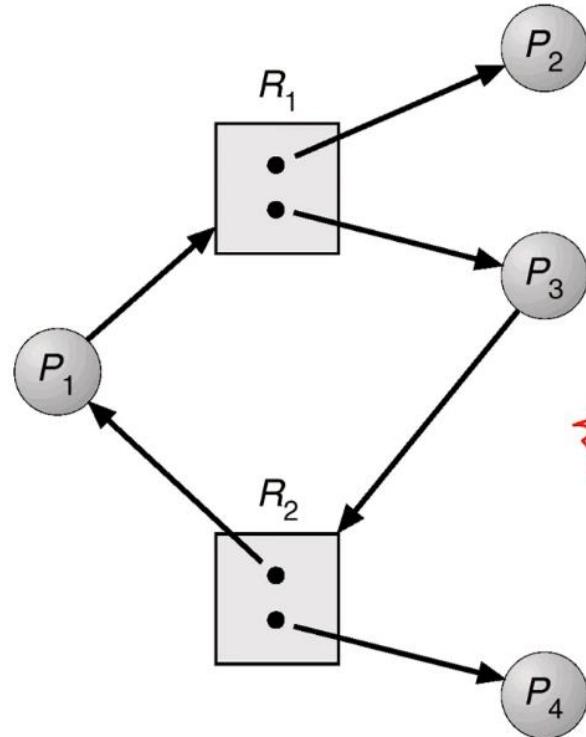
Resource Allocation Graph With A Deadlock



○

- Above, we have two circular waiting situations:
 - P_1 waits for P_2 , which waits for P_3 , which waits for P_1 .
 - P_2 waits for P_3 , which waits for P_2 .

Resource Allocation Graph With A Cycle But No Deadlock



13

- - Above: P_2 and P_4 can complete on their own, which will then allow P_1 and P_3 to complete.

Basic Facts

- If graph contains no cycles \Rightarrow no deadlock.
- If graph contains a cycle \Rightarrow
 - if only one instance per resource type, then deadlock.
 - if several instances per resource type, possibility of deadlock.
 - Strategies for dealing with Deadlocks
 - Ignore case: deadlock is rare/negligible/cost of fixing is too high (throughput suffers / low resource utilization or cost of time of writing code for prevention may be high); if it occurs, just reboot/rerun the program for example.
 - Dynamic avoidance case: when it is “not safe” (chance of deadlock), even if a resource is available, do not allow resource to be acquired at the moment or for a given process.
 - Con: Can cause low utilization of resources.

Strategies for dealing with Deadlocks

1. just ignore the problem altogether *optimistic*
2. prevention
 - negating one of the four necessary conditions
3. dynamic avoidance *Not safe*
 - careful resource allocation
4. detection and recovery
 - - The Ostrich Algorithm
 - Notice: since OSes sometimes use the ignore method, this is why it is often good to reboot a system to fix many problems with performance/correctness. For example, maybe you are having memory issues and your system is running slow; there may be built-up deadlocks and unused resources: so just reboot the system to gain resources back and start the program/machine at a fresh state and allow deadlocks to rebuild for a while before you solve the issue yet again: reboot.
 - It is a tradeoff: Reboot system when there are issues; or write code to prevent deadlocks, but that will use a lot of resources or prevent full utilization of resource, thus if that cost is too high, it may be counterproductive to use such solutions; more optimal solution may be to ignore, and simply reboot a system.
 - Also: if there is a small set of processes in a deadlock; you can simply just kill them manually and restart the processes if needed (for example, an internet browser process that may be suffering performance due to deadlock issues; just close window to kill process, or kill the entire application process and start it again).

The Ostrich Algorithm

- Pretend there is no problem
- Reasonable if
 - deadlocks occur very rarely
 - cost of prevention is high
- UNIX and Windows takes this approach
- It is a trade off between
 - convenience
 - correctness
- - Widely adopted approach.
- Deadlock Prevention: Attacking the Mutual Exclusion Condition

Deadlock Prevention

Attacking the Mutual Exclusion Condition

- Some devices (such as printer) can be spooled
 - only the printer daemon uses printer resource
 - thus deadlock for printer eliminated
- Not all devices can be spooled
 - Spooled device:
 - A line printer used to print the output of a number of jobs is an example of a spooled device. Spool is an acronym for simultaneous peripheral operations on-line. Spooling is a process in which data is temporarily held to be used and executed by a device, program or the system.
 - Thus, data is held inside memory caches inside the printer's hardware device, so that it can manage what prints and in which order, thus allowing all devices in the network to "share" the printer.
- Deadlock Prevention: Attacking the Hold and Wait Condition

Attacking the Hold and Wait Condition

- Request all resources before starting
 - a process never has to wait for what it needs
- Request resources only when the process has none.
 - process must give up all resources
 - then request all immediately needed
- Examples
 - If need 2 chopsticks, request both at the same time
 - How the phone company avoids deadlock 
 - Technique used in Ethernet/some multiprocessor nets
 - Everyone speaks at once. On collision, back off and retry
- Problems
 - may not know required resources at start of run
 - also ties up resources other processes could be using
 - starvation
 - Phone company example: end-to-end connection attempt; if you get all circuits along a path but eventually reach a circuit that you need to reach your destination that is in use, then you have to wait (lets say for a minute), but after a certain amount of time, you will have to release all acquired resources along the route to your destination and try again to get all circuits along the path necessary to reach destination: basically, you will get a "line busy" error signal and then the call will end (release all resources).
 - Problems: could cause resource underutilization, starvation, etc..
- Deadlock Prevention: Attacking the No Preemption Condition

Attacking the No Preemption Condition

- Applied to resources whose state can be easily saved and restored later
 - Ex: CPU registers and memory space
- This is not a viable option for most resources
- Consider a process given the printer
 - halfway through its job
 - now forcibly take away printer
 - !?!



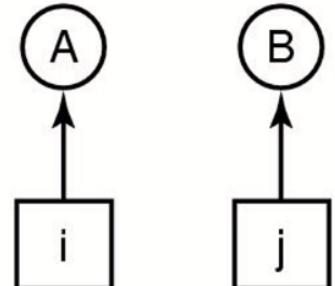
-
- Deadlock Prevention: Attacking the Circular Wait Condition
 - Essentially, use set theory: give resources a number reference, and then make sure processes request resources in a certain order, and that the set of ordered resources requested by each process will not cause circular waiting.

Attacking the Circular Wait Condition (1)

- ✓ 1. Imagesetter
- ✗ 2. Scanner
- 3. Plotter
- ✗ ✓ 4. Tape drive
- ✗ 5. CD Rom drive

(a)

1, 4
2, 4, 5



(b)

- Normally ordered resources
- A resource graph

- ○ Deadlock due to not ordering resource requests properly example:

Deadlock Example

```

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

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

```

- - Notice the order by which each thread/program (do work one and do work two) requests and acquires a resource (mutex lock in this example); the bad ordering allows for the potential of circular waiting deadlock; thus change the order (if plausible in a given program depending on its application/functionality) to avoid deadlock possibility (or even to reduce it).
 - Thus: make each thread request the first and second mutex in the same order instead of opposite order.
- Another Deadlock example with Lock Ordering:

Deadlock Example with Lock Ordering

```

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

```

From A → B
From B → A

Transactions 1 and 2 execute concurrently. Transaction 1 transfers \$25 from account A to account B, and Transaction 2 transfers \$50 from account B to account A

- - Transferring money from one account to another; so the transaction thread needs to acquire a lock for both accounts in order to exclusively deduct from one account and deposit to the other.
 - If you have two transaction threads (threads 1 and 2), where thread 1 tries to send money from A to B, and thread 2 tries to send money from B to A, and both threads run concurrently, then you can see the potential for deadlock as the locks for both accounts will be requested by both threads concurrently (or simultaneously in multiprocessing) in opposite orders: thread 1 tries to acquire A lock, then B; thread 2 tries to acquire B lock, then A; thus thread 1 and thread 2 could both end up circularly waiting for their respective locks for A or B indefinitely.
 - Solution: order account number **NOT** using *from* and *to* account, but instead just acquire the lock based on the account number ordering; for example, maybe acquire locks in order from smallest account number to greatest account number; that way no matter how many threads in the system, even if they are trying to send transactions in the opposite direction, the **order** in which they try to acquire a lock for each account involved in the transaction will always be the **same**, thus preventing any deadlock from circular waiting.
- Summary of Approaches to Deadlock Prevention

Summary of approaches to deadlock prevention

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Problems:

- low resource utilization
- reduced system throughput

◦
◦

22

Lecture Video 08-2 (week7 – 2/21/22): Deadlock

- Deadlock Avoidance Approach

Deadlock Avoidance

Requires that the system has some additional *a priori* information available.

- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.
 - Conservative method: uses worst case scenario to determine MAX number of resources that may be needed; thus, this solution can cause waste of resources.
- Safe State

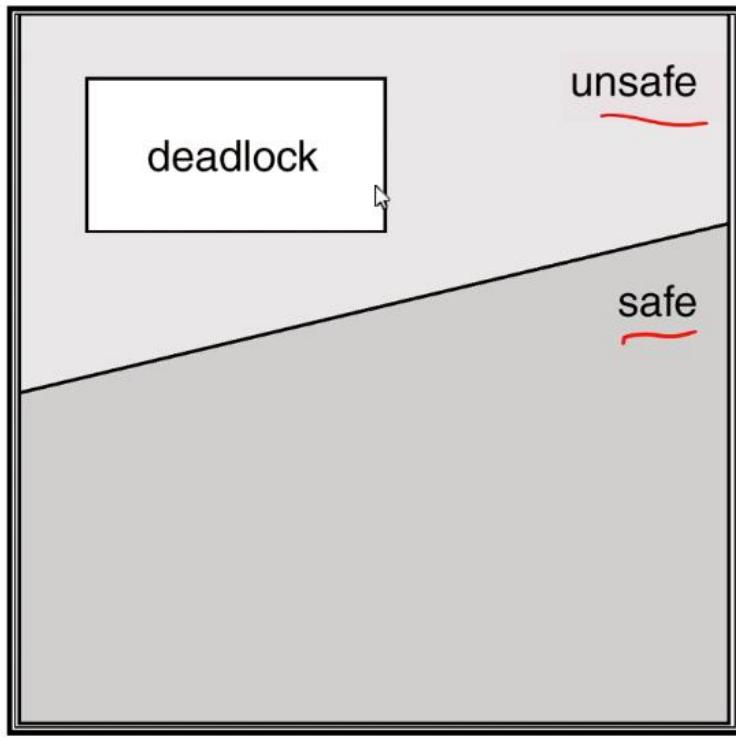
Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a *safe state*.
- System is in safe state if there exists a safe sequence of all processes.
- Sequence $\langle P_1, P_2, \dots, P_n \rangle$ is safe if for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$.
 - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
 - When P_j is finished, P_j can obtain needed resources, execute, return allocated resources, and terminate.
 - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.
- Essentially, a given process is safe (can request another resource) in a sequence of processes if for that given process there are enough resources available to request up to the max resources it may need + all resources currently held by all prior processes ($j < i$).
- This is the avoidance algorithm/method used to define “safe” v “unsafe” state.

Basic Facts

- If a system is in safe state \Rightarrow no deadlocks.
- If a system is in unsafe state \Rightarrow possibility of deadlock.
Avoid 
- Avoidance \Rightarrow ensure that a system will never enter an unsafe state.
 -

Safe, Unsafe , Deadlock State

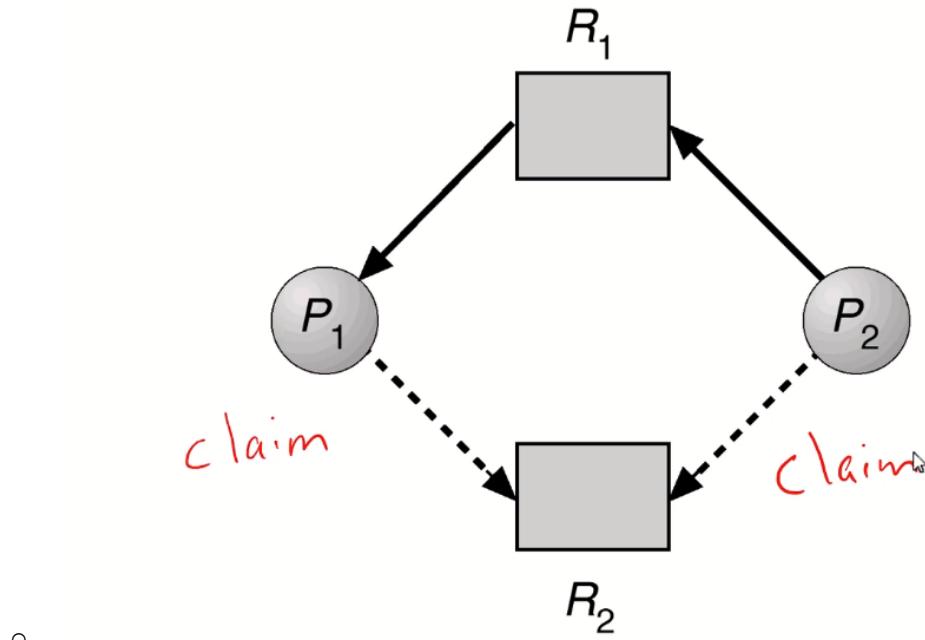


- Resource -Allocation Graph Algorithm

Resource-Allocation Graph Algorithm

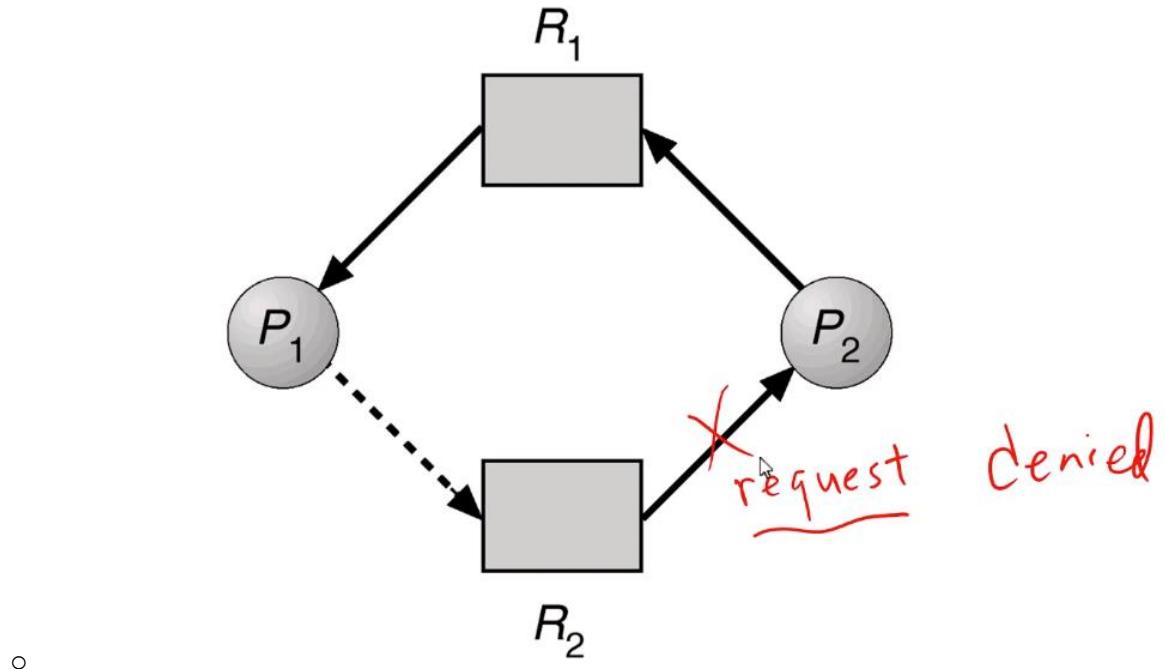
- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j ; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.
 - A Priori – “based on the past”; deduction.

Resource-Allocation Graph For Deadlock Avoidance



- Safe state; P_1 or P_2 claim will be allowed when requested.

Unsafe State In Resource-Allocation Graph



- Unsafe state would be created if P2 allowed to request R2; P2 request will be denied (to keep state safe) since as we see if it were not there would be in an unsafe state where there could be circular waiting caused between P1 and P2, if P1 ended up requesting R2 at the same time P2 held R2 and is waiting for P1 to release R1.
- Banker's Algorithm
 - Similar idea to when bankers evaluate a loan application, they always want to stay on the safe side: make sure the person applying for the loan will be able to pay it back.

Banker's Algorithm

- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.
- Example of Banker's Algorithm

Example of Banker's Algorithm

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

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

$\langle P_1, P_3, P_0, P_2, P_4 \rangle$ safe

- Example of “A Priori” (calculating NEED): Need (claim) = MAX - ALLOCATION
- Starting out at the T_0 snapshot, we know that P_1 will be able to complete since all of its needs of each resource are less than (or equal to) currently available of each resource.
 - Then when P_1 completes, available will become 5-3-2 (for A-B-C, respectively); simply take allocation of P_1 values and add that to Available.
 - Repeat process: now P_3 can definitely complete, then total available will become 7-4-3...etc...now at this point, all remaining processes will be able to definitely complete in any remaining order based on available resources; use this order to ensure that the state is always in a safe state.
 - Thus in this example: $\langle P_1, P_3 \rangle$ must be completed in that order, and the remaining processes can be completed in any order to ensure safe state at all times.

Example (Cont.)

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

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

- The system is in a safe state since the sequence $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria.
 - See in the example above, the processing order is a little different than what the professor used, since as long as the key processes P_1 and P_3 are in their proper order (just as the professor did) and come before all other processes to keep the state safe at all times; thus, for example if P_4 tries to request a certain number of resources and tries to complete before P_1 and P_3 , then it will be blocked /resource request will be denied (wait).

Example P_1 Request (1,0,2) (Cont.)

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

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0	5	3	2
P_2	3	0	1	6	0	0	7	4	3
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

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

- Previously, available was 3-3-2; thus since $1-0-2 \leq 3-3-2$, request is granted; thus new available is 2-3-0.
- Now, solve for the order again by which processes must be completed; in this case, P1, followed by P3 is still the order of the first two necessary in order to keep system safe.
- Request will be denied for P4 (using initial state from previous example where available is 3-3-2), since even though request \leq available, there will be no process where all needs \leq the new available amount if P4 was granted the requested resources.

	Allocation	Need	Avail.
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	0 0 2
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	$\rightarrow 100$

- Executing safety algorithm shows that sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement.
- Can request for (3,3,0) by P_4 be granted? Denied
- Request by P_0 will be granted (using initial state from previous example where available is 3-3-2) since request (0-2-0) \leq available (3-3-2) = new available (3-1-2) and there is an order that allows for safety: P_3, P_1, P_2, P_0, P_4 ; where the order of P_3 and P_1 in this case is essential:

Example P_1 Request (1,0,2) (Cont.)

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

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	3 1 2
P_2	3 0 1	6 0 0	5 2 3
P_3	2 1 1	0 1 1	7 2 5
P_4	0 0 2	4 3 1	$\langle P_3, P_1, P_2, P_0, P_4 \rangle$

- Executing safety algorithm shows that sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement.
- Can request for (3,3,0) by P_4 be granted? Denied
- Can request for (0,2,0) by P_0 be granted? Yes
- Data Structures for the Banker's Algorithm

Data Structures for the Banker's Algorithm

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

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

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

- Safety Algorithm
 - Determines the state: safe, or unsafe.

Safety Algorithm

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

Work = Available

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

2. Find an i such that both:

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

(b) Need_i \leq Work

If no such i exists, go to step 4

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

Finish[i] = true

go to step 2

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

- - If you reach step 4 and finish[i] for all i is NOT TRUE, then the system is NOT in a safe state; requests will be denied.
- Resource-Request Algorithm for Process P_i

Resource-Request Algorithm for Process P_i

$Request_i$ = request vector for process P_i . If

$Request_i[j] = k$ then process P_i wants k instances of resource type R_j

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

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe \Rightarrow the resources are allocated to P_i
- If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored
- So notice the Safety Algorithm and the Resource-Request Algorithm formally laid out is what we did in the examples of the Banker's Algorithm.
- Deadlock Detection

Deadlock Detection

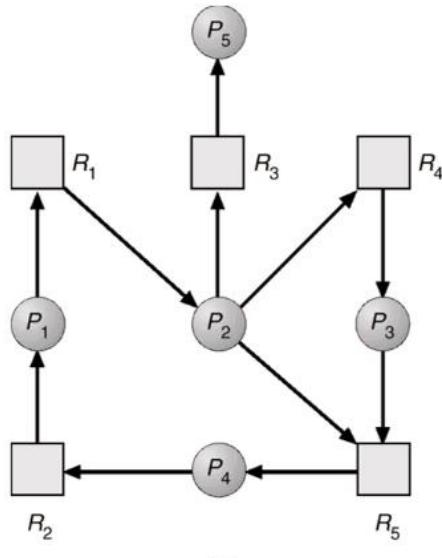


- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme
 - Single Instance of Each Resource Type

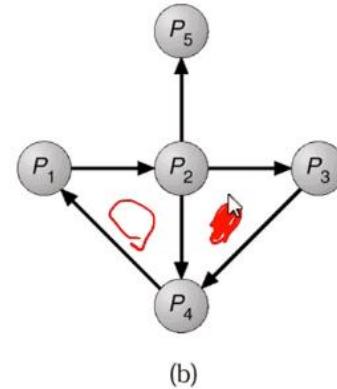
Single Instance of Each Resource Type

- Maintain *wait-for* graph
 - Nodes are processes.
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j .
- Periodically invoke an algorithm that searches for a cycle in the graph.
 $\text{O}(n^3)$
 - An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.
 - Simply need to find and recover from a circular deadlock.
 - This is not a cheap (it is resource intensive) algorithm since there may be hundreds or thousands for processes in a system.
 - Big O = N^2 ; expensive operation.
 - Resource-Allocation Graph and Wait-for Graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph

- Several Instances of a Resource Type
 - Similar to Banker's algorithm: can we find an order for all processes to complete.

Several Instances of a Resource Type

- *Available*: A vector of length m indicates the number of available resources of each type.
- *Allocation*: An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- *Request*: An $n \times m$ matrix indicates the current request of each process. If $\text{Request}[i,j] = k$, then process P_i is requesting k more instances of resource type R_j .
 - Detection Algorithm
 - (remember, this is similar to Banker's algorithm)

Detection Algorithm

1. Let $Work$ and $Finish$ be vectors of length m and n , respectively Initialize:
 - (a) $Work = Available$
 - (b) For $i = 1, 2, \dots, n$, if $Allocation_i \neq 0$, then $Finish[i] = \text{false}$; otherwise, $Finish[i] = \text{true}$.
2. Find an index i such that both:
 - (a) $Finish[i] == \text{false}$
 - (b) $Request_i \leq Work$

If no such i exists, go to step 4.

○
○

Example of Detection Algorithm

- Five processes P_0 through P_4 ; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0 :

	<u>Allocation</u>			<u>Request</u>	<u>Available</u>	
	A	B	C	A	B	C
P_0	0	1	0	<u>0</u>	<u>0</u>	<u>0</u>
P_1	2	0	0	2	0	2
P_2	3	0	3	<u>0</u>	<u>0</u>	<u>0</u>
P_3	<u>2</u>	1	1	1	0	0
P_4	0	0	2	0	0	2

- Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = \text{true}$ for all i .
 - In the example, P0 and P2 can run to completion without any problems since they have 0 total requests (0-0-0, for resources A-B-C, respectively); whenever they run to completion, we add their allocation values to the available vector, then check for the next process that can be completed if it has no requests, or if it does have requests and there is now enough resources available that can be allocated to allow a process to complete and then release its resources...etc.
 - Just like Banker's algorithm, we are trying to find an order (if there exists one) by which to all tasks can be completed; if there is an order found, thus, there is no deadlock detected; if not, then a deadlock is detected. Note the difference between this and the Banker's algorithm is that there is no "need" == possible requests of a process; there are only Request == already requested and is waiting (and thus there could be a deadlock that could be detected).

Example (Cont.)

- P_2 requests an additional instance of type C.

<u>Request</u>	<u>Aval</u>
$A \ B \ C$	$A \ B \ C$
$P_0 \ 0 \ 0 \ 0$	$0 \ 1 \ 0$
$P_1 \ 2 \ 0 \ 1$	
$P_2 \ 0 \ 0 \ 1$	
$P_3 \ 1 \ 0 \ 0$	
$P_4 \ 0 \ 0 \ 2$	

- State of system?
 - Can reclaim resources held by process P_0 , but insufficient resources to fulfill other processes; requests.
 - Deadlock exists, consisting of processes P_1, P_2, P_3 , and $\underline{P_4}$.
 - Above, no path (in the graph)/sequence (order) by which to complete all processes (only P_0 can complete); thus, a deadlock has been detected.
- Detection-Algorithm Usage

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
 - How often a deadlock is likely to occur?
 - How many processes will need to be rolled back?
 - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.
 - remember: resource intensive (N^2); so we don’t want to invoke the detection algorithm too often.
- Recovery from Deadlock: Process Termination

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
 - Priority of the process.
 - How long process has computed, and how much longer to completion.
 - Resources the process has used.
 - Resources process needs to complete.
 - How many processes will need to be terminated.
 - Is process interactive or batch?

- Recovery from Deadlock: Resource Preemption
 - Common technique in databases: two PCs have two phases of commitment; the first phase you try, the second phase is commitment; before you commit to a connection, you can always rollback.
 - Rollback can cause starvation
 - Same process may always be chosen as the victim whenever there is a conflict; solution is to keep track of rollback count so: a process that has already been rolled back many times will eventually get priority and not become the victim again in the next conflict.

Recovery from Deadlock: Resource Preemption

- Preempt resources without killing off thread
 - Take away resources from thread temporarily
 - Doesn't always fit with semantics of computation
- Rollback – return to some safe state, restart process for that state.
 - For bridge example, make one car roll backwards (may require others behind him)
 - Common technique in databases (transactions) 2PC
 - Of course, if you restart in exactly the same way, may reenter deadlock once again
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.
- Summary: most systems take the Windows and Linux approach of optimism: ignore deadlock situations assuming it rarely happens and depend on manually killing/restarting a process or rebooting a system, in order to save time and resource overhead and also make sure resources don't go underutilized.

MIDTERM REVIEW (week8 – 2/28/22)

- Know the difference between a process and a thread; and know what threads of the same process shares and does not share (and thus know what states are saved in the PCB and the TCB).
 - Threads of the same process share the same text/code, global data variables, and Heap (dynamically allocated memory)
 - They do NOT share the CPU state, or the STACK memory (used for local functions and their local data variables)
- Remember the three states of a thread:

- Running, Ready, and Waiting
 - For waiting, there could be many queues: semaphore conditional queues, etc...but all of those queues put the thread in a waiting state.
- Understand how a new process is started using FORK, JavaThread, and Pthread
- CPU scheduling: algorithms to select threads from the running queue to run.
 - For RR: make sure your context switching time is no more than 10% of your overall running time (making progress); otherwise you will spend more time context switching than running threads. Your context switching time should not be more than 10% of your set quantum value. But still, you don't want your q to be too long, then it will reduce the effectiveness and efficiency of Round Robin schema by reducing response time.

Lecture Video 09-1 (week9 – 3/7/22): Memory Management Protection

- Resource Virtualization

Resource Virtualization

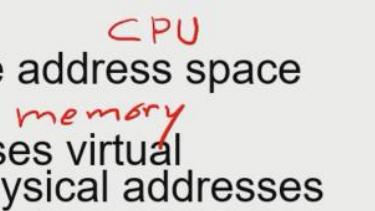
- Physical Reality: Different Processes/Threads share the same hardware
 - Need to multiplex CPU (Just finished: scheduling)
 - Need to multiplex use of Memory (Today)
 - Need to multiplex disk and devices (later in term)
- Why worry about memory sharing?
 - The complete working state of a process and/or kernel is defined by its data in memory (and registers)
 - ~~Consequently~~, cannot just let different processes of control use the same memory
 - Physics: two different pieces of data cannot occupy the same locations in memory
 - Probably don't want different processes to even have access to each other's memory (protection)
- Important Aspects of Memory Multiplexing
 - In most OS, the CPU only sees the virtual memory addresses/space and uses an address translation table to access the actual physical address.

Important Aspects of Memory Multiplexing

- **Controlled overlap:**

- Separate state of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
- Conversely, would like the ability to overlap when desired (for communication)

- **Translation:**

- Ability to translate accesses from one address space (virtual) to a different one (physical)

- When translation exists, processor uses virtual addresses, physical memory uses physical addresses
- Side effects:
 - Can be used to avoid overlap
 - Can be used to give uniform view of memory to programs

- **Protection:**

- Prevent access to private memory of other processes
 - Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
 - Kernel data protected from User processes
 - Processes protected from themselves

-
- Address Binding
 - When we actually map the virtual address to the physical address...
 - 3 stages: symbol → virtual address → physical address

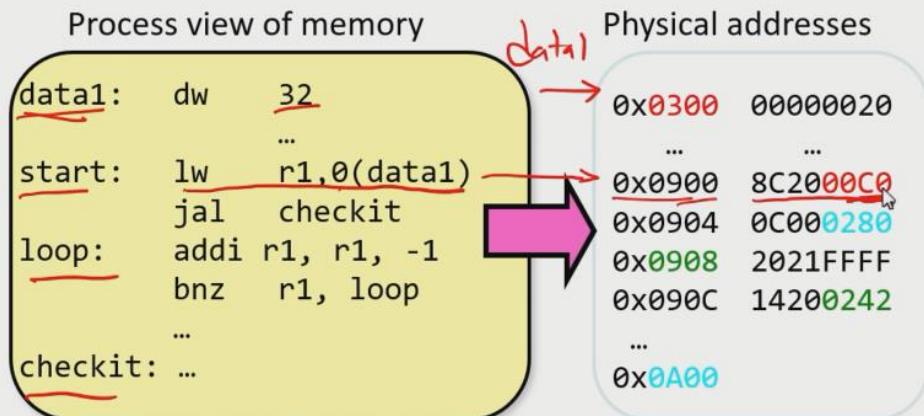
Address Binding

- Programs on disk, ready to be brought into memory to execute from an **input queue**
 - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
 - How can it not be?
- Further, addresses represented in different ways at different stages of a program's life
 - Source code addresses usually symbolic
 - Compiled code addresses **bind** to relocatable addresses
 - i.e. “14 bytes from beginning of this module”
 - Linker or loader will bind relocatable addresses to absolute addresses
 - i.e. 74014
 - Each binding maps one address space to another
- - Binding of Instructions and Data to Memory
 - Data1: = data segment
 - Start: = beginning of main function
 - Loop: = a loop
 - Checkit: = a function outside of main
 - lw refers to “load”
 - so in the example, we want to load data1, offset by 0, into register r1.

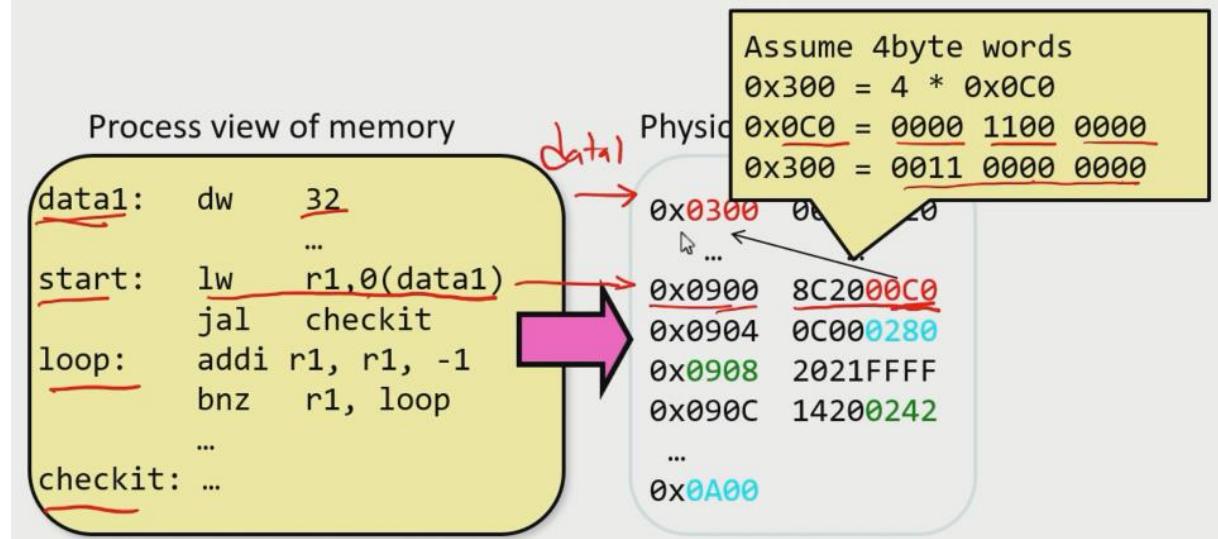
*virtual**physical*

main
func1
var1
var2

Binding of Instructions and Data to Memory

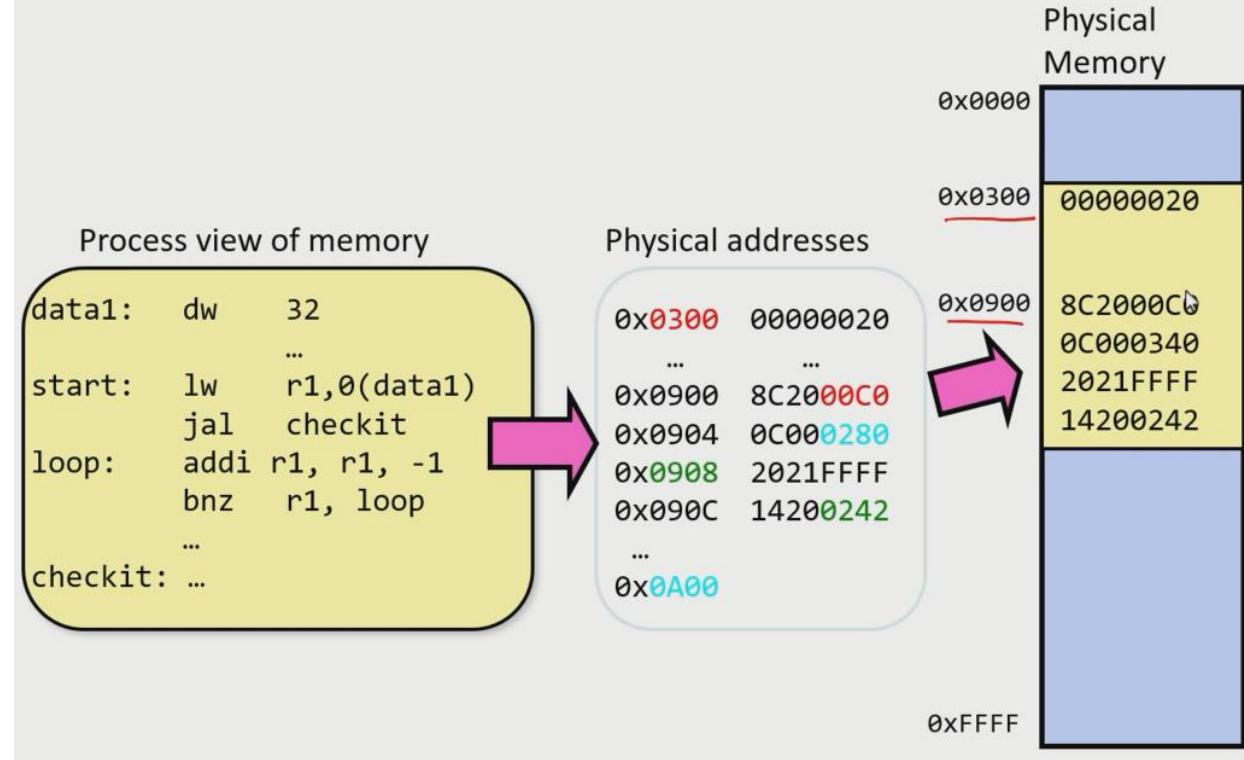


Binding of Instructions and Data to Memory



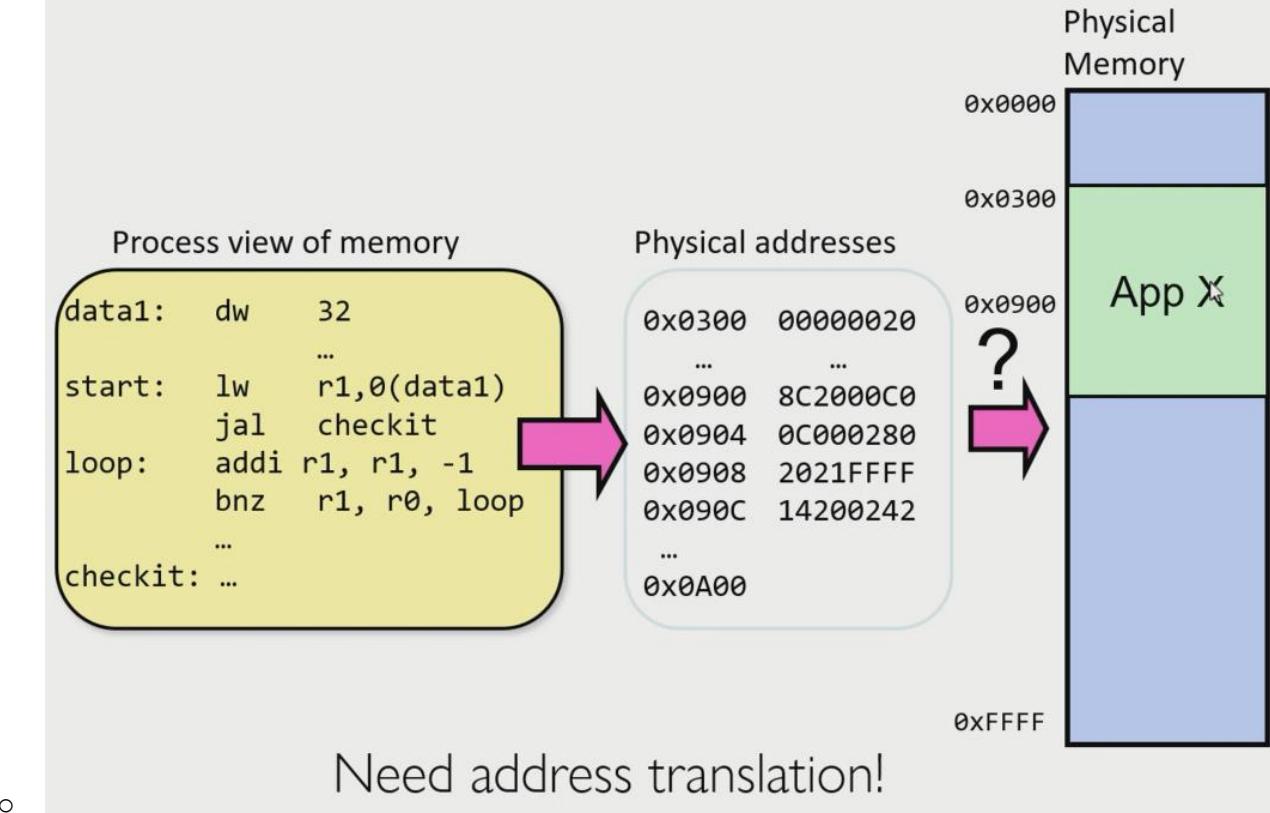
- 8C2 is the lw operator = load word
- 00C0 refers to memory location 0C0, offset 0.

Binding of Instructions and Data to Memory

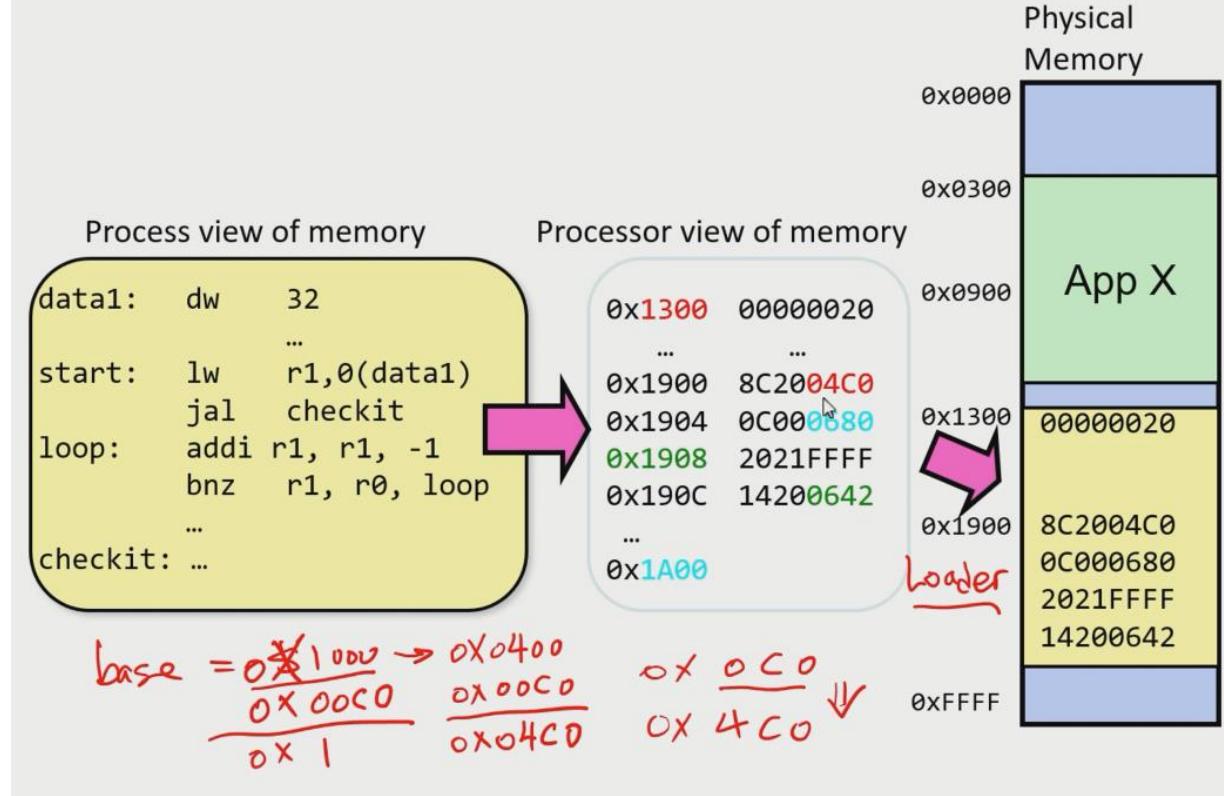


- - Above, mapping to physical memory at the given location will work as long as there is not another process running there.

Second copy of program^X from previous example

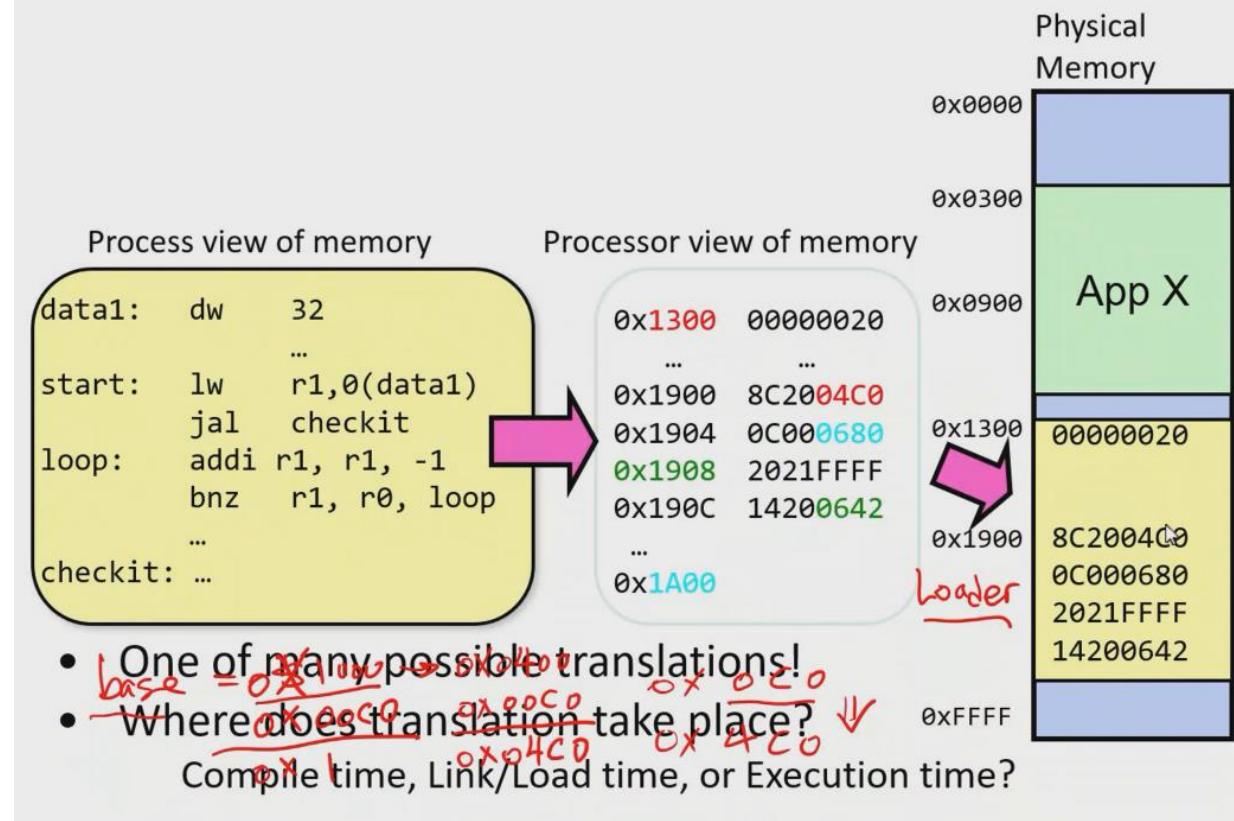


Second copy of program from previous example



- Above, the base is used to translate a process's memory to another location during runtime (dynamic); or it could be done at compile time, or link and load time; if you do it at compile time, the program (process) then is not relocatable → you must always load the same physical address (virtual address cannot be changed, thus when translated, it will always be at the same physical location).
- During runtime, add the base to all virtual addresses to get (translate to) the physical address space.

Second copy of program from previous example

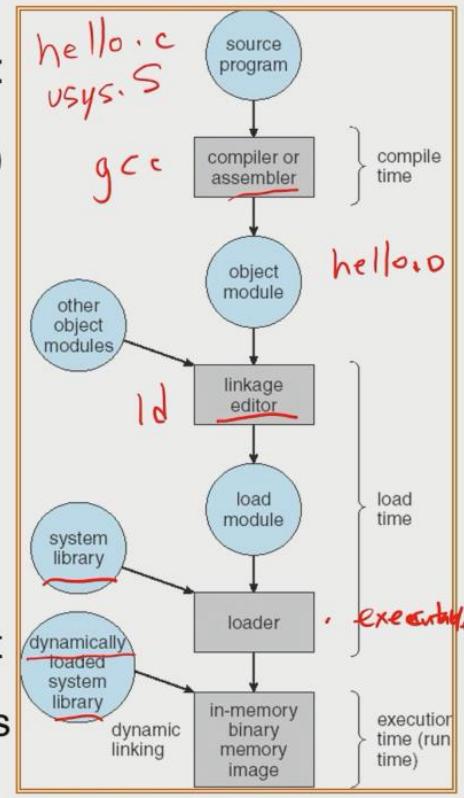


Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
 - **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
 - **Load time:** Must generate **relocatable code** if memory location is not known at compile time
 - **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another
 - Need hardware support for address maps (e.g., base and limit registers)
 - Address offset cannot be greater than limit
- Multi-step Processing of a Program for Execution
 - New innovation: dynamic linking: dynamically loaded system libraries; this allows for you to only have one copy of the library (usually read-only libraries) in the physical memory which can be shared by multiple processes at the same time, which only access these libraries during execution (dynamic link; not linked during load or compile time; only during runtime); this obviously can save a lot of memory.
 - Windows .dll files (dynamically linked library file types)

Multi-step Processing of a Program for Execution

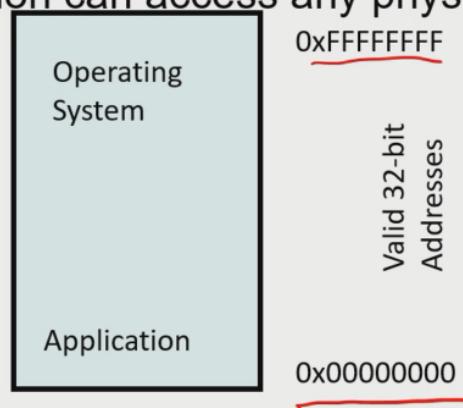
- Preparation of a program for execution involves components at:
 - Compile time (i.e., "gcc")
 - Link/Load time (UNIX "ld" does link)
 - Execution time (e.g., dynamic libs)
- Addresses can be bound to final values anywhere in this path
 - Depends on hardware support
 - Also depends on operating system
- Dynamic Libraries
 - Linking postponed until execution
 - Small piece of code, *stub*, used to locate appropriate memory-resident library routine
 - Stub replaces itself with the address of the routine, and executes routine



- Recall: Uniprogramming

Recall: Uniprogramming

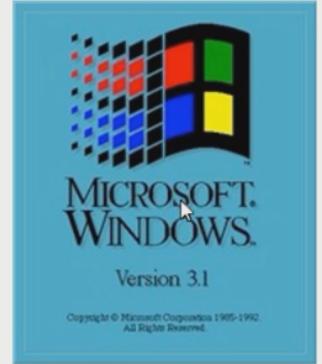
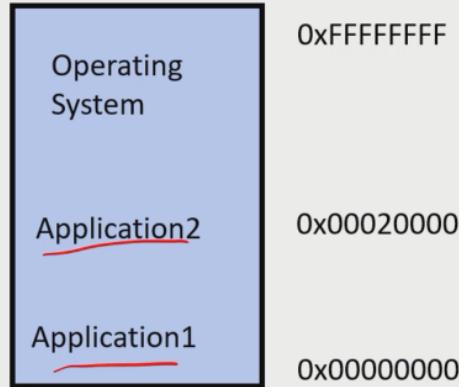
- Uniprogramming (no Translation or Protection)
 - Application always runs at same place in physical memory since only one application at a time
 - Application can access any physical address



- Application given illusion of dedicated machine by giving it reality of a dedicated machine
 - Virtual address same as physical address, and binding done at compile time.
- Multiprogramming (primitive stage)

Multiprogramming (primitive stage)

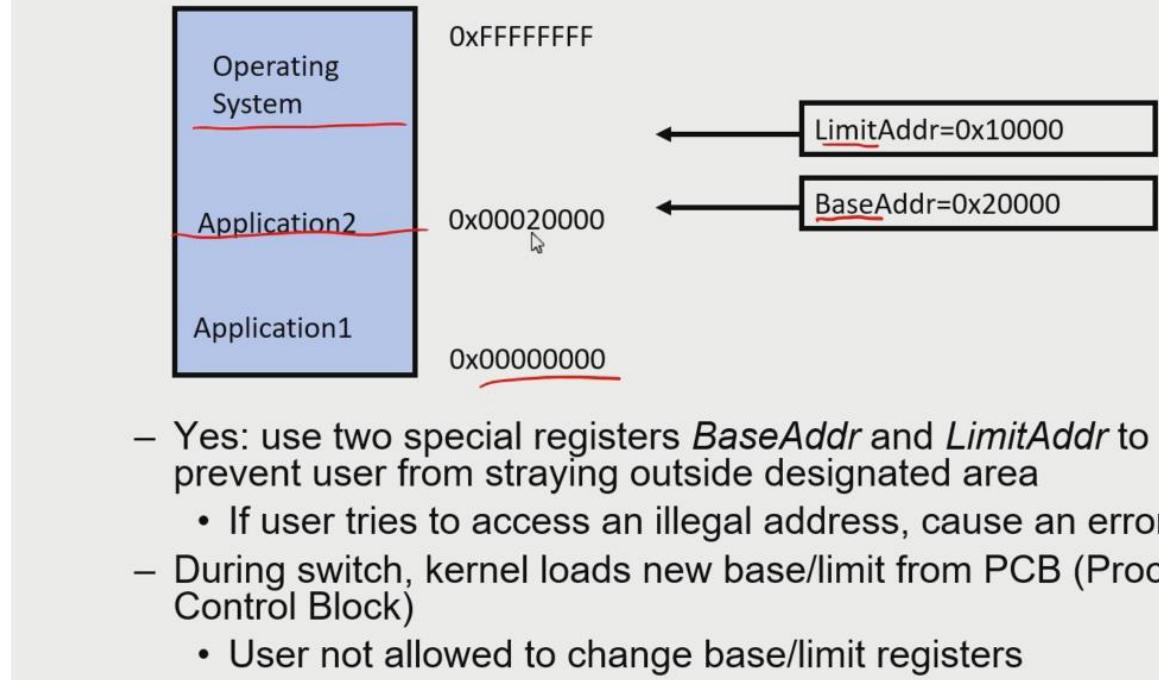
- Multiprogramming without Translation or Protection
 - Must somehow prevent address overlap between threads



- Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
 - Everything adjusted to memory location of program
 - Translation done by a linker-loader (relocation)
 - Common in early days (... till Windows 3.x, 95?)
- With this solution, no protection: bugs in any program can cause other programs to crash or even the OS
 - Multiprogramming (Version with Protection)
 - Hardware support for protection using two registers: base address and limit address registers.
 - Example: Linux segmentation fault: error often occurs when a process tries to access memory outside of its allocation range.

Multiprogramming (Version with Protection)

- Can we protect programs from each other without translation?



- Yes: use two special registers *BaseAddr* and *LimitAddr* to prevent user from straying outside designated area
 - If user tries to access an illegal address, cause an error
- During switch, kernel loads new base/limit from PCB (Process Control Block)
 - User not allowed to change base/limit registers
 - Base and limit registers must be done in kernel mode by the OS (otherwise, if user apps could change it, there would be no/less protection)
- Logical (virtual) vs. Physical Address Space

Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
 - **Logical address** – generated by the CPU; also referred to as **virtual address**
 - **Physical address** – address seen by the memory unit
 - Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
 - **Logical address space** is the set of all logical addresses generated by a program
 - **Physical address space** is the set of all physical addresses generated by a program
-
- Protection: Kernel and Address Spaces

Protection: Kernel and Address Spaces

- Goal of Protection
 - Keep user programs from crashing OS
 - Keep user programs from crashing each other

Physical Memory	Abstraction: Virtual memory
No protection	Each program isolated from all others and from the OS
Limited size	Illusion of infinite memory 2^{32} , 2^{64}
Shared visible to programs	Transparent – can't tell if memory is shared
Easy to share data between programs	Ability to share code, data

- Illusion of infinite memory; for 32 bit, virtual space is 2^{32} ; 64 bit, it is 2^{64} (but only 48 bits are used); those are nearly an “infinite” amount of address space.
- Requires Hardware Support
 - Remember CPU can be in two states with hardware support: kernel (kernel bit set during operations, in this case set is bit = 0), and user mode (kernel bit not set during operations, bit = 1); this bit is controlled by OS (kernel part of the OS).

Requires Hardware Support

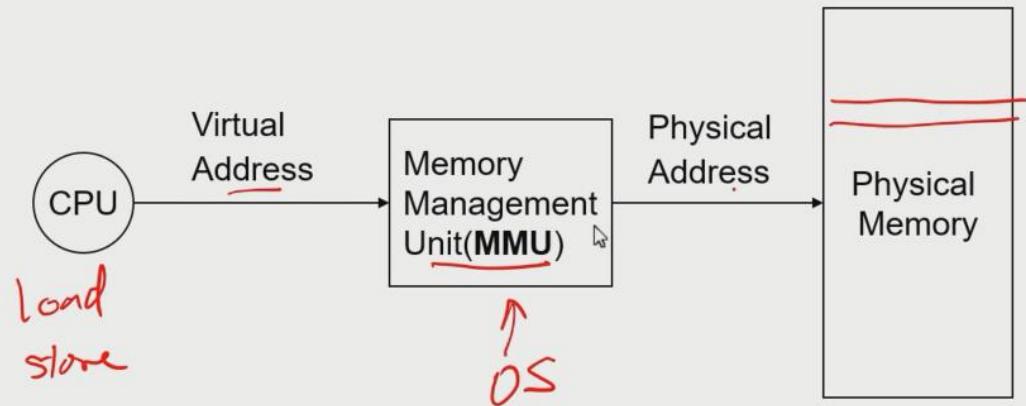
8086

- Early PCs offered none (ex: Intel 8088)
- More modern architectures do
- 2 primary types of support
 - Address translation
 - Dual mode operation: kernel vs. user mode
- Address Translation
 -

Address Translation

- Address space:
 - Consists of code, data, files
 - Literally, all the addresses a program can touch. All the state that a program can affect or be affected by.
- Restrict what a program can do by restricting what it can touch
 -
- Address Translation in Modern Architectures (use MMU = Memory Management Unit; a type of hardware support for memory management and protection)
 - MMU controlled by OS; mapping table that it stores can only be updated in the kernel mode.

Address Translation in Modern Architectures



- Hardware translates every memory reference from virtual address to physical addresses; software sets up and manages the mapping in the translation box.
 - Two Views of Memory

Two Views of Memory

- View from the CPU – what program sees, virtual memory
- View from memory – physical memory
- Translation box (MMU) converts between the two views

Lecture Video 09-2 (week9 – 3/7/22): Memory Management Protection

- How does address translation help?

How does address translation help?

- Completely encapsulate address space
 - Have only virtual addresses
 - MMU translates them to physical addresses
 - No way to know where others' address space
 - Ex: suppose first line of code of OS is at physical address 0 – MMU won't allow any translation to address 0 from user program



Think of MMU's translation as table lookup

- - Think of Domain Name Service (DNS → it translates a text address into an IP address) or Network Address Translation (NAT → translates private IP addresses into public addresses, while managing the public addresses to ensure there is no overlap/use of a public address space already used by another domain).
- How to manage MMU?

How to manage MMU?

- OS is allowed to change MMU tables
- What if user could modify them?
 - Could get to all of physical memory
 - Could crash OS, or other users
- Need to avoid this
 - Again hardware helps: dual mode operation
- Dual Mode Operation
 - In x86, 0 = kernel mode, 3 = user mode; 1 and 2 are in between privilege levels; hypervisor is usually set to level 1.

Dual Mode Operation

- Processor Status Word (PSW) has mode bit
 - A special purpose register
- If 0, kernel mode; if 1, user mode x86
 - In kernel mode, can do anything (OS has control)
 - In user mode, can only do certain things
 - Cannot modify MMU
 - Cannot turn interrupts off and on
 - Cannot read directly from disk
 - Cannot halt machine

How to share CPU between kernel and users?

- From Kernel to User

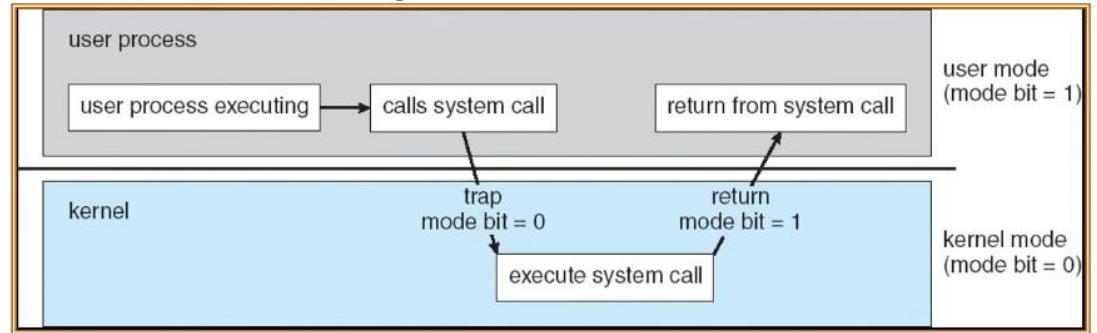
From Kernel to User

- Suppose kernel wants to run user program
- Create a new thread to:
 - Allocate and initialize address space
 - Read program off disk into memory
 - Initialize translation tables (MMU) and registers
 - Set hardware register to correct translation tables
 - There are many, need correct user's
 - Change mode bit to 1
 - Jump to first line of code
-
- From User to Kernel
 - Software "trap" == software interrupt
 - Bus error (bad address, e.g. unaligned access) means for example that an address for an integer is not a multiple of 4, thus it is a bad address; thus that is a bus error.

From User to Kernel

- Voluntary
 - System call – like doing procedure call to kernel
 - Often called a software “trap” instruction
- Involuntary
 - *Synchronous Exceptions:*
 - Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
 - Segmentation Fault (address out of range)
 - Page Fault (for illusion of infinite-sized memory)
 - Interrupts are Asynchronous Exceptions
 - Examples: timer, disk ready, network, etc....
 - Interrupts can be disabled, traps cannot!
- On system call, interrupt, or exception: hardware atomically:
 - Save state of user program (registers, PC, etc.)
 - Sets processor states to kernel and jump in
- System Calls
 - Remember during system call, you cannot execute arbitrarily any program; it is a call that specifically jumps to a OS handler program.

System Calls



- System call – special instruction to jump to a specific operating system handler.
- Can the user program call any routine in the OS?
 - No. Just specific ones the OS says is ok. Always start running handler at same place, otherwise, problems!
 - System call ID encoded into system call instruction
 - Index forces well-defined interface with kernel
 - Remember system call IDs and their respective location is stored inside of a system call vector (likely function pointers to the associated system call ID value called upon).
 - In Xv6, system call arguments can be pushed into the stack of the user program, so that the kernel can get those arguments when the user program does a system call.
 - Buffer overflow can happen if an argument is passed to the kernel that is too long; which can cause kernel memory to be overwritten, which is of course very dangerous, so argument checking is critical → you cannot assume correct system call arguments will be passed.

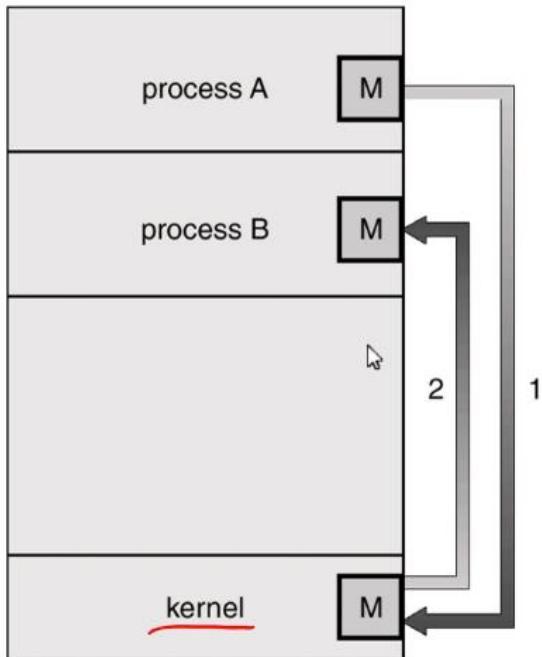
System Calls

- User needs to get back to into kernel
 - Examples:
 - Execute a new program (in new address space) *exec*
 - Wait for another program to complete
 - Do file I/O
 - Communicate with other address spaces
- How does the system call pass arguments?
 - Use registers
 - Write into user memory, kernel copies into its memory
- How does OS know that system call arguments are as expected? *buffer overflow*
 - It can't – kernel has to check all arguments.
 - Communication between address spaces
 - ps | grep apt → ps command "lists (output) all processes", but using the pipe operator, you can redirect that output (stdout == a system call) as the input (stdin == a system call) to the next function call, which in this case is the grep command (global regular expression; finds a phrase or word), and the parameter "apt" is also passed in as a parameter to the grep call , which means find the phrase "apt" in a given set of text → thus technically the pipe (| operator) operator made it so that a call to grep was this:
 - grep (stdin:(the stdout of ps), "apt") → find "apt" in the text of the output from the ps command.
 - So now you see, the pipe command takes the output from one function call and makes it the (or one of the) input(s) to a subsequent function call.
 - Message passing == networking; sending and receiving data over the network via a socket between two or more hosts, or even among the same host.
 - File system, communicate through shared file: two or more processes (or even hosts) are reading and writing to (updating) the same file; thus in doing this they can communicate by making decisions based on read or updated shared file contents. Access control is necessary so that only certain processes or hosts can access certain parts of the file and have various permissions at those parts (read or write, or both), and also so that we can avoid race conditions.

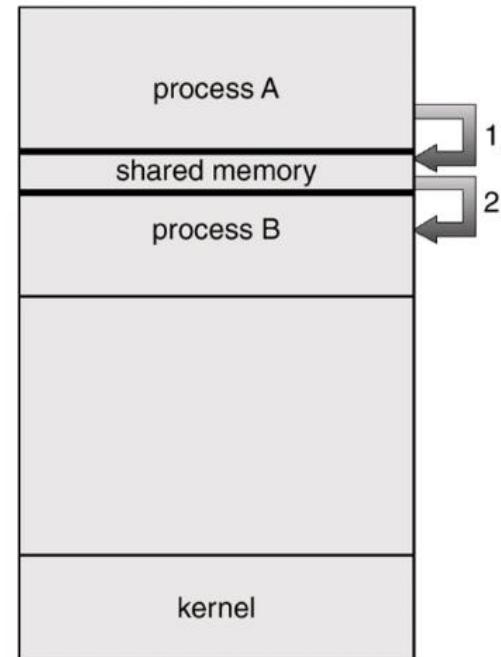
Communication between address spaces

- Share a region of memory
- Inter-process communication, communication has to go through kernel via system calls
 - Byte stream producer/consumer. Ex: communicate through pipes connecting stdin/stdout *ps | grep apt*
 - Message passing (send/receive) *socket*
 - File system (read and write files). File system is shared state!
- Communication Models

Communication Models



Message Passing



Shared Memory

- Shared memory is a faster way to communicate.
- So you can either communicate through the kernel via sending information to Kernel via a system call, which then another process can access that passed information via another system call; or, you can do a system call once that will create a shared memory segment for two or more processes (faster communication method; less overhead, but more risk for race conditions → need to handle and protect against these).
- Memory Management

Memory Management

- Modern OS's allow multiple address spaces in memory
 - Require sharing physical memory
 - Many different techniques exist
 - Depends on hardware and on OS design
- No Hardware Support

No Hardware Support

- Might still have uni- or multi- programming
- One program at a time in physical memory
 - MS-DOS
- Multiple program in memory
 - Several fixed partitions
 - Must deal with relocation
 - Address must be modified based on partition
 - Use linker-loader to place programs in memory
 - Linker identifies addresses to be modified, loader actually modifies those addresses

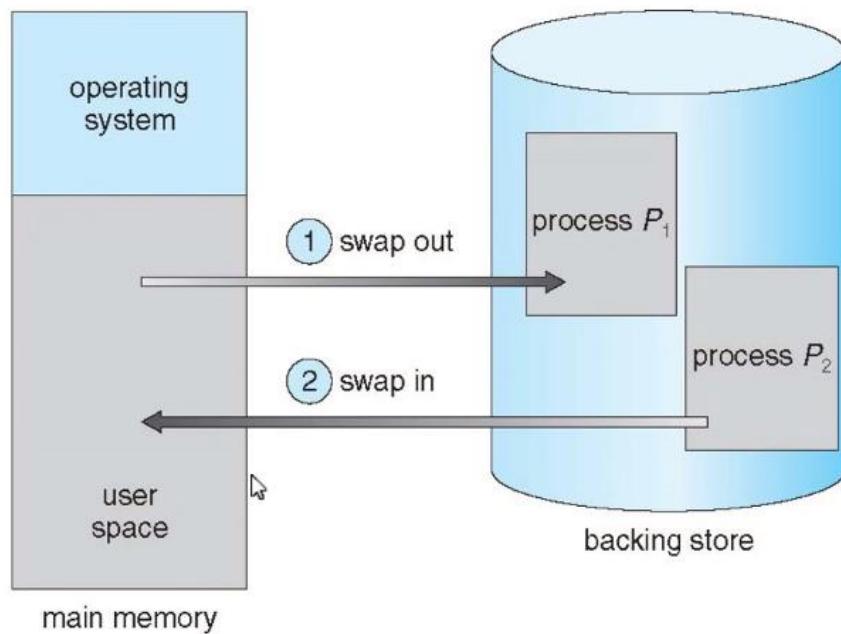
- Linker = combines all object modules into an executable.
- Loader = load program from disc to memory, does static linking.

- Swapping

Swapping

- Move entire thread + address space to disk
- Expensive
 - Not used in many systems
 - Some UNIX versions swap a process when too many in physical memory
 - MS-Windows allows “concurrent executions” via swapping

Schematic View of Swapping



- Contiguous Allocation
 - We need some kind of memory management that can support dynamic binding:

Contiguous Allocation

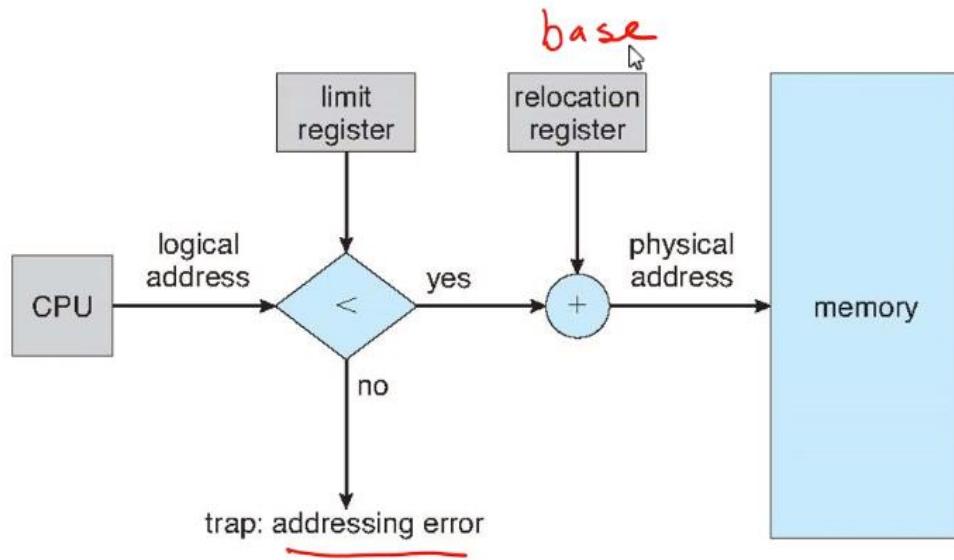
- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
 - Resident operating system, usually held in low memory with interrupt vector
 - User processes then held in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

◦

Contiguous Allocation (Cont.)

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
 - Base register contains value of smallest physical address
 - Limit register contains range of logical addresses – each logical address must be less than the limit register
 - MMU maps logical address *dynamically*
 - Can then allow actions such as kernel code being **transient** and kernel changing size
 - All virtual addresses need to add the base (relocation) register in order to translate to the physical address.
- Hardware Support for Relocation and Limit Registers

Hardware Support for Relocation and Limit Registers

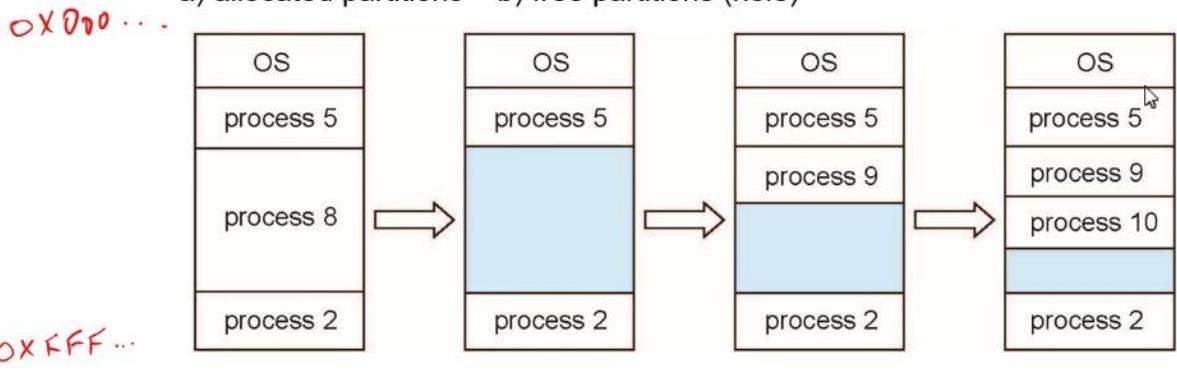


- Multiple-partition allocation
 - Notice the two partitions: OS first (starting at the first physical address; its ending is what determines the base register value), then all other processes go on the stack next into the second partition.
 - We can do multiple partitions for a given process's needs; brings efficiency.

Multiple-partition allocation

- Multiple-partition allocation

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



- ○ **cannot combine** holes if they are not *adjacent* to each other.

- Dynamic Storage-Allocation Problem

- Best fit: could end up with a lot of small holes in your memory
- Worst fit: could end up with holes that are not too small, but not big enough for larger processes.

Dynamic Storage-Allocation Problem

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

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

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

-
- Fragmentation (applies to RAM memory, and disc memory)
 - Need contagious memory in order to allocate memory resources to a process; issue when memory is fragmented – it becomes unusable.
 - (external) Fragmentation occurs when there are many holes in the memory that are not adjacent and thus cannot be combined; thus you may have lots of available memory, but with so many holes that cannot be combined, thus, the memory is fragmented and if it is fragmented enough, the memory will become unusable.
 - Think about disc defrag utility in windows; now you know why it would be useful to have such a defragmentation utility.
 - Internal fragment is simply when memory resources are overallocated to a process; thus some of that memory dedicated to that process becomes a fragment since it is unused; its really an allocated hole/fragment.

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
4KB
2.5KB
- First fit analysis reveals that given N blocks allocated, $0.5 N$ blocks lost to fragmentation
 - 1/3 may be unusable -> **50-percent rule**

○

Fragmentation (Cont.)

- Reduce external fragmentation by **compaction**
 - Shuffle memory contents to place all free memory together in one large block
 - Compaction is possible *only* if relocation is dynamic, and is done at execution time
 - I/O problem
 - Latch job in memory while it is involved in I/O
 - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

○

- Note: disc can also be fragmented just like memory.

- Contiguous Allocation: Evaluation
 - Need to make sure the space between stack and heap is big enough so they don't at some point overlap each other (throw a runtime error).

- Chrome is referring to the web browser application; i.e. how to share chrome application code for multiple processes; they all use the same chrome.exe code, but they just have different data; for contiguous allocation, it is more difficult to share the code segment and run multiple processes based on that same code and just have different data.
- Stack and heap are virtual memory types; they can grow or shrink; but in physical memory, the max size of stack and heap must be pre allocated, hence the limitation is that when that allocated space in the physical memory runs out, they can bump into each other. Solution may be then to dynamically allocate more space in the actual physical memory, or recompile code and set initial allocated heap and stack memory to a larger value and thus assign it more space in the physical memory at compile time.

Contiguous Allocation: Evaluation

- Pros:
 - Simple and fast
- Cons:
 - How to share between programs?
 - Ex: would like to share chrome code
 - How to allow address space to grow?
 - Ex: have to start stack and heap somewhere; at some point stack might “bump” into heap!
 - How to allocate memory simply?
-

Lecture Video 10-1 (week10 – 3/14/22): Address Translation

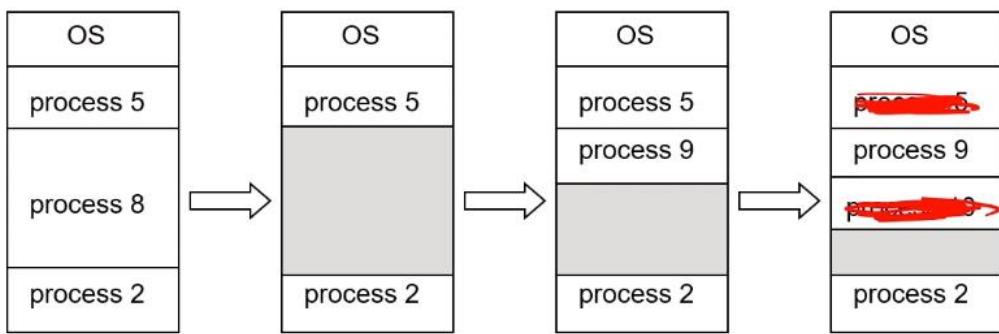
- Address Translation
 - bit maps: indicate if memory is free or allocated; one page per bit.

Address Translation

- Memory Allocation
 - Linked lists
 - Bit maps
- Options for managing memory
 - Base and Bound
 - Segmentation
 - Paging
 - Paged page tables
 - Inverted page tables
 - Segmentation with Paging
 - Memory Management with Linked Lists

Memory Management with Linked Lists

- *Hole* – block of available memory; holes of various size are scattered throughout memory.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Operating system maintains two **linked lists** for
 - a) allocated partitions
 - b) free partitions (hole)



- Dynamic Storage-Allocation Problem
 -

Dynamic Storage-Allocation Problem

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

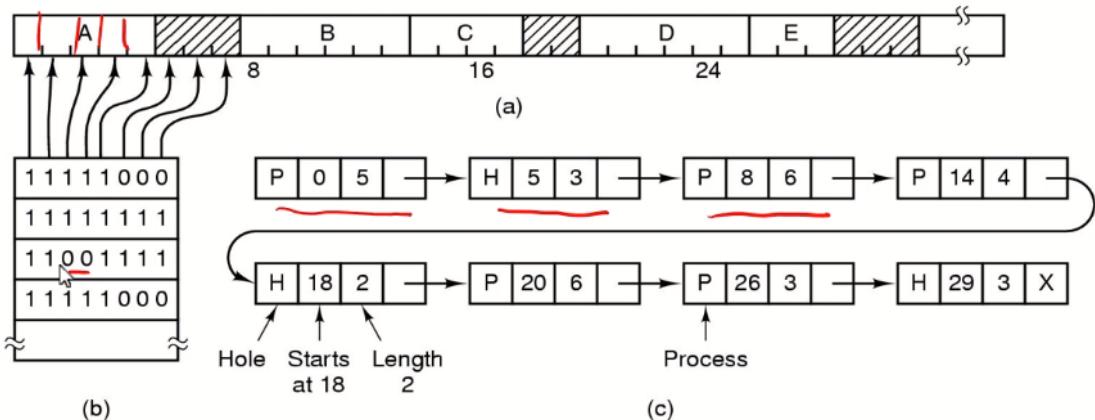
- **First-fit:** Allocate the *first* hole that is big enough.
- **Best-fit:** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit:** Allocate the *largest* hole; must also search entire list. Produces the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

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

- Memory allocation with Bitmaps

Memory Allocation with Bit Maps

Pages



- Part of memory with 5 processes, 3 holes
 - tick marks show allocation units
 - shaded regions are free
- Corresponding bit map
 - Same information as a list
- Break up physical memory into many pages
 - Typically, the size of 1 page is 4kb
 - 1 = allocated memory
 - 0 = free memory
- Memory allocation is easy (no need to do first fit or best fit, but it depends on the address translation mechanism).
- Can be represented as a linked list
- P = partition
- H = hole
- # # = start location, and size(number of pages)
- SUMMARY: Two ways to track allocated and free memory: either maintain a linked list, or use a bit map.
- Fragmentation (review)

Fragmentation

- **External Fragmentation**
 - Have enough memory, but not contiguous
 - Can't satisfy requests
 - Ex: first fit wastes 1/3 of memory on average
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
paging
- Reduce external fragmentation by **compaction**
 - Shuffle memory contents to place all free memory together in one large block.
- Segmentation
 -

Segmentation

- A **segment** is a region of logically contiguous memory.
- Idea is to generalize base and bounds, by allowing a table of base&bound pairs.
- Break virtual memory into segments
- Use contiguous physical memory per segment
- Divide each virtual address into:
 - segment number
 - segment offset
 - Note: compiler does this, not hardware

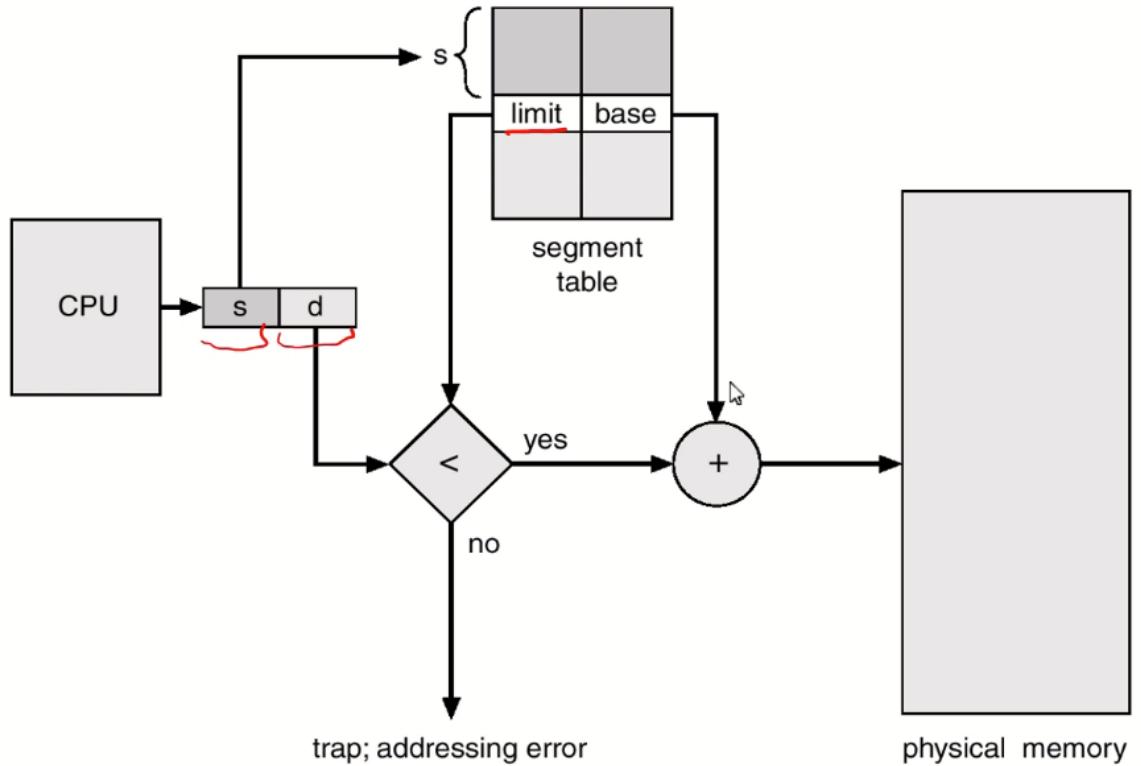
Code
data
stack
heap

- Segmentation – Address Translation

Segmentation – Address Translation

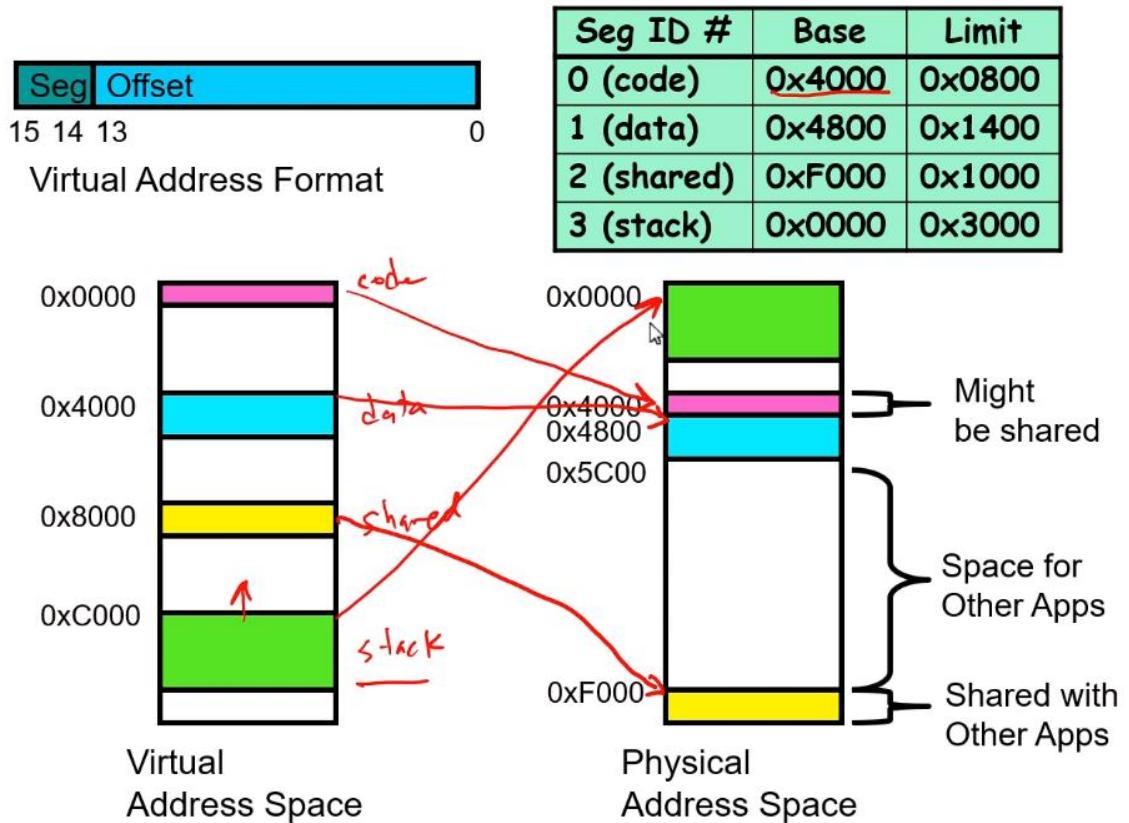
- Use segment # to index into segment table
 - segment table is table of base/limit pairs
- Compare to limit, Add base
 - exactly the same as base/bound scheme
- If greater than limit (or illegal access)
 - the segmentation fault
 -
- Segmentation Hardware
 - CPU gives virtual address (there are two parts)
 - s refers to segment table number
 - d refers to offset
 - limit = size of memory of a segment.
 - Offset (size of memory requested) has to be within the limit (otherwise segmentation fault); offset requested < limit of a segment.
 - if within limit, then add base address for the given segment (each segment has a segment number, a base number, and a limit (memory size) that correspond to the physical memory); add offset + base to get the corresponding physical memory location.
 -

Segmentation Hardware



- Example: Four Segments (16-bit addresses)
 - It's okay if the virtual address space is very sparse since it is just virtual; and we need it to be flexible to dynamic memory like heap and stack.
 - We just need to make sure that each segment (contiguous memory) (data, code, shared, stack, heap, etc.) correspond to a *contiguous* physical memory location.

Example: Four Segments (16 bit addresses)



-
- Example of Segment Translation
 - Where do we maintain the segment tables?
 - Each process only has several segments; no more than ten segments
 - The segment table can easily be maintained using the registers (different register for each segment to keep the base address).
 - We can keep entire table inside of the registers (speeds up transmission)
 - Another option: if too many segments (thus, you cannot fit table in the registers), we put segment table in main memory; but this slows down translation (since main memory slower than register memory). In x86, there are six registers for a segment (one for each type of segment, ie., stack, heap, code..etc.).
 - We only need to translate virtual addresses to physical addresses when we *fetch*. So for example, move 0x4050 (virtual address) to register \$a0, or move PC+4 into PC register (each instruction is 4 bytes); both move operations do not require address translation → the virtual address will suffice in these operations.
 - jal strlen means jump to address strlen; thus move address of strlen into PC.
 - For instruction “lb \$t0, (\$a0)” means access data stored in the a0 register (0x4050)
 - So we only need to do translations when we need to load an instruction, or store or load from main memory data sections.

VA

Example of segment translation

0x240	main:	la \$a0, varx
0x244		jal strlen
...		...
0x360	strlen:	li \$v0, 0 ;count
0x364	loop:	lb \$t0, (\$a0)
0x368		beq \$r0,\$t1, done
...		...
0x4050	varx	dw 0x314159

Seg ID #	Base	Limit
0 (code)	0x4000	0x0800
1 (data)	0x4800	0x1400
2 (shared)	0xF000	0x1000
3 (stack)	0x0000	0x3000

Let's simulate a bit of this code to see what happens (PC=0x240): 0x0240

1. Fetch 0x240. Virtual segment #? 0; Offset? 0x240
Physical address? Base=0x4000, so physical addr=0x4240
Fetch instruction at 0x4240. Get "la \$a0, varx"
Move 0x4050 → \$a0, Move PC+4→PC
2. Fetch 0x244. Translated to Physical=0x4244. Get "jal strlen"
Move 0x0248 → \$ra (return address!), Move 0x0360 → PC
3. Fetch 0x360. Translated to Physical=0x4360. Get "li \$v0,0"
Move 0x0000 → \$v0, Move PC+4→PC
4. Fetch 0x364. Translated to Physical=0x4364. Get "lb \$t0,(\$a0)"
Since \$a0 is 0x4050, try to load byte from 0x4050
Translate 0x4050. Virtual segment #? 1; Offset? 0x50
Physical address? Base=0x4800, Physical addr = 0x4850,
Load Byte from 0x4850→\$t0, Move PC+4→PC

0100
0x4050

- Segmentation (continued)
 - Core dump: crash process, and in its memory place some core files so you can analyze what happened/what issue occurred.

Segmentation (cont.)

- This should seem a bit strange: the virtual address space has **gaps** in it! Each segment gets mapped to contiguous locations in physical memory, but may be gaps between segments.
- But, a correct program will never address gaps; if it does, trap to kernel and then core dump.
- Minor exception: stack, heap can grow. In UNIX, sbrk() increase size of heap segment. For stack, just take fault, system automatically increase size of stack.
- Detail: Need protection mode in segmentation table. For example, code segment would be read-only. Data and stack segment would be read-write.
 - Segmentation Pros and Cons

Segmentation Pros & Cons

- + Efficient for sparse address spaces
 - Can keep segment table in registers
- + Easy to share whole segments (for example, code segment)
- + Easy for address space to grow
- Complicated Memory allocation: still need first fit, best fit, etc., and re-shuffling to coalesce free fragments, if no single free space is big enough for a new segment

How do we make memory allocation simple and easy?

- How to make memory allocation simple and easy? Answer: Bitmap Paging.
- Paging

- Key idea: each page is a *chunk* of memory.; each bit on the bitmap represents one page of the physical memory.
- Now segment doesn't need to be contiguous anymore; now we can map freely between the virtual and physical memory on a by-page bases instead of on a by-segment basis; as a standard unit, pages are generally relatively smaller than segments.

Paging

- Allocate physical memory in terms of fixed size chunks of memory, or pages.
- Simpler, because allows use of a bitmap. What is a bitmap?

00111100000001100

Each bits represents one page of physical memory – 1 means allocated, 0 means unallocated. Lots simpler than base & bounds or segmentation

- OS controls mapping: any page of virtual memory can go anywhere in physical memory

○

Paging (cont.)

- Avoids fitting variable sized memory units
- Break physical memory into frames
 - Determined by hardware
- Break virtual memory into pages
 - Pages and frames are the same size
- Divide each virtual address into:
 - Page number
 - Page offset
- Note:
 - With paging, hardware splits address
 - With segmentation, compiler generates segmented code.
- - Paging – Address Translation

Paging – Address Translation

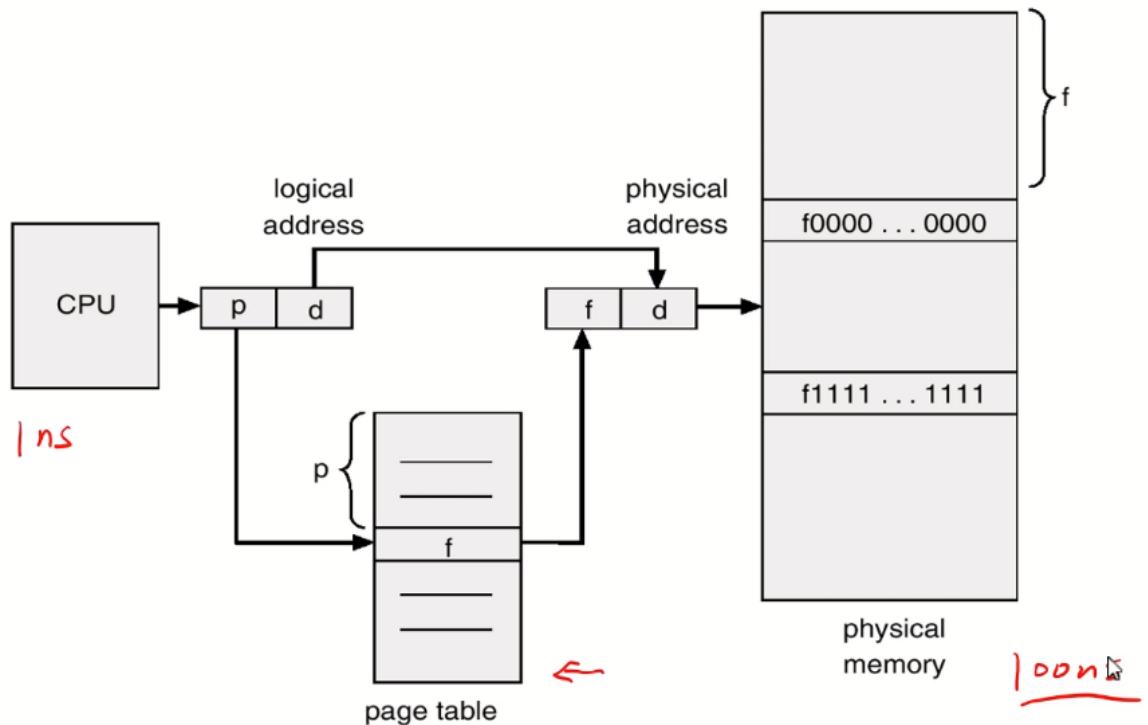
- Index into page table with high order bits
 - Get physical frame
- Append that to offset
- Now present new address to memory



Note: kernel keeps track of free frames can
be done with a bitmap

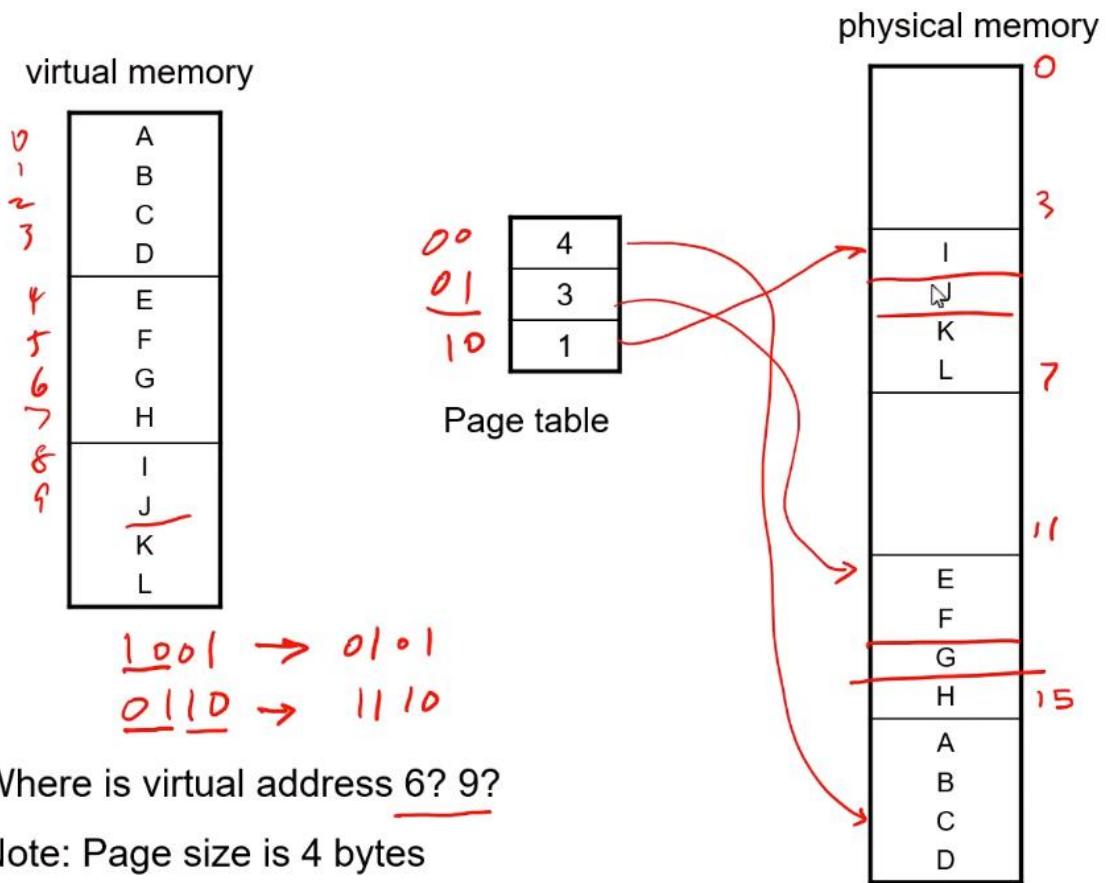
- Address Translation Architecture
 - Page table is too large to fit in registers; so it must be in main memory.

Address Translation Architecture



- Since page table is in main memory, then this address translation scheme requires too many physical memory accesses; this can reduce speed greatly.
 - So 200 nanoseconds versus using CPU register memory like in segmentation translation.
- Paging Example

Paging Example



- Notice: for example, the virtual address decimal 6 == 0110 binary; so now the first two bits represent the location in main memory to check the page table; now the value stored at the corresponding page table memory location (page 3 for this example) is the page (block of memory) that the given virtual address maps to; now take that value of 3 == 11, and replace the high order bits (01) with what was found at location 01 of the page table (11); thus physical memory location is 1110 == decimal 14; thus in summary: virtual location 0110 (decimal 6) maps to physical location 1110 (decimal 14).
- Paging (Continued)
 - For paging, there is no external fragmentation since it does not require your memory address to be contiguous; can go anywhere in the physical memory; but you do have internal fragmentations.
 - Internal fragmentation: memory allocated to a process, but that goes unused.
 - Worst case: your process only uses 1 byte out of the entire frame.
 - Average case: approximately half of a frame goes unused (internally fragmented).

Paging (Cont.)

- Calculating internal fragmentation
 - Page size = 2,048 bytes *2KB*
 - Process size = 72,766 bytes
 - 35 pages + 1,086 bytes
 - Internal fragmentation of 2,048 - 1,086 = 962 bytes
 - Worst case fragmentation = 1 frame – 1 byte
 - On average fragmentation = 1 / 2 frame size
 - So small frame sizes desirable?
 - But each page table entry takes memory to track
 - Page sizes growing over time *4KB, 2MB, 1GB*
 - Solaris supports two page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
- Process can only access its own memory
 - Process can only access its own memory: provides protections.
- Shared Pages
 - Sharing read-write pages is prone to race conditions that must be addressed, but it is much more efficient form of **inter-process** communications than message passing.
Remember a key difference: threads of the same process share the same address space; but between processes or between threads from different processes, we need a way to share data and communicate (if desired) since for the most part, we keep inter-process communication limited for the sake of protections.

Shared Pages

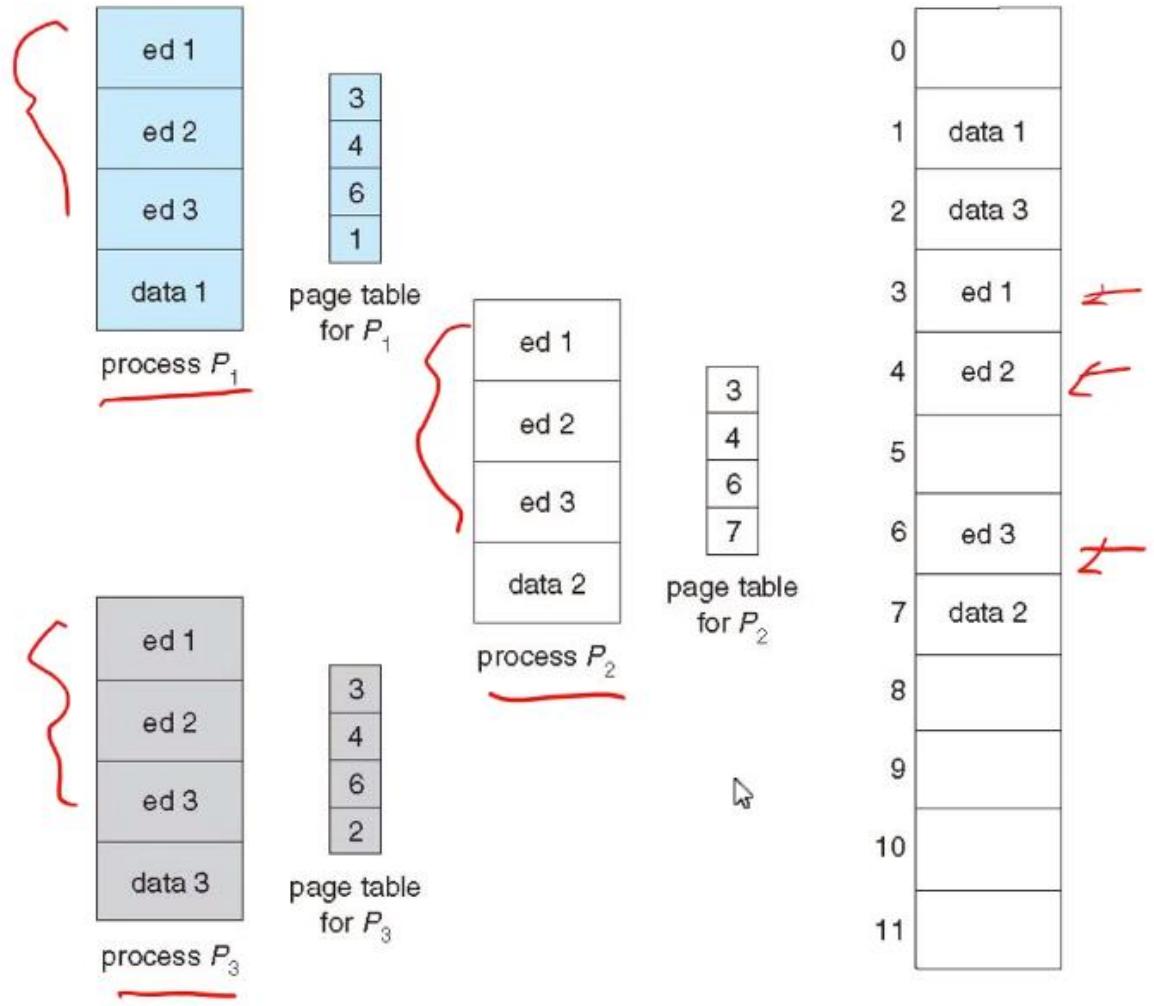
- **Shared code**

- One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
- Similar to multiple threads sharing the same process space
- Also useful for inter-process communication if sharing of read-write pages is allowed

- **Private code and data**

- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space
 - Shared Pages Example
 - Notice first three entries in the page table are the same except for the last entry (map to the data segment).

Shared Pages Example



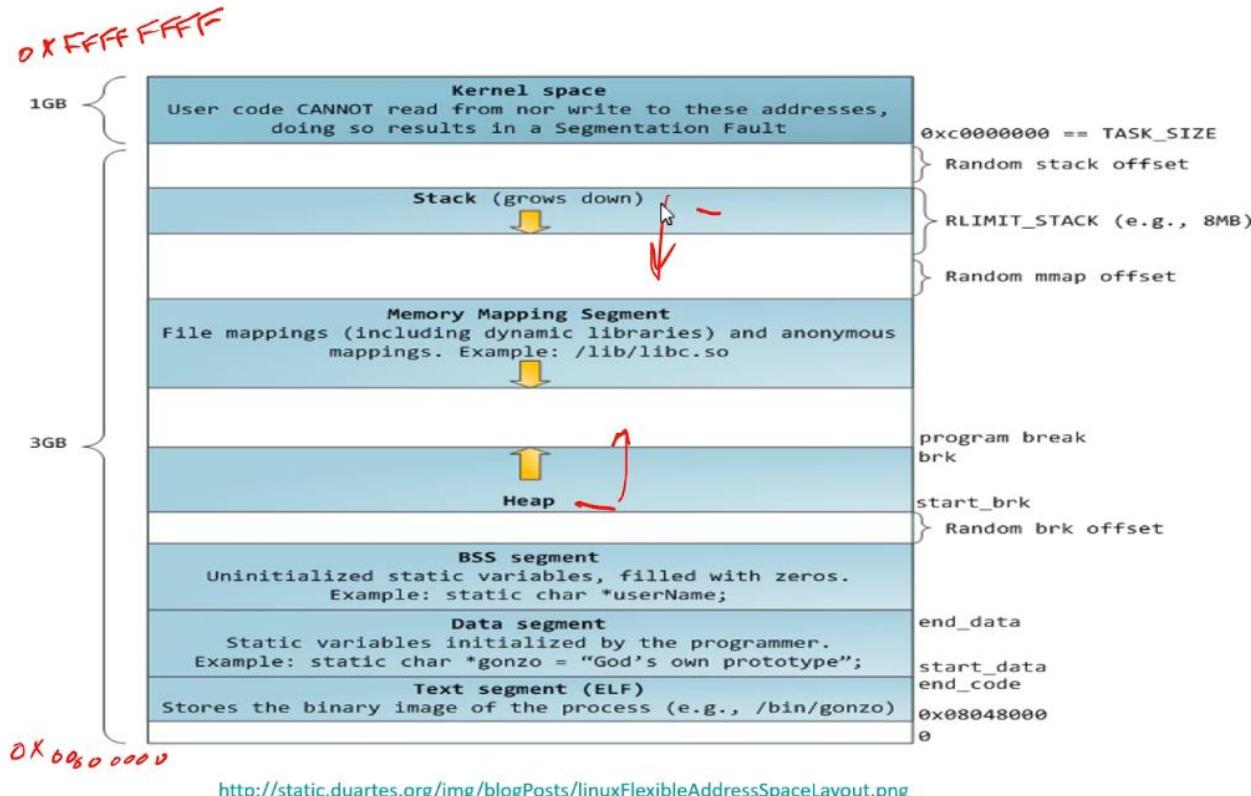
- Where is page sharing used?
 - U->K = user mode to kernel mode
 - What does kernel need to do to access other user processes address space?
 - Switch to a different page table.
 - We have the same OS/kernel for all processes, thus the memory location where the OS is running can be shared among all processes; of course that memory can only be accessed in kernel mode.
 - For different processes running same binary, i.e., multiple chrome windows open, you can share the same binary (code segment).
 - “this is a limited form of sharing that threads have within a single process”, translates more precisely to “this is a limited form of sharing, **similar to** the sharing that threads have within a single process”.

Where is page sharing used ?

- The “kernel region” of every process has the same page table entries
 - The process cannot access it at user level
 - But on U->K switch, kernel code can access it AS WELL AS the region for THIS user
 - What does the kernel need to do to access other user processes?
- Different processes running same binary!
 - Execute-only, but do not need to duplicate code segments
- User-level system libraries (execute only)
- Shared-memory segments between different processes
 - Can actually share objects directly between processes
 - Must map page into same place in address space!
 - This is a limited form of the sharing that threads have within a single process
 -
- Memory Layout for Linux 32-bit (Pre-Meltdown patch!)
 - Text segment == code segment.
 - Data segment: needs to be saved in the executable and loaded into memory when the process starts for the given program.
 - File mappings: speed up file access; you could map a file into the address space for easy read and write access to files.

4.6.3

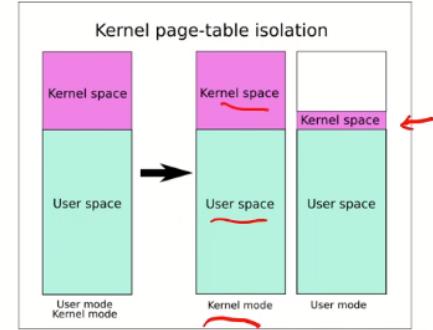
Memory Layout for Linux 32-bit (Pre-Meltdown patch!)



- Some simple security measures

Some simple security measures

- Address Space Randomization
 - Position-Independent Code \Rightarrow can place user code anywhere in address space
 - Random start address makes much harder for attacker to cause jump to code that it seeks to take over
 - Stack & Heap can start anywhere, so randomize placement
- Kernel address space isolation
 - Don't map whole kernel space into each process, switch to kernel page table
 - Meltdown \Rightarrow map none of kernel into user mode!
 - There was a big meltdown event even with limited kernel access in user mode mapping; somehow hackers were still able to access the kernel without a fault occurring and thus take over a system; in modern day, we map none of kernel into user mode.



Lecture Video 10-2 (week10 – 3/14/22): Address Translation

- Paging Issues

Paging Issues

- Fragmentation
- Page Size
- Protection and Sharing
- Page Table Size
- What if page table too big for main memory?
 -
- Fragmentation and Page Size

- Larger page size: smaller page table size; better I/O because reduce number of I/O needed to load in a page and store a page.

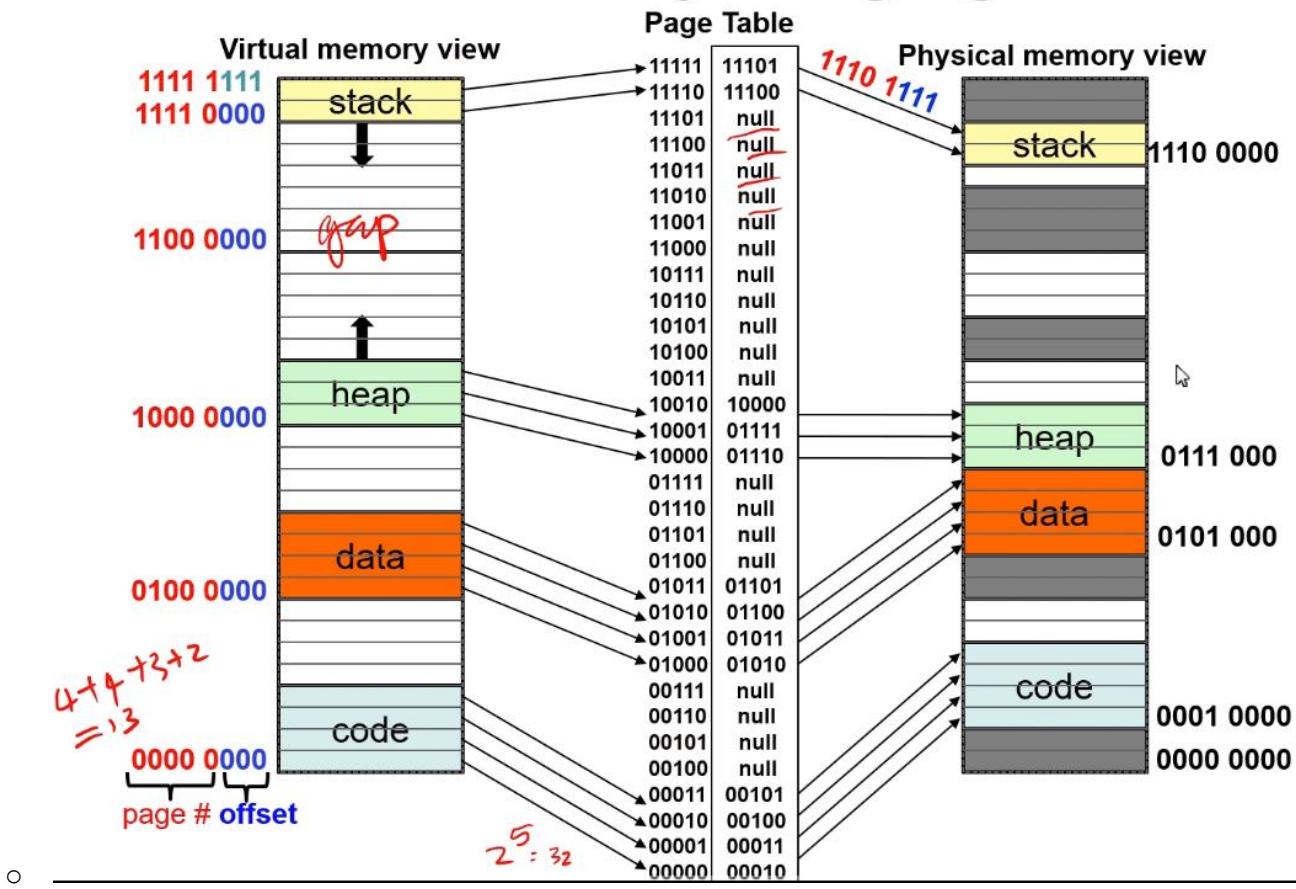
Fragmentation and Page Size

- Fragmentation
 - No external fragmentation
 - Page can go anywhere in main memory
 - Internal fragmentation
 - Average: $\frac{1}{2}$ page per address space
- Page Size
 - Small size?
 - Reduces (on average) internal fragmentation
 - Large size?
 - Better for page table size
 - Better for disk I/O
 - Better for number of page faults (Demand Paging)
 - Typical page sizes: 4K, 8K, 16K
 - Typical page size is around 4K
- Protection and Sharing

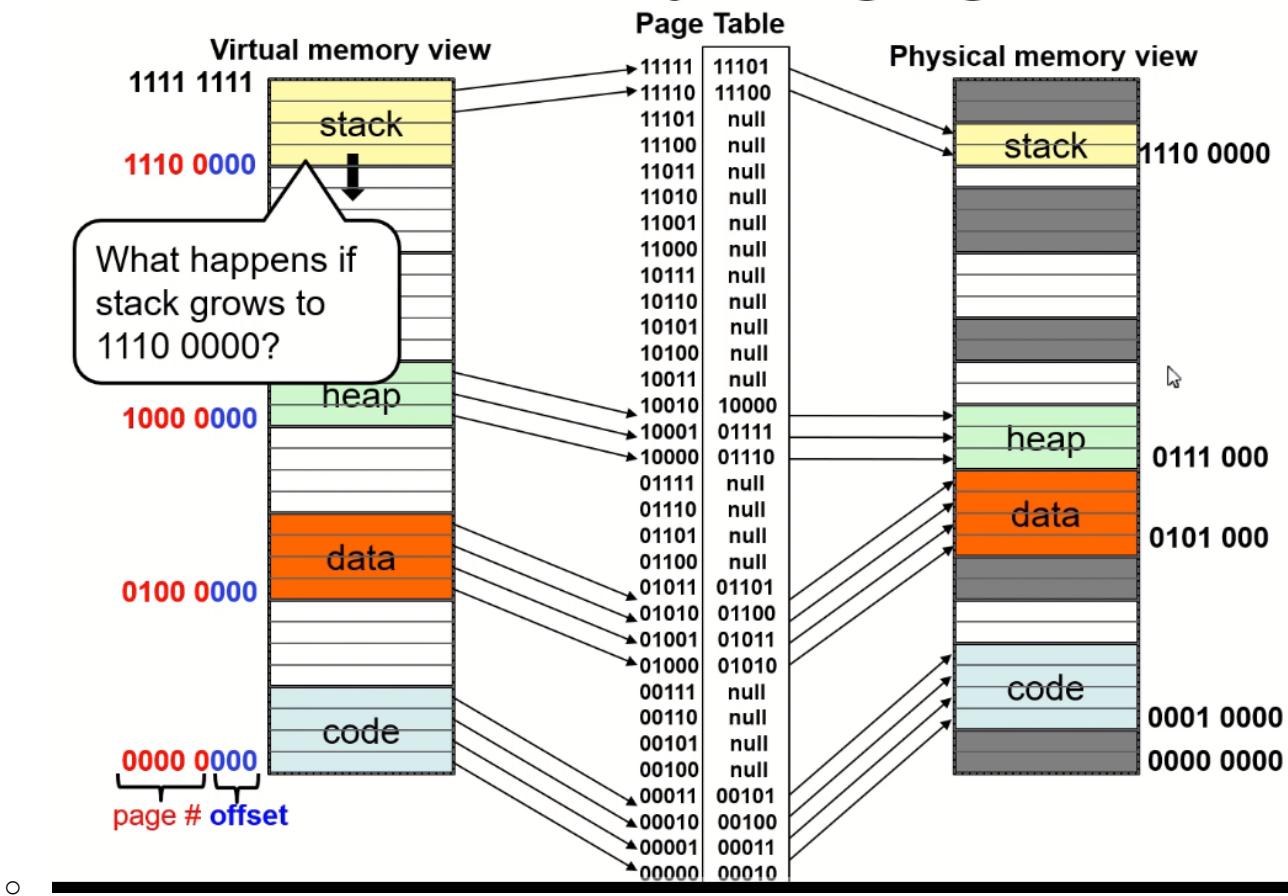
Protection and Sharing

- Page protection – use a bit
 - code: read only
 - data: read/write
- Sharing
 - Just “map pages in” to your address space
 - Ex: if “vi” is at frames 0 through 10, all processes can adjust their page tables to “point to” those frames.
 - Code (text section/program code) much easier to share than data section.
- Summary: Paging
 - For this example, address space is 8 bits (1 byte): 00000000 to 11111111
 - Offset is 3 bits.
 - Page number is 5 bits; $2^5 = 32$ entries in the page table
 - Stack is using 2 pages, heap is using 3, data is using 4, and code is using 4 = 13 pages out of 32 page capacity in total is being used for this process.

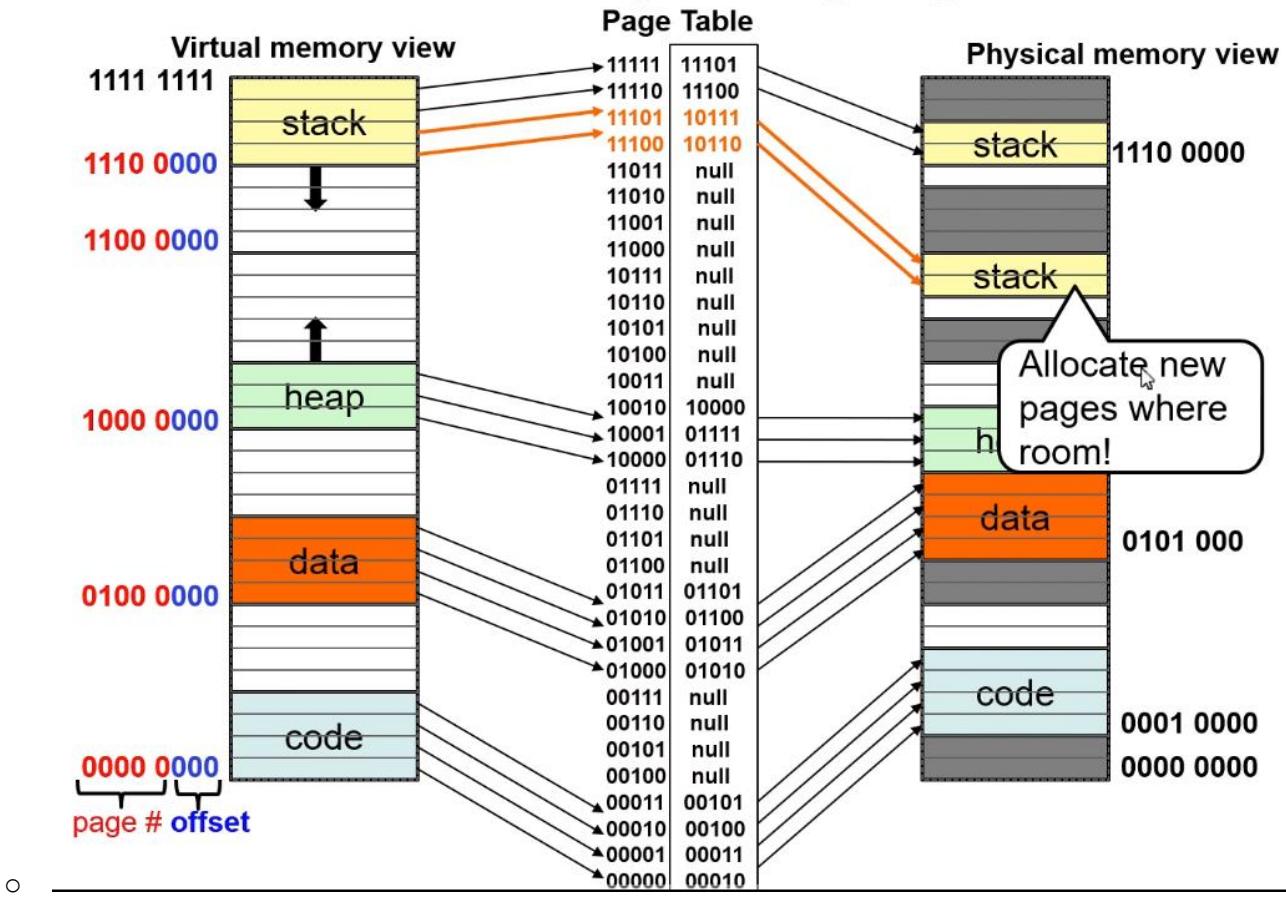
Summary: Paging



Summary: Paging

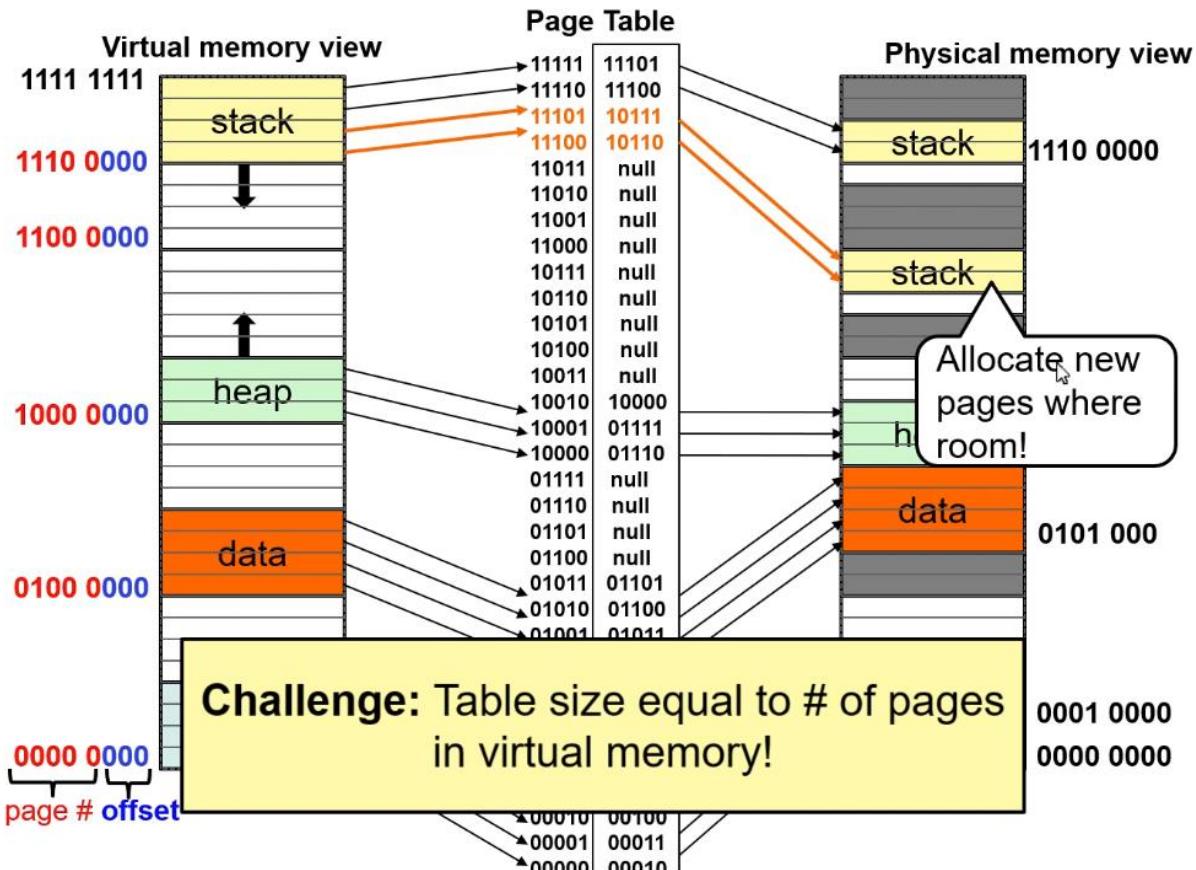


Summary: Paging



- Virtual memory stack and heap are made sparse so they can have room to grow dynamically.

Summary: Paging

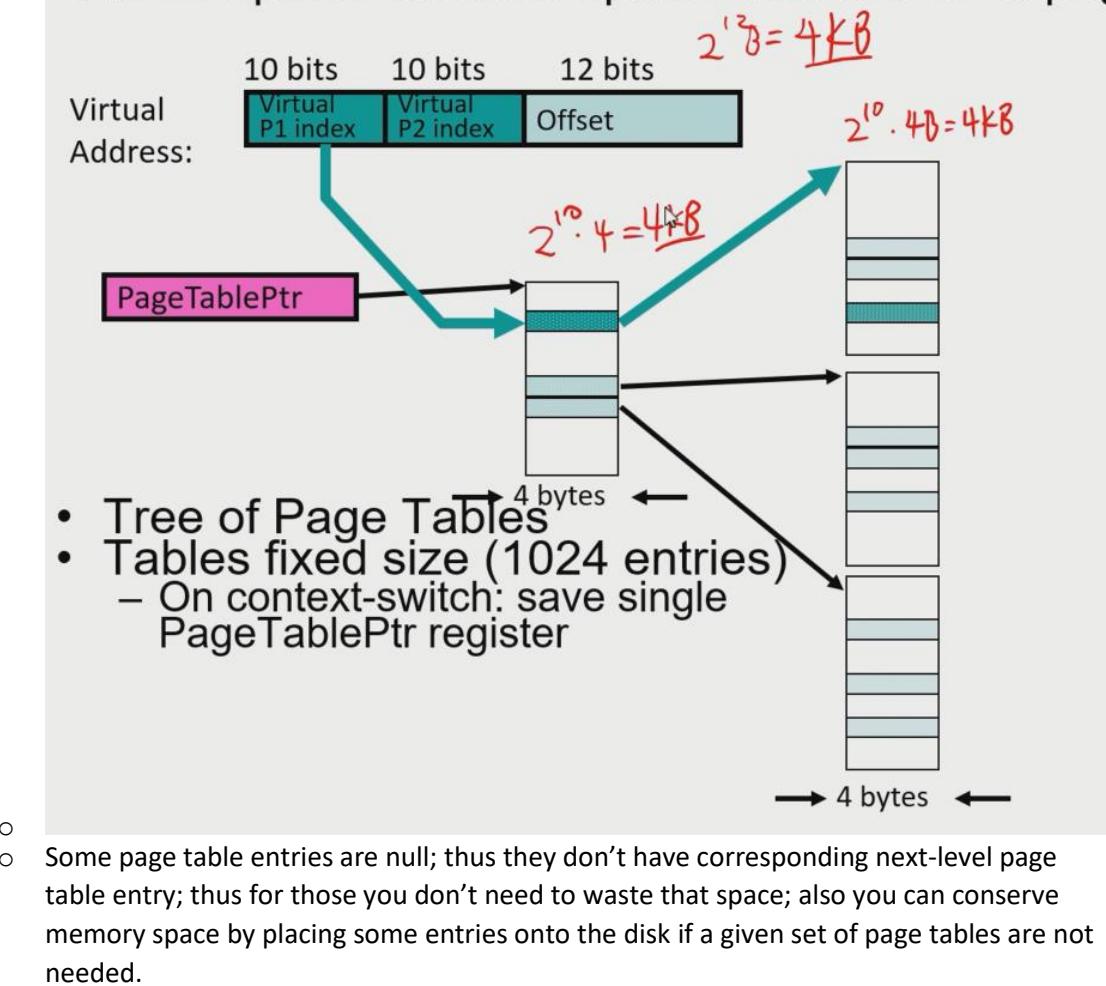


- Problem: there are many null address (unused address spaces even though they are allocated to the process).
- How can we minimize the null addresses in the page table? We don't need to keep them in the page table.
- Page Table Discussion
 - Page table pointer and limit (upper limit – largest page number; usually a constant that you don't need to save) are both stored in a register.
 - Eg: 0 to $2^{31}-1$ means your process spans 2^{31} addresses; with 1K frames == 1Kb frames (page size), == size of 2^{10} , you would need $2^{31}/2^{10} == 2^{21}$ pages == more than 2million page table entries (2.097152 million to be precise) to map the virtual memory to the physical memory.
 - Most processes actually need a few hundred page table entries or less; so 2million is a large waste (large internal fragment).

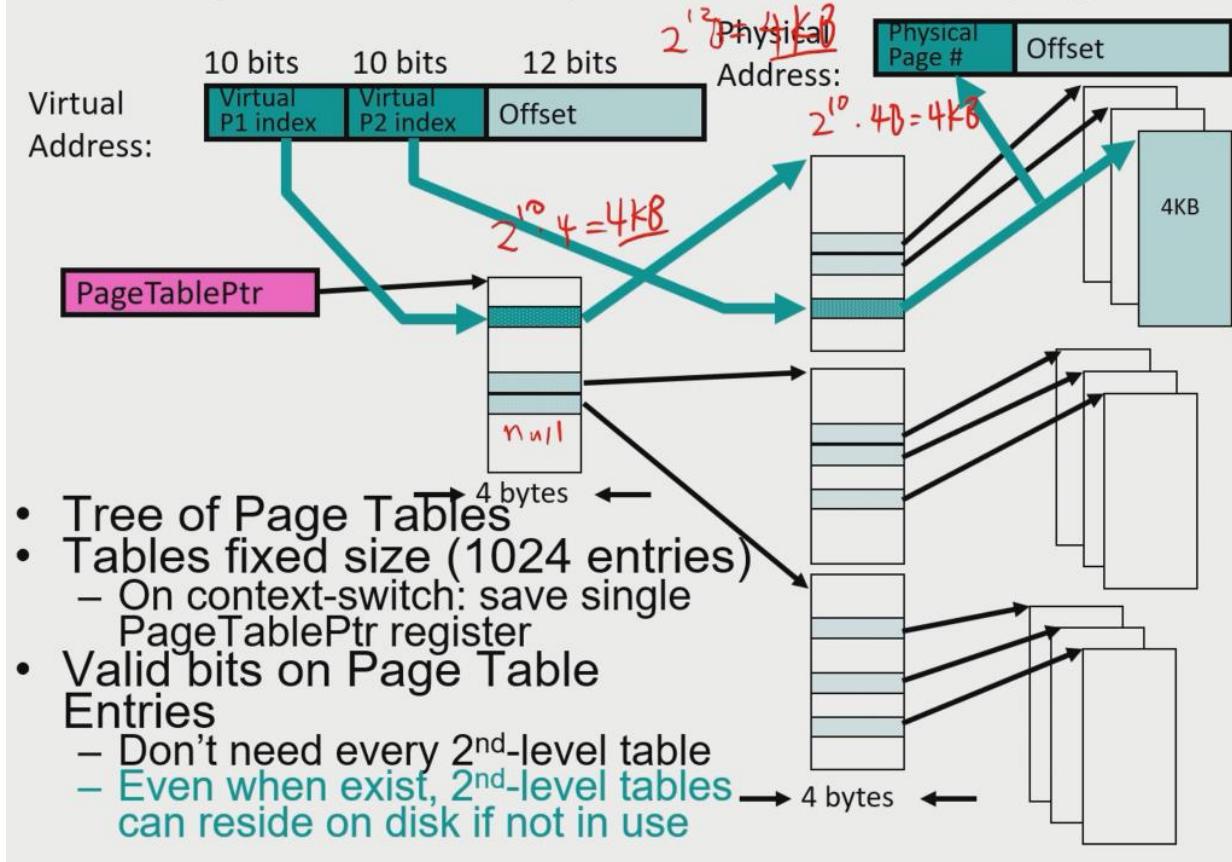
Page Table Discussion

- What needs to be switched on a context switch?
 - Page table pointer and limit
- Analysis
 - Pros
 - Simple memory allocation
 - Easy to share
 - Con: What if address space is sparse? $2^{21} \cdot 2^{10}$
 - E.g., on UNIX, code starts at 0, stack starts at $(2^{31}-1)$
 - With 1K pages, need 2 million page table entries!
 - Con: What if table really big?
 - Not all pages used all the time \Rightarrow would be nice to have working set of page table in memory
- How about multi-level paging or combining paging and segmentation?
 - Fix for (excessively) sparse address space (32-bit architecture): The two-level page table
 - Easy to manage: each table entry fits nicely inside a single page in memory.
 - Each page/frame is 4KB; so it works out nicely that both page table levels require 4KB of space == exactly one frame/page.
 - For each process, we need a register to save the address of its page table.
 - 3 steps: find index in first 10 bit table, then index the second 10 bit table, then you find the corresponding physical page and load it; from offset you find exactly which byte you want to read.
 - But notice each added level of page table adds overhead since each level means there is an additional memory access needed to reach the final access in the desired physical memory location.
 - For example; 2 levels means 3 memory accesses needed; 4 page table levels means you need 5 total memory accesses, etc.

Fix for sparse address space: The two-level page table

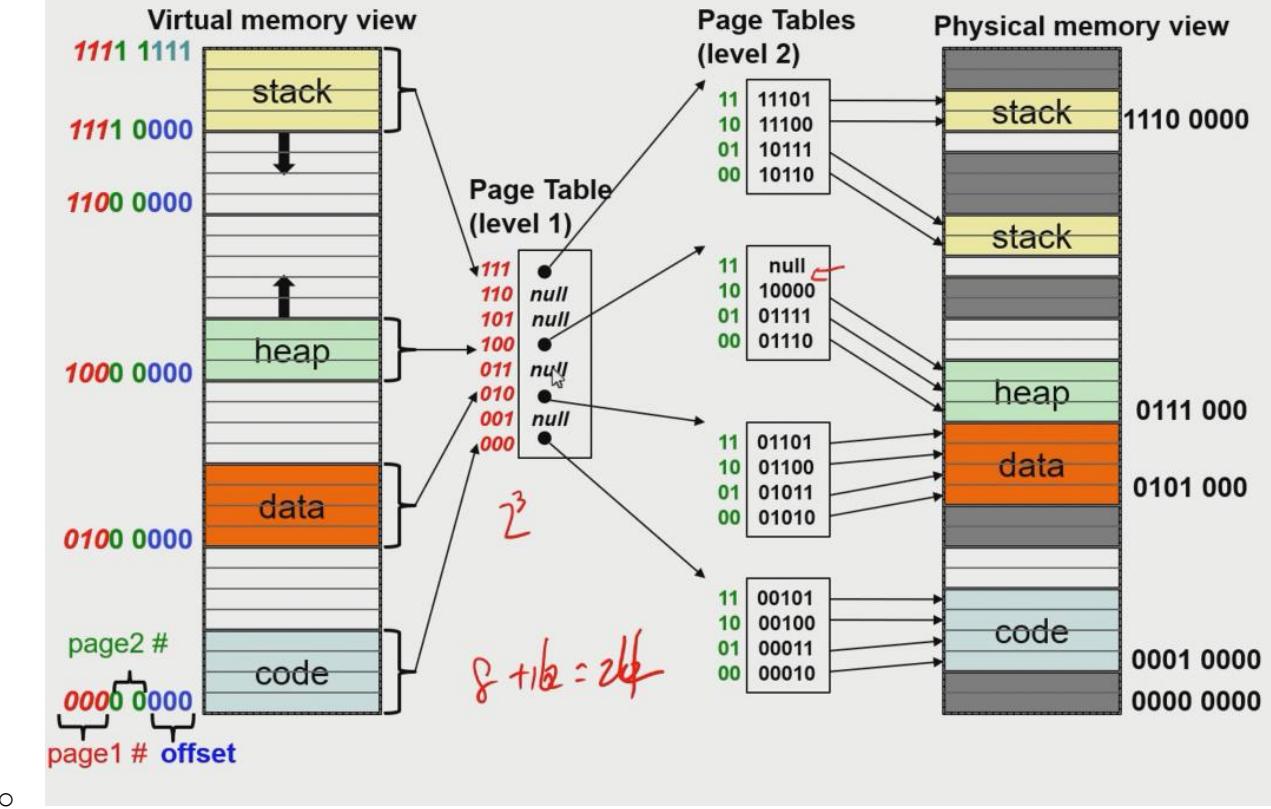


Fix for sparse address space: The two-level page table

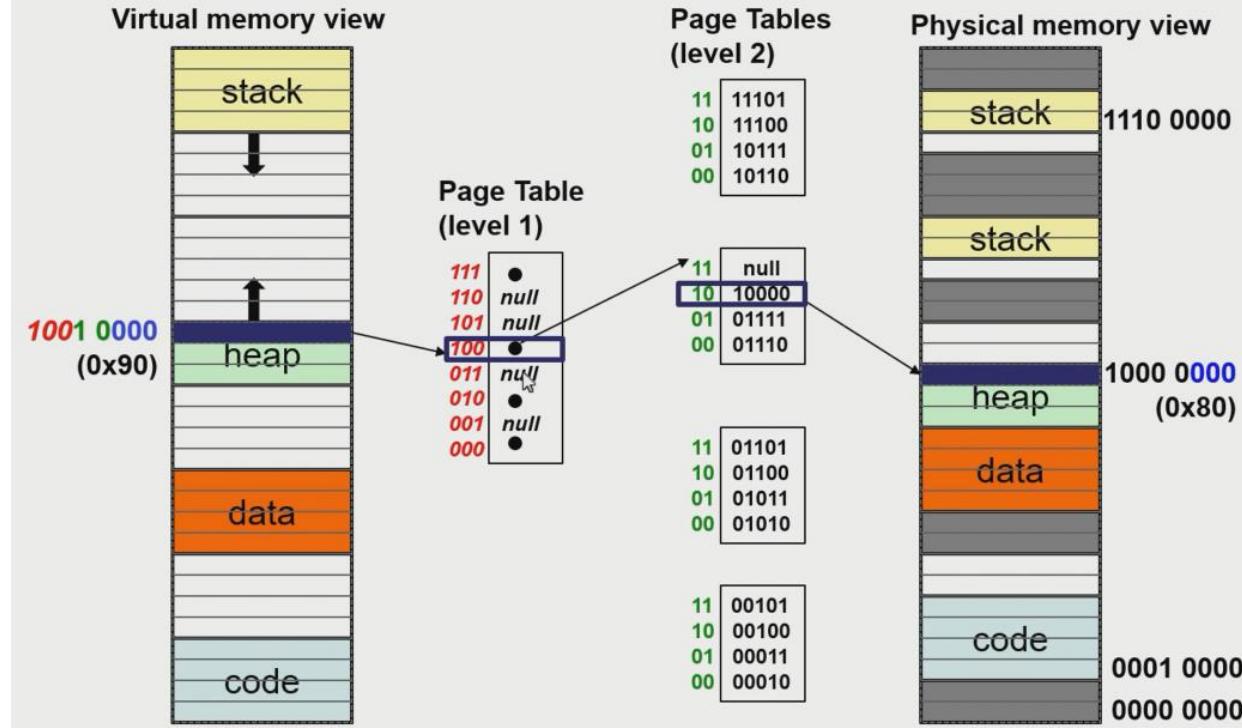


- Tree of Page Tables
- Tables fixed size (1024 entries)
 - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
 - Don't need every 2nd-level table
 - Even when exist, 2nd-level tables can reside on disk if not in use
-
- Summary: Two-Level Paging
 - For one-level paging, in the previous example we needed 32 table entries; now with this two-level paging version, we only need $8+16 = 24$ entries.
 - Remember this example is using 8-bit memory, but in practice, we have 32-bit or 64-bit address architectures – so virtual address space is very sparse so that you can save even more unnecessary null page entries.

Summary: Two-Level Paging

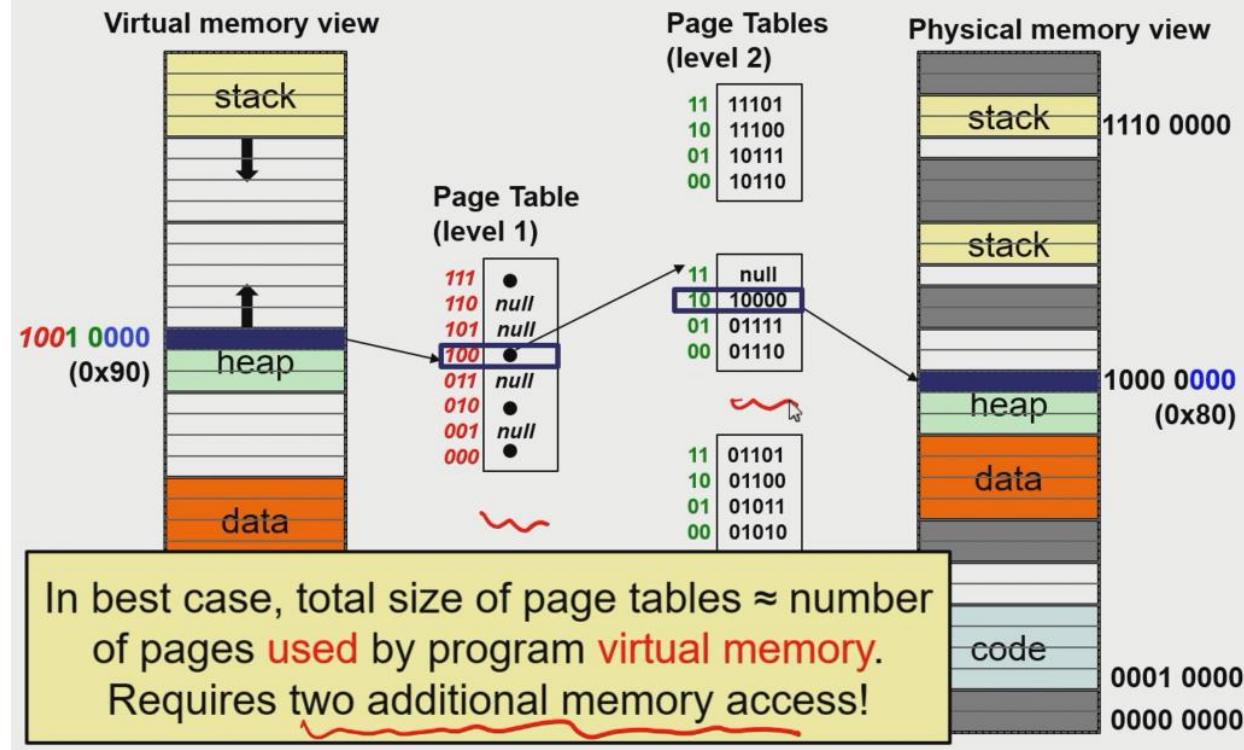


Summary: Two-Level Paging



-
- Remember, main memory speed is 100x slower than CPU memory; so every level of paging increases security and saves memory, but it dramatically slows down computer performance.
- So every time you have to access main memory, thus each level of paging, slow down the computer by a factor of 100 each time; i.e: 2 level = 200x slower, 3 levels = 300x slower, etc.
 - We need to find a solution for this later.

Summary: Two-Level Paging



- Multi-level Translation: Paging the Segments
 - Combine frame number and offset == physical address.

Multi-level Translation: Paging the Segments

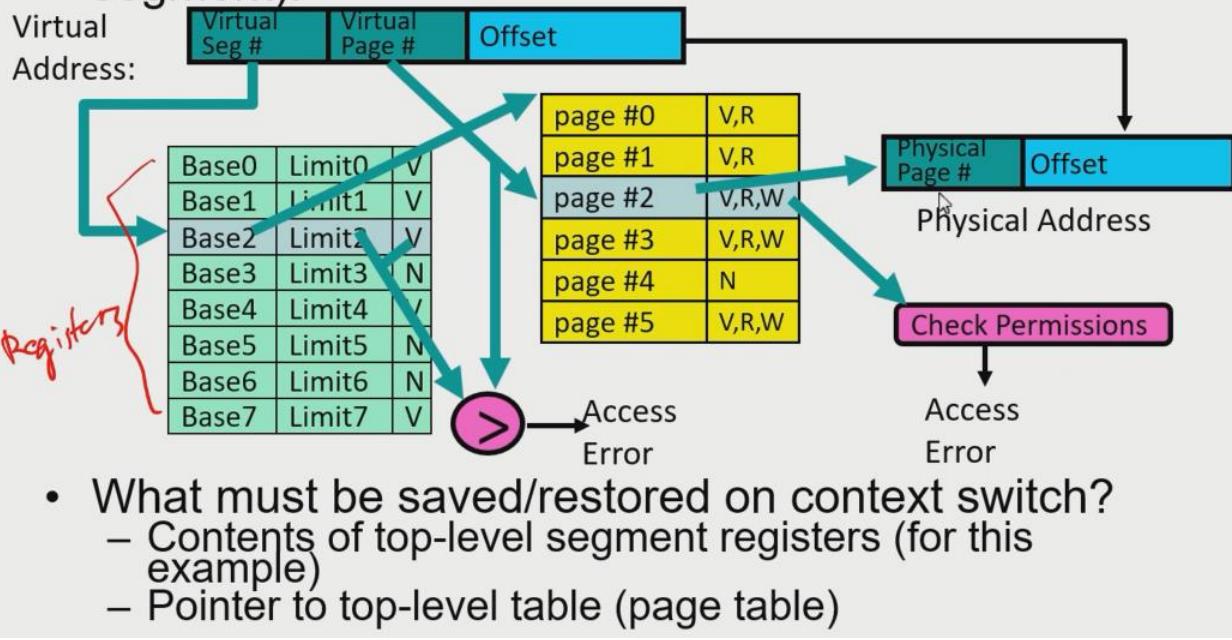
- Divide address into three parts
 - Segment number
 - Page number
 - Offset
 - Segment table contains addr. of page table
 - Use that and page # to get frame #
 - Combine frame number and offset
-
- What does this buy you?

What does this buy you?

- Simple management of physical memory
 - Paging (just a bitmap)
- Maintain logical structure
 - Segmentation
- However,
 - Possibly 3 memory accesses to get to memory!
- Paging with Segments
 - If page number is >limit; there will be an access error.
 - Number of segments not that large, so segment table can be stored in a register (much faster than main memory), which can save memory access time.
 - Page table has flags to indicate if an entry is valid or null, read only, or read and write; usually a single bit is used to save if a memory location is read only (0) or read and write (1).

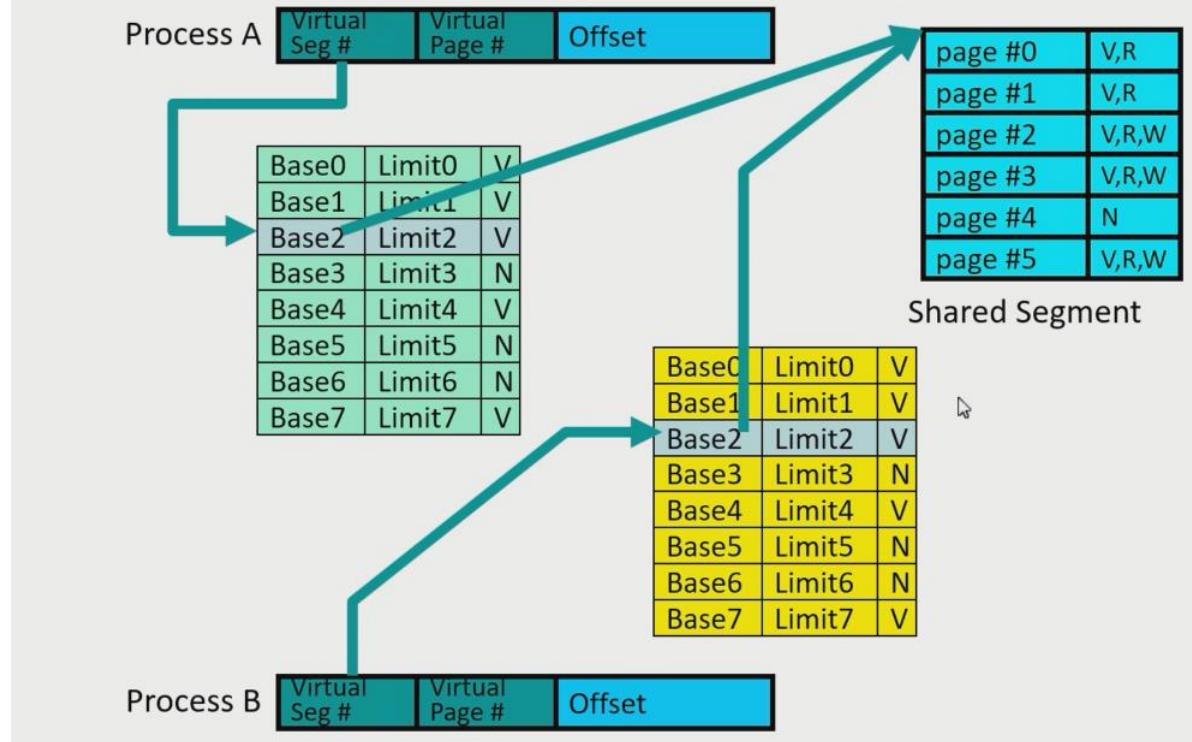
Paging the Segments

- What about a tree of tables?
 - Lowest level page table \Rightarrow memory still allocated with bitmap
 - Higher levels often segmented
- Could have any number of levels. Example (top segment):



- What must be saved/restored on context switch?
 - Contents of top-level segment registers (for this example)
 - Pointer to top-level table (page table)
- What about Sharing (Complete Segment)?
 - We can have a complete segment shared.
 - This logically make more sense because a segment is more meaningful, so that a shared segment such as the data segment will have same semantic meanings, i.e: data segment == shared segment.

What about Sharing (Complete Segment)?



-
- Intel x86-64 Page Table
 - Each page level index is 9 bits; thus 2^9 entries * 8bytes for each entry = $2^9 * 8 = 2^{12}$ = 4KB. So again, a page table size is exactly one page (default); but for intel you could have page size be 2MB (2^{21} ; combine OFFSET and L1), or 1GB (2^{30} ; combine OFFSET + L1 + L2).
 - Page table entry will indicate if it points to next page table level or just a large page.

x86-64 Page Table

- In x86-64 virtual address has 64 bits, but only the first 48 bits are *meaningful*. We have 9 bits to index into each page table level, and 12 bits for the offset. This means 16 bits are left over and unused.

63-48	47-39	38-29	29-21	20-12	11-0
L4	L3	..	L2	L1	OFFSET

- Page sizes: $4KB$, $2MB$, $1GB$

$$2^{12}$$

$$2^{21}$$

$$2^{30}$$

$$2^9 \cdot 8 = 2^{12}B = 4KB$$

- Multi-level Translation Analysis

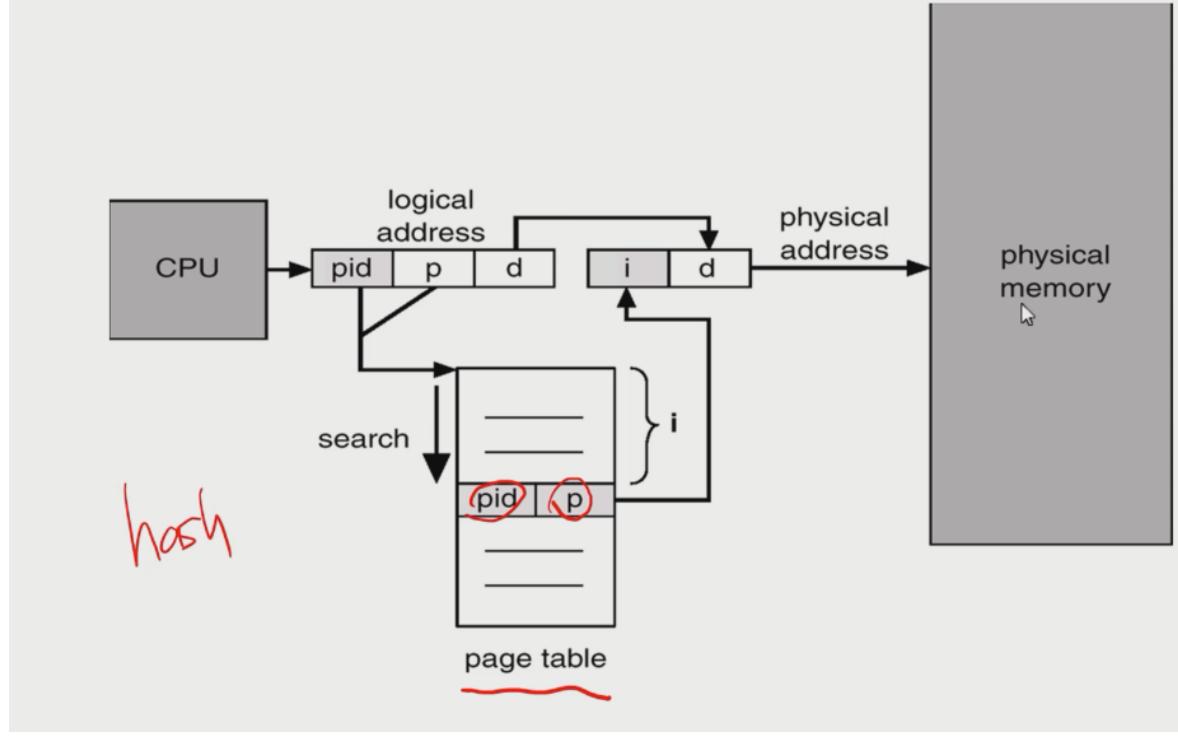
Multi-level Translation Analysis

- Pros:
 - Only need to allocate as many page table entries as we need for application
 - In other words, sparse address spaces are easy
 - Easy memory allocation *bitmap*
 - Easy Sharing
 - Share at segment or page level (need additional reference counting)
- Cons:
 - One pointer per page (typically 4K – 16K pages today)
 - Page tables need to be contiguous
 - However, previous example keeps tables to exactly one page in size
 - Two (or more, if >2 levels) lookups per reference
 - Seems very expensive!
- Inverted Page Table (“Core Map”)
 - With this scheme, it is more expensive since we have to actually scan the table (with bitmaps, we simply knew exactly how to convert to the physical address and did not have to search); so solution is to use hash table to limit searching.

Inverted Page Table (“Core Map”)

- One entry for each real page of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.
- Good for page replacement
 -
- Inverted Page Table Architecture
 - pid == process id
 - p = virtual page number
 - remember for hash table: if you want to reduce number of hash table size increases, make the table double the size.
 - Size of hash table is relatively proportional to the size of the physical memory (not the virtual memory), which means it is pretty good at not wasting space, since this hash table is used for all pages of physical memory.
 - Interesting idea, but not widely used.

Inverted Page Table Architecture



- Address Translation Comparison

Address Translation Comparison

	Advantages	Disadvantages
Simple Segmentation	Fast context switching (segment map maintained by CPU)	External fragmentation
Paging (Single-Level)	No external fragmentation Fast and easy allocation	Large table size Internal fragmentation
<u>Paged Segmentation</u>	Table size ~ # of pages in virtual memory	Multiple memory references per page access
<u>Multi-Level Paging</u>	Fast and easy allocation	
Inverted Page Table	Table size ~ # of pages in physical memory	Hash function more complex No cache locality of page table

- No cache locality means (I think) no entries can be stored in a register or CPU cache because hash table means there is a hash function that must always be solved in order to get the key to the correct hashed location.
- Summary

Summary

- Segment Mapping
 - Segment registers within processor
 - Segment ID associated with each access
 - Often comes from portion of virtual address
 - Can come from bits in instruction instead (x86)
 - Each segment contains base and limit information
 - Offset (rest of address) adjusted by adding base
- Page Tables
 - Memory divided into fixed-sized chunks of memory
 - Virtual page number from virtual address mapped through page table to physical page number
 - Offset of virtual address same as physical address
 - Large page tables can be placed into virtual memory
- Multi-Level Tables
 - Virtual address mapped to series of tables
 - Permit sparse population of address space
-
-

Lecture Video 11 (week11 – 3/21/22): Caching and TLB

- Speed up memory access when using virtual address schemes via caching
- CIS-310 concepts
- TLB = Translation Look Aside Buffer (accelerates virtual memory access)
 - A form of caching; if you did one virtual address translation before, you can attempt to use caching to remember the translation and do the translation faster via the cache buffer. This is much faster than traversing the page tables or having to do several main-memory lookups.

Caching and TLB

Virtual → Physical

- - How is the translation accomplished?
 - PTE = Page Table Entry

How is the Translation Accomplished?



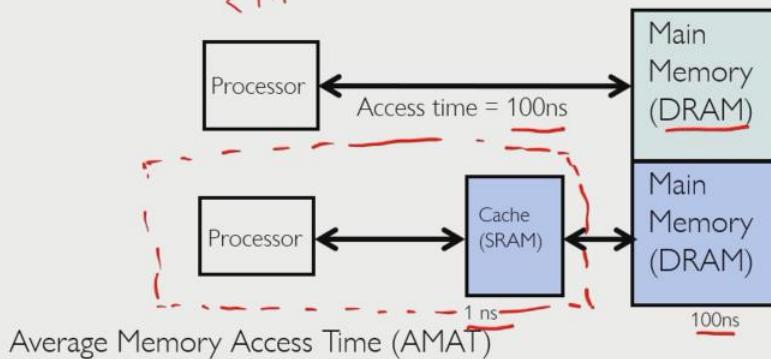
- The MMU must translate virtual address to physical address on:
 - Every instruction fetch
 - Every load
 - Every store
- What does the MMU need to do to translate an address?
 - 1-level Page Table
 - Read PTE from memory, check valid, merge address
 - Set “accessed” bit in PTE, Set “dirty bit” on write
 - 2-level Page Table
 - Read and check first level
 - Read, check, and update PTE
 - N-level Page Table ...
- **MMU does Page Table Tree Traversal to translate each address**
 - Remember page table tree traversal is very slow since every traversal to the next level is an additional main memory access.
- General Concept: Caching

General Concept: Caching

- Cache “safe place to hide or store things”
 - With computers, can provide significant speedup
 - Make frequent case fast and infrequent case less dominant
- Caching underlies many techniques used today to make computers fast
 - Can cache: memory locations, address translations, pages, file blocks, file names, DNS, Web Pages, etc...
- Average Access time =
$$(\text{Hit Rate} \times \text{Hit Time}) + (\text{Miss Rate} \times \text{Miss Time})$$
- Only good if:
 - Frequent case frequent enough and
 - Infrequent case not too expensive
- Caching is the key to memory system performance
 - 1 GHz CPU clock speed means its memory has a access speed of about 1 ns; 4GHz has an access speed of about 0.25ns.
 - SRAM is on the same chip as the CPU; part of CPU memory
 - Place frequently accessed memory inside of SRAM cache

Caching is the key to memory system performance

$1/67/2 \rightarrow 0.25\text{ ns}$
 $< 1\text{ ns}$



$$\text{HitRate} = 90\% \Rightarrow \text{AMAT} = (0.9 \times 1) + (0.1 \times 100) = 11.1 \text{ ns}$$

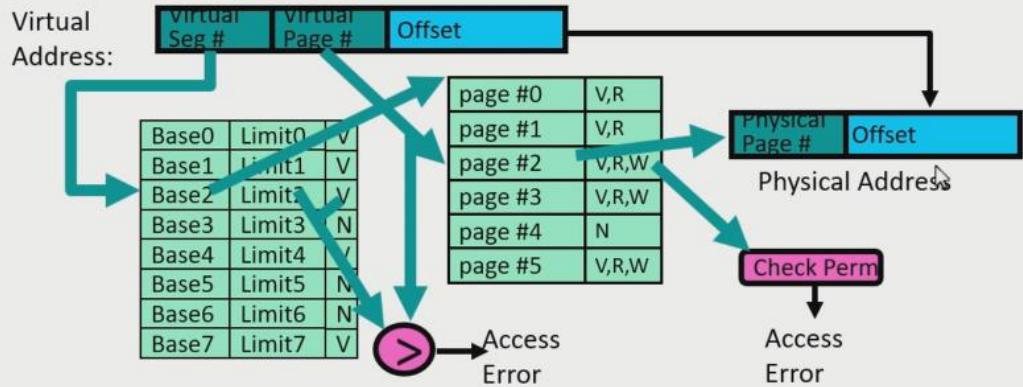
$$\text{HitRate} = 99\% \Rightarrow \text{AMAT} = (0.99 \times 1) + (0.01 \times 100) = 2.01 \text{ ns}$$

MissTime_{L1} includes $\text{HitTime}_{L1} + \text{MissPenalty}_{L1} \equiv \text{HitTime}_{L1} + \text{AMAT}_{L2}$

$$= 1 + 100 = 101 \text{ ns}$$

- Another Major Reason to Deal with Caching
 - From previous paging the segment example
 - In this example (provided segment table, which can be stored in register/cached on the CPU, is also in main memory), you have to do **three** DRAM accesses per actual DRAM access: first page the segment table, then page table, then physical page.
 - Perhaps even slower lookups if some of the tables are stored on the disc; thus I/O operations needed which is greatly slower (Demand Paging)
 - It would be bad if we use caching to make memory access faster for only one of the levels of this lookup tree, if the overall chain of translation is slower, it won't really help; it will be a waste of cache memory use.

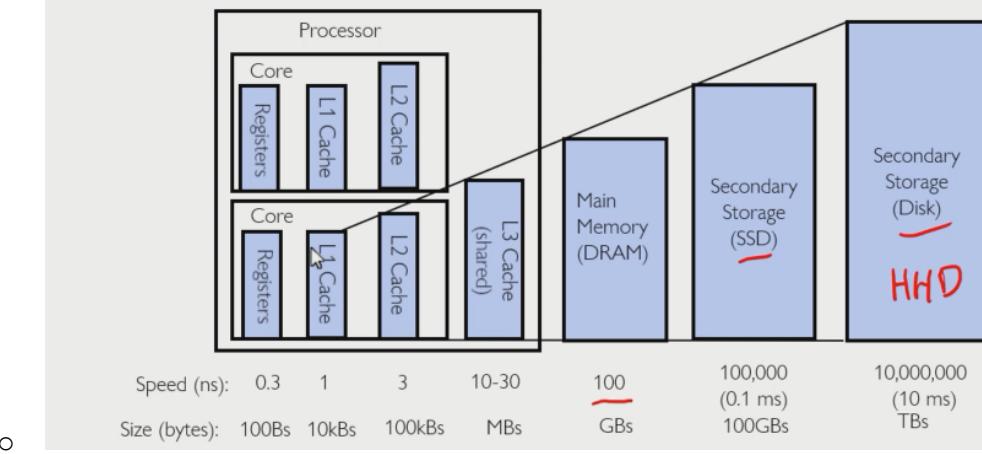
Another Major Reason to Deal with Caching



- Cannot afford to translate on every access
 - At least three DRAM accesses per actual DRAM access
 - Or: perhaps I/O if page table partially on disk!
- Even worse: What if we are using caching to make memory access faster than DRAM access?
- Solution? Cache translations!
 - Translation Cache: TLB (“Translation Lookaside Buffer”)
-
- Memory Hierarchy (typical x86 memory structure)
 - L3 cache is shared by all cores
 - But all cores, registers, and cache levels are all on the same CPU chip
 - All cache memory are SRAM
 - For reading and writing, the speed indicated for each type of memory is in regards to the average time to read or write to one block (frame) of memory.

Memory Hierarchy

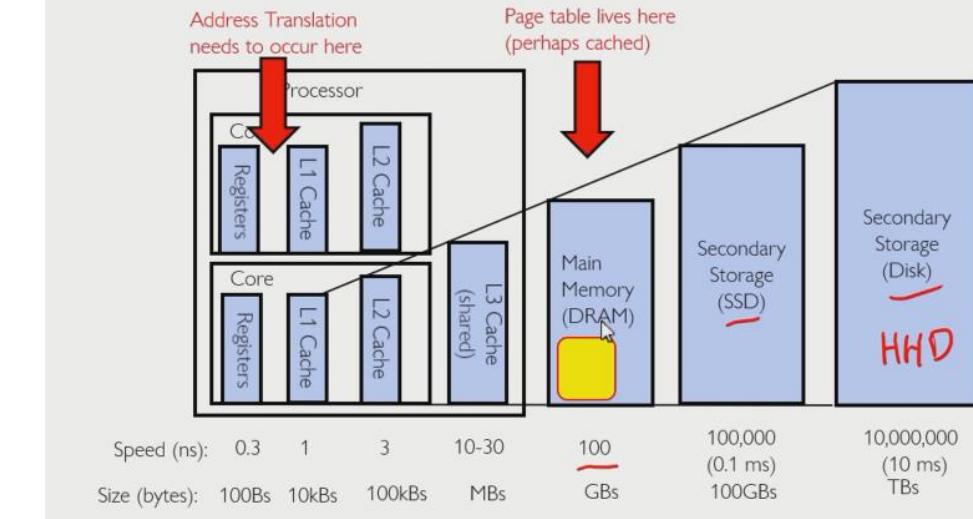
- Caching: Take advantage of the principle of locality to:
 - Present the illusion of having as much memory as in the cheapest technology
 - Provide average speed similar to that offered by the fastest technology



- For HHD he meant to write HDD == Hard Disk Drives

Memory Hierarchy

- Caching: Take advantage of the principle of locality to:
 - Present the illusion of having as much memory as in the cheapest technology
 - Provide average speed similar to that offered by the fastest technology



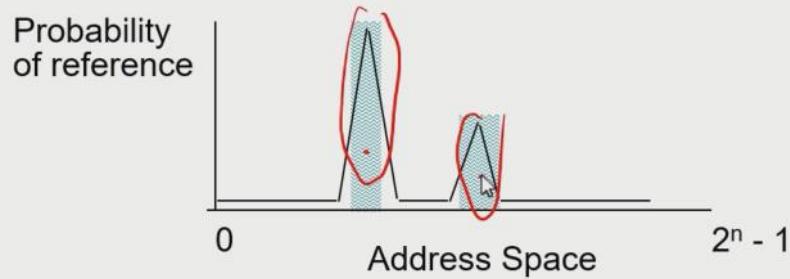
-

- Memory Hierarchies
 - Programs do behave regularly; they often spend 90% of the time in only 10% of the code that it is running. This is because most processes run programs that contain **loops** where most of the work is done, and usually accessing the same memory locations; thus, caching is often very effective at providing the principle of locality.

Memory Hierarchies

- Cache => fast access, expensive, small
- Main memory => slow access, cheap, large
- Want to:
 - put frequently accessed data in cache
 - All else in cheap memory
- This works if:
 - Programs behave regularly
 - They do – 90 % of time in 10 % of code (why?)
 -
- When do memory hierarchies work?
 - You find that programs spend most of the time in a small part of code segment (instructions), and a small part of the data segment; thus they have a relatively good locality.
 - Example of spatial locality: if you access element 1 of an array, you are likely to access element 2 and all other elements or **nearby** elements of that array/location of the array being accessed.
 - These are what make it possible to have a high hit rate (cache hit).

When do memory hierarchies work?

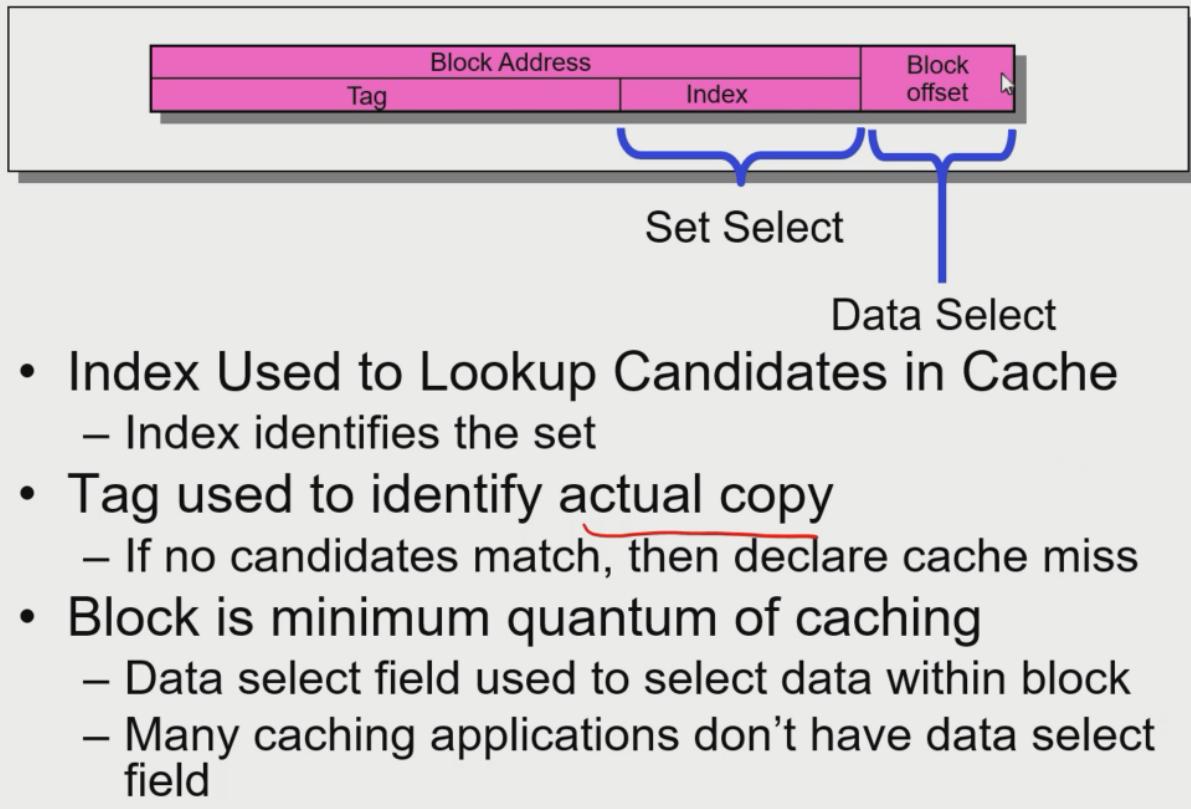


- Temporal locality
 - Reference same locations as in recent past
 - Occurs because of loops
- Spatial locality
 - Reference locations close by
 - Occurs because of loops and structured data
- Caching Issues
 - P() means “probability of”

Caching Issues

- What is the effective access time?
 - $P(\text{hit}) * \text{time for hit} + P(\text{miss}) * \text{time for miss}$
 - Better hit a lot
- How do you figure out if data is in cache?
- How do you make room if necessary?
- How do you keep cache consistent?
 -
- How is a Block found in a Cache? (review concepts from CIS-310 course – Computer Organization and Assembly)
 - Mostly done in hardware (not in OS).
 - 3 parts: Tag, Index, and offset. (tag and index make up block address)

How is a Block found in a Cache?

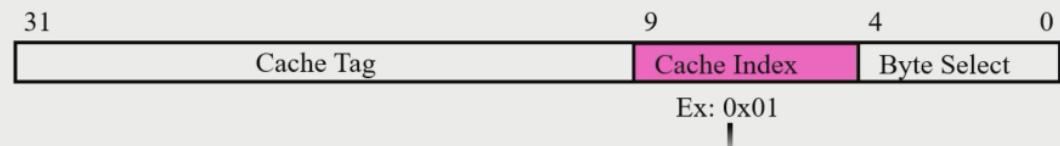


- Index Used to Lookup Candidates in Cache
 - Index identifies the set
- Tag used to identify actual copy
 - If no candidates match, then declare cache miss
- Block is minimum quantum of caching
 - Data select field used to select data within block
 - Many caching applications don't have data select field
- - Direct Mapped Cache (32-bit)
 - In this example, Cache Index = $M=5$ bits; thus cache each block is $2^5 = 32$ bytes; so, the quantum (minimum size / discrete size for each block that can be stored in cache) for caching in this instance is 32bytes.
 - In this example, the cache capacity is 1KB; each row (block) is 32 bytes; thus there are 32 rows (0 to 31), and thus $32\text{blocks} \times 32\text{bytes} = 1024$ bytes of cache (bytes 0 to 1023).

Direct Mapped Cache

- **Direct Mapped 2^N byte cache:**
 - The uppermost $(32 - N)$ bits are always the Cache Tag
 - The lowest M bits are the Byte Select (Block Size = 2^M)
 - Example: 1 KB Direct Mapped Cache with 32 B Blocks
 - Index chooses potential block

$$m = 5$$

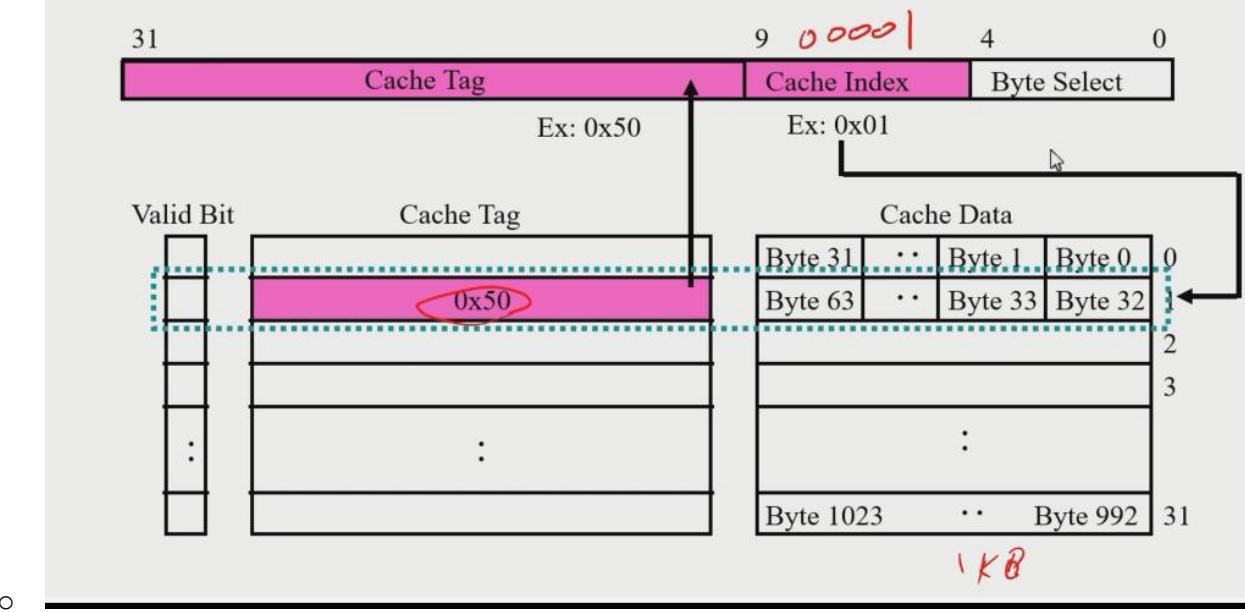


1KB

- How do we know if we have a cache hit? Look at the tag; tag should match address we are comparing to see if it is cached.
 - It is called **Direct** Mapped Cache, because each physical memory location is only possibly cached once into the cache.

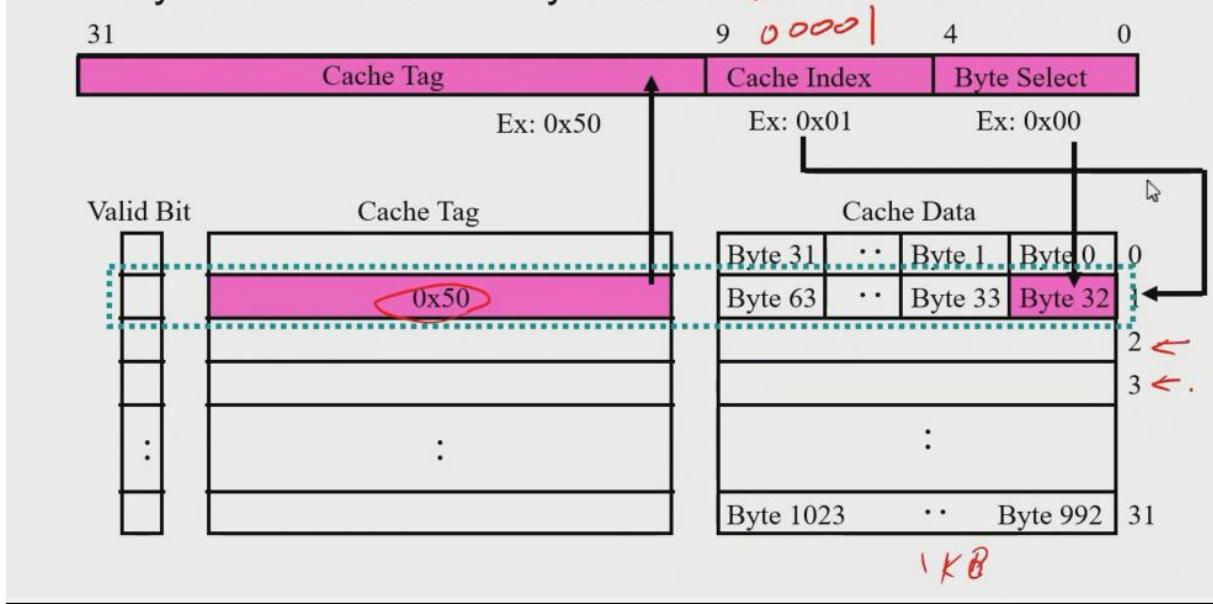
→ Direct Mapped Cache

- **Direct Mapped 2^N byte cache:**
 - The uppermost $(32 - N)$ bits are always the Cache Tag
 - The lowest M bits are the Byte Select (Block Size = 2^M)
 - Example: 1 KB Direct Mapped Cache with 32 B Blocks
 - Index chooses potential block
 - Tag checked to verify block
- $m = 5$



→ Direct Mapped Cache

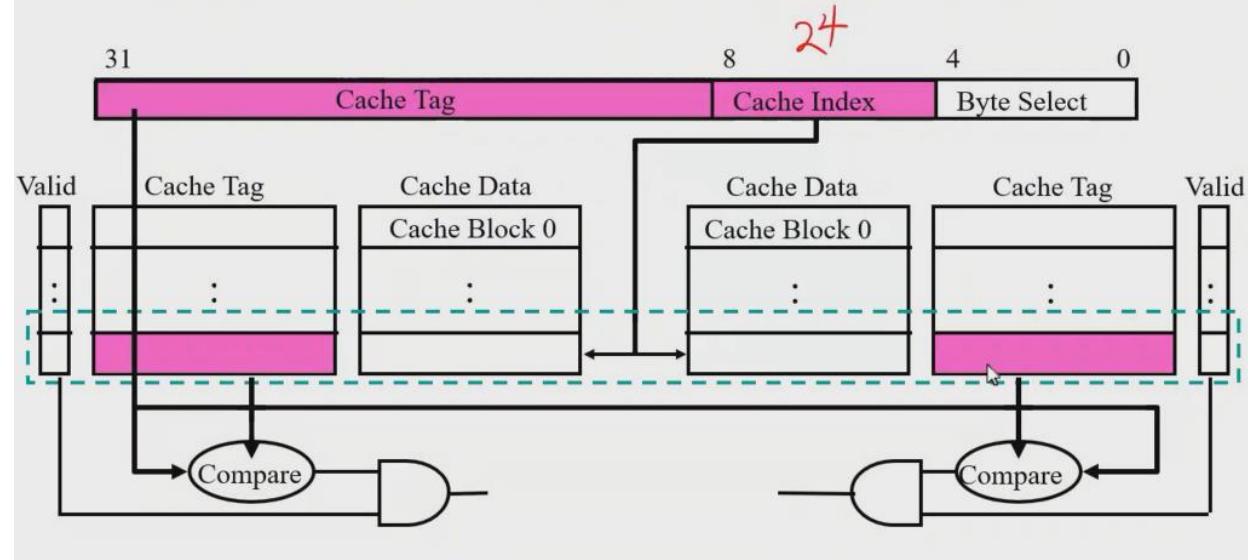
- **Direct Mapped 2^N byte cache:**
 - The uppermost ($32 - N$) bits are always the Cache Tag
 - The lowest M bits are the Byte Select (Block Size = 2^M)
- Example: 1 KB Direct Mapped Cache with 32 B Blocks
 - Index chooses potential block
 - Tag checked to verify block
 - Byte select chooses byte within block



-
- Set Associative Cache
 - Every physical memory can map to N different cache positions inside cache.
 - In this example, we now only use 4 bits for cache index; $2^4 = 16$ entries possible. Cache block size is still based on 5 bits = $2^5 = 32$ bytes = M (quantum); and thus you can select any one of the 32 bytes' address on a given 32-byte block.
 - Two way associate; so at each entry you can either go left or right. If entries from both left and right are compared to memory location being attempted do not have at least 1 match, then cache miss.

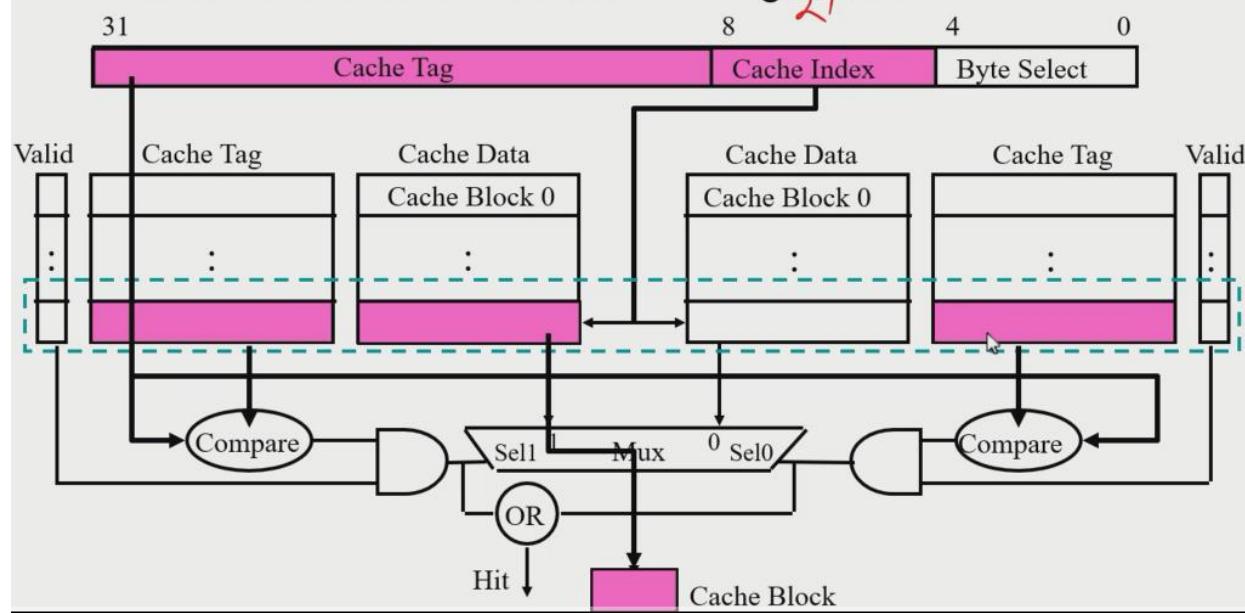
Set Associative Cache

- **N-way set associative:** N entries per Cache Index
 - N direct mapped caches operate in parallel
- **Example: Two-way set associative cache**
 - Cache Index selects a “set” from the cache
 - Two tags in the set are compared to input in parallel



Set Associative Cache

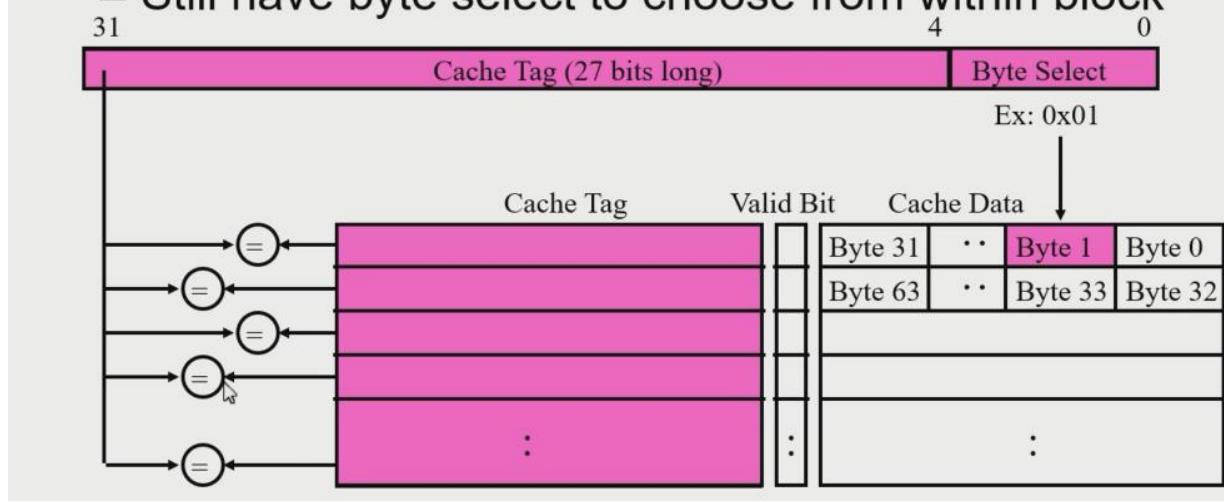
- **N-way set associative:** N entries per Cache Index
 - N direct mapped caches operates in parallel
- **Example: Two-way set associative cache**
 - Cache Index selects a “set” from the cache
 - Two tags in the set are compared to input in parallel
 - Data is selected based on the tag result



- If cache miss, then load from main memory into cache (update tag; cache the memory since it was not), and then also access that memory needed from main memory via the newly cached version of it.
- This method requires more complex hardware; you need to do this comparison in parallel.
- Fully Associative Cache
 - Simply directly compare the tags
 - Only 5 bits are for the byte select; remaining 27 bits used for cache tag.
 - When checking for a cache hit, we do a parallel comparison (32 in this case' 32 parallel comparison (simultaneous comparisons via hardware)).
 - Again: if cache miss, then load from main memory into cache and update cache tag, then access the needed memory from the cache: cache time + main memory time.
 - Fully associative: for every physical memory (RAM) location you can have many places to cache it into physical cache.
 - Generally more expensive and slower than direct mapped cache (set associative is in the middle).

Fully Associative Cache

- **Fully Associative:** Every block can hold any line
 - Address does not include a cache index
 - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
 - We need N 27-bit comparators
 - Still have byte select to choose from within block

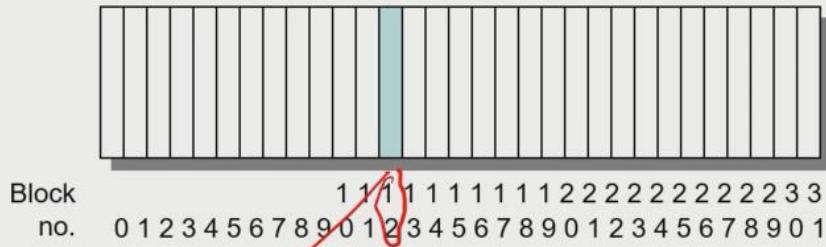


- Where does a block get placed in Cache?

Where does a Block Get Placed in a Cache?

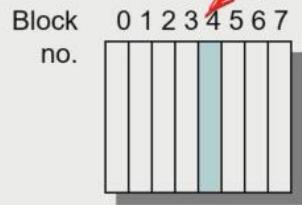
- Example: Block 12 placed in 8 block cache

32-Block Address Space:



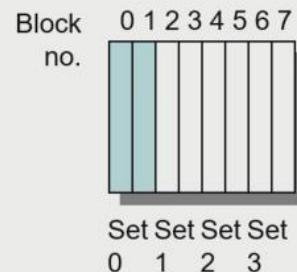
Direct mapped:

block 12 can go
only into block 4
($12 \bmod 8$)



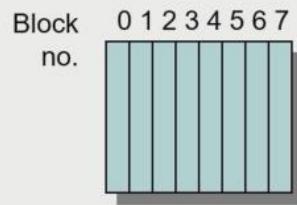
Set associative:

block 12 can go
anywhere in set 0
($12 \bmod 4$)



Fully associative:

block 12 can go
anywhere

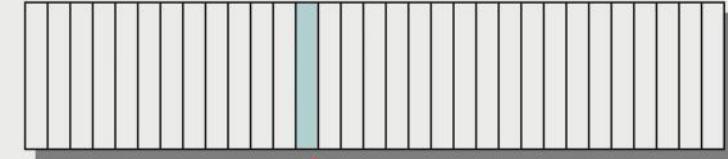


- Fully associative is more efficient, but slower, and more complex and more expensive.

Where does a Block Get Placed in a Cache?

- Example: Block 12 placed in 8 block cache

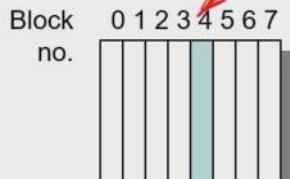
32-Block Address Space:



Block no. 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

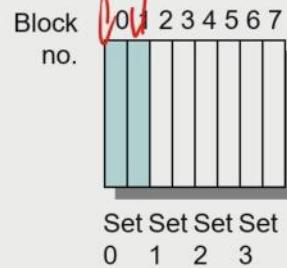
Direct mapped:

block 12 can go
only into block 4
($12 \bmod 8$)



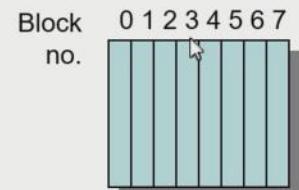
Set associative:

block 12 can go
anywhere in set 0
($12 \bmod 4$)



Fully associative:

block 12 can go
anywhere



- Sources of Cache Misses

Sources of Cache Misses

- **Compulsory** (cold start or process migration, first reference): first access to a block
 - “Cold” fact of life: not a whole lot you can do about it
 - Note: If you are going to run “billions” of instruction, Compulsory Misses are insignificant
- **Capacity:**
 - Cache cannot contain all blocks access by the program
 - Solution: increase cache size
- **Conflict (collision):**
 - Multiple memory locations mapped to the same cache location
 - Solution 1: increase cache size
 - Solution 2: increase associativity
- **Coherence (Invalidation): other process (e.g., I/O) updates memory**
 - With coherence, cache memory is invalidated; so then it needs to be pushed out (cleaned/wiped) and basically restarted; thus all programs will essentially be running as a “cold start”.
- What happens on a write?
 - Clean: block is not updated yet/recently.
 - Dirty: block was updated and needs to write back to memory; if block is clean, then no need to write back to memory – it is safe to replace the block with another memory location without writing back.

What happens on a write?

- **Write through:** The information is written to both the block in the cache and to the block in the lower-level memory
- **Write back:** The information is written only to the block in the cache.
 - Modified cache block is written to main memory only when it is replaced
 - Question is block clean or dirty?
- Pros and Cons of each?
 - WT:
 - PRO: read misses cannot result in writes
 - CON: Processor held up on writes unless writes buffered
 - WB:
 - PRO: repeated writes not sent to DRAM processor not held up on writes
 - CON: More complex
Read miss may require writeback of dirty data
- Caching Address Translations
 - For all caching, locality is essential; the locality must be strong enough to make caching effective.

Caching Address Translations

- When translate virtual address to physical address
- Use a Translation Lookaside Buffer (**TLB**) *cache*
 - Small cache of frequently used virtual to physical translation – (why would this be helpful?)
 - Part of MMU
 - On the chip, so latency is very small (1 – 5ns)
- Question is one of page locality: does it exist?
 - Instruction accesses spend a lot of time on the same page (since accesses sequential)
 - Stack accesses have definite locality of reference
 - Data accesses have less page locality, but still some... →
- How do we:
 - Tell if translation is in TLB
 - Make room if needed
 - Keep TLB consistent.
- • Finding Translation in TLB
 - It makes sense that TLB uses fully associative scheme, since the TLB cache memory size (capacity) is so small; fully associative is generally slower, but it is more memory-efficient; thus in this case, it is best to use it for TLB.

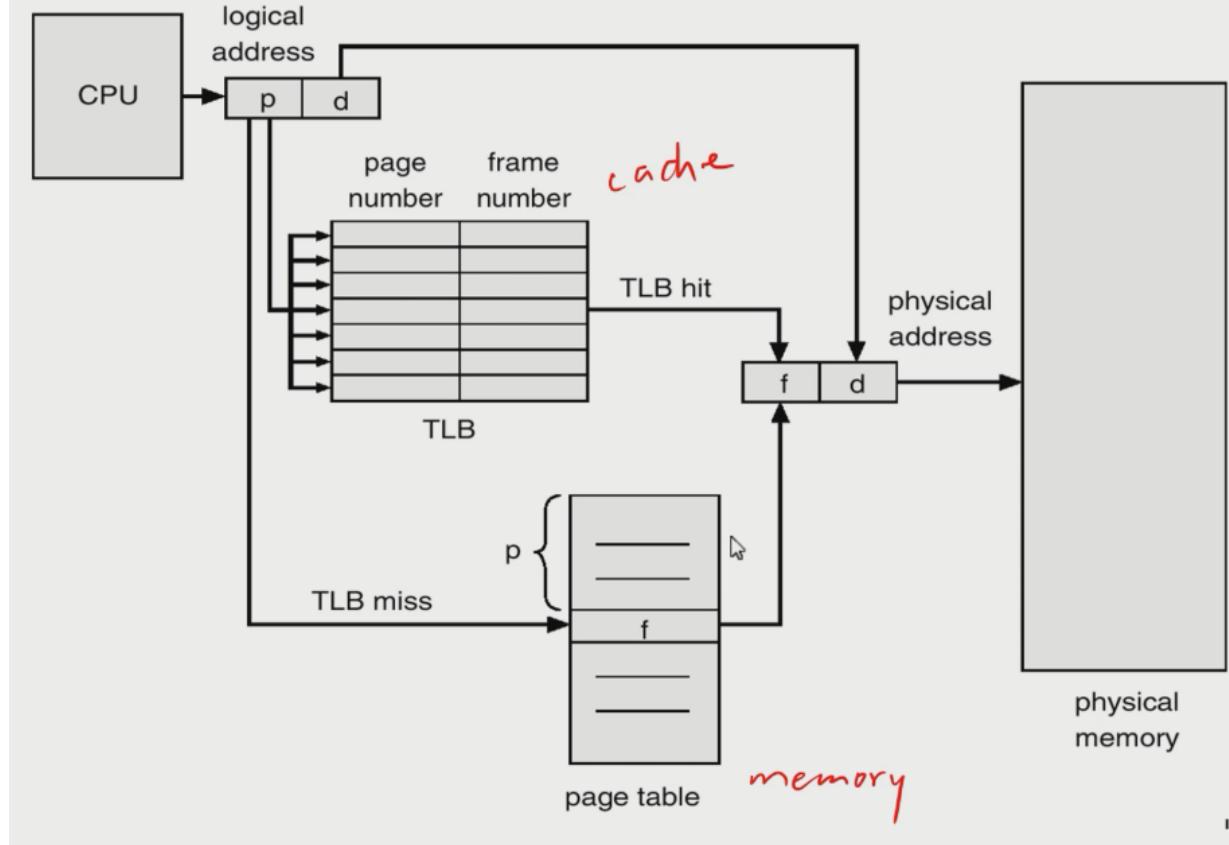
Finding Translation in TLB

- TLB can be:
 - Direct mapped, meaning entry can be only one possible place
 - n-way set associative, meaning entry can be in “n” different places (assume $n < \text{size(TLB)}$).
 - ✓ – Fully associative, meaning entry can be anywhere
 - (note, check all in parallel)

TLBs generally small and fully associative:
memory caches generally direct mapped or n-way set associative where n is small.

- - Paging Hardware With TLB
 - p = virtual page table number
 - d = offset
 - fully associative, and thus uses hardware which can do all comparison simultaneously (in parallel) via a set of OR gates.
 - Notice, for a cache hit, you only have to do one cache access, and one main memory access.
 - f = frame number
 - if cache miss, you have to check page table, which of course is in memory, thus you have to access main memory twice to get the desired physical address.

Paging Hardware With TLB



- Choosing an Item to Replace
 - TLB is fully associative; so when you have a miss, you can replace any of the cached entries.

Choosing an Item to Replace

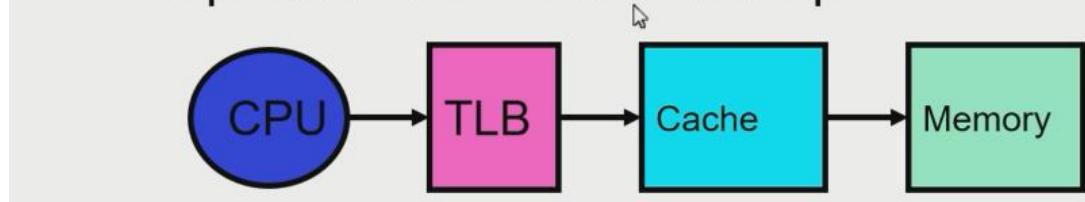
- On TLB miss, probably will access this page again
 - So, OS puts this translation in TLB
 - Must replace some entry; many methods for this.
 - Must be fast here - random choice is LRU reasonable
 - Hit in TLB cuts memory access time by 50-75%, whereas hit in disk cache cuts disk access time by several orders of magnitude
-
- What about consistency?
 - AS = address space, VP = virtual page number, PF = physical frame (page) number
 - Notice between processes, there could be two VP with the same number, but since they are different processes, they could map to different physical pages; one solution is to just validate the TLB on context switch.
 - But if you context switch a lot, then TLB will be constantly invalidated; thus translation will be slow more frequently and thus stunt performance.
 - Architectural solution is the one more commonly used right now.

What about consistency?

- On context switch between threads in different address spaces:
 - New address space has completely different page table
 - Can get aliasing in the TLB
 - AS1: VP 4 == PF 6
 - AS2: VP 4 == PF 1
- Options?
 - Invalidate TLB: simple but might be expensive
 - What if switching frequently between processes?
 - Include ProcessID in TLB
 - This is an architectural solution: needs hardware
- TLB Organization

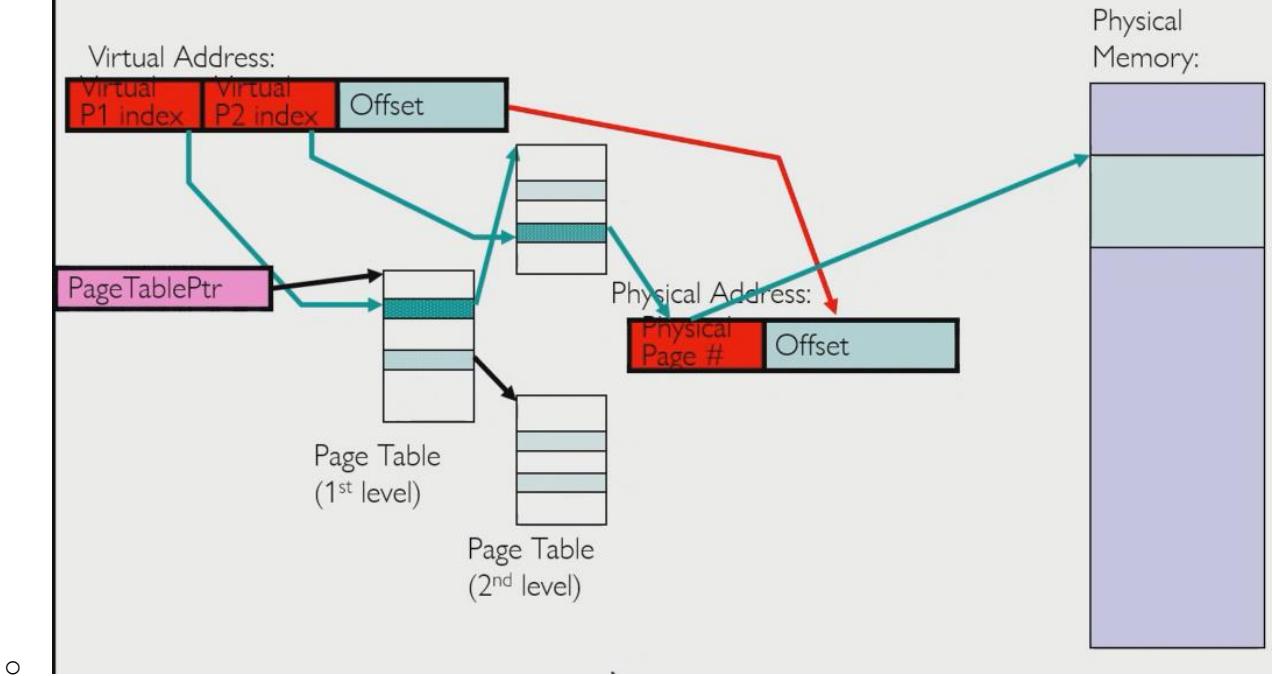
TLB Organization

- How big does TLB actually have to be?
 - Usually small: 128-512 entries
 - Not very big, can support higher associativity
- TLB usually organized as fully-associative cache
 - Lookup is by Virtual Address
 - Returns Physical Address + other info
- When does TLB lookup occur?
 - Before cache lookup?
 - In parallel with cache lookup?



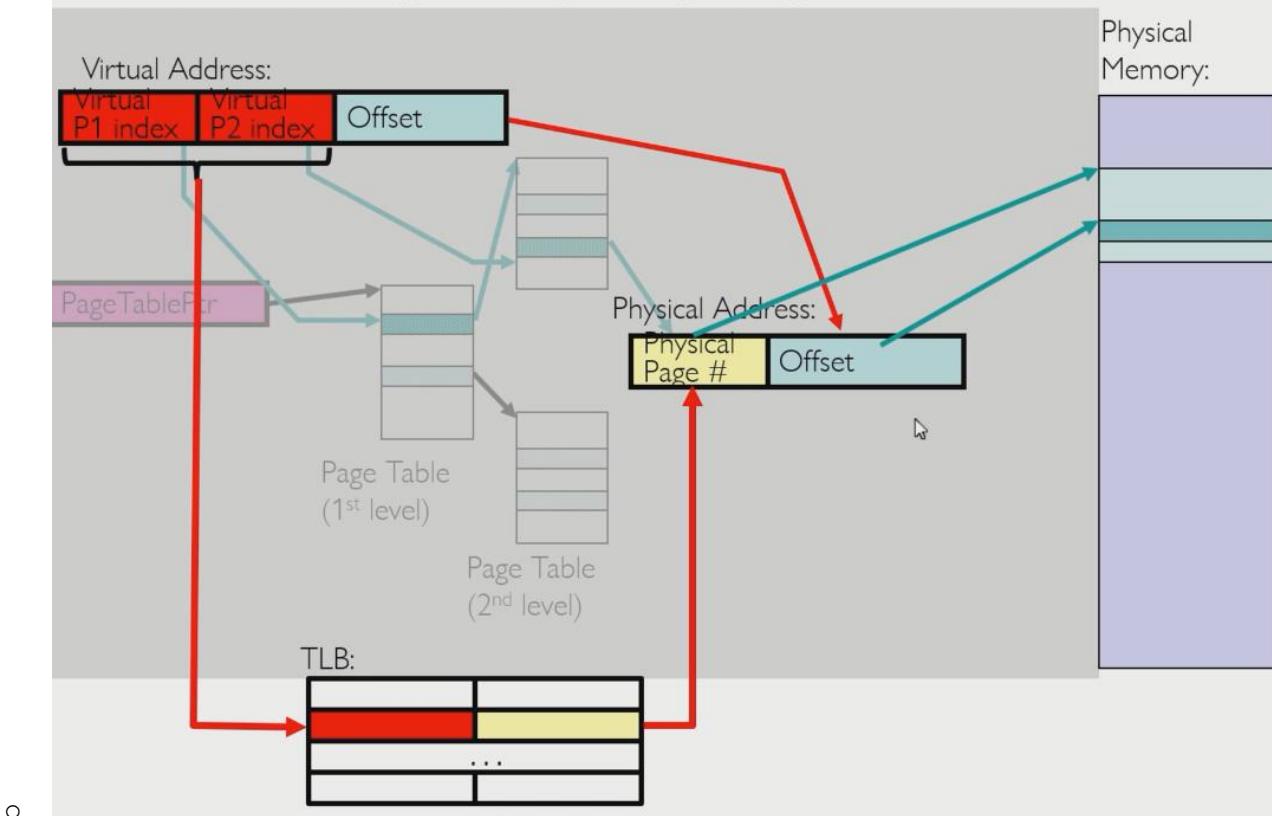
- Putting Everything Together: Address Translation (two-level paging system)
 - Slow pace:

Putting Everything Together: Address Translation



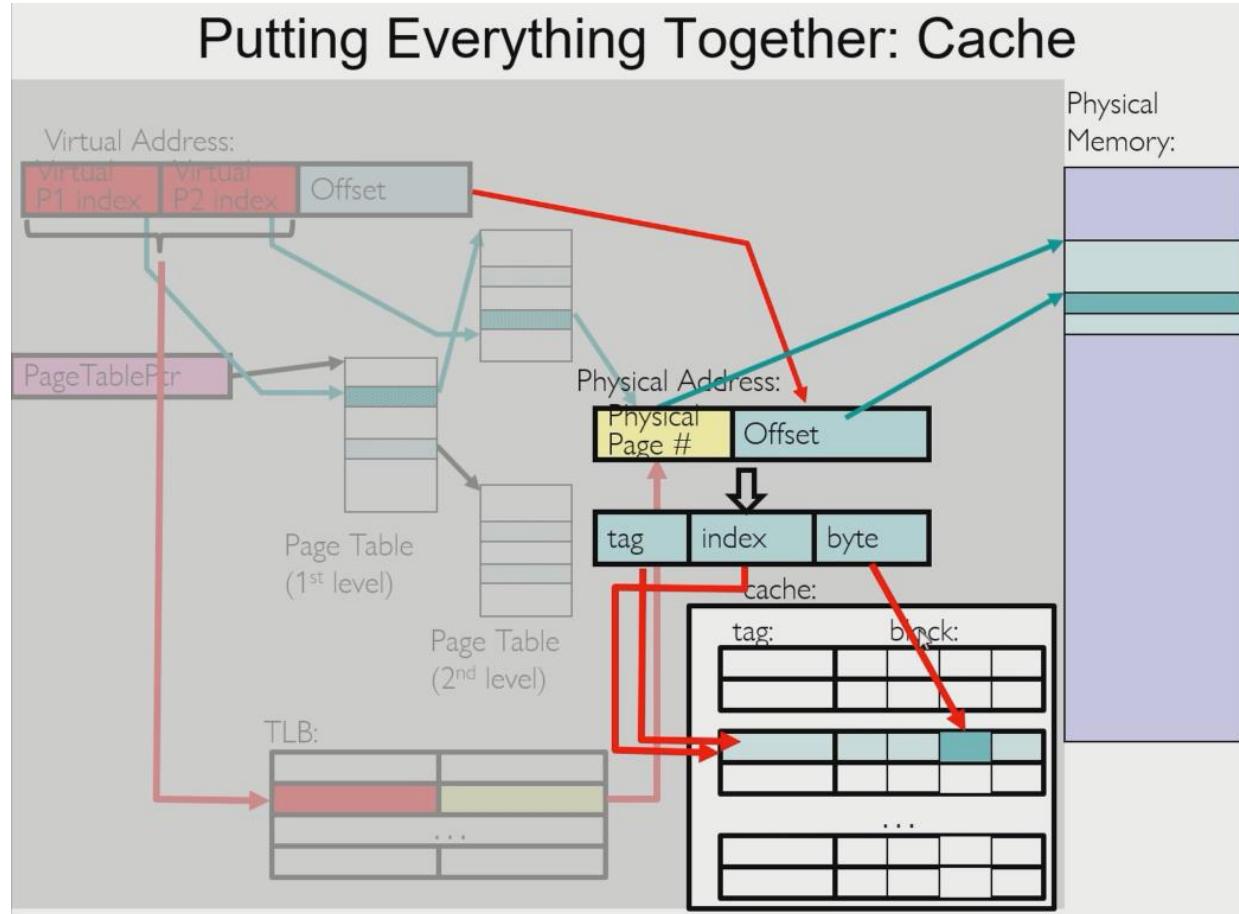
- o Faster pace (using TLB):

Putting Everything Together: TLB



- o

- Fastest page lookup via TLB and cache lookup:
 - Remember for L1, L2, AND L3 cache, it is often read and write to via Set Associative scheme.



Lecture Video 12-1 (week11 – 3/21/22): Demand Paging

- Virtual Memory vs. Physical Memory

Virtual Memory vs. Physical Memory

- So far, all of a job's virtual address space must be in physical memory
- However, many parts of programs are never accessed
 - Unlikely error conditions
 - Wasted malloc/new
 - Large arrays
- Processes don't use all their memory all of the time
 - 90-10 rule: programs spend 90% of their time in 10% of their code
 - Wasteful to require all of user's code to be in memory
- Large startup cost on a context switch
 - Must load all of program into physical memory
 - Limits degree of multiprogramming
 - Demand Paging: Using Main Memory as a Cache for the Disk
 - Most of a process's address space (and data stored at the address) can be stored on the disc, and we can use main memory as a cache for the disk.
 - Some of the pages (virtual memory) of a process is never brought into the main memory – hence the 90-10 rule, and hence why we do not need all pages of the entire page table to be valid for every process to be placed in main memory and take up that valuable faster memory space.
 - This is why total virtual memory can be much larger than actual available physical memory – because of demand paging.
 - This gives us lots of flexibility; for example, we don't really have to worry about running out of virtual address space as much, and thus we don't have to worry about the stack and heap running into each other, etc.
 - For the old Sparc machines, you can use 4GB of virtual address space for EACH process via demand paging.

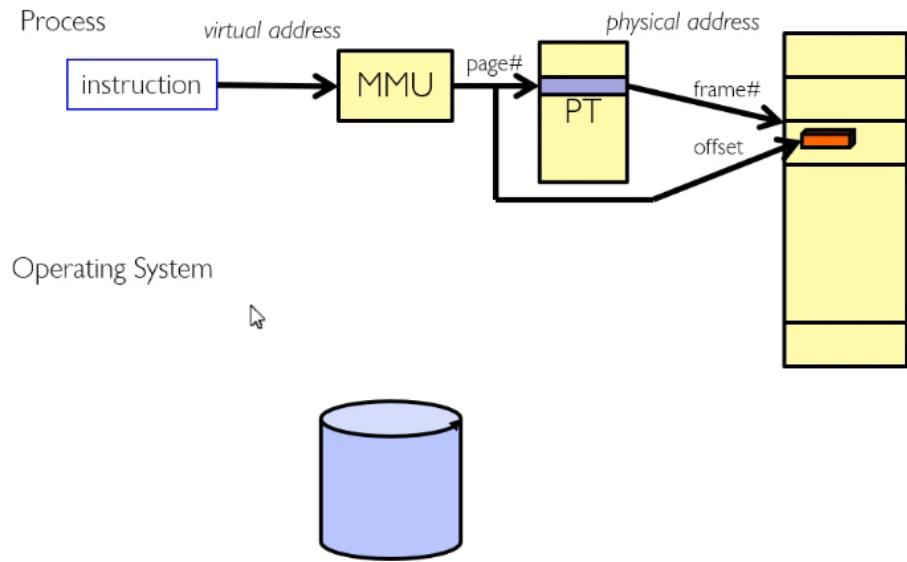
Demand Paging: Using Main Memory as a Cache for the Disk

- Demand paging follows that pages should only be brought into memory if the executing process demands them.
- Allow Virtual Memory > Physical Memory
 - Many advantages
 - Larger degree of multiprogramming
 - Can write programs for large virtual memories
 - Ex: on Sparc, 32 bit virtual addresses
 - Can use 4GB of virtual address space
 - Physical memory typically 16-64 MB

Warning: needs to be implemented efficiently

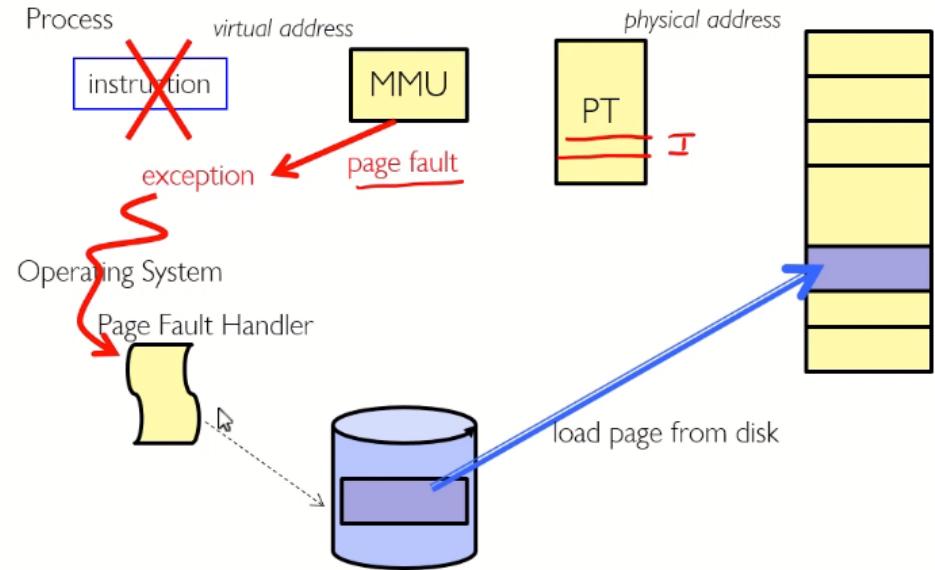
- Page Fault → Demand Paging

Page Fault \Rightarrow Demand Paging

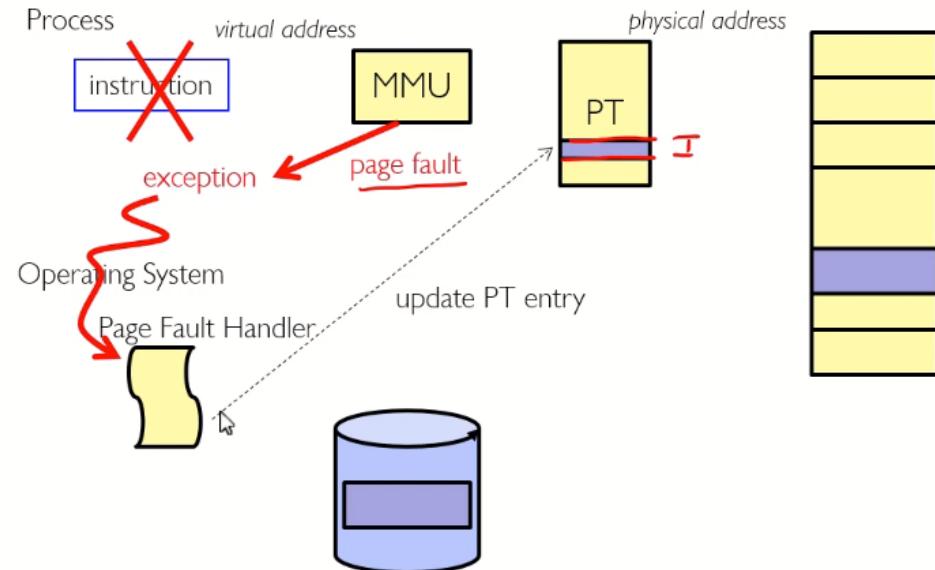


- - Above, is a successful translation to an address to fetch an instruction (via the page table, which is in main memory, and then going to the physical location of the instruction in the main memory after going through the memory management unit and to the page table).
 - Remember, with demand paging, the entire virtual address space is not needed for a process, thus we can shorten the amount of pages in main memory (which are frames of course) and thus shorten the main memory usage since page table will not be fully valid (since some pages will be on disc). And also, because of this fact, the virtual memory could be much larger than physical memory, thus, for some processes we could not physically store all pages of a page table in main memory.
 - Now below, a demonstration of a page fault when a given page is not in the page table (which is in main memory, as well the page table of course), and thus needs to be loaded from the disc. The process that needed an instruction but caused a fault will go to sleep (wait) so that the page fault handler can run, and then once it completes, then the process that went to sleep will be made awake (runnable) in order to retry fetching the needed instruction:

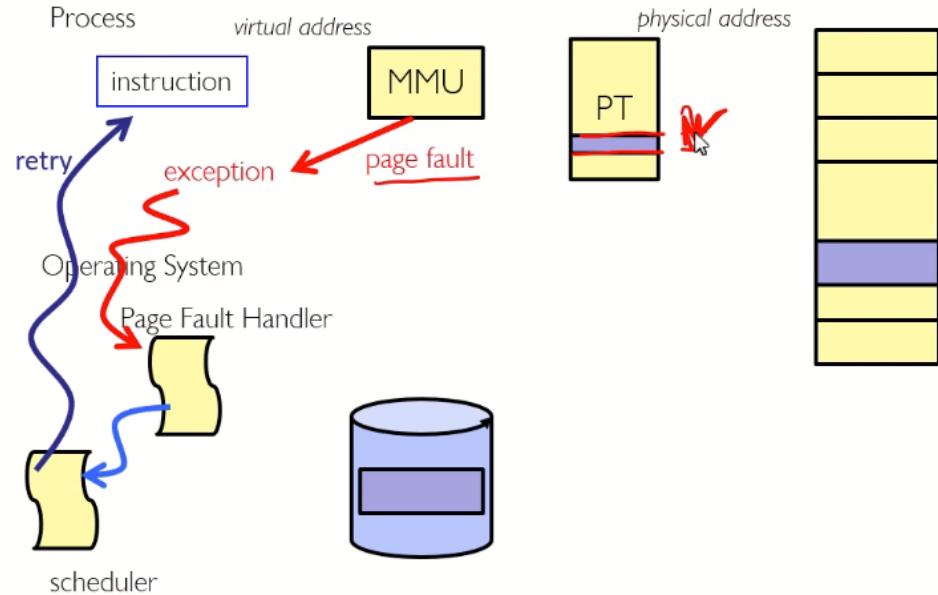
Page Fault \Rightarrow Demand Paging



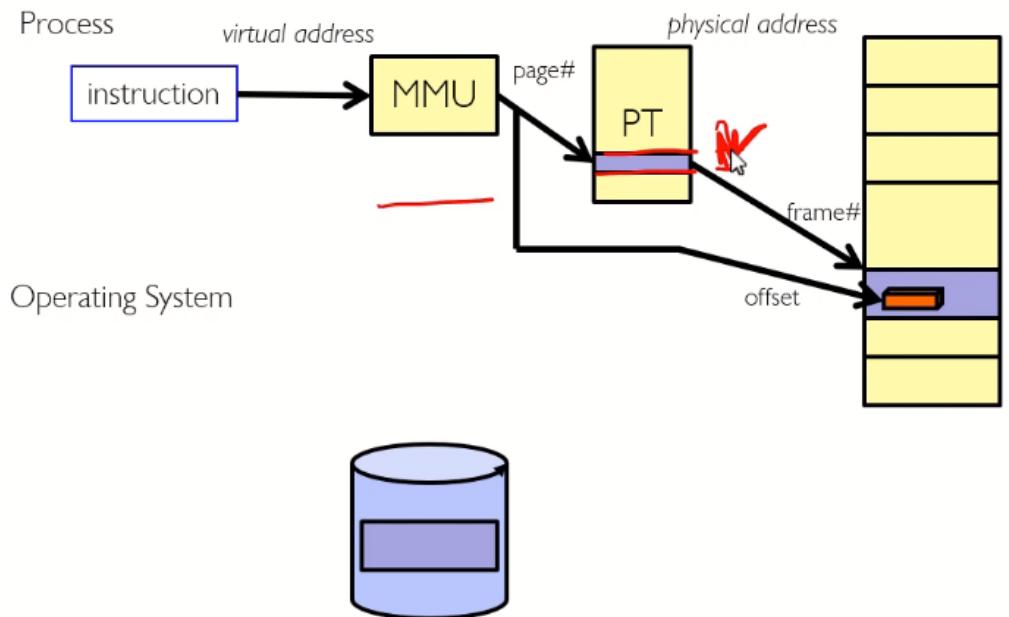
Page Fault \Rightarrow Demand Paging



Page Fault \Rightarrow Demand Paging



Page Fault \Rightarrow Demand Paging



- Demand Paging is Caching

- Fully associative: no limits on the physical memory space that the virtual address space can map to. This is a must for demand paging.
- Remember from caching: TLB = Translation Look Aside Buffer (accelerates virtual memory access)
- Random page replacement scheme is not good enough (unlike with the caching scheme) because we don't want to replace pages we will need later. Need a better scheme → LRU.
- Disc is very slow, so we definitely need write-back scheme when accessing a page: we need to load from the disc and load it to main memory, and then write back any updates to the page to the disc. We need a dirty bit to tell us if a page has been updated.

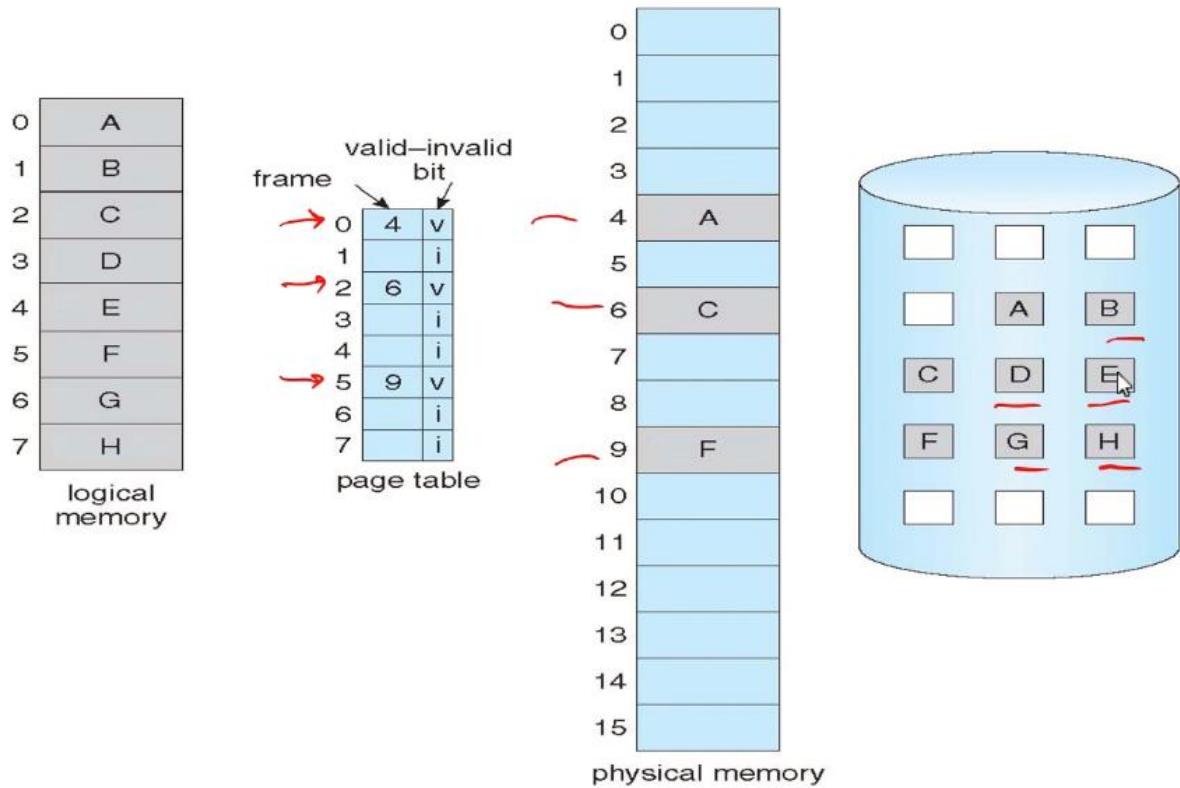
Demand Paging is Caching

- Since Demand Paging is Caching, must ask:
 - What is block size?
 - 1 page
 - What is organization of this cache (i.e. direct-mapped, set-associative, fully-associative)?
 - Fully associative: arbitrary virtual → physical mapping
 - How do we find a page in the cache when look for it?
 - First check TLB, then page-table traversal *V, I*
 - What is page replacement policy? (i.e. LRU Random...)
 - This requires more explanation... (kinda LRU)
 - What happens on a miss?
 - Go to lower level to fill miss (i.e. disk)
 - What happens on a write? (write-through, write back)
 - Definitely write-back. Need dirty bit!
- Translating Virtual Addresses to Physical Addresses
 - Valid/invalid bit = dirty bit → needed for each page table entry; the bit will say if page is in main memory or not; if it is in main memory, then the page is valid; invalid if on disc.

Translating Virtual Addresses to Physical Addresses

- Modify page table
 - Add valid/invalid bit
- Program still generate virtual addresses
 - Now look up physical frame and check valid bit
 - If invalid (“page fault”) means 1 of 2 things
 - Address is illegal => kill of thread *Segmentation fault*
 - Address is legal but on disk => get it.
- - Page Table When Some Pages Are Not in Main Memory
 - To run a process, we just need its instructions and data in main memory; we don't need everything else. This is what makes demand paging useful – to save memory. Also, it can sacrifice a reasonable amount time/speed for convenience of being able to increase multiprogramming and virtual address space size and other flexibility, by making use of the disc – which is usually always much larger but just slower.

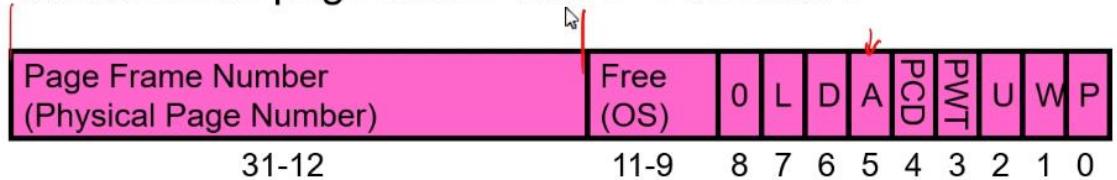
Page Table When Some Pages Are Not in Main Memory



- What is in a Page Table Entry (PTE)?
 - When dirty bit is set (true), then page needs to be written back to (updated on) the disc.
 - L = 1 (true) means page is large (4MB). (L = 0 means page size is 4KB).

What is in a Page Table Entry (PTE)?

- What is in a Page Table Entry (or PTE)?
 - Pointer to next-level page table or to actual page
 - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
 - Address same format previous slide (10, 10, 12-bit offset)
 - Intermediate page tables called “Directories”



P: Present (same as “valid” bit in other architectures)

W: Writeable

U: User accessible

PWT: Page write transparent: external cache write-through

PCD: Page cache disabled (page cannot be cached)

A: Accessed: page has been accessed recently

D: Dirty (PTE only): page has been modified recently

L: L=1 \Rightarrow 4MB page (directory only).

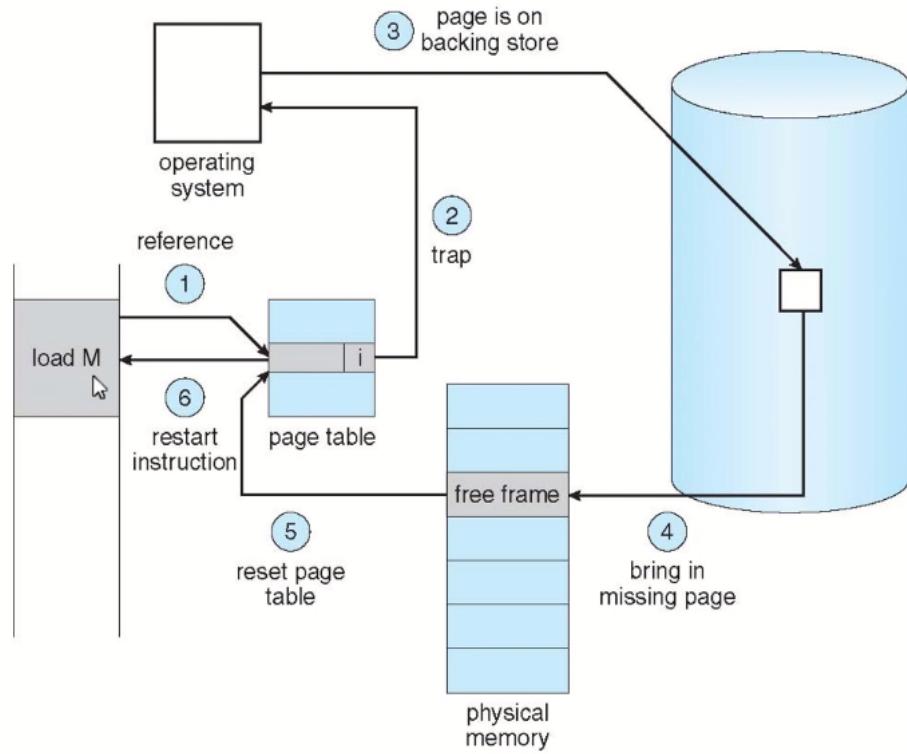
Bottom 22 bits of virtual address serve as offset

- What happens on a page fault?

What happens on a page fault?

- Trap to OS
 - Get a free physical frame (may need to reclaim)
 - If frame has been modified, schedule disk write
 - Invalidate that address space's page table entry
 - Schedule disk read (note: takes long time)
 - Update this address space's page table entry
 - Queue thread on disk queue
 - Context switch to new thread
 - (when disk read done move waiting thread to ready queue)
 - Steps in Handling a Page Fault

Steps in Handling a Page Fault

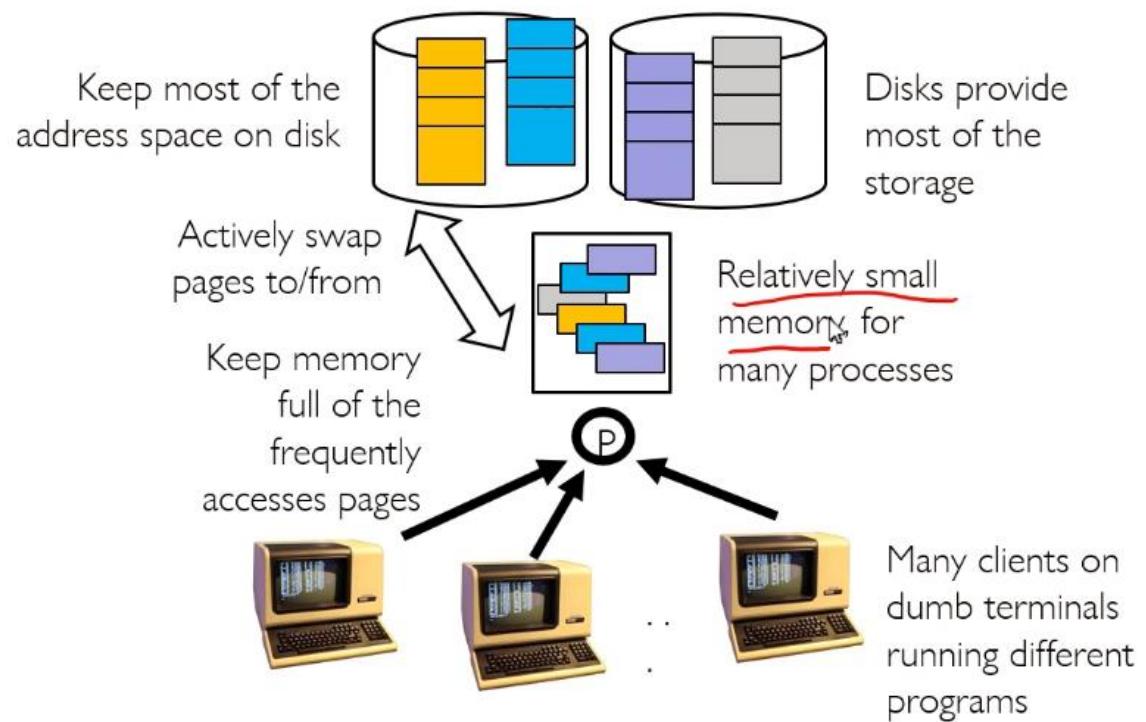


- Demand Paging Cost Model
 - Note: Hit rate + miss rate = 1 → as in 100%.
 - Thus miss rate = 1 – hit rate, and hit rate = 1 – miss rate
 - Miss time = hit time + miss penalty
 - Page-fault service includes mostly accessing the disc, which is very slow, and remember the time it takes for a disc access is the same amount of time that a million or more instructions can be executed – so remember disc access is very slow.
 - So if too many page faults occur (probability of miss is too high), then the performance of processes will be too slow/greatly reduced.

Demand Paging Cost Model

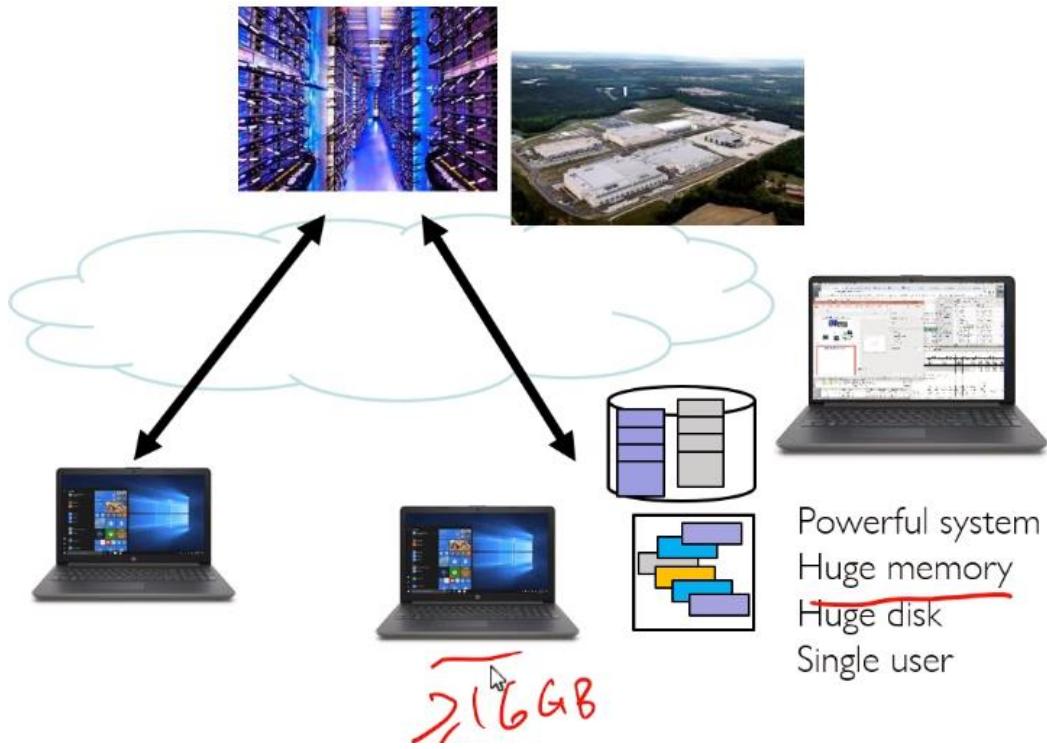
- Since Demand Paging like caching, can compute average access time! ("Effective Access Time")
 - $EAT = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Time}$ $\leftarrow (\text{hit time} + \text{miss penalty})$
 - $EAT = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$
- Example:
 - Memory access time = 200 nanoseconds
 - Average page-fault service time = 8 milliseconds
 - Suppose p = Probability of miss, $1-p$ = Probably of hit
 - Then, we can compute EAT as follows:
 $EAT = 200\text{ns} + p \times 8\text{ms}$ $1\text{ms} = 1,000,000\text{ ns}$
 $220\text{ ns} \leq 200\text{ns} + p \times 8,000,000\text{ns}$
- If one access out of 1,000 causes a page fault, then $EAT = 8.2\text{ }\mu\text{s}$:
 - This is a slowdown by a factor of 40!
- What if want slowdown by less than 10%?
 - $EAT < 200\text{ns} \times 1.1 \Rightarrow p < 2.5 \times 10^{-6}$
 - This is about 1 page fault in 400,000!
- Origins of Demand Paging
 - Remember demand paging increases the degree of multiprogramming since most of the address space of processes is stored on the disc; thus you can run more processes concurrently since main memory is not the primary limitation; of course you just do sacrifice having a smaller size limitation for much less speed due to disc access being slow.

Origins of Demand Paging



- Very Different Situation Today
 - Since today's machines have such large memory, rarely need to access the disc to demand (replace) a page; most machines don't use more than 50% of their memory, so page fault will be very rare.

Very Different Situation Today



- - But we still have need and use of demand paging today.
- Many Uses of Virtual Memory and “Demand Paging”

Many Uses of Virtual Memory and “Demand Paging” ...

- Extend the stack
 - Allocate a page and zero it
- Extend the heap
- Process Fork
 - Create a copy of the page table
 - Entries refer to parent pages – NO-WRITE
 - Shared read-only pages remain shared
 - Copy page on write
- Exec
 - Only bring in parts of the binary in active use
 - Do this on demand
- MMAP to explicitly share region (or to access a file as RAM)
Load Store
 - How does TLB affect things?

How does TLB affect things?

- First check in TLB
- If not there, trap to OS
- OS checks its page tables
 - If in memory, load page tables entry into TLB
 - If not in memory, page fault
- Aspects of Demand Paging
 - First access of a new/initialized processes results in pure demand paging – cannot be avoided; we have to start somewhere in terms of loading some of the pages into main

memory until all of the demanded/frequently used pages are loaded into main memory (if there is space – hence then normal demand paging begins). That's why starting a program always will take longer and run slower initially – once those pages are loaded into memory for the first time, then process can start to run smoother/faster since subsequent accesses will be much faster.

Aspects of Demand Paging

- Extreme case – start process with no pages in memory
 - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
 - And for every other process pages on first access
 - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
 - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
 - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
 - Page table with valid / invalid bit
 - Secondary memory (swap device with swap space)
 - Instruction restart
 - One detail left out: transparency
 - When we restart an instruction, we essentially restore the original state of the CPU for that process when that instruction was originally tried for fetching and execution.

One detail left out: transparency

- When faulting thread finally rescheduled:
 - **must restart instruction**
 - Instruction has never completed
 - Hardware must save enough state
 - What instruction was, complete state
 - Same as info saved on regular context switch
-
- Demand Paging Optimizations
 - So, part of a disc can have a special space called a “swap space”, which is faster than the other file system space on that same disc.
 - File system space: we have to do multiple things to find the correct disc block.
 - Swap space: we already know exactly which block on the disc to go to.
 - For file system related memory (statically created) (a file loaded into RAM so it can be read and written to easier) if a page is not dirty (dirty bit set since page was modified), no need to write it back to disc; so discard it (i.e., clear the space so another page can be written there). If a page is dirty, then of course it must be written back before the page is replaced.
 - For binaries stack and heap memory, which are dynamically created and can be changed and is often referenced (with functions that often have loops, etc.), is referred to as anonymous memory, which must always be written back to swap space before being replaced since it is so often accessed and changed.

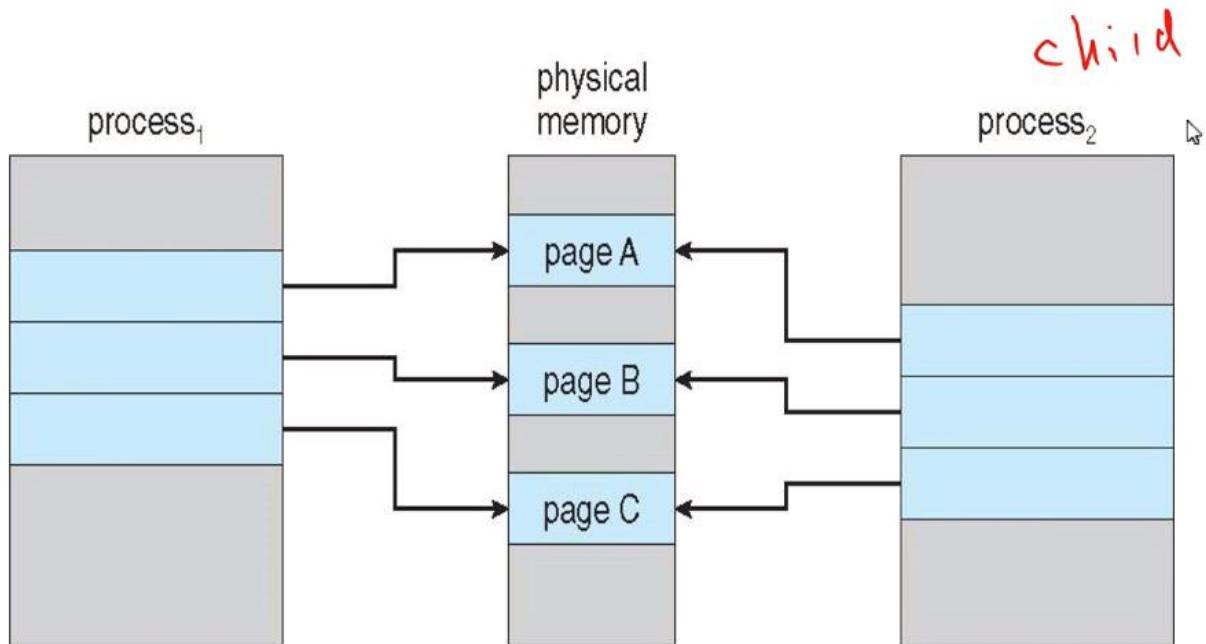
Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
 - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
 - Then page in and out of swap space
 - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
 - Used in Solaris and current BSD
 - Still need to write to swap space
 - Pages not associated with a file (like stack and heap) – anonymous memory
 - Pages modified in memory but not yet written back to the file system
- Copy-on-Write (COW)
 - Mainly for fork()

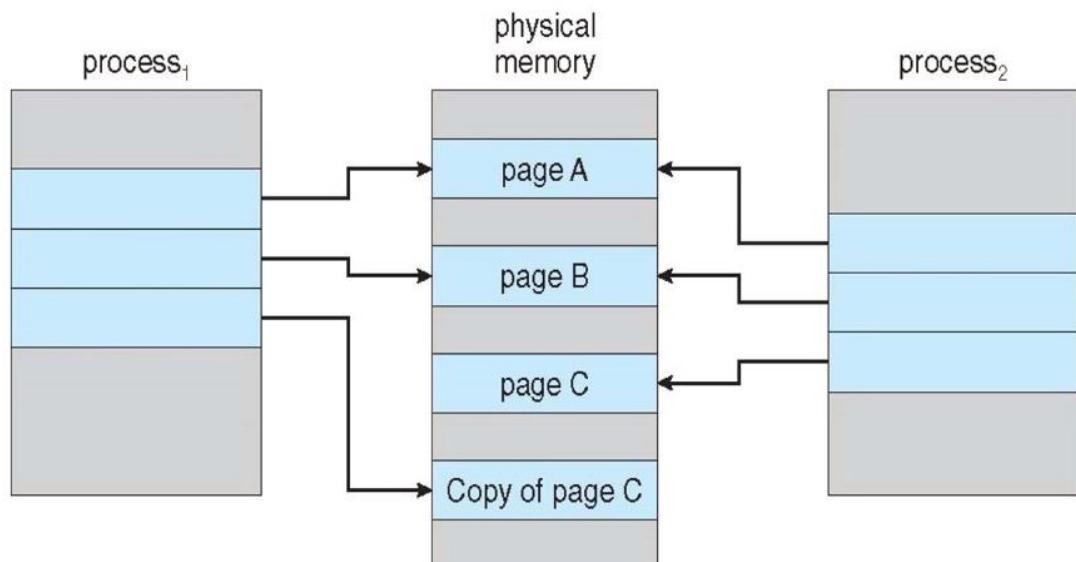
Copy-on-Write

- **Copy-on-Write (COW)** allows both parent and child processes to initially **share** the same pages in memory
 - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
 - Designed to have child call exec()
 - Very efficient

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



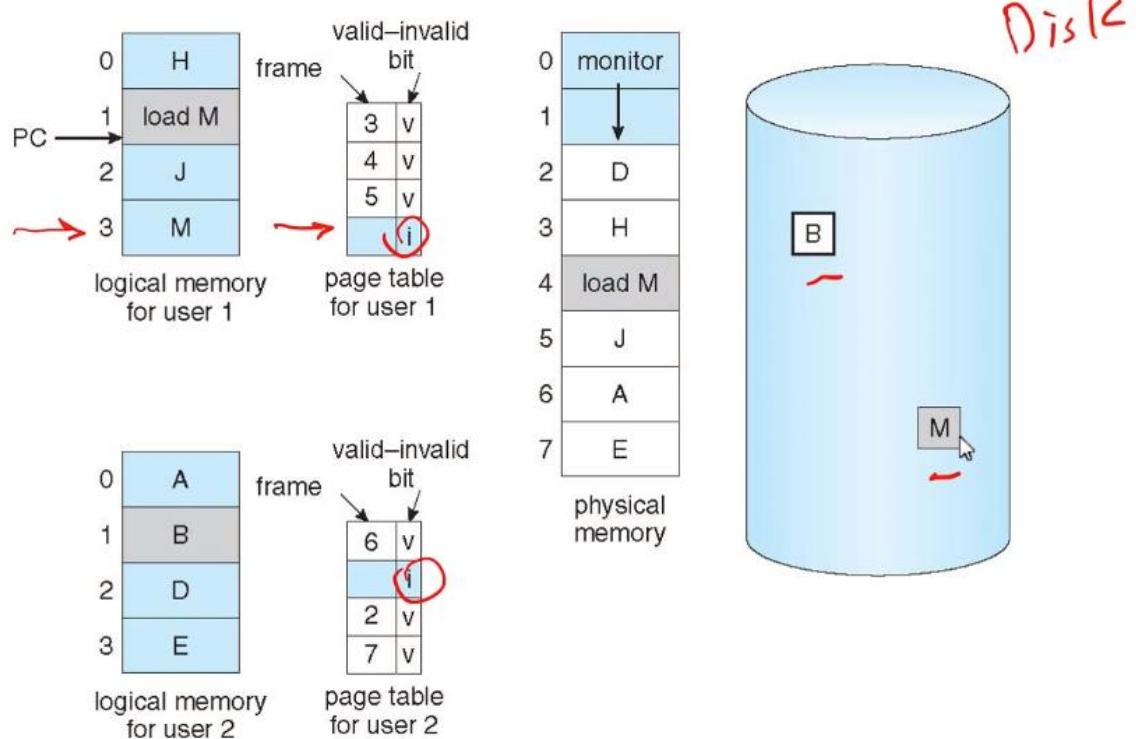
Lecture Video 12-2 (week11 – 3/21/22): Demand Paging

- What Happens if There is no Free Frame (no available block of memory)?

What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
 - Algorithm – terminate? swap out? replace the page?
 - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times
 - - Especially want to avoid same page brought into memory several times
- Need For Page Replacement

Need For Page Replacement



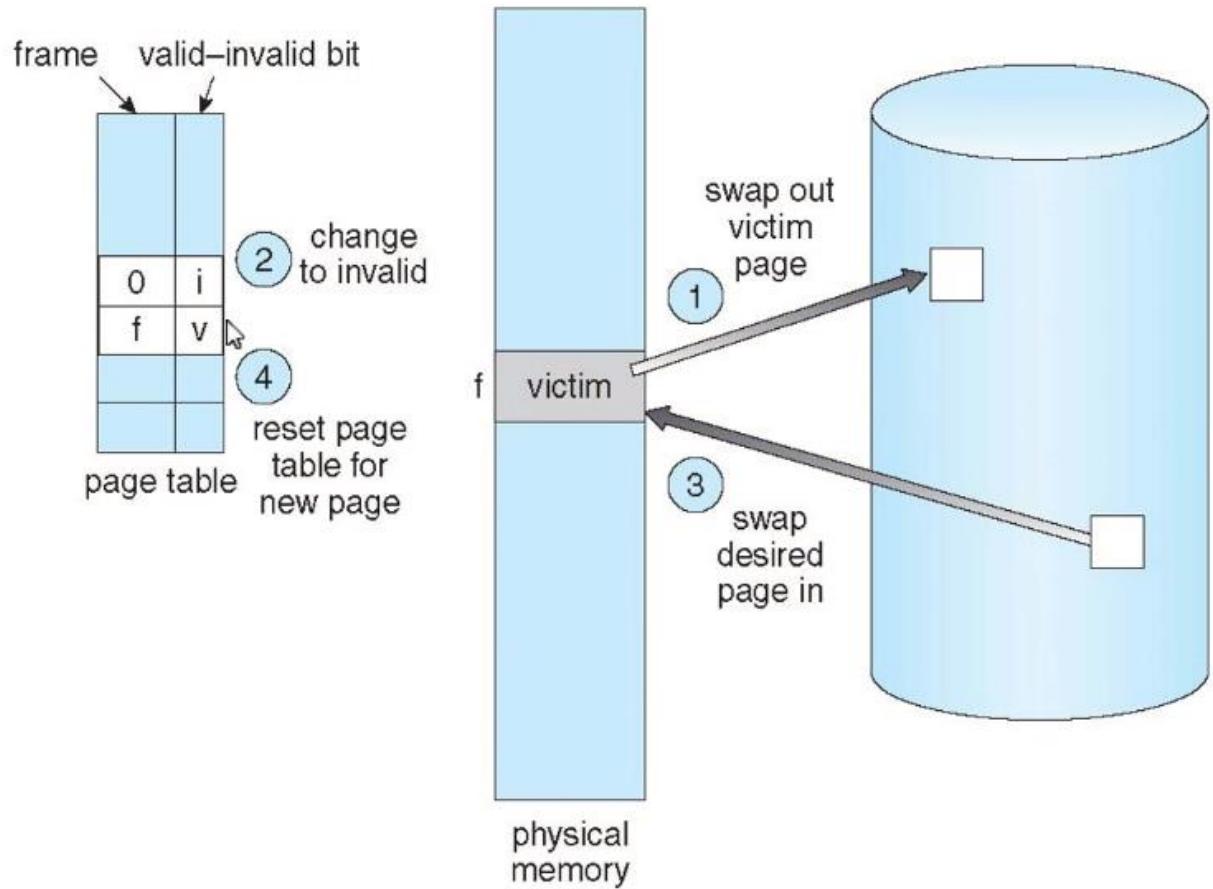
-
- Notice M and B are on the disk; thus they are not in main memory and thus since memory is full either with I/O pages or with user pages, a page needs to be replaced so that we can access M or access B. Notice how Load M is also in memory, since it is an instruction; thus we can access the instruction, but the instruction includes accessing M, which is on the disk.
- Basic Page Replacement

Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a victim frame
 - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

- So, in the previous example, load M instruction is what caused the trap, so we would have to restart that instruction since it was never executed to completion.
- Page Replacement

Page Replacement



-
- Notice the labeled steps above: 1, 2, 3, and 4.
- What Factors Lead to Misses?

What Factors Lead to Misses?

- **Compulsory Misses:**
 - Pages that have never been paged into memory before
 - How might we remove these misses?
 - Prefetching: loading them into memory before needed
 - Need to predict future somehow! More later.
- **Capacity Misses:**
 - Not enough memory. Must somehow increase size.
 - Can we do this?
 - One option: Increase amount of DRAM (not quick fix!)
 - Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!
- **Conflict Misses:**
 - Technically, conflict misses don't exist in virtual memory, since it is a "fully-associative" cache
- **Policy Misses:**
 - Caused when pages were in memory, but kicked out prematurely because of the replacement policy
 - How to fix? Better replacement policy
 - Page and Frame Replacement Algorithms

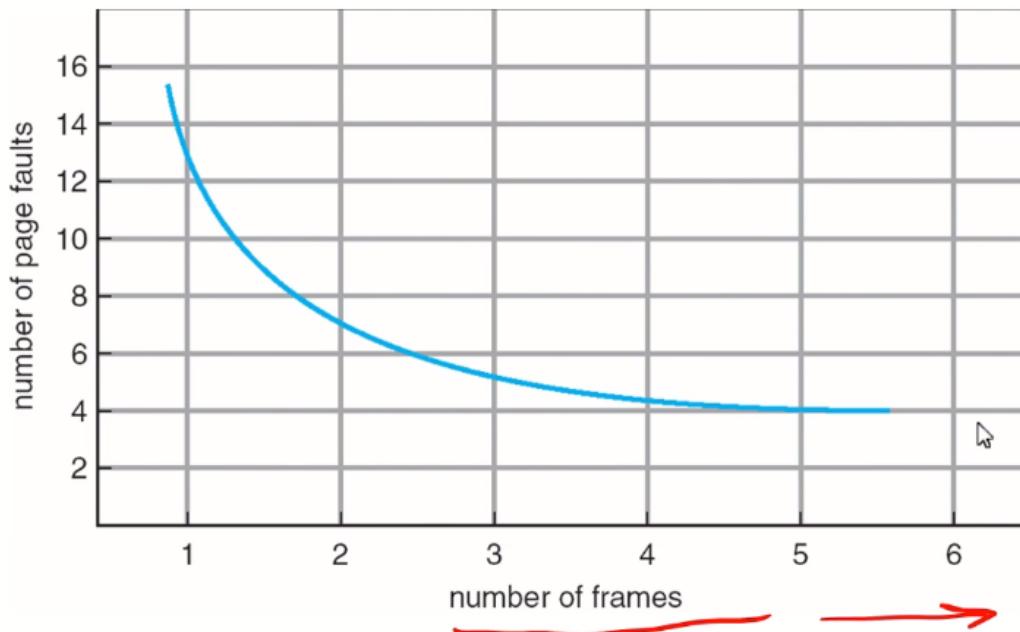
Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
 - How many frames to give each process
 - Which frames to replace
- **Page-replacement algorithm**
 - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
 - String is just page numbers, not full addresses
 - Repeated access to the same page does not cause a page fault
 - Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

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

- - Graph of Page Faults Versus The Number of Frames (for a single process, or even in general adding more total frames so that all processes can have more frames or allow to exist more processes, which can increase multiprogramming)

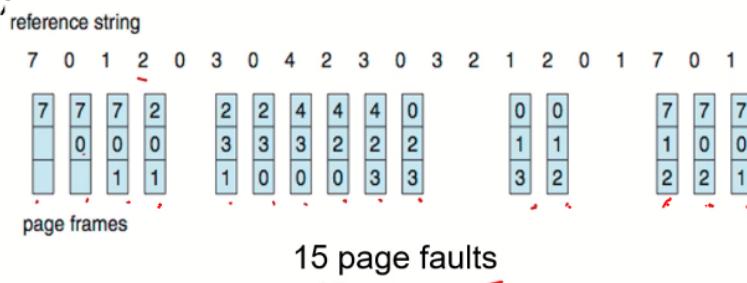
Graph of Page Faults Versus The Number of Frames



-
- First in First out (FIFO) Algorithm
 - Not as an efficient algorithm
 - Issue that can arise: first in should not be first out necessarily because what if the first in will be used frequently, and in a staggered order?
 - The example shown below shows the reference string for a single process, and how since there is only 3 available memory blocks (frames/pages), only three pages of the process at a time can be valid as the process executes.
 - Belady's Anomaly is only sometimes, but it can occur if the correct page access order occurs, as shown in the example.

First-In-First-Out (FIFO) Algorithm

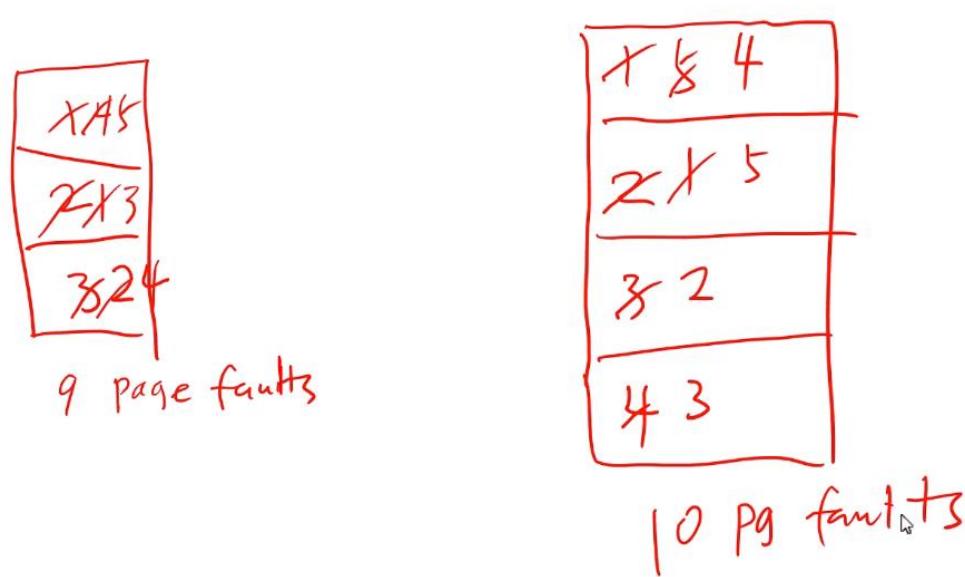
- Reference string:
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)



- Can vary by reference string: consider
1,2,3,4,1,2,5,1,2,3,4,5
 - Adding more frames can cause more page faults!
 - Belady's Anomaly
- How to track ages of pages?
 - Just use a FIFO queue

○

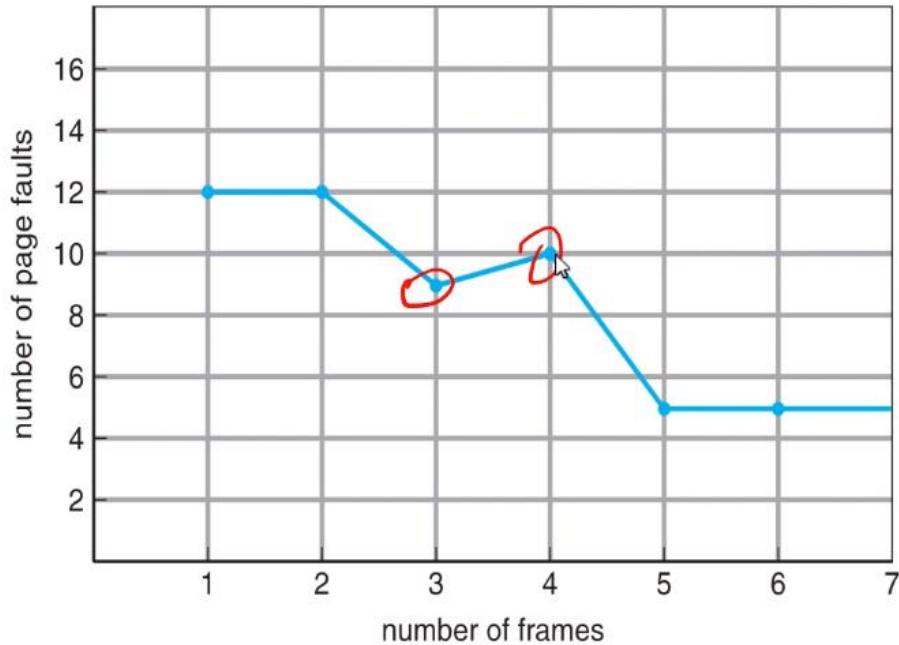
1,2,3,4,1,2,5,1,2,3,4,5



○

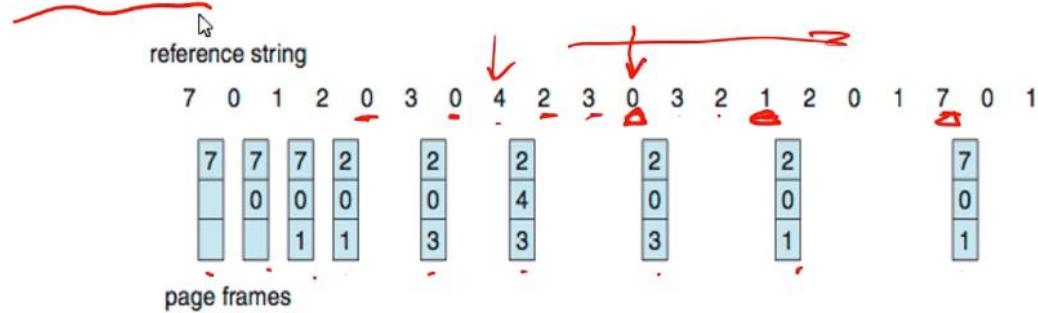
- So it is easy to implement (using queue), but not efficient.

FIFO Illustrating Belady's Anomaly



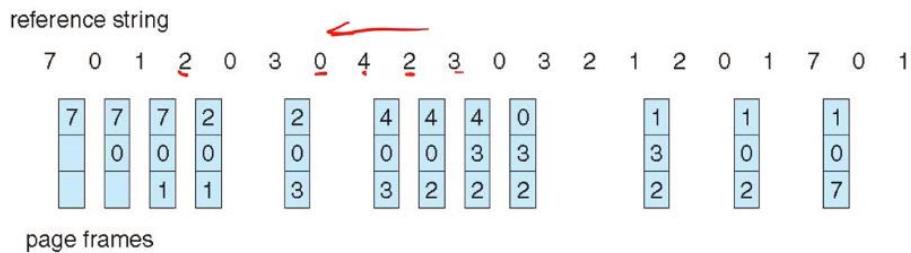
- Optimal Algorithm: MIN (minimum)
 - Use reference string as a way to look into future, and then set up page replacement such that the page that will not be used for the longest period of time (ticks perhaps) will be the victim frame.
 - But we cannot know the future, so this is only used as a standardized test to compare the performance of other prediction algorithms or even inefficient algorithms such as FIFO.

Optimal Algorithm: MIN (minimum)



- Replace page that will not be used for longest period of time
 - 9 is optimal for the example
- How do you know this?
 - Can't read the future
- Used for measuring how well your algorithm performs
 - Number of page faults in the above example is 9 (6 of which were compulsory, since the process has 6 pages).
- Least Recently Used (LRU) Algorithm
 - Use the idea of "locality" (loops, arrays, etc.)
 - These parts of code are likely to be needed again soon, but also when they have not been used recently, it can also be a good chance that loop or array in the program will no longer be referenced or any time soon (relatively speaking).
 - Simply select the least recently used page as the victim frame via using a time variable.
 - But updating the time stamp every time a process is used is an expensive operation; we need some sort of approximation...

Least Recently Used (LRU) Algorithm

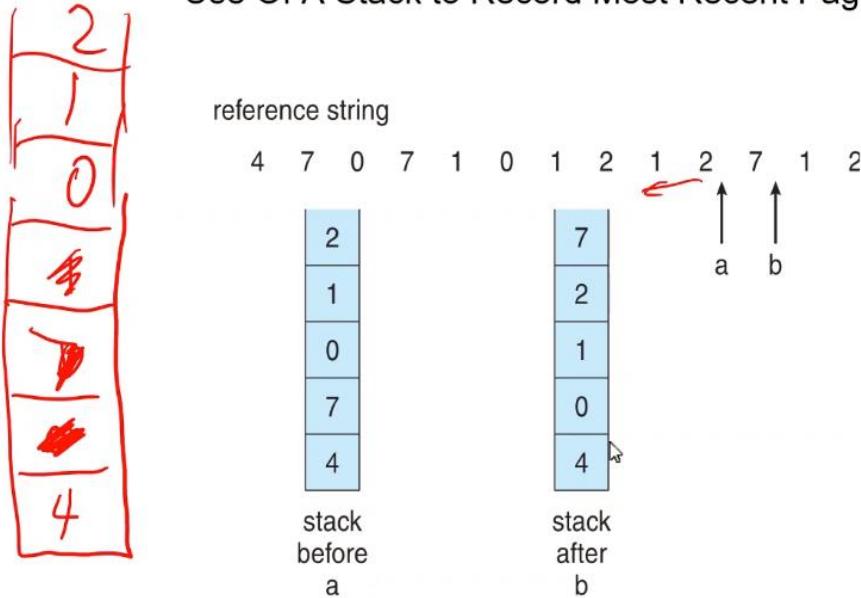


- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page
- 12 faults – better than FIFO but worse than MIN
- Generally good algorithm and frequently used
- But how to implement?
 - LRU Algorithm (cont.)
 - Stack implementation can be used, but each update is more expensive; although no search needed since bottom of stack is the page that needs to be replaced

LRU Algorithm (Cont.)

- Stack implementation
 - Keep a stack of page numbers in a double link form:
 - Page referenced:
 - move it to the top
 - requires 6 pointers to be changed
 - But each update more expensive
 - No search for replacement
- LRU and MIN are cases of **stack algorithms** that don't have Belady's Anomaly
 - Contents of memory with N frames is subset of contents of memory with N+1 frame
 - - Notice: with FIFO, the order that pages are loaded determines order of replacement, and that order can change when you add more frames, so contents of memory with N frames is not going to necessarily be a subset of the contents of memory with N+1 frames. With LRU, order of least recently used for N frames will always be a subset of the order when there are N+1 frames.
 - Using stack pointers is too expensive still because more pointer updates will be needed when there are more frames added – each frames pointer would need to be updated as to where it is on the LRU stack. We need to use some sort of approximation.
 - Example of using a stack to record most recent page references:

Use Of A Stack to Record Most Recent Page References

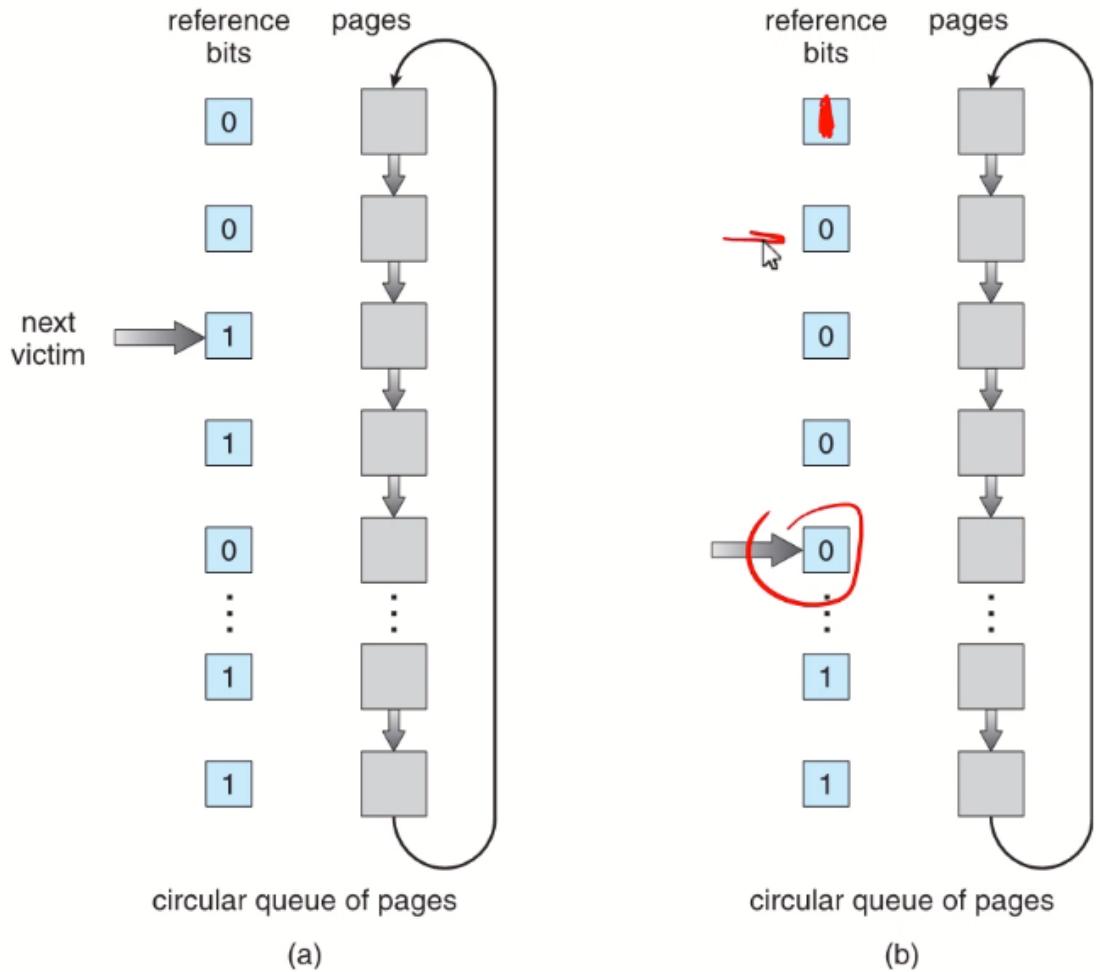


- LRU Approximation Algorithms

LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit**
 - With each page associate a bit, initially = 0
 - When page is referenced bit set to 1
 - Replace any with reference bit = 0 (if one exists)
 - We do not know the order, however
- **Second-chance algorithm**
 - Generally FIFO, plus hardware-provided reference bit
 - **Clock** replacement
 - If page to be replaced has
 - Reference bit = 0 → replace it not recently used
 - reference bit = 1 then:
 - set reference bit 0, leave page in memory
 - replace next page, subject to same rules
- Second-Chance (clock) Page-Replacement Algorithm (LRU approximation)

Second-Chance (clock) Page-Replacement Algorithm



- Enhanced Second-Chance Algorithm
 - Referenced = page was used (*referenced*); Modify = page was *modified* (dirty bit set).

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)
 - 1. (0, 0) neither recently used nor modified – best page to replace
 - 2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
 - 3. (1, 0) recently used but clean – probably will be used again soon
 - 4. (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
 - Might need to search circular queue several times
 - ○ No system actually implements LRU directly because it is still overall so expensive to track, even with the best search for recently used approximation.
 - So many times, we are just happy with (it is fast and efficient enough) finding one page that is not recently used (bit = 0); thus, we can even randomly select pages and check their recent bit (and set it to 0 if it is 1, then select another random page).

38

Lecture Video 12-3 (week11 – 3/21/22): Demand Paging

- Advanced Paging Issues

Advanced Paging Issues

- Swap Area
 - Transparency
 - Core Map
 - Global vs. Local Replacement
 - Thrashing
 - Prepaging.
-
- Swap Area
 - When you format a disk, you can reserve a swap space/area on the disk.

Swap Area

- “Swap Space” – special area on disk
 - Dedicated to holding an entire address space
 - One swap file per address space
 - Limits size of virtual memory
 - Managed differently than rest of disk
 - Know where data will go; can make disk access very fast

○

- More on Transparency
 - Page fault is totally transparent to the user process (user process cannot see/does not know anything about the page fault; it occurs and is resolved in the background by the kernel via a trap).

More on Transparency

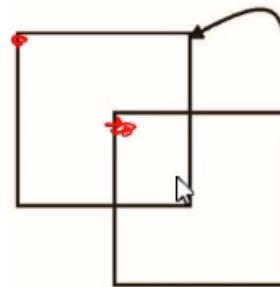
- Executing an instruction has several steps
 - Instruction Fetch
 - Instruction Decode
 - Execute *Load* *Store* ↗
- Page fault can happen at fetch or execute
 - So in general, just restart whole instruction

Block copies are a problem, as is autoincrement

- Instruction Restart
 - With the advanced block move instruction, which is rarely used as is with other advanced instructions, we cannot know how much of the block was copied before a fault occurred; so if fault occurs, we just have to restart the entire execution.

Instruction Restart

- Consider an instruction that could access several different locations
 - block move



- auto increment/decrement location
- Restart the whole operation?
 - What if source and destination overlap?

- Recall: Core Map

Recall: Core Map

- Maps physical frame to virtual page
- Very useful in page replacement
 - Just scan through each frame
 - Wouldn't want to go through every single address space, because could be many
 - What about sharing?
 - One physical frame can be used by several address spaces
 - ▪ Potential problem with core map: multiple virtual pages can share the same physical address (frame). So how do we know which virtual page is being referenced since they are mapped to the same frame?
- Global vs. Local Replacement
 - One hog that has too many frames will cause too many replacements and will slow down the overall system performance.

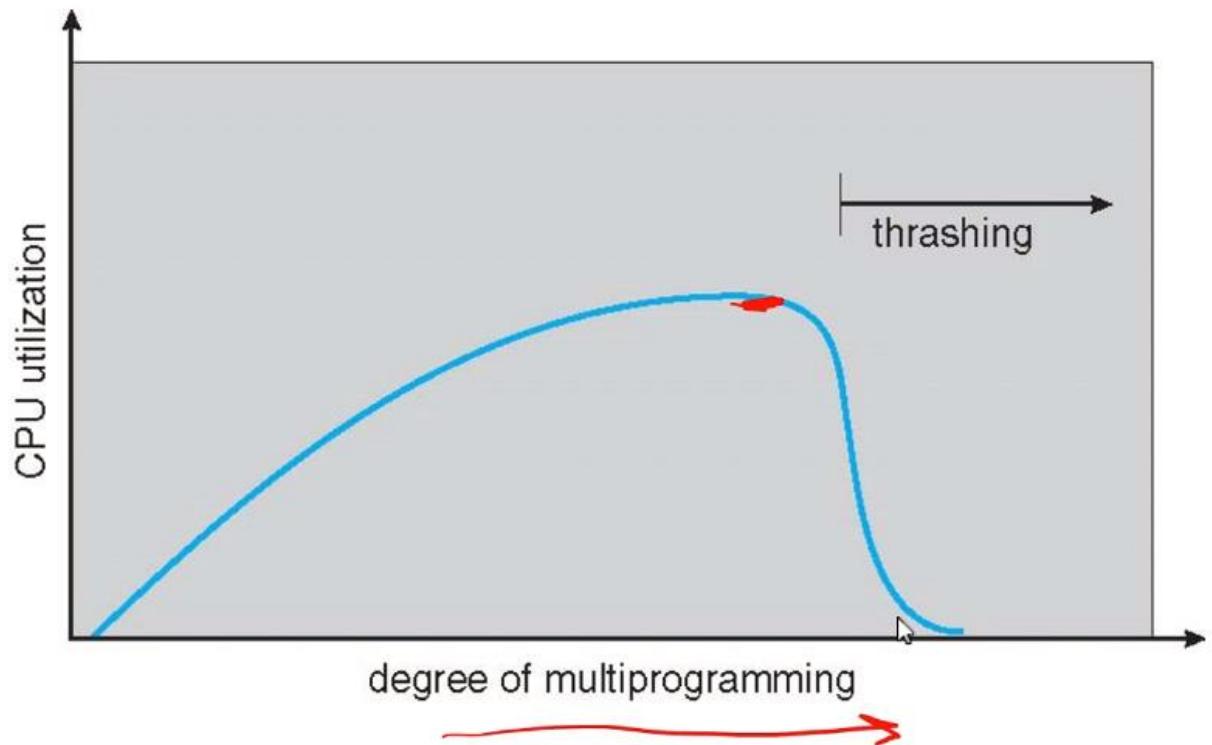
Global vs. Local Replacement

- Should OS reclaim
 - a) Most appropriate frame being used by the faulting address space? (“local replacement”)
 - b) Most appropriate frame in memory? (“global replacement”)
- Better throughput with global replacement
- One hog can ruin everything, however.
 - Local: if a process needs one of its pages, then replace one of its current loaded pages with another one.
 - Global: if a process needs one of its pages, replace any frame in the entire memory – whether it be a page that is a part of the same process, or another process.
 - Most systems use global replacement.
 - This is why a hog can reduce the performance of the entire system.
 - If it were local, the process would be limited to only slowing its own performance down – not the entire system (all other processes/threads that can be context switched in or threaded in parallel, etc.).
 - So with global replacement, there is no proper isolation between processes (and their associated frames), which is why hogs can arise.
- Thrashing

Thrashing

- Pages replaced when still needed
 - Thread can spend more time page faulting than getting any useful work done
- Very common on timesharing systems
 - Ex: log more and more users into the system, eventually:
total # of pages needed > total # of pages available
- Adding more processes can actually decrease CPU utilization
- Need to figure out needs of a process
 -

Thrashing (Cont.)

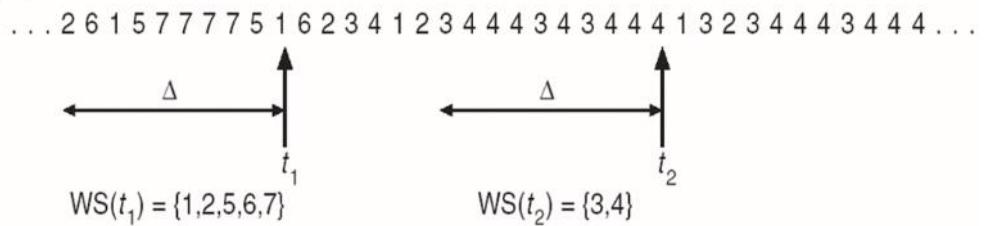


- Working-Set Model
 - Figure out how many pages a process needs.
 - Need: (instructions, used data, stack and heap)
 - But some processes have data that is spread out among many different pages, or have instructions that cause the program to jump around a lot; or could need more pages to manipulate or read a file placed in memory (for faster access, especially when it will be frequent).
 - Working set is different from process to process.

Working-Set Model

- Working Set (Denning, MIT, mid-60's)
 - Informally, collection of pages process is using right now
- $\Delta \equiv$ working-set window \equiv a fixed number of page references
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
- $D = \sum WSS_i \equiv$ total demand frames, approximation of locality
- if $D > m$ (the total number of physical frames) \Rightarrow Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

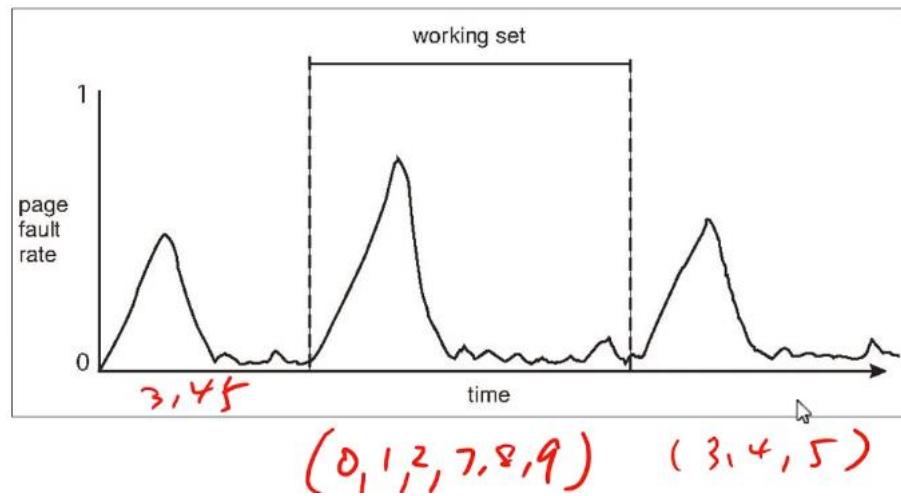
page reference table



- $D = \sum WSS_i$
- $D =$ sum of all processes working set window in the system over a given interval of time (delta). Gives you essentially the set of all pages used, and thus the total number of unique pages referenced by all processes, and thus the total number of pages needed by all processes at a given time in the system.
- Tracking working set of all processes is not that easy.
- Working Sets and Page Fault Rates
 - For a process, initially when you switch from one set of pages to another, initially (of course) there will be a lot of page faults, since the new needed set is not in memory.
 - This is why you will see lots of peaks and valleys for the page fault v working set over the lifetime of a process.

Working Sets and Page Fault Rates

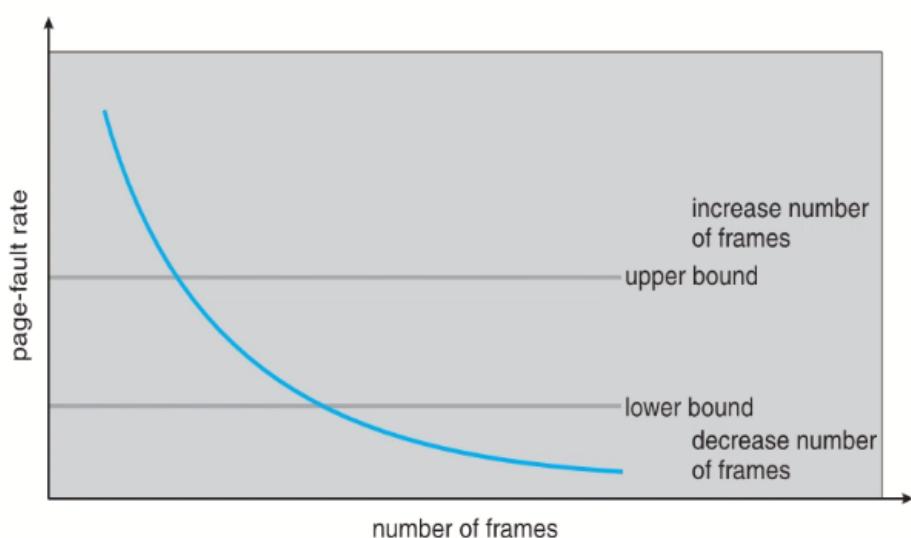
- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



- Page-Fault Frequency
 -

Page-Fault Frequency

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



- - What about Compulsory Misses?
 - Clustering tries to take advantage of the principle of locality.
 - Locality: pages nearby likely needed in the future; due often because of looped code, or arrays, etc.
 - Working set tracking: track working set so when one of the pages from that set is loaded in memory, then when the process is swapped back in, also swap not only that single page, but all of the pages of the working set that the page belongs to (if there is enough frames available..etc.).

What about Compulsory Misses?

- Recall that compulsory misses are misses that occur the first time that a page is seen
 - Pages that are touched for the first time
 - Pages that are touched after process is swapped out/swapped back in
 - Clustering:
 - On a page-fault, bring in multiple pages “around” the faulting page
 - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
 - Working Set Tracking:
 - Use algorithm to try to track working set of application
 - When swapping process back in, swap in working set
 $(3, 4)$ $(0, 1, 2, 7, 8, 9)$

structure
For the example, each int is 4 bytes, and assume each page is $128 \times 4 = 512$ bytes; thus we need 128 pages total for the program.
For program 1, we have to do a lot of page replacements because we are accessing the array column-wise.
Program 2 accesses the array row-wise, which is how each page stores the memory, thus there is only 128 faults – significantly less!
This demonstrates that no matter what page-fault algorithms is used, the performance of a process largely depends on how its program is written.

 - Often times, when code is written inefficiently (or perhaps it is written with optimization for a particular system in mind), the compiler will often have an optimization algorithm to optimize the program to best match the memory architecture of the system that it will run on.

Program Structure

- Writing stupid programs => slow execution

- int[128,128] data;
- Each row is stored in one page
- Program 1

```
for (j = 0; j < 128; j++)
for (i = 0; i < 128; i++)
    data[i, j] = 0;
```

128 x 128 = 16,384 page faults

- Program 2

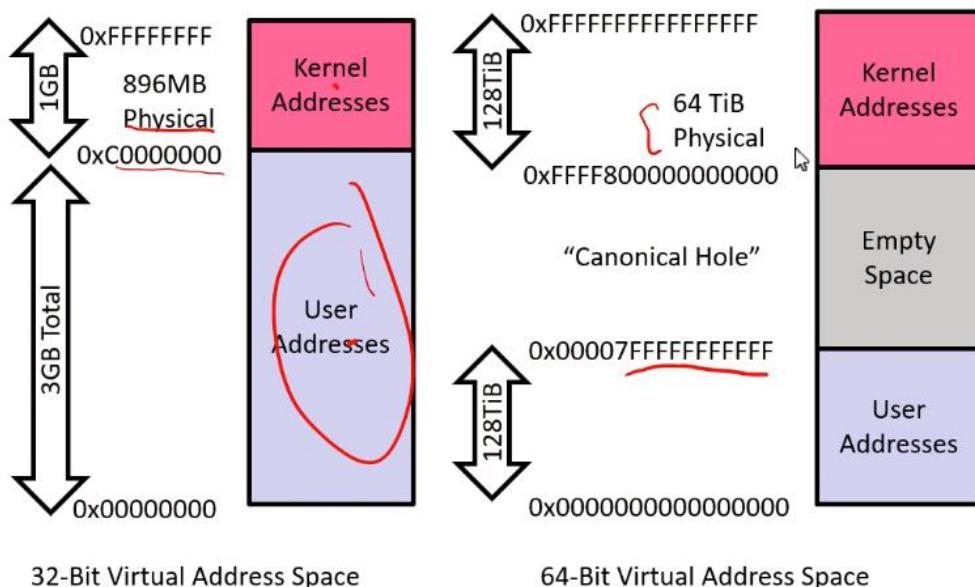
```
for (i = 0; i < 128; i++)
for (j = 0; j < 128; j++)
    data[i, j] = 0;
```

128 page faults

51

- Linux Virtual Memory Map (Pre-Meltdown)

Linux Virtual Memory Map (Pre-Meltdown)



○

- The kernel maps to each user's address space.

Pre-Meltdown Virtual Map (Details)

- Kernel memory not generally visible to user
 - Exception: special VDSO (virtual dynamically linked shared objects) facility that maps kernel code into user space to aid in system calls (and to provide certain actual system calls such as `gettimeofday()`)
- Every physical page described by a “page” structure
 - Collected together in lower physical memory
 - Can be accessed in kernel virtual space
- For 32-bit virtual memory architectures:
 - When physical memory < 896MB
 - All physical memory mapped at 0xC0000000
 - When physical memory \geq 896MB
 - Not all physical memory mapped in kernel space all the time
 - Can be temporarily mapped with addresses $>$ 0xCC000000
- By “physical memory”, it means a “virtual” address that directly maps to the same physical memory location; so its basically a 1:1 address, but many times it is still considered virtual space, such as virtual kernel address space so users can access system call section of the kernel code.
- Post Meltdown Memory Map
 - Speculative execution = auto execution in order to speed up execution; most cpu can try to run speculatively – if speculation was incorrect, then the executions can roll back.
 - This is not visible to the user process, until the speculation is confirmed as correct.
 - However there is a side effect: when you do speculative execution, data will be loaded from memory to cache, etc; the data is not accessible by user process, however, the fact that the data is loaded into the cache – cannot be rolled back. So it can be rolled back if necessary from a process/main memory, but the cached data for the execution of the process cannot be rolled back – cached data is much faster, so it was good especially for speculative execution, but the kernel map was left in the cache as well – vulnerable memory location. By exploiting this bug, a user process is able to see all of the contents of the kernel memory. So if you shared a virtual machine in the cloud, people could read your kernel's content in the cloud.
 - This bug was identified in 2017. But they kept it secret from the public until 2018 when the issue was fixed by manufacturers and had some time to permeate.

2017

Post Meltdown Memory Map ^{Spectre}

- Meltdown flaw (2018, Intel x86, IBM Power, ARM)
 - Exploit speculative execution to observe contents of kernel memory

```

1: // Set up side channel (array flushed from cache)
2: uchar array[256 * 4096];
3: flush(array); // Make sure array out of cache
4: try { // ... catch and ignore SIGSEGV (illegal access)
5:   uchar result = *(uchar *)kernel_address; // Try access!
6:   uchar dummy = array[result * 4096]; // leak info!
7: } catch (...) // Could use signal() and setjmp/longjmp
8: // scan through 256 array slots to determine which loaded

```

100% +4
← cached
- Some details:
 - Reason we skip 4096 for each value: avoid hardware cache prefetch
 - Note that value detected by fact that one cache line is loaded
 - Catch and ignore page fault: set signal handler for SIGSEGV, can use setjmp/longjmp....
- **Patch:** Need different page tables for user and kernel
 - Without PCID tag in TLB, flush TLB twice on syscall (800% overhead!) Pid
 - Need at least Linux v 4.14 which utilizes PCID tag in new hardware to avoid flushing when change address space
- **Fix:** better hardware without timing side-channels
 - Will be coming, but still in works
- Simple fix: don't map the kernel to the user's address space anymore (as was done pre-meltdown).
 - Need different page tables for user and kernel.
 - Need to flush when no PCID (PID) present (occurs during speculative execution); but costs lots of overhead (unless you avoid double-flushing via the latest Linux versions that is deployed on newer hardware which contain the new PCID tags).
 - This causes overall slower performance since kernel now uses different page table than the user processes.
 - System code now more expensive; but a future fix that is more efficient will nullify the performance drop for the security patch.
- Summary
 - Thrashing: swap out process == free up memory so that a process that needs more frames can run (since with more frames available, it can now fit its working set in memory).

Summary

- Demand paging follows that pages should only be brought into memory if the executing process demands them.
- Replacement policies
 - FIFO: Place pages on queue, replace page at end
 - MIN: Replace page that will be used farthest in future
 - LRU: Replace page used farthest in past
- Clock Algorithm: Approximation to LRU
 - Arrange all pages in circular list
 - Sweep through them, marking as not “in use”
 - If page not “in use” for one pass, then can replace
- Working Set:
 - Set of pages touched by a process recently
- Thrashing: a process is busy swapping pages in and out
 - Process will thrash if working set doesn’t fit in memory
 - Need to swap out a process

○

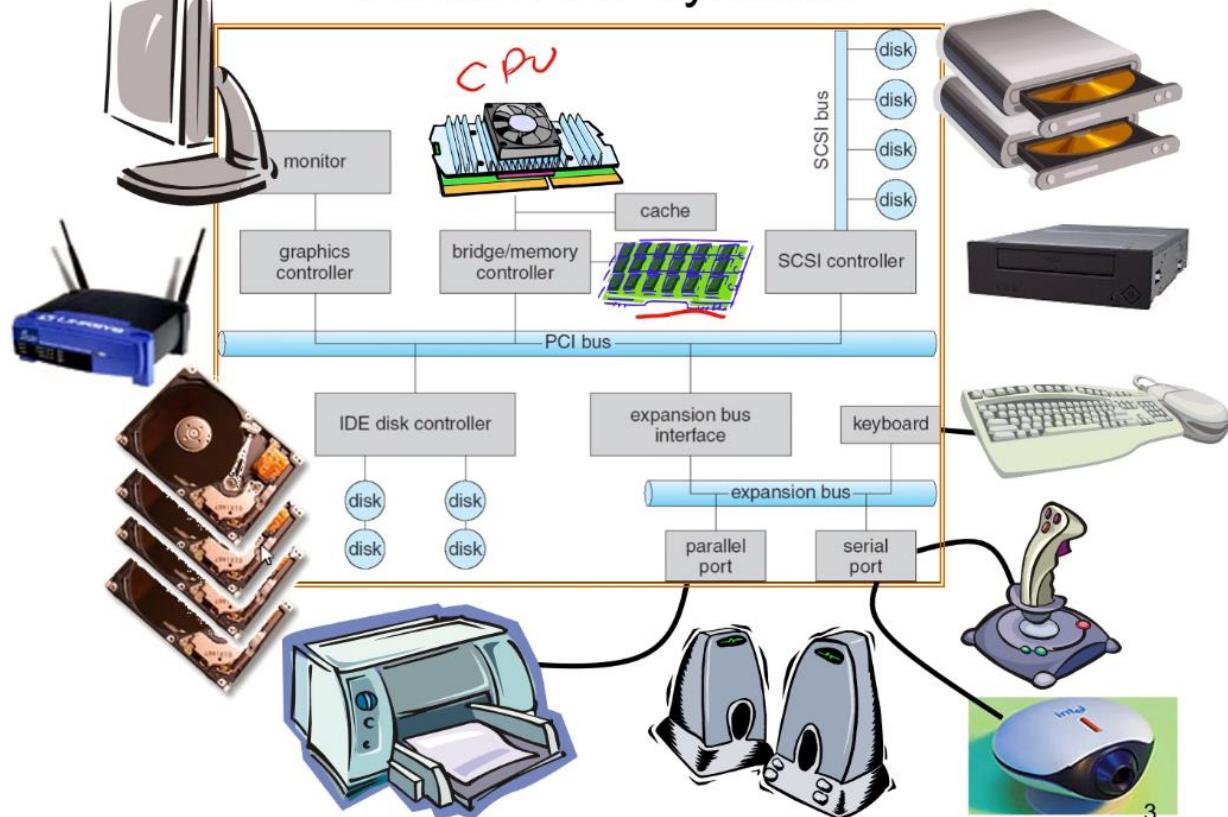
Lecture Video 13-1 (week12 – 3/28/22): I/O System

- The Requirements of I/O

The Requirements of I/O

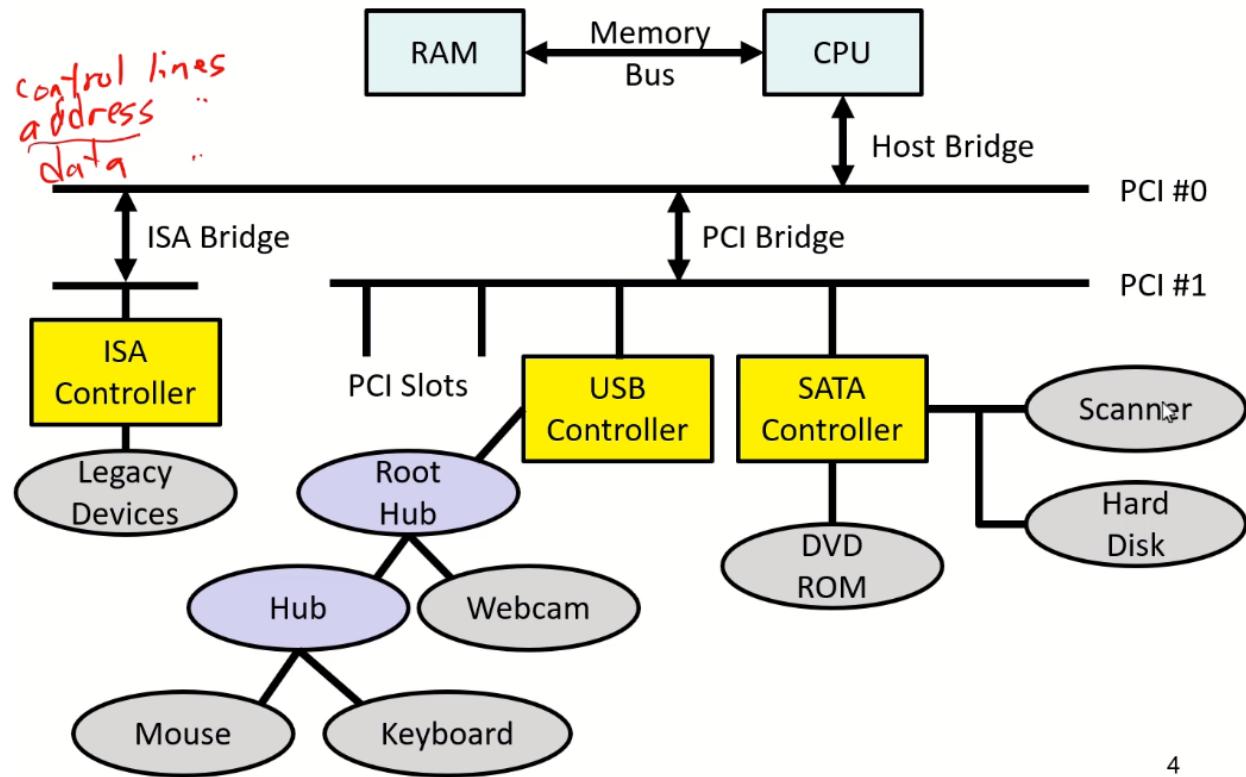
- What about I/O?
 - Without I/O, computers are useless (disembodied brains?)
 - But... thousands of devices, each slightly different
 - How can we standardize the interfaces to these devices?
 - Devices unreliable: media failures and transmission errors
 - How can we make them reliable???
 - Devices unpredictable and/or slow
 - How can we manage them if we don't know what they will do or how they will perform?
- Some operational parameters:
 - Byte/Block
 - Some devices provide single byte at a time (e.g. keyboard)
 - Others provide whole blocks (e.g. disks, networks, etc)
 - Sequential/Random
 - Some devices must be accessed sequentially (e.g. tape)
 - Others can be accessed randomly (e.g. disk, cd, etc.)
 - Polling/Interrupts
 - Some devices require continual monitoring
 - Others generate interrupts when they need service
- - Modern I/O Systems

Modern I/O Systems



- Peripheral Component Interconnect (PCI) Architecture
 - What is a bus?: it is a set of wires that are control lines for reading a writing communications between devices/components.

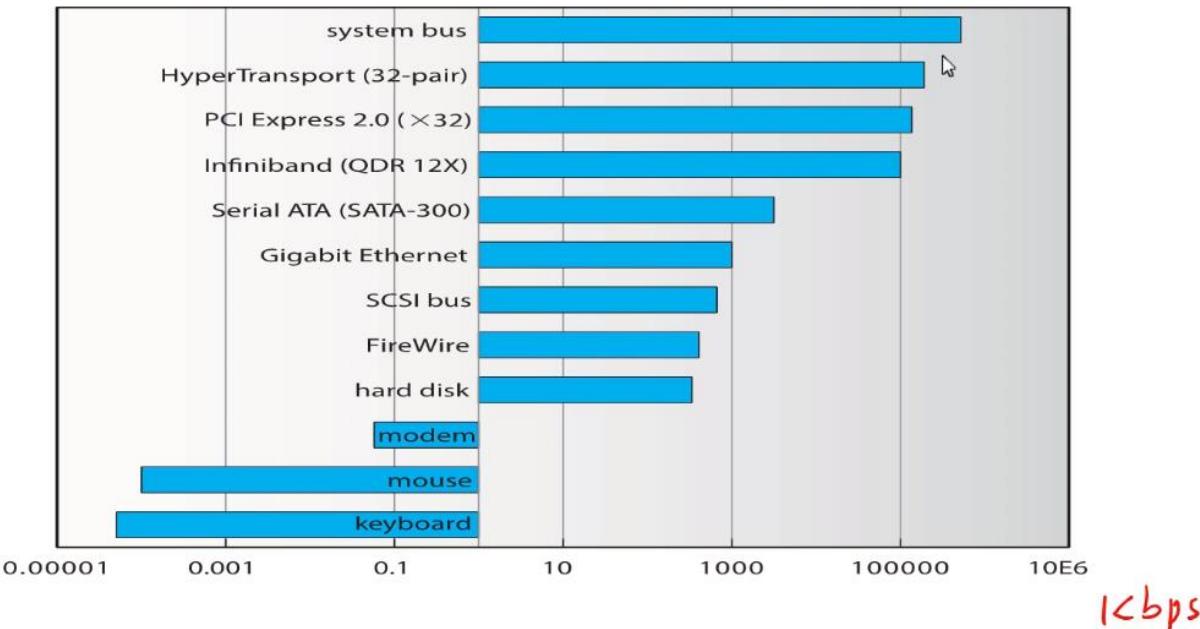
Peripheral Component Interconnect (PCI) Architecture



4

- Sun Enterprise 6000 Device-Transfer Rates
 -

Sun Enterprise 6000 Device-Transfer Rates



1Kbps

5

- Device Rates vary over 12 orders of magnitude
 - System better be able to handle this wide range
 - Better not have high overhead/byte for fast devices!
 - Better not waste time waiting for slow devices
-
- The Goal of the I/O Subsystem (I/O is part of the OS – it is a subset of it)
 - Most I/O devices are treated like a file (including for networking I/O, console window for std out, etc.).
 - (fd name given in the example is short for “file descriptor”).
 - Once you get the reference to the file descriptor, you can obviously read and write to it..etc.
 - Treating all IO like files gives a standard interface to all IO devices; applications do not need to know how a device works; we just need an interface to do input and output – no matter the device.

The Goal of the I/O Subsystem

- Provide Uniform Interfaces, Despite Wide Range of Different Devices

– This code works on many different devices:

```
FILE fd = fopen("/dev/something", "rw");
for (int i = 0; i < 10; i++) {
    fprintf(fd, "Count %d\n", i);
}
close(fd);
```

~~fread~~
~~fwrite~~

– Why? Because code that controls devices (“device driver”) implements standard interface.

- We will try to get a flavor for what is involved in actually controlling devices in rest of lecture.

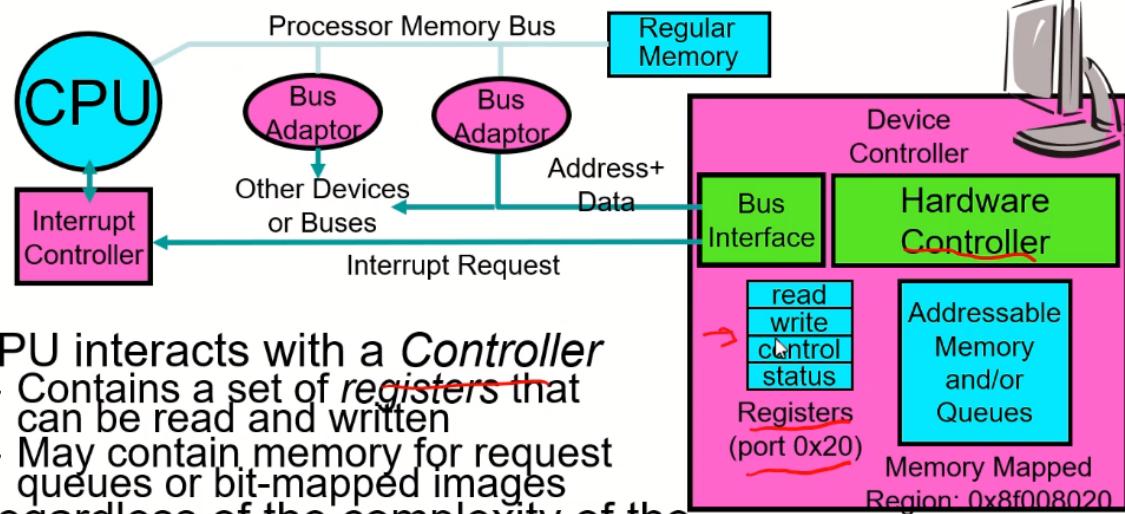
- Device driver actually controls the device.
- Want Standard Interfaces to Devices
 - Most often, a block is the same size as the size of a page for a given system (typically 4KB).
 - Each type of device has its own interface, but it is all a standardized file interface; for example, the socket interface is a type of file (network file) that can send and receive messages to local or nonlocal devices.
 - Select() system call allows for monitoring multiple sockets in parallel.

Want Standard Interfaces to Devices

- **Block Devices:** e.g. disk drives, DVD-ROM
 - Access blocks of data
 - Commands include `open()`, `read()`, `write()`, `seek()`
 - Raw I/O or file-system access
 - Memory-mapped file access possible *Load, store*
- **Byte Devices:** e.g. keyboards, mice, serial ports, some USB devices
 - Single characters at a time
 - Commands include `get()`, `put()`
 - Libraries layered on top allow line editing
- **Network Devices:** e.g. Ethernet, Wireless, Bluetooth
 - Different enough from block/character to have own interface
 - Unix and Windows include socket interface *Send recy*
 - Separates network protocol from network operation
 - Includes select() functionality

-
- How Does User Deal with Timing?
 - By default, most devices are blocking.
 - `ioctl()` system call says to make a device/operation/interface to be nonblocking.
 - Usually will cause no changes because it is often waiting on user input, but it will continue to run (without interrupting user process to complete its operation; the interface waits in line for the processor just like everyone else without putting user process who called for the I/O to be put to sleep.).
 - Asynchronous is nonblocking as well, and similarly user process continues to run, but the user process will be called back (via `callback()` function) when the I/O process (kernel-level thread/process) completes. So instead of continuously read or writing, even if no data to read or write, the I/O process simply waits for some I/O state/condition, and when it is fulfilled, then it calls user process to let it know its done; but user process is never blocked during this period, and again, the I/O process waits in line just like everyone else for the CPU.
 - Good for networking interfaces.
 - How does the processor actually talk to the device?
 - Memory mapped I/O is mainly for displays/monitors.
 - Makes sense, because the screen is always on.

How does the processor actually talk to the device?

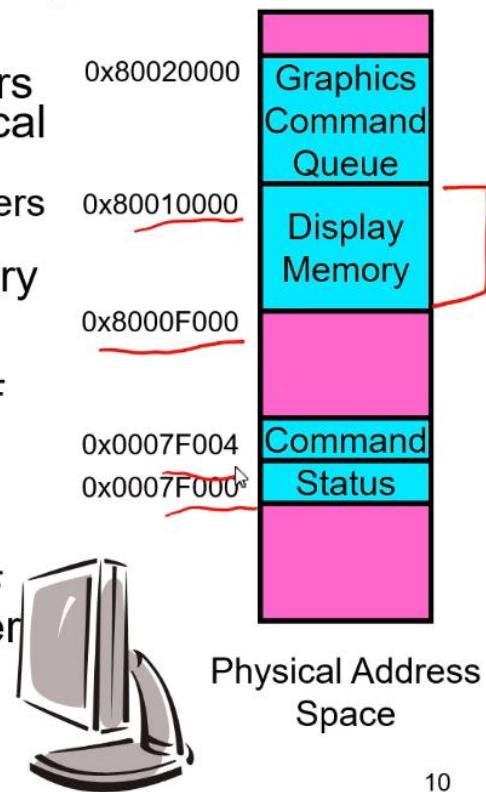


- CPU interacts with a *Controller*
 - Contains a set of *registers* that can be read and written
 - May contain memory for request queues or bit-mapped images
- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
 - **I/O instructions:** in/out instructions *in 0x20, EDX*
 - **Memory mapped I/O:** load/store instructions

- Example: Memory-Mapped Display Controller
 - For example: every pixel on a screen is 3 bytes.
 - So update these 3 bytes, you can update a pixel.
 - Often, displays have two display memories, so you can switch from one to the other very quickly (thinking about the source input from TV screen, etc.).

Example: Memory-Mapped Display Controller

- **Memory-Mapped:**
 - Hardware maps control registers and display memory into physical address space
 - Addresses set by hardware jumpers or programming at boot time
 - Simply writing to display memory (also called the “frame buffer”) changes image on screen
 - Addr: 0x8000F000—0x8000FFFF
 - Writing graphics description to command-queue area
 - Say enter a set of triangles that describe some scene
 - Addr: 0x80010000—0x8001FFFF
 - Writing to the command register may cause on-board graphics hardware to do something
 - Say render the above scene
 - Addr: 0x0007F004
 - ○ Command address space ex: display memory 1, or display memory 2 (change sources, essentially of which memory is to be output to the screen).
 - This memory is on the actual graphics card of the device, and the CPU can directly update the hardware memory (registers) of the device.
- Transferring Data To/From Controller



Transferring Data To/From Controller

- **Programmed I/O:**

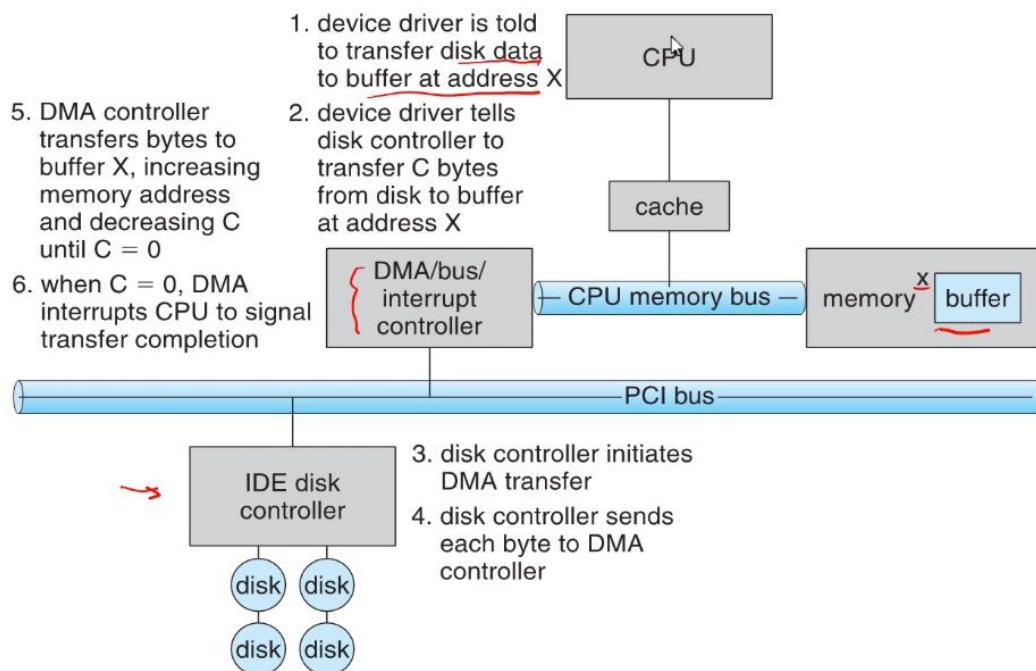
- Each byte transferred via processor in/out or load/store
- Pro: Simple hardware, easy to program
- Con: Consumes processor cycles proportional to data size

- **Direct Memory Access:**

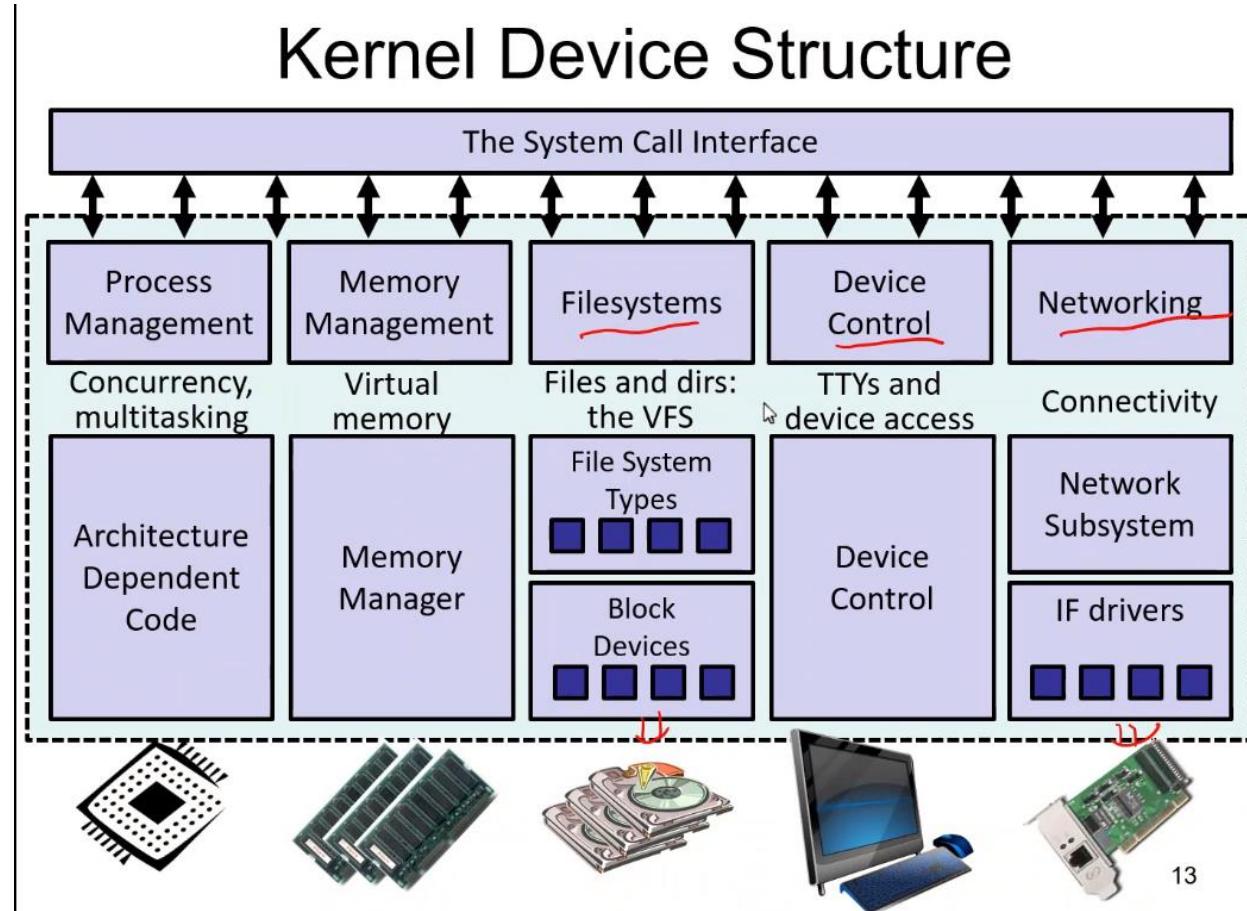
- Give controller access to memory bus
- Ask it to transfer data to/from memory directly

- Six Step Process to Perform DMA (direct memory access) Transfer
 - CPU and PCI busses usually 32 bit, and parallel (can read and write simultaneously).
 - DMA controller a very simple processor unit.

Six Step Process to Perform DMA Transfer



- Kernel Device Structure

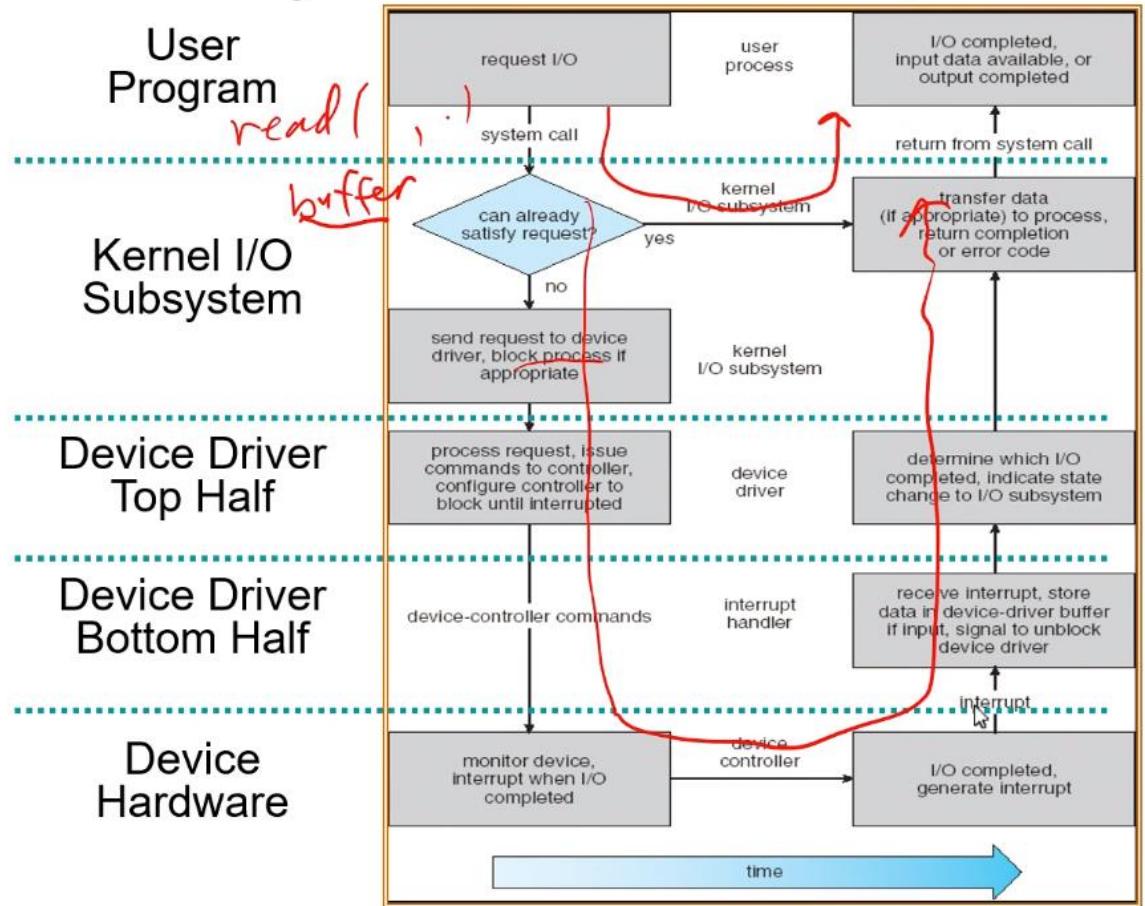


-
- Notice the subsections; you have the half that interfaces with system call/user interface; and you have the half that interfaces with the device directly.
- Device Drivers

Device Drivers

- **Device Driver:** Device-specific code in the kernel that interacts directly with the device hardware
 - Supports a standard, internal interface
 - Same kernel I/O system can interact easily with different device drivers
 - Special device-specific configuration supported with the `ioctl()` system call
- **Device Drivers** typically divided into two pieces:
 - Top half: accessed in call path from system calls
 - implements a set of **standard, cross-device calls** like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
 - This is the kernel's interface to the device driver
 - Top half will *start* I/O to device, may put thread to sleep until finished
 - Bottom half: run as interrupt routine
 - Gets input or transfers next block of output
 - May wake sleeping threads if I/O now complete
- So top half is standard interface part of the driver (this section will have similar code in it for all drivers for a given OS), bottom half is the part written by manufacturer specifically to control the type of device the driver is written for.
- Life Cycle of An I/O Request

Life Cycle of An I/O Request



- I/O Device Notifying the OS
 - Remember, with interrupts you have to save the state of the current process, then load the kernel-level interrupt code; this has relatively high overhead.
 - Polling directly checks device-specific status registers, thus low overhead.

I/O Device Notifying the OS

- The OS needs to know when:
 - The I/O device has completed an operation
 - The I/O operation has encountered an error
 - **I/O Interrupt:**
 - Device generates an interrupt whenever it needs service
 - Handled in bottom half of device driver
 - Often run on special kernel-level stack
 - Pro: handles unpredictable events well
 - Con: interrupts relatively high overhead
 - **Polling:**
 - OS periodically checks a device-specific status register
 - I/O device puts completion information in status register
 - Could use timer to invoke lower half of drivers occasionally
 - Pro: low overhead
 - Con: may waste many cycles on polling if infrequent or unpredictable I/O operations
 - Actual devices combine both polling and interrupts
 - For instance: High-bandwidth network device:
 - Interrupt for first incoming packet
 - Poll for following packets until hardware empty
-
- Kernel I/O Subsystem

Kernel I/O Subsystem

- Scheduling
 - Some I/O request ordering via per-device queue
 - Some OSs try fairness
- Buffering - store data in memory while transferring between devices
 - To cope with device speed mismatch
 - To cope with device transfer size mismatch
 - To maintain “copy semantics”
 - CPU is often faster than I/O device, but can often have an inadequate buffer size, so we may need larger buffer (block devices) and also synchronize the speed using buffers as well.

Kernel I/O Subsystem

- Caching - fast memory holding copy of data
 - Always just a copy
 - Key to performance
- Spooling - hold output for a device
 - If device can serve only one request at a time
 - i.e., Printing
- Device reservation - provides exclusive access to a device
 - System calls for allocation and deallocation
 - Watch out for deadlock
- Error Handling

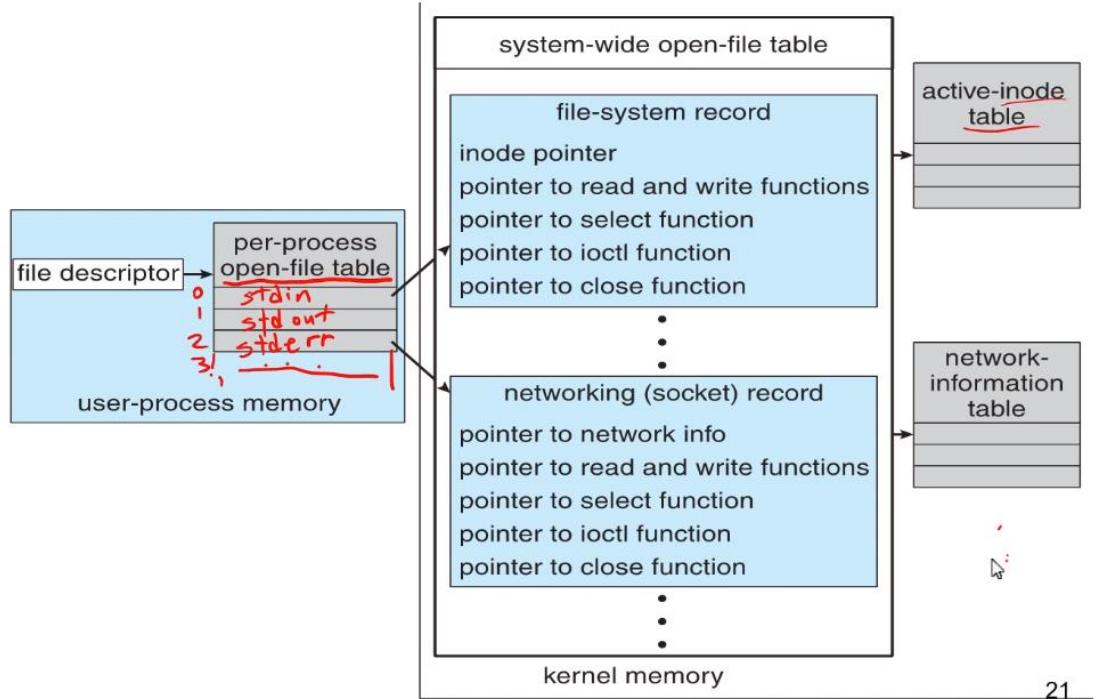
Error Handling

- OS can recover from disk read, device unavailable, transient write failures
- Most return an error number or code when I/O request fails
- System error logs hold problem reports
 - Kernel Data Structures
 - Cons with message passing: relatively slow due to system call being relatively slow.

Kernel Data Structures

- Kernel keeps state info for I/O components, including open file tables, network connections, character device state
- Many, many complex data structures to track buffers, memory allocation, “dirty” blocks
- Some use object-oriented methods and message passing to implement I/O
 - Windows uses message passing
 - Message with I/O information passed from user mode into kernel
 - Message modified as it flows through to device driver and back to process
 - Pros / cons?
- UNIX I/O Kernel Structure
 - Remember, we abstract each device as a separate file: some are real files on storage devices, others are interfaces with I/O devices (device files).

UNIX I/O Kernel Structure



Lecture Video 13-2 (week12 – 3/28/22): I/O System

- What Determines Bandwidth for I/O?
 - Depends on bus speed.

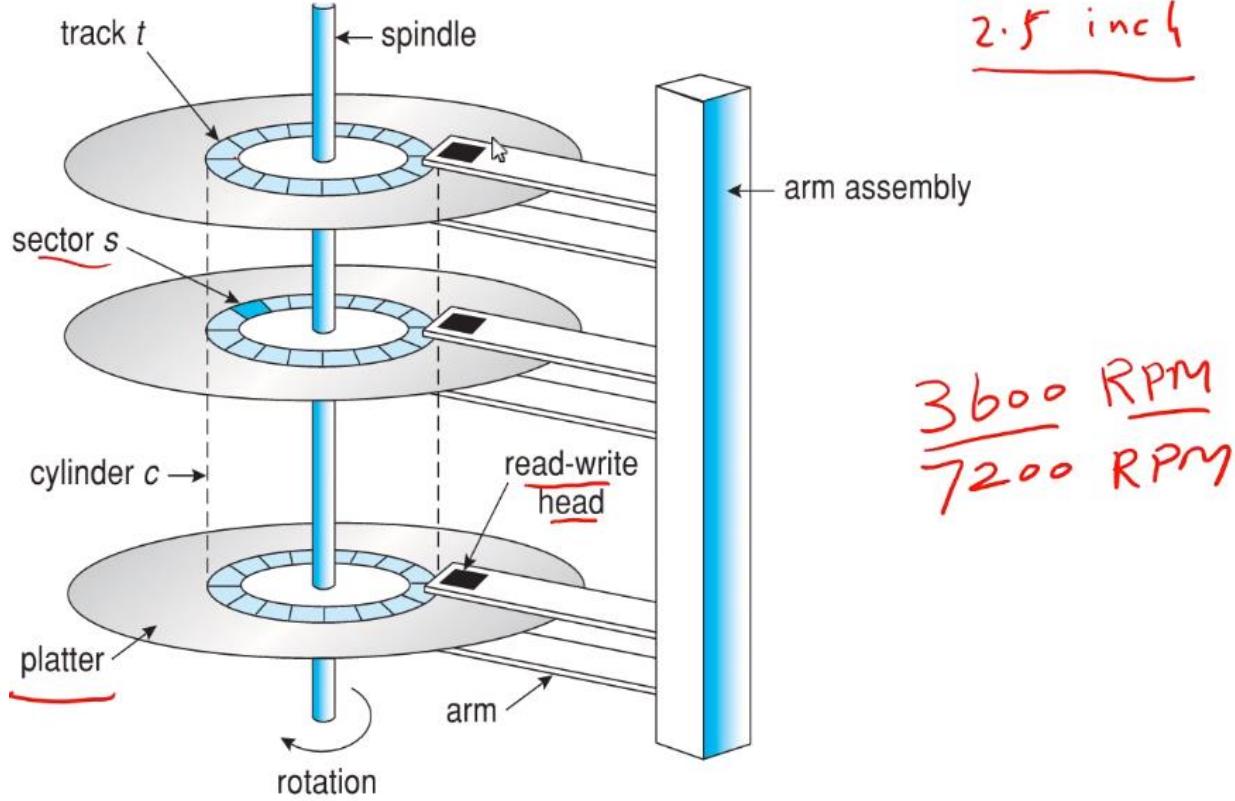
What Determines Peak BandWidth for I/O ?

- Bus Speed
 - PCI-X: 1064 MB/s = 133 MHz x 64 bit (per lane)
 - ULTRA WIDE SCSI: 40 MB/s
 - Serial Attached SCSI & Serial ATA & IEEE 1394 (firewire): 1.6 Gb/s full duplex (200 MB/s)
 - USB 3.0 – 5 Gb/s
 - Thunderbolt 3 – 40 Gb/s
- Device Transfer Bandwidth
 - Rotational speed of disk
 - Write / Read rate of NAND flash
 - Signaling rate of network link
- Whatever is the bottleneck in the path...
 - Signal rate (equivalent/essentially clock speed of the network device).
 - Rotational disk is usually the bottleneck, but for SSD (NAND flash), the bus speed is actually the bottleneck (slowest speed in the path).
- Storage Devices
 - For SSD: no overwriting – block must be erased first (= additional overhead); also, wear pattern is that with too many writes, the device is not that durable; after a certain number of writes, device will become bad.
 - Also, random write is 10x slower than random read for SSD

Storage Devices

- Magnetic disks HDD
 - Storage that rarely becomes corrupted
 - Large capacity at low cost
 - Block level random access
 - Slow performance for random access
 - Better performance for sequential access
- Flash memory SSD
 - Storage that rarely becomes corrupted
 - Capacity at intermediate cost (5-20x disk)
 - Block level random access
 - Good performance for reads; worse for random writes
 - Erasure requirement in large blocks
 - Wear patterns issue
- Disk Management
 - Desktops traditionally have 3.5in disk (faster @ 7200 rpm, and more storage space).
 - Laptops have 2.5in disk (slower speed @ 3600rpm, less storage).
 - Roughly need to wait for half a rotation on average to begin a read or write.
 - The same track location on several platters = a column (cylinder), as shown in the diagram (the radius of the a track t, creates a cylinder).
 - Many hard disk drives (HDD) have many platters (disks), which are magnetically coated and can be read/written on each side of the disk.

Disk Management



-
- Properties of a Hard Magnetic Disk
 - Each sector is typically 512Bytes, but with larger storage capacities, size of a sector may be much larger now, even up to 64KB or bigger.
 - Group of sectors = block (also called file block) = typically 1 page = typically 4KB,(or 8, 16, 32, 64KB..etc).
 - Can overwrite on the disk.
 - Constant bit density means that outer tracks must have more sectors, since the cover a larger radius; thus to keep **(num sectors/length) = constant**, you must *increase* number of sectors on a track as length increases (directly proportional relationship).

Properties of a Hard Magnetic Disk

- Properties
 - Head moves in to address circular **track** of information
 - Independently addressable element: **sector** 512B
 - OS always transfers groups of sectors together—"blocks"
 - Items addressable without moving head: **cylinder**
 - A disk can be rewritten in place: it is possible to read/modify/write a block from the disk
- Typical numbers (depending on the disk size):
 - 500 to more than 100,000 tracks per surface
 - 32 to 800 sectors per track
- Zoned bit recording
 - Constant bit density: more sectors on outer tracks
 - Speed varies with track location
 - As track length increases (larger radius as you move out from the center), then read and write speed increases, since you are covering more distance (a larger track) over the same amount of time (rpm = constant at all tracks) and thus can read/write more sectors (since also the larger the radius, the more sectors that are on a track that can be read and written to in the same amount of time – corresponding to larger track length). This is why the 2.5in laptop disks are slower and less spacious (smaller area, plus max speed is at outer edge of the disk and is limited to a track radius of 2.5in, while desktop disks have more area and more tracks that are also larger, with max speed at 3.5in radius of the desktop disk.).
 - So: read/write for outer tracks are much faster than inner tracks.
- Performance Model
 - Seek time = mechanical movement = slow; also depends on how far across the disk the head/arm has to move (depends on location of desired read/write).
 - Average is about a few milliseconds.

Performance Model

- Read/write data is a three-stage process:
 - Seek time: position the head/arm over the proper track (into proper cylinder)
 - Rotational latency: wait for the desired sector to rotate under the read/write head
 - Transfer time: transfer a block of bits (sector) under the read-write head
- Highest Bandwidth:
 - Transfer large group of blocks sequentially from one track
 - Disk Performance Example
 - Transfer rate (on a given track) = (#sectors * sector size)/time for 1 rotation.
 - In our example, since transfer rate is 4MB/s, then that means 32 sectors/8ms can be transferred to (with each sector = 1KB) → 8ms/32 sectors = 0.25ms/sector → transfer speed in ms/sector.
 - 8ms revolution/(2ms rot delay per revolution) = 4ms total rotational delay.
 - Approx. 10ms /1kb == 100kbyte/sec
 - Notice 4MB/s transfer speed, but because of all the mechanical delays, actual speed you get in only 100KBps

Disk Performance Example

- Assumptions:

- Ignoring queuing and controller times for now
- Avg seek time of 5ms,
- 7200RPM \Rightarrow Time for rotation: $60000 \text{ (ms/minute)} / 7200 \text{ (rev/min)} \approx 8 \text{ ms}$
- Transfer rate of 4 MByte/s , sector size of $1 \text{ Kbyte} \Rightarrow$
 $1024 \text{ bytes} / 4 \times 10^6 \text{ (bytes/s)} = 256 \times 10^{-6} \text{ sec} \approx .26 \text{ ms}$ $\frac{\# \text{ of sectors} \times \text{sector size}}{\text{Time for rotation}}$

- Read sector from random place on disk:

- Seek (5ms) + Rot. Delay (4ms) + Transfer (0.26ms)
- Approx 10ms to fetch/put data: 100 KByte/sec $= \frac{32 \times 1 \text{ KB}}{8 \text{ ms}} = 4 \text{ MB/s}$

- Read sector from random place in same cylinder:

- Rot. Delay (4ms) + Transfer (0.26ms)
- Approx 5ms to fetch/put data: 200 KByte/sec

- Read next sector on same track:

- Transfer (0.26ms): 4 MByte/sec

- Key to using disk effectively (especially for file systems) is to minimize seek and rotational delays

- Notice every degree higher of random access is slower; sequential access is fastest (read next sector on same track).
- Not as much control on rotational delays – but still some.

Lecture Video 13-3 (week12 – 3/28/22): I/O System

- Disk Scheduling
 - LBA = Logical block address = single number that gives address location on disk; used and implemented by disk hardware controllers (not OS). We must know track number (and header and sector number) for the OS to manage disk read/write scheduling.
 - Total time = seek time + rot delay time (time for 1 revolution) + transfer time.

Disk Scheduling

(# header, # tracks, sector #) disk block LBA
(logical block address)

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth
- Minimize seek time
- Seek time \approx seek distance
- Disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer

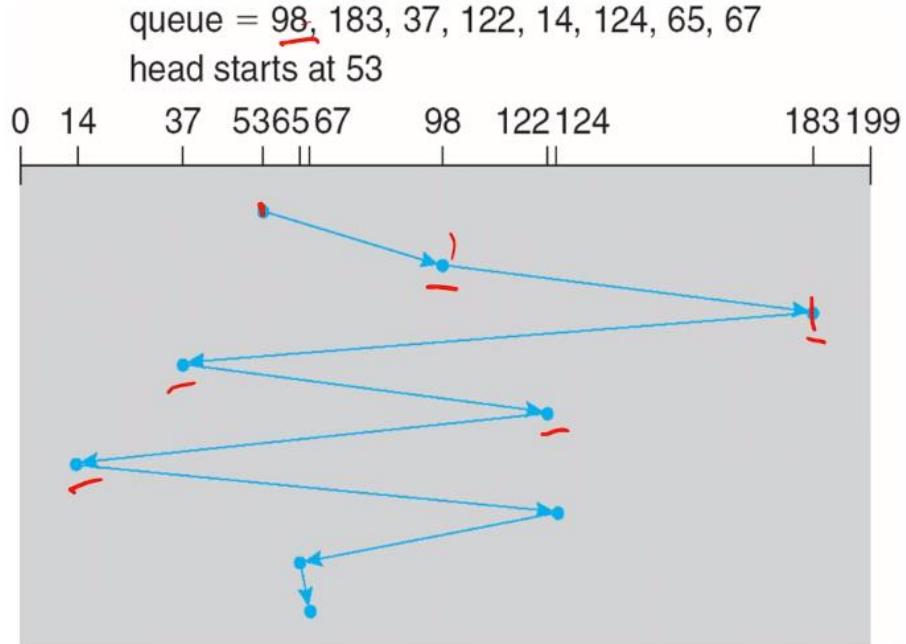
o

Disk Scheduling (Cont.)

- There are many sources of disk I/O request
 - OS
 - System processes
 - Users processes
- I/O request includes input or output mode, disk address, memory address, number of sectors to transfer
- OS maintains queue of requests, per disk or device
- Idle disk can immediately work on I/O request, busy disk means work must queue
 - Optimization algorithms only make sense when a queue exists
 - The requests number in the example for the algorithm uses track number (which determines/corresponds to the radius of a given cylinder).
 - Initial head pointer in the example is at track 53
- FCFS algorithm
 - Number of cylinders (tracks) traveled (seek distance) is proportional to seek time.
 - Notice how often and how much distance the read write head has to move.
 - Simple but not very efficient.

FCFS

Illustration shows total head movement of 640 cylinders



- SSTF(shortest seek time first) algorithm
 - Remember: minimizing seek time means minimizing seek distance.
 - Notice total head movement is much less than with FCFS.
 - Much more efficient, but some requests can get starved, if there are clusters of requests in the same locations, the relatively farther away (sparse) requests may not get served or starved. For example, if most requests are in the middle tracks, then outer and inner tracks (track number) may not get served or starved.
- SCAN Algorithm (aka elevator algorithm)

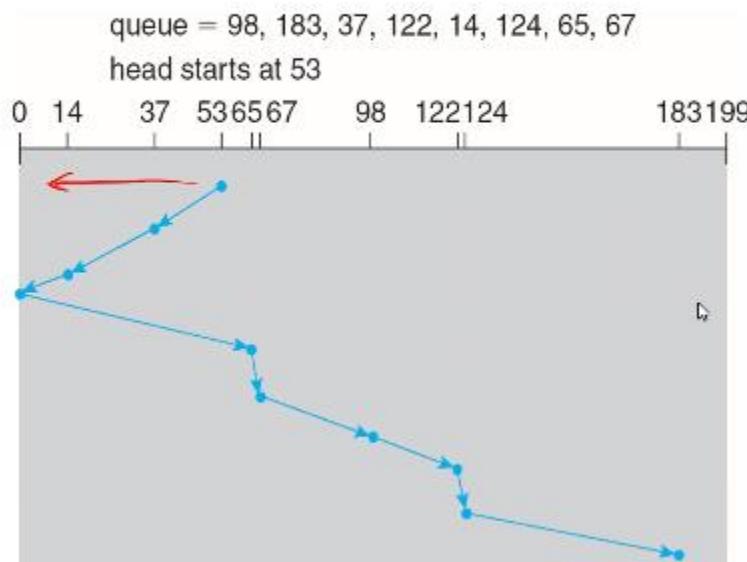
SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- **SCAN algorithm** Sometimes called the **elevator algorithm**
- Illustration shows total head movement of 208 cylinders
- But note that if requests are uniformly dense, largest density at other end of disk and those wait the longest

33

-
- Think about how elevators work: they service all requests in one direction before reversing directions and serving all requests in that next direction.
- Similar to shortest job first, but more fair

SCAN (Cont.)



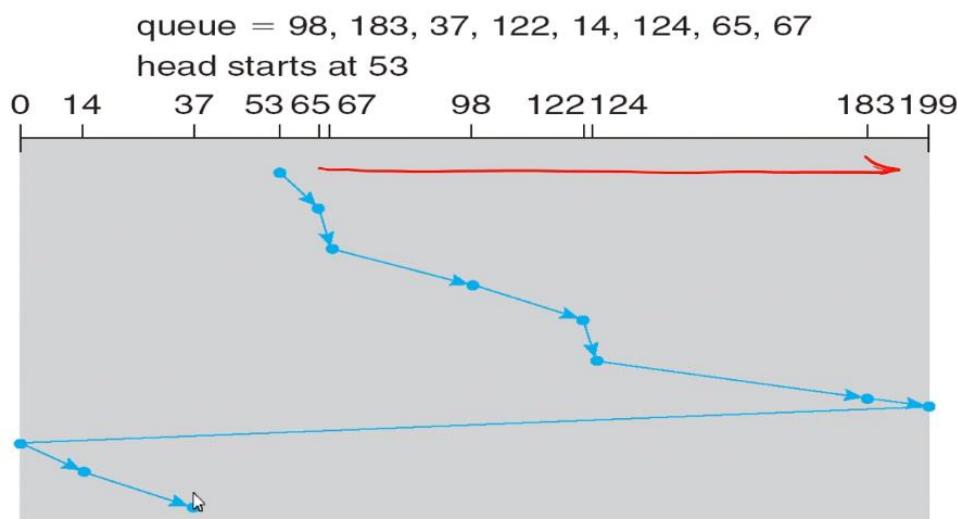
-

- One issue: requests at one end of disk may wait longer (provided there is uniform density of requests across the disk).
- C-SCAN

C-SCAN

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

C-SCAN (Cont.)



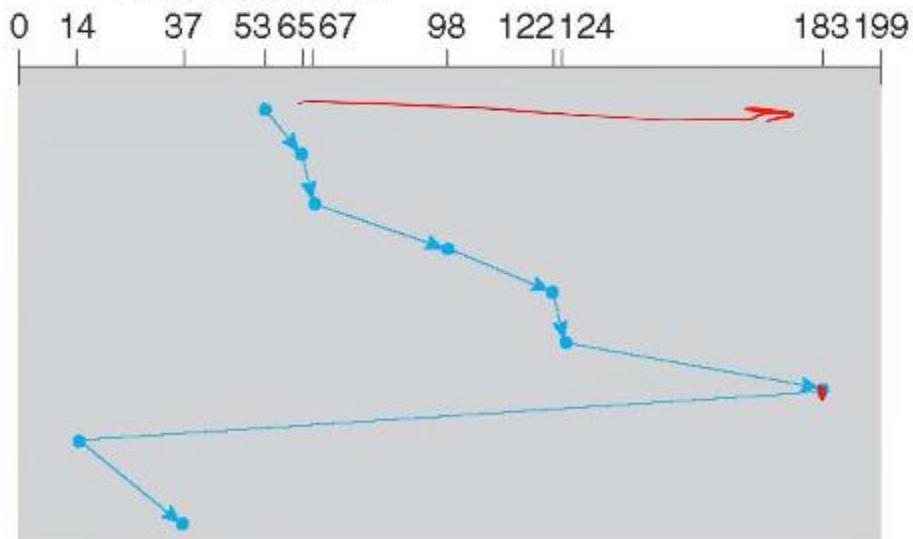
- C-LOOK
 -

C-LOOK

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

C-LOOK (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



- - Selecting A Disk-Scheduling Algorithm

Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk
 - Less starvation
- Either SSTF or LOOK is a reasonable choice for the default algorithm
- What about rotational latency?
 - Difficult for OS to calculate
- How does disk-based queueing effect OS queue ordering efforts?
 -
- Disk Management
 - Low-level (physical) formatting usually done by the disk manufacturer.
 - Logical (software) formatting needed for OS file data structures and booting code (if device is bootable – i.e. Booting from a Disk in Windows).

Disk Management

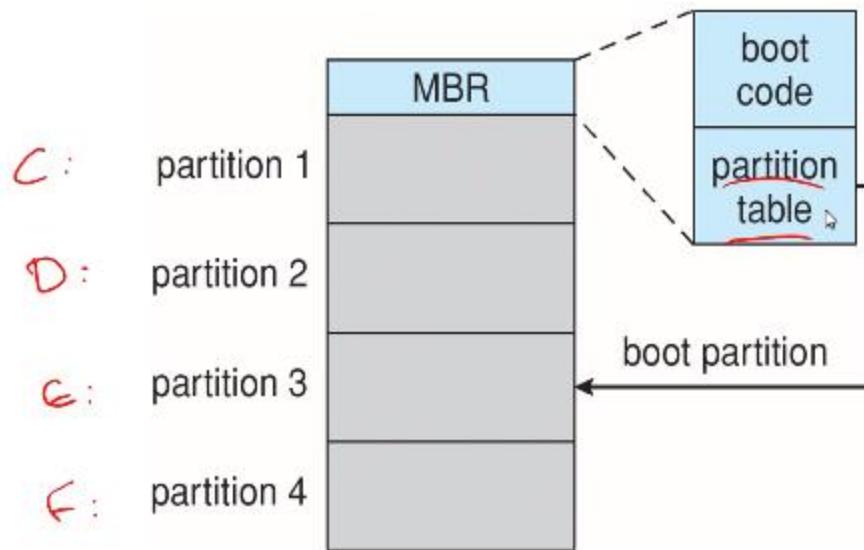
- **Low-level formatting**, or **physical formatting** — Dividing a disk into sectors that the disk controller can read and write
 - Each sector can hold header information, plus data, plus error correction code (**ECC**)
 - Usually 512 bytes of data but can be selectable
- To use a disk to hold files, the operating system still needs to record its own data structures on the disk
 - **Partition** the disk into one or more groups of cylinders, each treated as a logical disk
 - **Logical formatting** or “making a file system”
 -
- Typical booting process: from ROM (read-only memory → load MBR (master boot record == bootstrap loader), from MBR → load OS.

- Sector sparing: physically bad (broken/damaged) blocks can be skipped/not used.

Disk Management (Cont.)

- Raw disk access for apps that want to do their own block management, keep OS out of the way (databases for example)
- Boot block initializes system
 - The bootstrap is stored in ROM
 - **Bootstrap loader** program stored in boot blocks of boot partition
- Methods such as **sector sparing** used to handle bad blocks
 -
- Booting from a Disk in Windows
 - Each partition of the hard drive is a “soft drive”.
 - MBR = Master Boot Record (contains boot code and partition table – jump to beginning of OS to load the OS into memory).
 - Microsoft drive format only allows for 4 partitions in the MBR; need a larger table to have more partitions, thus if user wants more, the another partition table will be stored on one of the 4 partitions from the MBR table.

Booting from a Disk in Windows



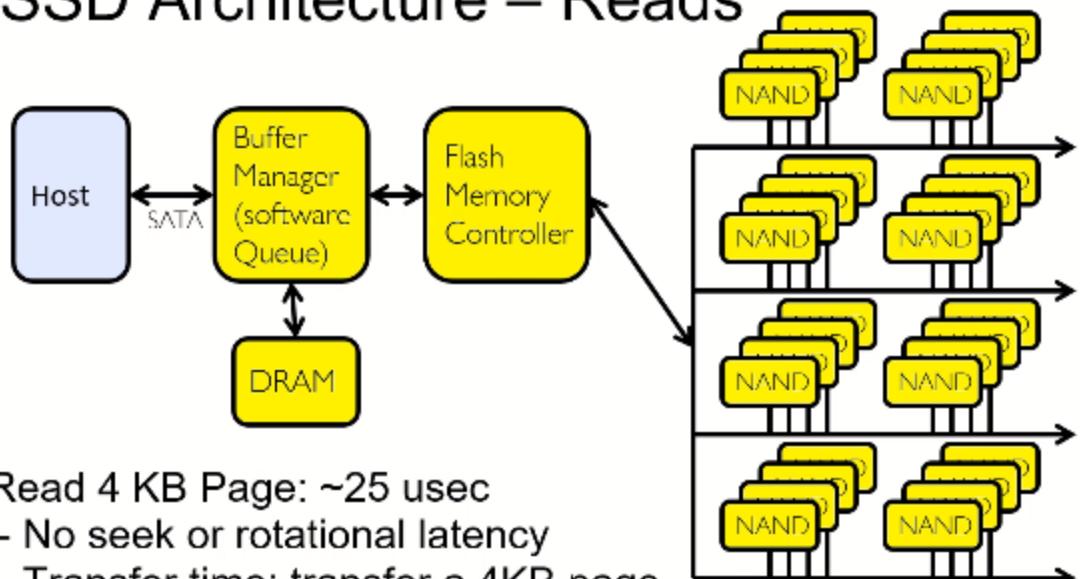
- ○ Partitioning a disk can protect OS boot code, and also protect other groups of tracks/cylinders from each other, and make accessing blocks from each partition faster (potentially).
- SSDs(solid state drives/"disks")
 - Used by most mobile devices, many laptops, tablets...etc.
 - Circuit-based memory.
 - Read speed similar to DRAM (but write is slower than DRAM).
 - Trapped electrons in capacitor cell give different state, which gives a distinguishable value (1, or 0).
 - With a 3-bit cell, you can represent $2^3 = 8$ different states.
 - Very low power use (thus increase battery life of a mobile device) since there are no moving parts (moving a mechanical arm or any mechanical part is very energy expensive).
 - Limited write cycles – you can only write to cells a limited amount of times before cells start to fail.
 - Lifetime is much shorter than traditional hard drives.

Solid State Disks (SSDs)



- 1995 – Replace rotating magnetic media with non-volatile memory (battery backed DRAM)
- 2009 – Use NAND Multi-Level Cell (2 or 3-bit/cell) flash memory
 - Sector (4 KB page) addressable, but stores 4-64 "pages" per memory block
 - Trapped electrons distinguish between 1 and 0
- No moving parts (no rotate/seek motors)
 - Eliminates seek and rotational delay
 - Very low power and lightweight
 - Limited "write cycles"
- Rapid advances in capacity and cost ever since!
-
- SSD Architecture – Reads
 - Each stack of NAND represents multiple pages together (a block).
 - Page-addressable through the flash memory controller.
 - Read is very fast; mainly limited by bus speed.
 - Read is similar to speed of DRAM.
 - Sequential and random access for read == same speed since you simply use busses (wires) to go directly to an address location to read in (just like RAM).

SSD Architecture – Reads



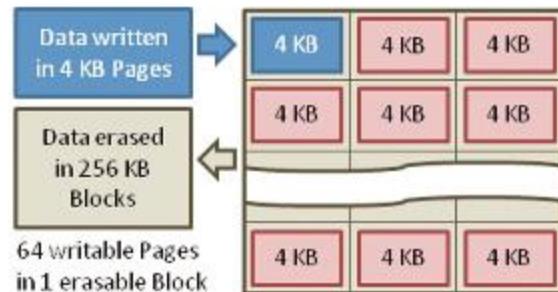
Read 4 KB Page: ~25 usec

- No seek or rotational latency
- Transfer time: transfer a 4KB page
 - SATA: $300\text{-}600\text{MB/s} \Rightarrow \sim 4 \times 10^3 \text{B} / 400 \times 10^6 \text{Bps} \Rightarrow 10 \text{us}$
 - **Latency = Queuing Time + Controller time + Xfer Time**
 - **Highest Bandwidth:** Sequential OR Random reads

- SSD Architecture – Writes
 - 10x slower than reads.
 - Cannot erase a single page; must erase an entire block when trying to erase a page so that you can write to it (must be empty!).
 - This means you must sometimes move entire blocks (set of pages) around in order to write. This costs CPU cycles, which is why it is much slower than read and produces additional overhead.
 - Remember : A block can have 4 to 64 pages.
 - Once you erase an entire block, then you can write to individual pages of that block.
 - So, write → 10x slower than read, and erase is → 10x slower than write.
 - Coalescing == come together to form one mass or whole; to amass.
 - Want to leave at least 20% reserve capacity in order to do coalescing more efficiently.
 - Changing energy levels cost much more energy when erasing or writing, which is why it is much slower as well.

SSD Architecture – Writes

- Writing data is complex! (~200 μ s – 1.7ms)
 - Can only write empty pages in a block
 - Erasing a block takes ~1.5ms
 - Controller maintains pool of empty blocks by coalescing used pages (read, erase, write), also reserves some % of capacity
- Rule of thumb: writes 10x reads, erasure 10x writes



Typical NAND Flash Pages and Blocks

https://en.wikipedia.org/wiki/Solid-state_drive

45

- SSD Summary
 -

SSD Summary

- Pros (vs. hard disk drives):
 - Low latency, high throughput (eliminate seek/rotational delay)
 - No moving parts:
 - Very light weight, low power, silent, very shock insensitive
 - Read at memory speeds (limited by controller and I/O bus)
- Cons
 - Small storage (0.1-0.5x disk), expensive (3-20x disk)
 - Hybrid alternative: combine small SSD with large HDD
 -
 - New algorithms for garbage collection make it such that the controller will move data around as needed (for writes, etc.), but in such a fashion so as to wear out all cells of the SSD more evenly (so, don't only move and write intensely on one section of cells) – which can improve the overall lifespan of the drive.
 - So for example, move read-only parts of disk (which is stored in cells not used often or at all in terms of writes) to frequently used cells to give those cells a break.

SSD Summary

- Pros (vs. hard disk drives):
 - Low latency, high throughput (eliminate seek/rotational delay)
 - No moving parts:
 - Very light weight, low power, silent, very shock insensitive
 - Read at memory speeds (limited by controller and I/O)

- Cons

- Small storage (0.1-0.5x disk) expensive
- Hybrid alternative: combine small SSD with large HDD
- Asymmetric block write performance: read pg/erase/write pg
 - Controller garbage collection (GC) algorithms have major effect on performance
- Limited drive lifetime
 - 1-10K writes/page for MLC NAND
 - Avg failure rate is 6 years, life expectancy is 9–11 years
- These are changing rapidly!

No longer true!

- I/O Performance
 - n = number of bytes (number of ops).
 - Notice as you increase n, your effective BW will be closer to the actual transfer rate.
 - (SB/n → as n approaches infinity, SB/n approaches 0.)
 - Queuing delay becomes a big factor at a certain point when the queue becomes large enough (starts to grow exponentially).
 - Lets say queuing delay time is (based on exponential graph) is greek letter row/1-row
 - So: $r/1-r$
 - Then if throughput (utilization) is 90%, then queuing delay could be $0.9/1-0.9 = 9 \rightarrow$ so 9x the typical base service time.
 - So you want to keep utilization under 80% to stay away from the steep queuing delay increase (as it is exponential as the graph shows, so it begins more linear initially, but then grow increasingly steeper with utilization).
 - So over 90% utilization, you will start to see your response time grow exponentially.

I/O Performance

The diagram illustrates the flow of an I/O request from a User Thread through a Queue (labeled 'Queue [OS Paths]') to a Controller, which then interacts with an I/O device. A red arrow points from the User Thread to the Queue, another from the Queue to the Controller, and a third from the Controller to the I/O device.

Below the diagram, the formula for Response Time is given as:

$$\text{Response Time} = \text{Queue} + \text{I/O device service time}$$

A graph on the right plots Response Time (ms) against Throughput (Utilization (% total BW)). The x-axis ranges from 0% to 100%, and the y-axis ranges from 0 to 300 ms. A black curve starts at approximately (0, 50) and rises sharply as utilization increases. Handwritten red text on the graph shows the formula for the curve: $\alpha \frac{\beta}{1-\beta} = \frac{\alpha q}{1-q} = q$.

- **Performance of I/O subsystem**
 - Metrics: Response Time, Throughput
 - Effective BW per op = transfer size / response time
 - $\text{EffBW}(n) = n / (S + n/B) = B / (1 + SB/n)$
 - Contributing factors to latency:
 - Software paths (can be loosely modeled by a queue)
 - Hardware controller
 - I/O device service time
 - **Queuing behavior:**
 - Can lead to big increases of latency as utilization increases
 - Solutions?
-

Lecture Video 14-1 (week13 – 4/04/22): File Systems

- File Systems

File Systems

- What is a file system?
 - A method for storing and accessing files
- We'll concentrate on:
 - User interface to files
 - Naming, access
 - System representation of and access to files
 - Structure, protection
 - Translation between user and system views
 - User pov = names
 - System pov = disk blocks.
 - File system is implemented by the OS
 - So Windows File Explorer is a user application, that calls the file system code (file-system system calls to OS)
- File System Components

File System Components

- Disk Management
 - How to arrange collection of disk blocks into files
- Naming
 - User gives file name, not track 50, platter 5, etc
- Protection
 - Keep information secure
- Reliability/durability
 - When system crashes, lost stuff in memory, but want files to be durable
 - Long-term Information Storage

Long-term Information Storage

1. Must store large amounts of data
2. Information stored must survive the termination of the process using it
3. Multiple processes must be able to access the information concurrently
 - Characteristics of Secondary Storage (like SSD or HDD)

Characteristics of Secondary Storage

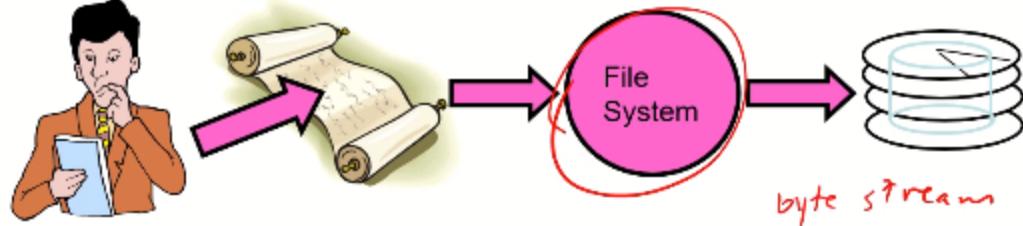
- Large amounts
 - TB
 - Slow to access
 - Milliseconds
 - It's free
 - Not quite, but close
 - It's there
 - Hangs around for a long time
- User vs. System View of a File

User vs. System View of a File

System View of a file	User's View of a File
Block oriented, may be scattered	Byte or record oriented, contiguous
<u>Physical block #'s</u>	<u>Name files</u>
No protection	Users <u>protected from each other</u>
Data might be corrupted if machine crashes	<u>Robust to machine failures</u>

-
- Translating from User to System View
 - System must operate at block level, while users operate at byte level (byte stream).
 - Need to do caching: system to read in whole blocks then extract the byte desired by user while in cache.
 - Good because often user will read/write from same block (principle of locality).
 - Entire block must be written even when user only needs to write one byte that is a part of the block.
 - One way to make writing efficient is to use caching for optimization: keep in memory until file is closed before writing out all changes at one time (instead of one write at a time, which means potentially writing the same block again and again just at different byte locations within the block).
 - Remember a block is a collection of pages → file is a collection of blocks.

Translating from User to System View



- What happens if user says: give me bytes 2—12?
 - Fetch block corresponding to those bytes
 - Return just the correct portion of the block
- What about: write bytes 2—12?
 - Fetch block
 - Modify portion
 - Write out Block
- Everything inside File System is in whole size blocks
 - For example, `getc()`, `putc()` → buffers something like 4096 bytes, even if interface is one byte at a time
- From now on, file is a collection of blocks
 -
- File Operations (User Interface)
 - Attributes == metadata (r/w permissions, size, owner, creation/modify date, access time, etc.)

Caching

7

File Operations (User Interface)

- | | |
|---|---|
| <ol style="list-style-type: none"> 1. Create 2. Delete 3. Open 4. Close 5. Read 6. Write 7. Append | <ol style="list-style-type: none"> 8. Seek <ul style="list-style-type: none"> • Change current <u>offset</u> 7. <u>Get attributes</u> 8. <u>Set Attributes</u> <ul style="list-style-type: none"> • Modifications times, protection bits 9. <u>Rename</u> |
|---|---|
- - An Example Program Using File System Calls

An Example Program Using File System Calls (1/2)

cp file1 file2

/* File copy program. Error checking and reporting is minimal. */

```
#include <sys/types.h>           /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);    /* ANSI prototype */

#define BUF_SIZE 4096             /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700          /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);        /* syntax error if argc is not 3 */

```

- fd name is given because it refers to "file descriptor" → opening a file will return an integer, and that int is the file descriptor, which allows you to reference the file table for the current process by at a particular location in the table via the integer file descriptor reference value.
- Protection bit number used (0700) gives owner read, write, and execute permissions (7 is 111 in binary; remember 0700 is an octet → 4 octets (1 for each type of user) and each user gets 3 permissions bits corresponding to read, write, or execute permissions.)

An Example Program Using File System Calls (2/2)

```

/* Open the input file and create the output file */
in_fd = open(argv[1], O_RDONLY); /* open the source file */
if (in_fd < 0) exit(2); /* if it cannot be opened, exit */
out_fd = creat(argv[2], OUTPUT_MODE); /* create the destination file */
if (out_fd < 0) exit(3); /* if it cannot be created, exit */

/* Copy loop */
while (TRUE) {
    rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
    if (rd_count <= 0) break; /* if end of file or error, exit loop */
    wt_count = write(out_fd, buffer, rd_count); /* write data */
    if (wt_count <= 0) exit(4); /* wt_count <= 0 is an error */
}

/* Close the files */
close(in_fd);
close(out_fd);
if (rd_count == 0) /* no error on last read */
    exit(0);
else
    exit(5); /* error on last read */
}

```

1

- - Alternative Interface: Memory-Mapped Files
 - When mapped to memory, you can simply use load and store commands to modify the file. Close the file of course, to officially write to/update the disk location where the actual file is stored.
 - Memory-mapped files are already used when we do demand paging for all executables (loading the code and data sections on demand for a given process == executable).
 - So we memory map the entire executable (data section + text/code section) to the process address space (data + code + stack + heap); then we only load mapped pages of the executable (or other parts of the address space) that are needed (on demand).
 - For example, if a certain function is called, then you may need a certain section from the code section, thus then you can load that portion of the mapped executable into memory (if it is not loaded already) so that it can be used; hence we are loading the code (or data) == parts of the entirely mapped executable on demand. Note: so entire executable mapped to main memory; but only portions of it is actually loaded into main memory as demanded.

Alternative Interface: Memory-Mapped Files

- Traditional file interface through system calls
 - 1) Open file
 - 2) Read data from file
 - 3) (Possibly) modify files
 - 4) Write data to file
 - 5) Close file
- Would be nicer to:
 - 1) Map file into address space, starting at addr. X
 - 2) (possibly) modify elements of X Load, store
 - 3) Close file
- Note: need swap files for shared datastreams which are on disk but not as a user file; thus, when we map a file into memory, no swap file needed since the file itself is by nature stored on the disk as permanent memory.

Memory-Mapped Files, cont

- Use virtual memory:
 - Provide familiar load/store access to files
 - Instead of read/write
 - Use file itself as backing store (no swapfile)
 - On close, file implicitly written to disk
- Advantages:
 - Less cumbersome, eliminate a copy
- Disadvantages:
 - What about large files, synchronizations, sharing?

UNIX provides this: mmap(fileId, virtAddr)

- Memory mapping is not that popular – it is not often used for most user applications; classic read/write open/close method is used.

- File Types
 - ASCII == plain text (human readability files; i.e. .c, .cpp, javascript files, .html, etc.); but not very efficient because of the space/size needed for each character.
 - Executables are often in binary, as long as multimedia files.
 - Regardless of name file extension used (.exe, .c, etc.), the actual file type is determined/set by magic # == file signature, which is stored in the first few bytes of the file so that OS and other applications know how to read/use the file, and if it is the correct format for a given application. When you rename a file with a different extension, in the background windows actually changes the magic# to the correct corresponding extension the user gave it – but magic number can be manually updated, which then the file extension name won't mean anything as it won't correspond to the actual magic #. So not only do executables have them, but all other binary files have magic #s too.
 - Directory is also a file, just a special kind of file → but its contents is a list of files and a pointer to the metadata about its contents and to blocks where its other contents start.
 - Other special files == I/O devices which are abstracted as files.

File Types

- Regular files
 - ASCII or binary
 - Executable file (binary) have headers with:
Magic #, text size, data size, etc..
*com
exe*
- Directories
- Character/Block special files
 - stdin
stdout*
- Directories (user interface)
 - Link == similar to Windows shortcut files/directories; same file belong to multiple directories, and can even have a different name. Unix/Linux use hardlink, but Windows uses softlink (which is what shortcuts are).
 - Soft link is basically a new file, linked to another one.
 - Hard link is only a single file, but (its metadata) includes several different reference names as part of the file, so that it can have different names and be a part of multiple directories (or even two different names of the same file, in the same directory).

Directories (user interface)

- Generally, tree-structured
- Consist of 0 or more files / *directories*
- Operations:
 - Create, delete, open, close, read, list
 - Link → *Unix / Linux*
 - Same file in multiple directories
 - Unlink
 - Remove a directory entry
- ^o Organize the Directory (Logically) to Obtain:

Organize the Directory (Logically) to Obtain

- **Efficiency** – locating a file quickly.
- **Naming** – convenient to users.
 - Two users can have same name for different files.
 - The same file can have several different names. *hard link, soft link / short cuts*
- **Grouping** – logical grouping of files by properties, (e.g., all Java programs, all games, ...)
- ^o Type of File Access

Types of File Access

- Sequential access
 - read all bytes/records from the beginning
 - cannot jump around, could rewind or back up
 - convenient when medium was magnetic tape
- Direct access (random access)
 - bytes/records read in any order *Seek*
 - essential for data base systems
 - read can be ...
 - move file marker (seek), then read or ...
 - read and then move file marker
- Content-based access
 - “find a certain type of data, e.g. person under 25 years old; this is really a database search
- How are files typically used?

How are files typically used?

- Most files are small (for example .login, .c files)
- Large files use up most of the disk space
 - Ex: image processing, multimedia
- Large files account for most of the bytes transferred to/from disk

Bad News: need everything to be efficient

- Need small files to be efficient, since lots of them
- Need large files to be efficient, since most of the disk space, most of the I/O due to them

○

Lecture Video 14-2 (week13 – 4/04/22): File Systems

- File System Implementation

File System Implementation

- Need to decide
 - How to translate between user and system view
 - How to store file on disk
 - How system accesses file on disk
 - How to manage disk spaces

↳

- Some Attributes of Open Files
 - Open count == how many processes are using the file.

Some Attributes of Open Files

- Offset
- Protection bits
- File size
- Modification times
- Pointers to disk blocks
- Open count
- Cache
 - Remember, file system is block level, but user application is byte wise, so need a cache in order to access each byte individually of a given block that the file system handles.
 - Mismatch handled by cache: blocks loaded into cache by file system.
 - Then user accesses the cache at given desired byte.
- Translating Between User and System (single-process OS only)

- FileID = file descriptor (integer, which indexes into the file table → file table maintains all the information about open files of a process).
- buf is in the user address space.

Translating Between User and System (single-process OS only)

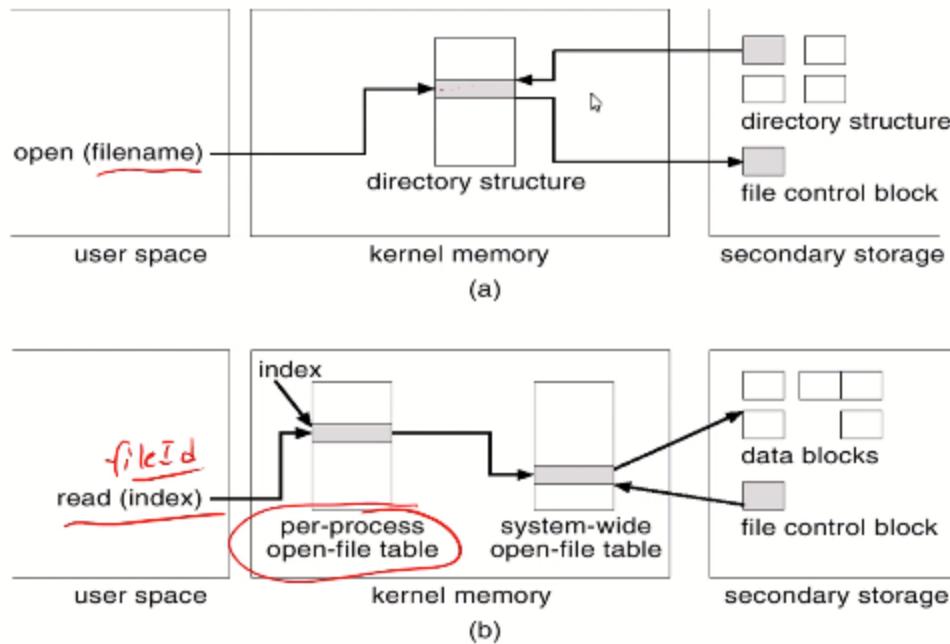
- Example:
 - fileId = Open ("foo");
 - Read(buf, nbytes, fileId);
- Open creates new openfile table entry
- On read, fileId indexes into openfile table
- Openfile table contains:
 - Everything on last slides

- Translating Between User and System (multiprogrammed OS)

Translating Between User and System (multiprogrammed OS)

- Same interface
 - Open, followed by Read
- More complicated implementation
 - Need to worry about files sharing
 - UNIX
 - File descriptor points to system-wide table
 - System wide-table contains offset, protection, pointer to "I-node"
 - I-node contains pointers to blocks, file size, etc.
- In-Memory File System Structures

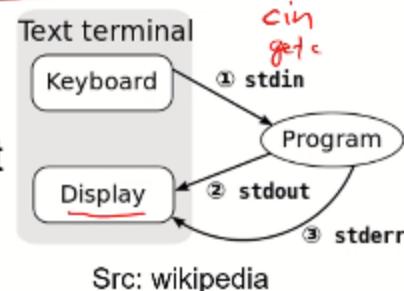
In-Memory File System Structures



-
- Directory (metadata of directory) not already in memory, so it must be loaded in, as shown by (a).
- As shown by (b), if a file is open by multiple processes, then we index into the table, and that indexed location simply contains a pointer to the system-wide open-file table; if file no open by any other process, then go to steps shown in (a).
- File Descriptors
 - You can explicitly close the default opened files, for example `stdin` (but then you won't be able to do any file input calls such as `cin`, `get`, which uses `stdin` file).
 - Both `stdout` and `stderr` by default go to the display; but you can redirect `stderr` (for example) to a storage file in order to essentially have a log file of `stderr` messages.

File Descriptor

- a file descriptor is an index for an entry in an open file table.
- There are three standard file descriptors, which every process should expect to have:
 - 0, stdin, standard input
 - 1, stdout, standard output
 - 2, stderr, standard error



- How is a file stored?
 -

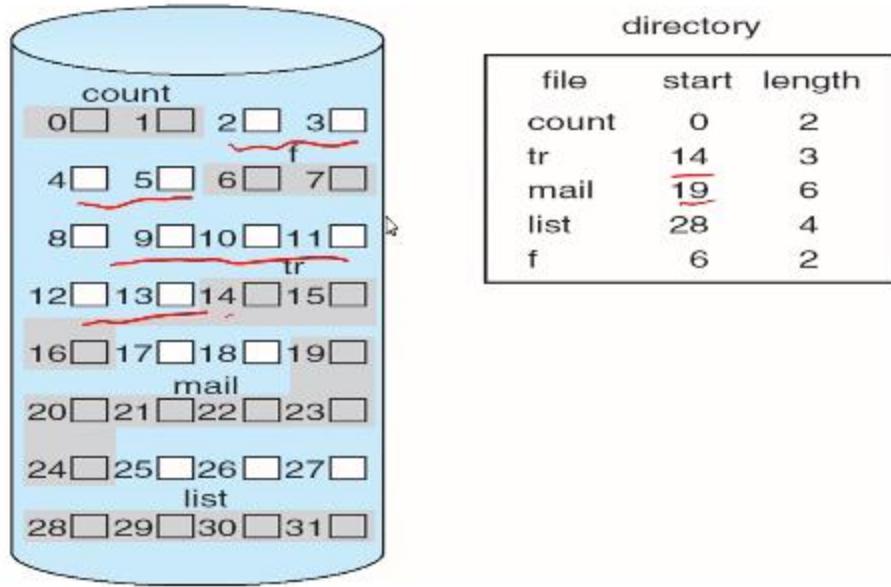
How is file stored?

- Contiguous allocation
 - Linked allocation
 - Couple variation
 - Indexed allocation
 - Many variation
- Contiguous Allocation
 -

Contiguous Allocation

- User must give file size in advance
- Find free space on disk using first fit/best fit
- Advantages
 - Fast sequential access
 - Fast random (direct) access
- Disadvantages
 - External fragmentation
 - Hard to increase file size (moving it very expensive)
- - Fast random access too because all blocks that make up the file are contiguous, so it is easy to index (jump around) into what is essentially an array of block (and thus pages and bytes), by simply using an offset value.
 - When disk becomes highly fragmented, can run disk defrag (just like the Windows system application that allows for this. It is very expense, hence why it usually takes a long time to run that program).
 - Contiguous (for disk/file allocation) is not as popular.

Contiguous Allocation



2

- - Linked Allocation

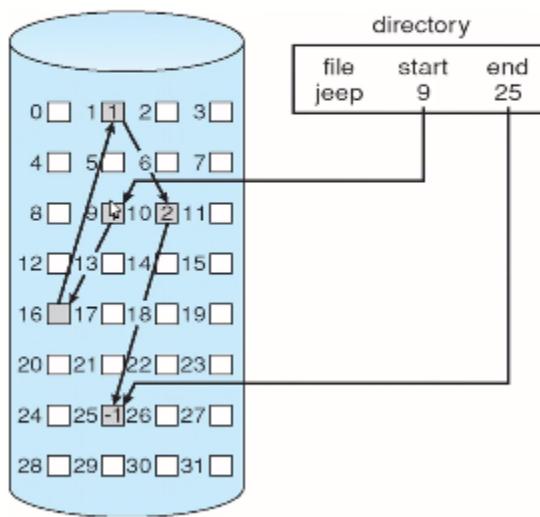
Linked Allocation

- Hard to increase file size with contiguous
- So, use linked blocks
 - Each block contains pointer to next block
 - Allows blocks to reside anywhere on disk
 - This is main advantage – can increase file size
- However
 - Sequential access: seek between each block
 - Direct access: horrible
 - Unreliable (lose block, lose rest of file)
- - Remember, direct access == random access
 - Very bad with linked allocation because you need to load in the entire file in order to locate the disk block needed), or starting with first block (to get the

head pointer), load in all subsequent blocks until desired block is found. So for random access, you have to always start from the beginning of the file.

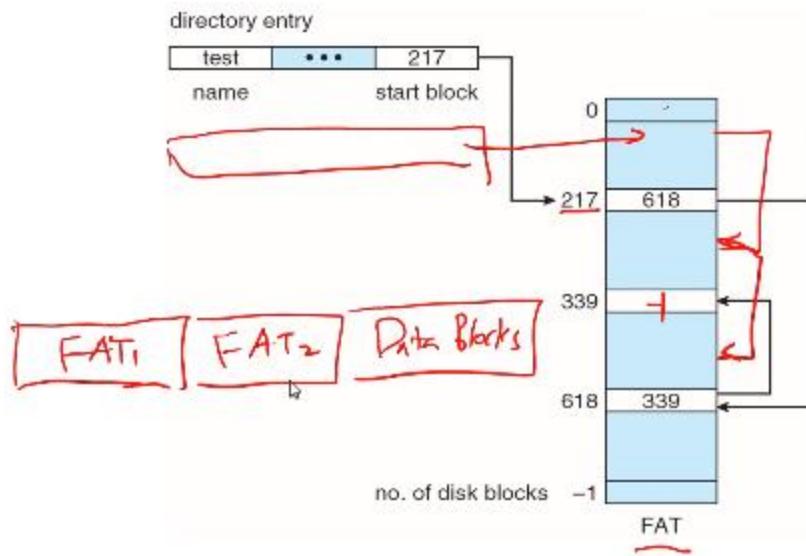
- Linked allocation uses pointers to point to next block; it is essentially a linked list.

Linked Allocation



- - File-Allocation Table (FAT format)
 - Solution to the links (pointers).
 - So instead of loading in entire block just to get the pointer to the next block, just maintain a file-allocation table, which is essentially a pointer table, that way you can have a table of pointers that maps all the locations that make up the entire file. Now, you simply need to load the file allocation table in order to get a certain desired block.
 - Cost is just that you need memory in order to store the table.
 - But also cost of loading table far less than cost of loading entire blocks just for a single pointer.
 - FAT file systems are simple; they are used for USB (flash drive) and memory-cards, which have relatively smaller memory sizes.
 - Usually there are two FAT tables, simply used for redundancy, just in case one of them crashes.

File-Allocation Table



-
- So instead of keeping linked list inside data blocks, we take the references out and save the space, and use that space instead for a FAT table (stored on the drive and used for all files), which can be loaded into cache (or memory) so that the file can be sequentially and directly accessed (random) very efficiently by simply reading the FAT table.
- FAT table is critical; if the disk block(s) storing the FAT table crashes, then entire file system (disk drive) will become inaccessible; so this is why we have redundancy (multiple copies for backup) of FAT tables.

FAT (File Allocation Table)

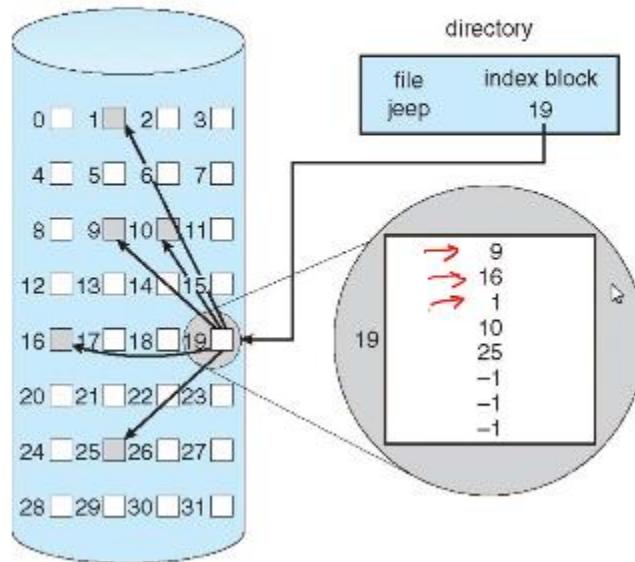
- Used in MS-DOS, *camera*, *USB*, *Memory-Card*
- Basically an offshoot of linked allocation
 - Keep the linked list in a separate part of the disk
 - Read FAT into memory (or cache it)
 - Find block by traversing FAT
 - Then go get it on disk
 - Reduces number of disk reads
- Indexed Allocation

- Index block is like a page table
- Similar to FAT, but each file stores its own table.

Indexed Allocation

- Still want to put free block anywhere on disk
- Bring together all pointers into one block
 - “index block”
 - File creation: set pointers to NULL
 - On write, find free block on disk, set pointer
 - Direct access: read index block, find right block
 - Similar mechanism to paging
- File metadata contains block location where the table is stored.

Example of Indexed Allocation



- In example above, only 5 blocks (+1 block for the table itself) is allocated to the process; the remaining pointer locations in the table is NULL/unused. Can use them later if file size increases.
- Index Block Problems

- Multilevel index is bad for small files since they may only be a few blocks and never need additional space (or not that much), so it will waste space since you will have empty but reserved index tables taking up blocks.

Index Block Problems

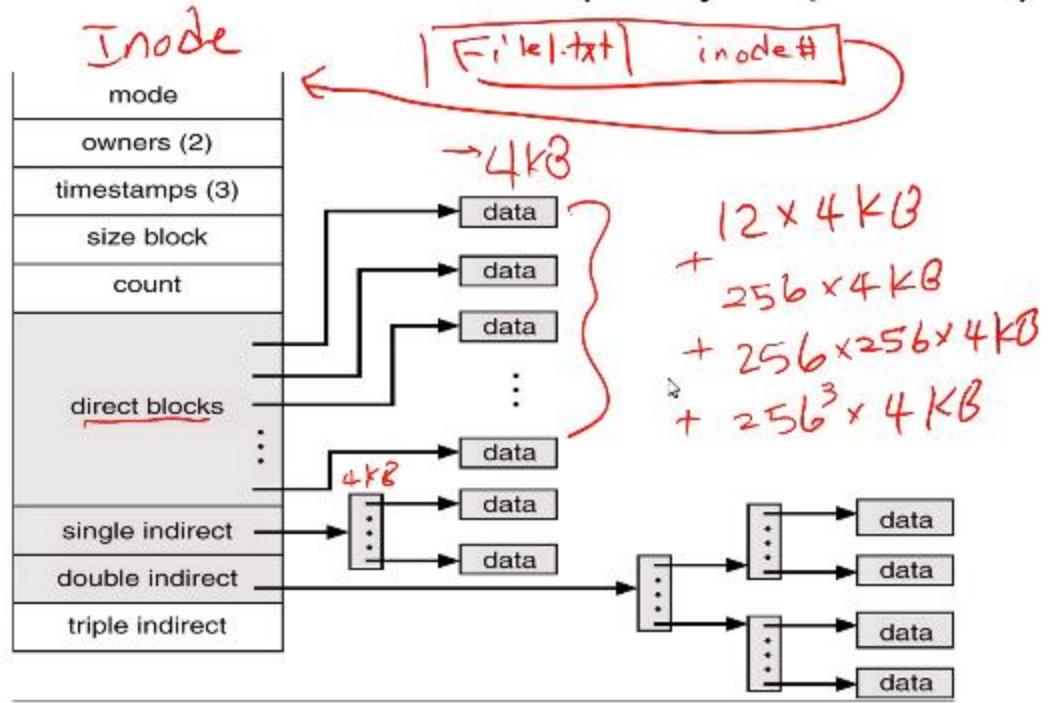
- How big should index block be?
 - Too large =>waste space for small files
 - Too small => painful when gets too large
- Options for large files
 - Link index blocks
 - Multilevel index
 - Read bunch of index blocks, then get data
 - Bad for small files
 - Combined scheme
 -
- Combined Schemes (used in UNIX and Linux)
 - Remember, I-node contains most of the metadata of a file.
 - So 3 out of the 15 pointers are essentially index trees and are as follows: a single pointer to another table, a double pointer to a list of table pointers, and a triple pointer to a list of double pointers of index tables. Good for small files since only need 15 total pointers, and the three following pointers only use minimal additional space, while also providing expandability for large files.

Combined Scheme (UNIX)

Linux EXT 2 & 3

- Keep some pointers to disk blocks
- Keep a few indirect pointers to index blocks
- Example (UNIX 4.2)
 - I-node contains 15 pointers
 - 12 to disk blocks (data)
 - 1 each to single, double, triple indirect block
 - Relatively simple, extensible, small files good
 - Large files: takes many read
- For unix, I-node (pointer to a metadata block) contains all metadata **except** for file name.
 - File name is in the directory (as well as the inode pointer).
- So for example, lets assume each block is 4KB, so that if your file size is more than 48KB ($12 \times 4\text{KB} = 48\text{KB}$), then you have to move to single indirect block (single pointer to the next index table) (also each index block is same size as a data block = 4KB).

Combined Scheme: UNIX (4K bytes per block)

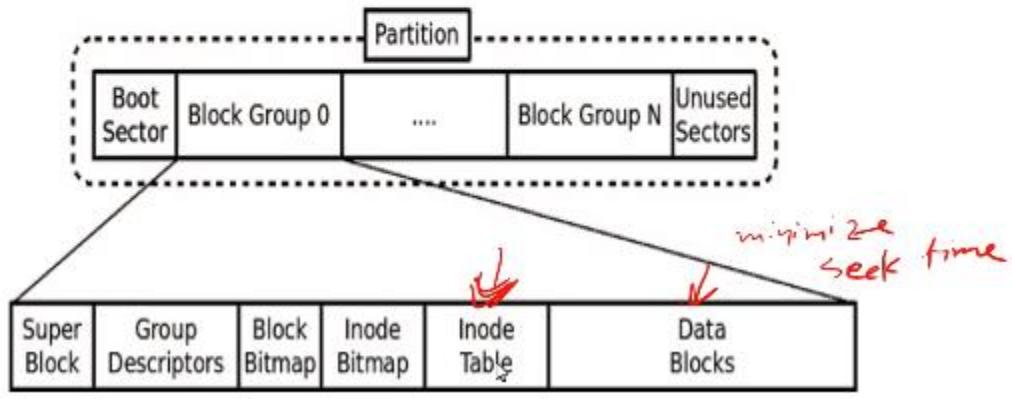


-
- Inode contains other data as well as the 12 direct pointers to blocks
- But for indirect tables, they can use the full size of the block (they don't contain metadata), and can thus use the full 4KB of a block and point to 256 pointer entries.
 - Thus if file size $> 12 * 4\text{KB} = 48\text{KB}$, use single indirect pointer to an index table, which that index table uses the full 4KB of the block size and can contain 256 direct block pointer entries; thus the file can be an additional $256 * 4\text{KB}$ in size. Then if you need even more size, use double direct, which contains 256 pointers to 256 index tables, and each index table points directly to 256 blocks. Thus now we have an additional size of $256^2 * 4\text{KB}$. And for the max size, use all 256 triple pointers, which point to 256 double pointers, which each points to 256 index tables, which each table points directly to 256 direct blocks. Thus you have an additional size of $256^3 * 48\text{KB}$.
 - Thus, max size of a file with 4KB blocks is:
 - $12 * 4\text{KB} + 256 * 4\text{KB} + 256^2 * 4\text{KB} + 256^3 * 4\text{KB}$, which is about 62 to 64GB
 - So, this solution is efficient for both large and small files.
- File Structures in Linux Ext4
 - We group Inode Table and Data blocks next to each other to minimize seek time on traditional hard drives, since we will use inode table to then read a data block.
 - Block bitmap tells if block is free or not (0 = free, 1 = allocated)
 - Same applies to Inode bitmap

- When you partition a disk, you could have lots of small files in a group; and number of files = number of inodes; so if you run out of inodes due to many small files, you could run out of space, even though there may actually be a lot of data blocks still free.

File Structures in Linux Ext4

- Improved management of large files and offered more flexibility
- Support volumes with sizes up to 1 exbibyte (EiB) and single files with sizes up to 16 tebibytes (TiB) with the standard 4 KiB block size



-

File Structures in Ext4

- Boot sector
 - Block is a disk allocation unit of at least 512 bytes
 - Contains the bootstrap code
 - UNIX/Linux computer has only one boot block, located on the main hard disk

-

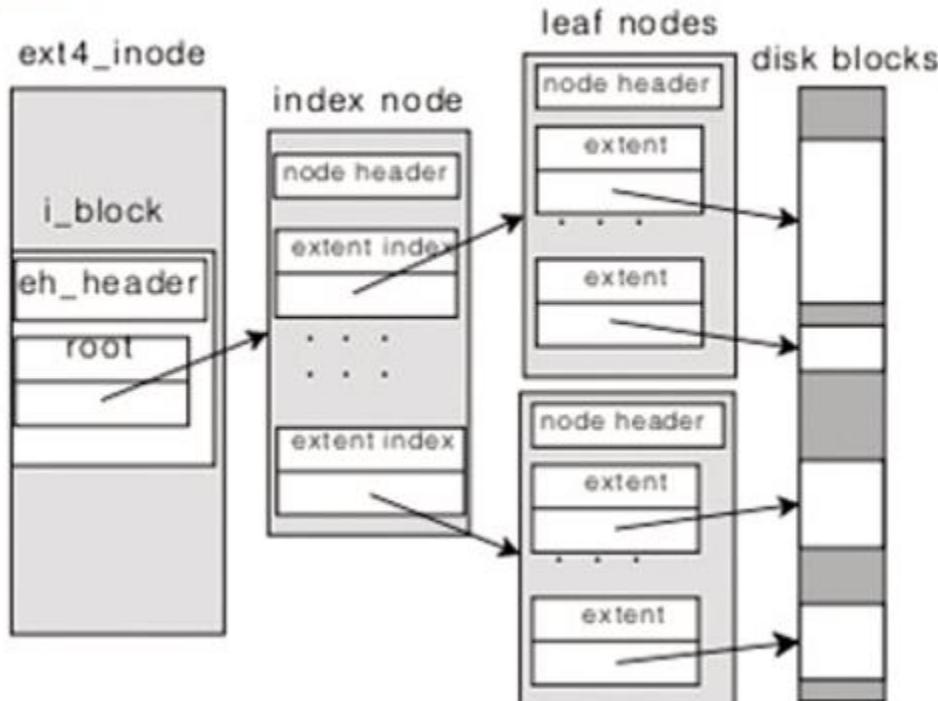
File Structures in Ext4

- Superblock
 - Indicates disk geometry, available space, and location of the first inode
 - Manages the file system
- Inode blocks
 - Contain file and directory metadata
 - Also link data stored in data blocks
- Data blocks
 - Where directories and files are stored
 - This location is linked directly to inodes
- - Extents (EXT4) (similar to NTFS developed by Microsoft)
 - For extent, just tell the start address, and the last (a range).
 - Works for if a file is made up of many contiguous blocks; instead of wasting space using many references (using the index table), just use extents (exts) instead. The purpose of index table anyway is to allow for efficient, non-contiguous memory allocation for large and small files if necessary. But often, file allocation can often still be contiguous, especially for large files, so then no need for many reference locations; just need the start and end, then we can index into the file over a large range since many or even all blocks that make up the file are contiguous. So, this combines contiguous, with non-contiguous allocation → allowing for combination of performance by taking advantage of both styles of file allocation; allow for more space efficiency in terms of contiguous storage (since reduced reference tables), and in terms of still being able to jump around and fill in blocks and not waste space (reduce fragmentation) by non-contiguous allocation.

~~TESTS~~
~~TESTS~~

Extents (EXT4)

- Instead of represent each block, represent large contiguous chunks of blocks with an extent (190, 100) (365, 259)
- An extent is a range of contiguous physical blocks, improving large-file performance and reducing fragmentation. A single extent in ext4 can map up to 128 MiB of contiguous space with a 4 KiB block size.
- There can be four extents stored directly in the inode. When there are more than four extents to a file, the rest of the extents are indexed in a tree.
 - Linux previously used EXT2 and EXT3, now it uses EXT4.

Linux

- Directories
 - inode number == inode number

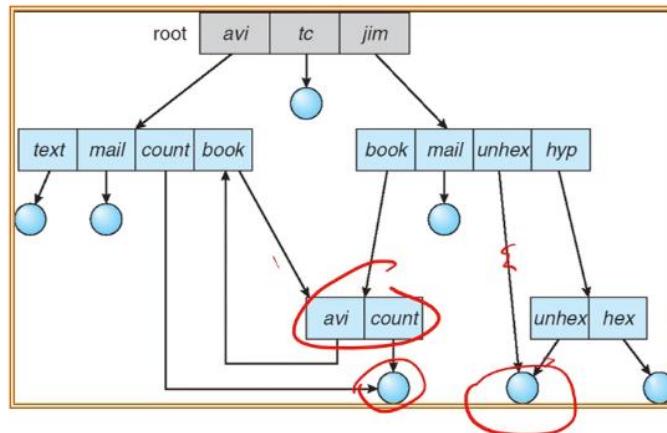
- remember, a directory is really just a special file; the contents of the file is simply the table of (file name, inumber) pairs.
- When we open a file, we usually load the directory to which the file belongs to in cache, since there is a good chance other files from the same directory will be queried and opened.

Directories

- **Directory:** a relation used for naming
 - Just a table of (file name, inumber) pairs
- How are directories constructed?
 - Directories often stored in files
 - Reuse of existing mechanism
 - Directory named by inode/inumber like other files
 - Needs to be quickly searchable
 - Options: Simple list, Hashtable, or B-tree
 - Can be cached into memory in easier form to search
- How are directories modified?
 - Originally, direct read/write of special file
 - System calls for manipulation: `mkdir`, `rmdir`
 - Ties to file creation/destruction
 - On creating a file by name, new inode grabbed and associated with new file in particular directory
- Remember, inode/inumber is what the system needs in order to associate ASCII file name in a particular directory with the actual disk block information (in order to talk to the hardware).
- Directory Organization/Structure

Directory Organization

- Directories organized into a hierarchical structure
 - Seems standard, but in early 70's it wasn't
 - Permits much easier organization of data structures
- Entries in directory can be either files or directories
 - Usually directories pointer entries that point to its current directory, and also to its parent directory.
 - Inodes contain a value called LinkCount: this tells number of directories linked to the same file or directory.
 - Allows for tracking number of hardlinks.
 - Use ln (LN) command to link a file or directory to a directory in Linux.
 - Use ln (LN) -s parameter to make it a softlink instead of a hardlink.
 - Softlinks have their own separate inodes



- Not really a hierarchy!
 - Many systems allow directory structure to be organized as an acyclic graph or even a (potentially) cyclic graph
 - Hard Links: different names for the same file
 - Multiple directory entries point at the same file
 - Soft Links: “shortcut” pointers to other files
 - Implemented by storing the logical name of actual file
- **Name Resolution:** The process of converting a logical name into a physical resource (like a file)
 - Traverse succession of directories until reach target file
 - Upon first access to a directory, it can take many disk reads to get to the desired directory. But after that, it is cached, so next/recent time you try to access the directory, since the location is cached then, it will be much faster to access since many (or not as many) disk reads will not be necessary.
 - Also, every process has a current working directory (the directory to which the process/executable belongs to on the disk).
 - Likely the CWD (current working directory) is already cached, which allows for relative filename instead of absolute path.

Directory Structure (Con't)

- How many disk accesses to resolve "/my/book/count"?
 - Read in file header for root (fixed spot on disk)
 - Read in first data block for root
 - Table of file name/index pairs. Search linearly – ok since directories typically very small
 - Read in file header for "my"
 - Read in first data block for "my"; search for "book"
 - Read in file header for "book"
 - Read in first data block for "book"; search for "count"
 - Read in file header for "count"

- Current working directory: Per-address-space pointer to a directory (inode) used for resolving file names
 - Allows user to specify relative filename instead of absolute path (say CWD="/my/book" can resolve "count")
- File System Caching
 - Can afford LRU page replacement overhead since disk access is not that frequent (compared to main memory).
 - Some systems allowing applications to request other replacement policy is useful for security, but also it allows the application to tell the system that the default policy will not work as well/efficiently based on the type of process the application is running (for example, a virus scan process scans all system files, thus the cache will be flushed with data (path mappings in this case) used only once, thus this is very inefficient; this is why sfc scannow virus scan or checkdisk Windows applications are very slow processes, since many disk reads will be needed as caching is not really effective since all system paths are checked).

File System Caching

- Key Idea: Exploit locality by caching data in memory
 - Name translations: Mapping from paths→inodes
 - Disk blocks: Mapping from block address→disk content
- **Buffer Cache:** Memory used to cache kernel resources, including disk blocks and name translations
 - Can contain “dirty” blocks (blocks not yet on disk)
- Replacement policy? LRU
 - Can afford overhead of timestamps for each disk block
 - Advantages:
 - Works very well for name translation
 - Works well in general as long as memory is big enough to accommodate a host’s working set of files.
 - Disadvantages:
 - Fails when some application scans through file system, thereby flushing the cache with data used only once
- Other Replacement Policies?
 - Some systems allow applications to request other policies
 - Example, ‘Use Once’:
 - File system can discard blocks as soon as they are used

45

- Too much memory allocated to buffer cache is not as big of a concern (it used to be a big concern in the past when main memory capacity was much lower), since on most modern machines, memory usage is less than 50% (well below the 90% threshold to keep system running smoothly and efficiently). Nevertheless, in an even where a user does need to run a large enough amount of applications that require a lot of memory, then there is a chance that OS could allocate too much buffer cache (which uses same main memory as processes do), thus taking up too much main memory and not leaving sufficient space for more processes to run concurrently/in parallel.
- Note: paging and file access are essentially the same idea, where you use cache (either cpu buffer or use main memory as a buffer or both), to access a block of data; in fact you must page the correct block of disk memory in order to load in the directory/file metadata (which gives location of the file on the disk).
- Remember, the kernel is just a process; thus it has its own address space (data, code, heap, stack, etc.); so when you see “kernel memory”, it means the address space of the kernel process. And remember too, the kernel process is the process that is always running; it is a central command process used by/is apart of the OS and determines how OS functions.
- In modern day, the disk controller have a lot of memory and can even cache entire tracks, so as to help greatly optimize and speed up disk access (and assist in pre fetching).

File System Caching (con't)

- Cache Size: How much memory should the OS allocate to the buffer cache vs virtual memory?
 - Too much memory to the file system cache \Rightarrow won't be able to run many applications at once
 - Too little memory to file system cache \Rightarrow many applications may run slowly (disk caching not effective)
 - Solution: adjust boundary dynamically so that the disk access rates for paging and file access are balanced
- **Read Ahead Prefetching:** fetch sequential blocks early
 - Key Idea: exploit fact that most common file access is sequential by prefetching subsequent disk blocks ahead of current read request (if they are not already in memory)
 - How much to prefetch?
 - Too many imposes delays on requests by other applications
 - Too few causes many seeks (and rotational delays) among concurrent file requests
 - Note: a buffer is simply a memory location – usually volatile, and can have various speeds (i.e., L1 cache buffer vs. a main memory cache buffer).
 - Remember, kernel cache is visible/can be accessed via system calls, thus if you copy user buffer to kernel buffer, then all other processes can see that updated kernel buffer and use that cache universally (shared among all process).
 - Using the flush() command tells OS to flush all (or a specific) buffers (clear them out, and thus write any dirty data to the file).
 - This is why sometimes cout will not immediately output display; if you want it to be immediate, you need to add flush() of the cout buffer in your program (cout.flush()) after using cout so that it can be written to display without delay (as it is cached, but in this case, it is not a I/O disk file, but a console/monitor output abstracted as an I/O file, thus if data is written in kernel cache and not yet written to console I/O file, then it will not show up on the screen, even though other processes can "see" that output via kernel cache).

File System Caching (con't)

- **Delayed Writes:** Writes to files not immediately sent out to disk
 - Instead, `write()` copies data from user space buffer to kernel buffer (in cache)
 - Enabled by presence of buffer cache: can leave written file blocks in cache for a while
 - If some other application tries to read data before written to disk, file system will read from cache
 - Flushed to disk periodically (e.g. in UNIX, every 30 sec) *flush*
 - Advantages:
 - Disk scheduler can efficiently order lots of requests
 - Disk allocation algorithm can be run with correct size value for a file
 - Some files need never get written to disk! (e.g. temporary scratch files written /tmp often don't exist for 30 sec)
 - Disadvantages
 - What if system crashes before file has been written out?
 - Worse yet, what if system crashes before a directory file has been written out? (lose pointer to inode!) 47
 - After a system crash, when it reboots, you will often see the OS file system try to repair the file system since it can be in an inconsistent state (due to lost data from kernel cache never being written to the disk).
 - Solution for repair/recovery is done through log files that the file system keeps on the disk, so that it has some information to work with in piecing back together the file system/state of all files on the disk of the system. This allows for rolling back or moving forward (if possible, which the recovery code determines using the log files).
 - NTFS and EXT4 implement this, and there has been many improvements recently.
 - What about Crashes?

What about crashes?

- Better reliability through use of log
 - All changes are treated as transactions.
 - A transaction either happens completely or not at all
 - A transaction is committed once it is written to the log
 - Data forced to disk for reliability
 - Process can be accelerated with NVRAM
 - Although File system may not be updated immediately, data preserved in the log

Commit or rollback

-
- Other ways to make a file system durable?
 - For distributed systems and most cloud computing, there is the 3 X 2 rule: in a single data center, you maintain 2 or 3 copies of all files, and that data also needs to be replicated (copied) to at least 2 different data centers == 6 total copies spread among two data centers.

Other ways to make file system durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
 - Can allow recovery of data from small media defects
- Make sure writes survive in short term
 - Either abandon delayed writes or
 - use special, battery-backed RAM (called non-volatile RAM or NVRAM) for dirty blocks in buffer cache.
- Make sure that data survives in long term
 - Need to replicate! More than one copy of data!
 - Important element: independence of failure
 - Could put copies on one disk, but if disk head fails...
 - Could put copies on different disks, but if server fails...
 - Could put copies on different servers, but if building is struck by lightning....
 - Could put copies on servers in different continents...

replication

3 X 2

-
- Free-Space Management

Free-Space Management

- File system maintains **free-space list** to track available blocks/clusters
 - (Using term “block” for simplicity)
 - **Bit vector** or **bit map** (n blocks)



$$\text{bit}[i] = \begin{cases} 1 & \Rightarrow \text{block}[i] \text{ free} \\ 0 & \Rightarrow \text{block}[i] \text{ occupied} \end{cases}$$

Block number calculation

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit

CPUs have instructions to return offset within word of first “1” bit

○

Free-Space Management (Cont.)

- Bit map requires extra space
 - Example:

block size = 4KB = 2^{12} bytes

disk size = 2^{40} bytes (1 terabyte)

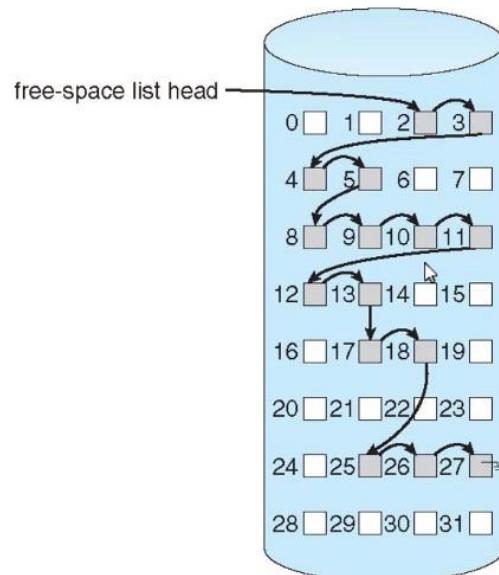
$n = 2^{40}/2^{12} = \underline{2^{28} \text{ bits}}$ (or 32MB) $\overset{2^{25}}{=} 2^5 \text{ MB}$
if clusters of 4 blocks \rightarrow 8MB of memory

- Easy to get contiguous files

- - 2^{28} bits needed to represent 2^{28} 4KB-blocks of memory. 8 bits in 1 byte, or 2³ bits in 1 byte; thus $2^{28}/2^3 = 2^{25}$ bytes = 32MB.
- Linked Free Space List on Disk
 - Most file systems use and prefer bitmap (NTFS and EXT); but linked free space list is another choice.

Linked Free Space List on Disk

- Linked list (free list)
 - Cannot get contiguous space easily
 - No waste of space
 - No need to traverse the entire list (if # free blocks recorded)



Free-Space Management (Cont.)

- Grouping
 - Modify linked list to store address of next $n-1$ free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)
- Counting
 - Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering
 - Keep address of first free block and count of following free blocks
 - Free space list then has entries containing addresses and counts
 - Through grouping (for free space linked list or counting (for bitmap), this is how you maintain (search for a desired number of free blocks for allocation) free space without having to traverse entire free space list.

Lecture Video 15 (week13 – 4/04/22): New Technologies File System (NTFS)

- Used by Microsoft Windows systems
 - Started as a joint project between IBM and Microsoft to develop OS/2 (OS designed for a PC).
- NTFS

NTFS

- Default file system for Windows 10, 8, 7, Vista, XP, 2000, NT, and Windows Servers
- No published spec from Microsoft that describes the on-disk layout
- Good sources for NTFS information
 - www.ntfs.com
 - Dr. Guo teaches Digital Forensics course at UM-Dearborn, which covers many file systems such as NTFS, EXT..etc..

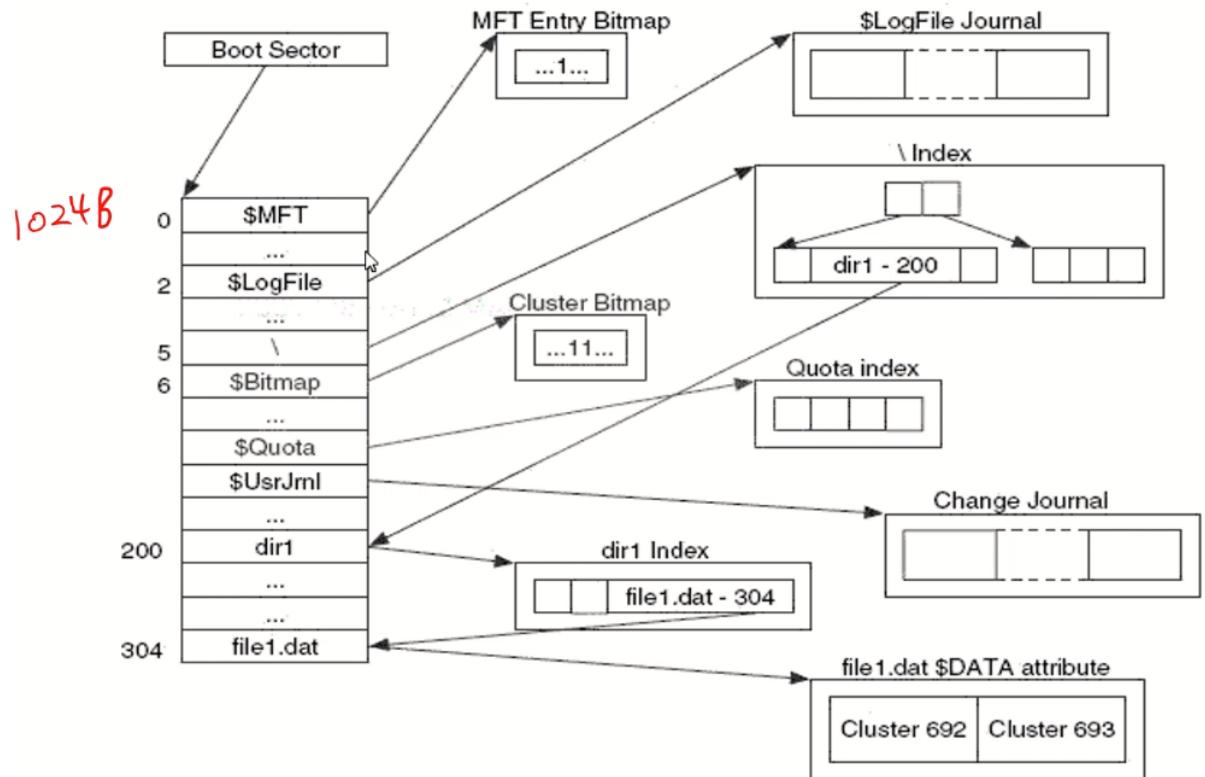
NTFS

- NTFS was designed for
 1. Reliability,
 2. Security, and
 3. Scalability.
- Scalability is provided by the use of generic data structures that wrap around data structures with specific content.
 - This is a scalable design because the internal data structure can change over time as new demands are placed on the file system, and the general wrapper can remain constant.
 - e.g. a generic wrapper in which every byte of data in an NTFS file system is allocated to a file
 -
- Everything is a File
 - So unlike EXT (uses inode and data and super block and bitmap areas) or FAT (uses FAT Table and data areas), there is no special region for the metadata (file attributes, etc.).
 - These types of systems essentially have a data area, and a metadata area, and of course the boot sector and boot code.
 - In NTFS, everything is a file; all the metadata is saved in each file; the file itself contains the metadata information for itself.
 - Most NTFS allocate first 16 sectors for the boot code.

Everything is a File

- An NTFS file system does not have a specific layout like other file systems, e.g., FAT.
 - The entire file system is considered a data area, and any sector can be allocated to a file.
- The only consistent layout is that the **first sectors of the volume contain the boot sector and boot code.**
- NTFS Example
 - MFT = master file table
 - \$ symbol before name means it is a reference (pointer)

NTFS Example



- Each record/file is 1024 bytes (includes data and metadata for each file).
- Master File Table
 - A file itself; it is the most important file.
 - NTFS considers each file just a list of attributes, with data being one of those attributes.

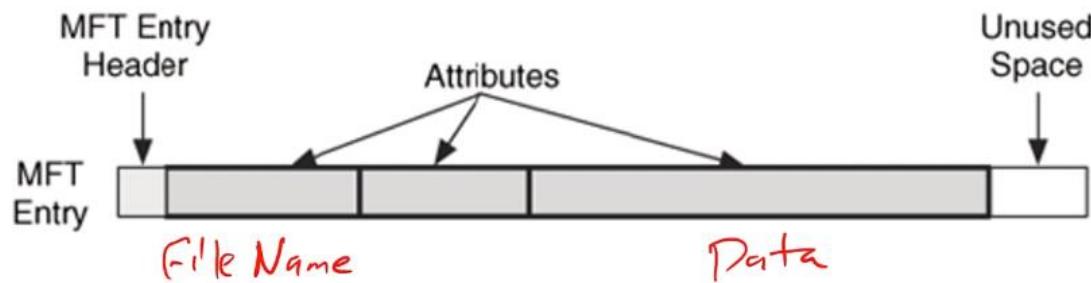
Master File Table ✓

- Contains information about all files and directories
- Every file and directory has at least one entry in the table
- Each entry is simple
 - Size: 1 KB
 - Entry header is first 42 bytes
 - Remaining bytes store attributes

- MFT Concepts

MFT Concepts

- The Master File Table (MFT) is the heart of NTFS.
- Each entry is 1 KB in size, BUT
 - only the first 42 bytes have a defined purpose.
 - the remaining bytes store attributes, which are small data structures that have a very specific purpose.
 - For example, one attribute is used to store the file's name, and another is used to store the file's content.
 - Figure below shows the basic layout of an MFT entry where there is some header information and three attributes.



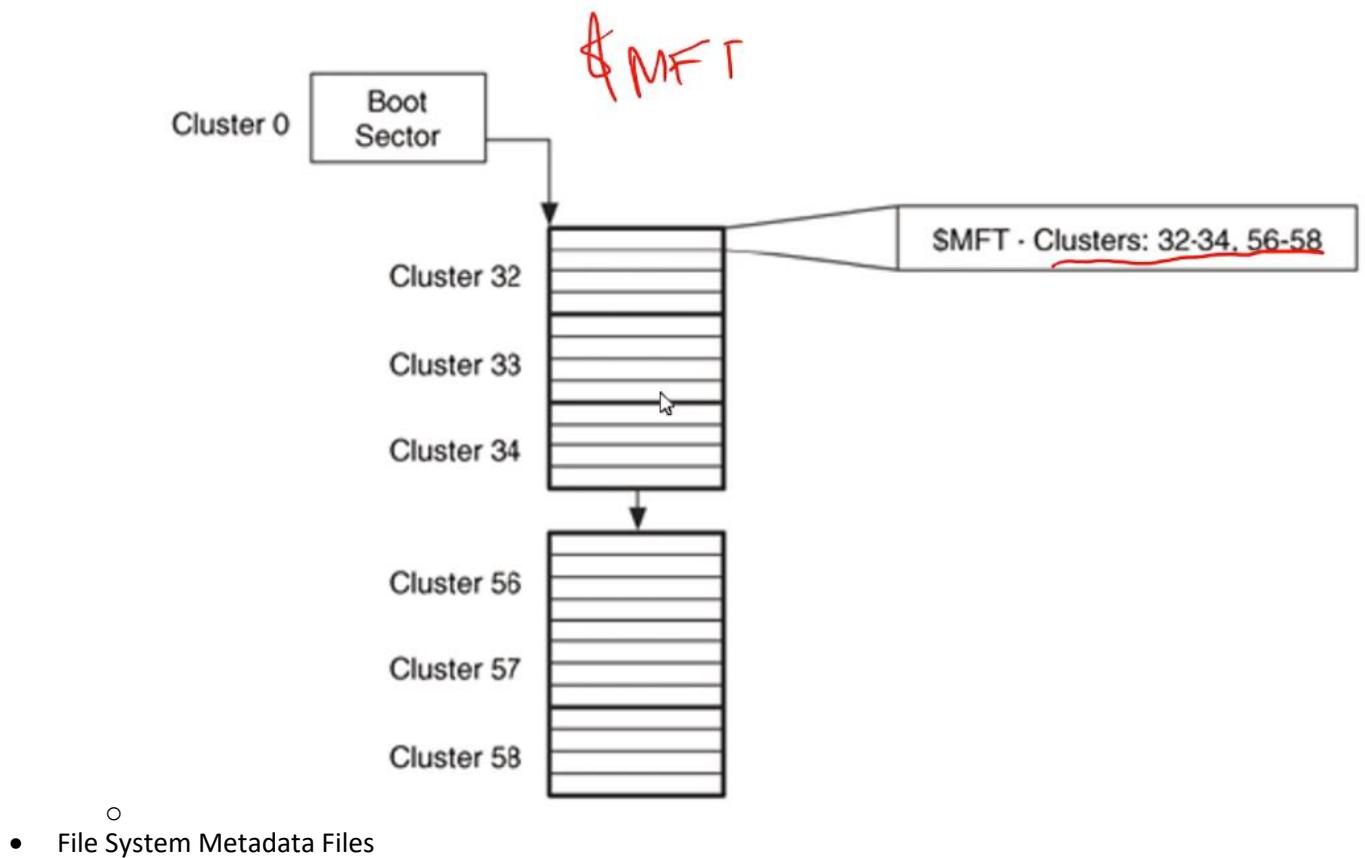
- MFT Concepts
 - MFT entry 0 is a record of the MFT table itself.

MFT Properties

- Microsoft calls each entry in the table **a file record**, but calling each entry **an MFT entry** is simpler.
- Each entry is given an address based on its location in the table, **starting with 0**. 
- To date, **all entries have been 1,024 bytes in size**, but the exact size is defined in the boot sector.
 - We can easily change (redefine) size of each entry to be larger (let's say for example 2048 bytes) → scalability.

MFT Properties: *The MFT is a file*

- The MFT has an entry for itself.
- The first entry in the table is named \$MFT, and it describes the on-disk location of the MFT.
- In fact, it is the only place where the location of the MFT is described; therefore, you need to process it to determine the layout and size of the MFT.
- The starting location of the MFT is given in the boot sector, which is always located in the first sector of the file system.
- Consider, for example, the case where the MFT is fragmented and goes from clusters 32 to 34 and 56 to 58.
 - **The boot sector is used to find the first MFT entry.**
 - MFT does not have to be contiguous; it can be fragmented (probably for security).



File System Metadata Files

- In NTFS file system, every byte in the volume is allocated to a file, therefore, how to find **the file system's administrative data or metadata files (according to Microsoft)**.
 - Microsoft documentation says it reserves only **the first 16 entries**, but in practice the first entry that is allocated to a user file or directory is **entry 24**. Entries 17 to 23 are sometimes used as overflow when the reserved entries are not enough.
- Microsoft reserves the **first 16 MFT entries** for **file system metadata files**.
- These reserved and unused entries are in an allocated state and have only basic and generic information.
- Every file system metadata file is listed in the root directory, although they are typically hidden from most users.
- The name of each file system metadata file begins with a "\$," and the **first letter is capitalized**.
 - The standard file system metadata files
 - Remember, logfiles can be used for recovery if system crashes.

The standard NTFS file system metadata files.

Entry	File Name	Description
0	\$MFT	The entry for the MFT itself.
1	\$MFTMirr	Contains a backup of the first entries in the MFT.
2	\$LogFile	Contains the journal that records the metadata transactions .
3	\$Volume	Contains the volume information such as the label, identifier, and version.
4	\$AttrDef	Contains the attribute information, such as the identifier values, name, and sizes.
5	.	Contains the root directory of the file system.
6	\$Bitmap	Contains the allocation status of each cluster in the file system.
7	\$Boot	Contains the boot sector and boot code for the file system.
8	\$BadClus	Contains the clusters that have bad sectors.
9	\$Secure	Contains information about the security and access control for the files (Windows 2000 and XP version only).
10	\$Upcase	Contains the uppercase version of every Unicode character.
11	\$Extend	A directory that contains files for optional extensions. Microsoft does not typically place the files in this directory into the reserved MFT entries.

- \$MFTMirr File Overview

- Small file (just 4 KB)

\$MFTMirr File Overview

- As the \$MFT file is used to find all other files, therefore, it has the potential of being a single point of failure if the pointer in the boot sector or the \$MFT entry is corrupt.
 - To fix this problem, there exists a backup copy of the important MFT entries that can be used during recovery.
 - **MFT entry 1** is for the **\$MFTMirr** file, which has a non-resident attribute that contains a backup copy of the first MFT entries.
 - The **\$DATA** attribute of the **\$MFTMirr** file allocates clusters in the middle of the file system and saves copies of at least the first four MFT entries, which are for **\$MFT**, **\$MFTMirr**, **\$LogFile**, and **\$Volume**.
-
- \$Boot File Overview

\$Boot File Overview

- The \$Boot file system metadata file is located in MFT entry 7.
 - It contains the boot sector of the file system.
 - This is the only file system metadata file that has a static location.
 - Its \$DATA attribute is always located in the first sector of the file system because it is needed to boot the system.
 - Microsoft typically allocates the first 16 sectors of the file system to **\$Boot**, but only the first half has non-zero data.
 - The NTFS boot sector is very similar to the FAT boot sector, and they share many fields, e.g., share the 0xAA55 signature.
 - The boot sector gives you basic size information about the size of each cluster, the number of sectors in the file system, the starting cluster address of the MFT, and the size of each MFT entry.
 - The boot sector also gives a serial number for the file system.
-

- \$Bitmap File Overview
 - Cluster is same thing as a block (Windows uses the term “cluster”; it is the smallest unit allocated to a file).

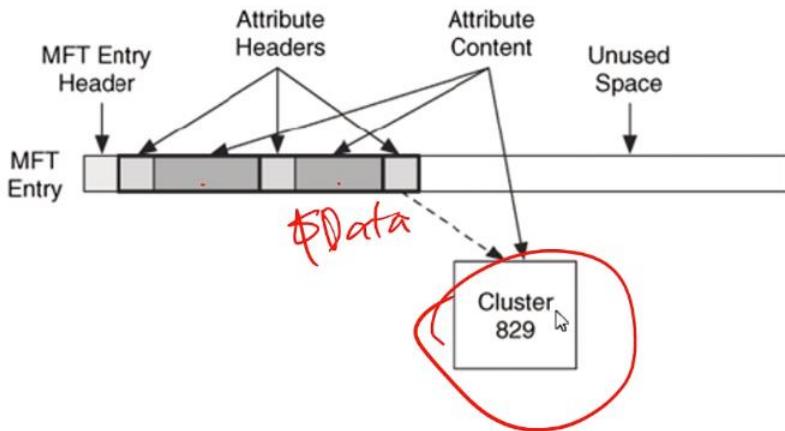
\$Bitmap File Overview

- The \$Bitmap file system metadata file can be used to determine allocation status of a cluster *block*
- It is located in **MFT entry 6**.
- It has a **\$DATA attribute** that has one bit for every cluster in the file system.
- For example, bit 0 corresponds to cluster 0, and bit 1 corresponds to cluster 1.
 - If the bit is set to 1, the cluster is allocated; if it is set to 0, it is not.
- Content Category

Content Category

- A cluster is a group of consecutive sectors, and the number of sectors per cluster is a power of 2 (i.e., 1, 2, 4, 8, 16).
- Each cluster has an address, starting with 0.
- Cluster 0 starts with the first sector of the file system.
- A cluster address can be mapped to a sector address as,
$$\text{SECTOR} = \text{CLUSTER} * \text{sectors_per_cluster}$$
- An NTFS file is a collection of attributes, some of which are resident (store the content in the MFT entry) and others are non-resident (store the content in clusters).
- NTFS does not have strict layout requirements, and any cluster can be allocated to any file or attribute, with the exception of \$Boot always allocating the first cluster.
- Resident vs Non-Resident Attribute

Resident vs Non-resident Attribute

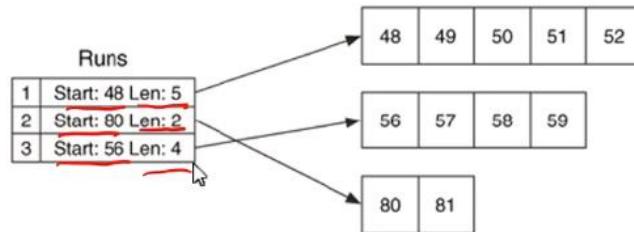


Resident vs Non-Resident Attributes

- A resident attribute stores its content in the MFT entry
 - A non-resident attribute stores its content in external clusters
 - Non resident attributes are stored in cluster runs
 - The attribute header gives the starting cluster address and its run length
- Non-resident Attributes
 - Cluster run is the equivalent of an extent from Linux EXT4 file system.

Non-resident Attributes

- Non-resident attributes are stored in cluster runs, which are consecutive clusters, and a cluster run is documented using **the starting cluster address** and **the run length**.
- Example:**
 - Consider an attribute that has allocated clusters 48, 49, 50, 51, and 52, it has a run that starts in cluster 48 with a length of 5.
 - If the attribute also allocated clusters 80 and 81, it has a second run that starts in cluster 80 with a length of 2. A third run could start at cluster 56 and have a length of 4.



- Standard Attribute Types
 - Standard Attribute Types

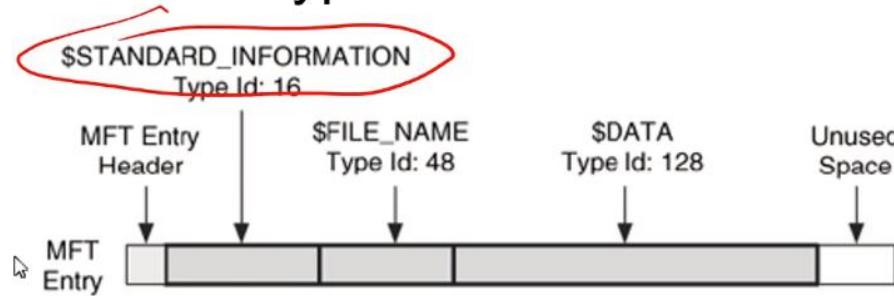
Standard Attribute Types

- The standard attributes have a **default type value** assigned to them, it can also be redefined in the **\$AttrDef file system metadata file**.
- In addition to a number, **each attribute type has a name**, and it has **all capital letters** and **starts with "\$."**
- Some of the default attribute types and their identifiers are given in the Table next.
- Not all these attribute types and identifiers will exist for every file.
- Note: Microsoft sorts the attributes in an entry using default type value.
- Standard Attribute Type: Example

- Standard information includes 4 time stamps (as well a few other pieces of information): creation date, modify date, when MFT table last updated, last accessed.

Standard Attribute Type: Example

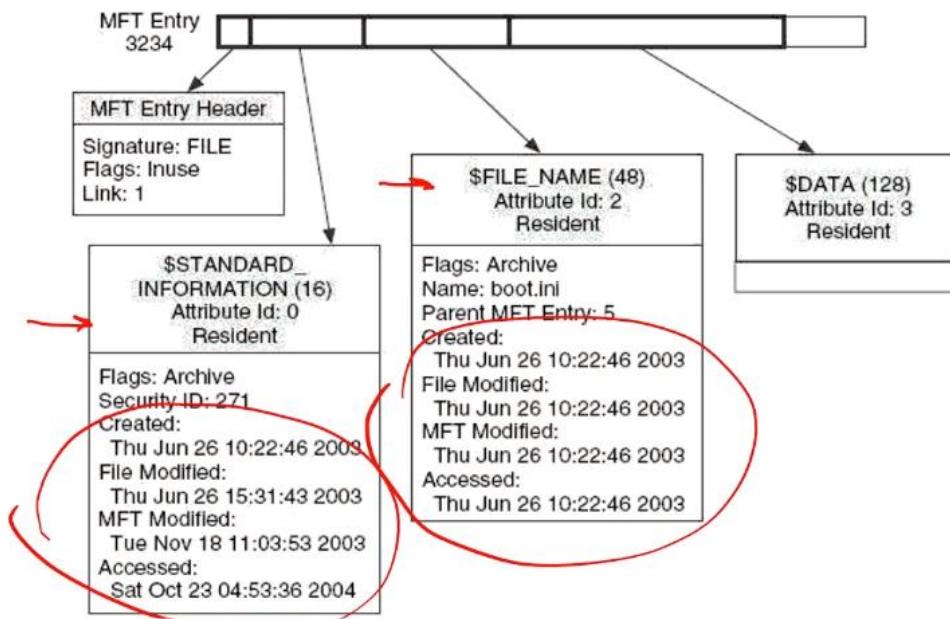
- Consider the following MFT entry with its names and types.



- It has the three standard file attributes. In this example, all the attributes are resident.
 - Metadata
 - For directory file types, they include the filename and standard information attributes (remember a directory is also just a file), which have redundant time stamp information in each attribute – you will find that time stamps in the filename is less frequently updated than the standard information attribute; so in order to see most recent reflection of changes and status of a directory file-type, using the standard information attribute (which is more frequently updated) may be more useful.

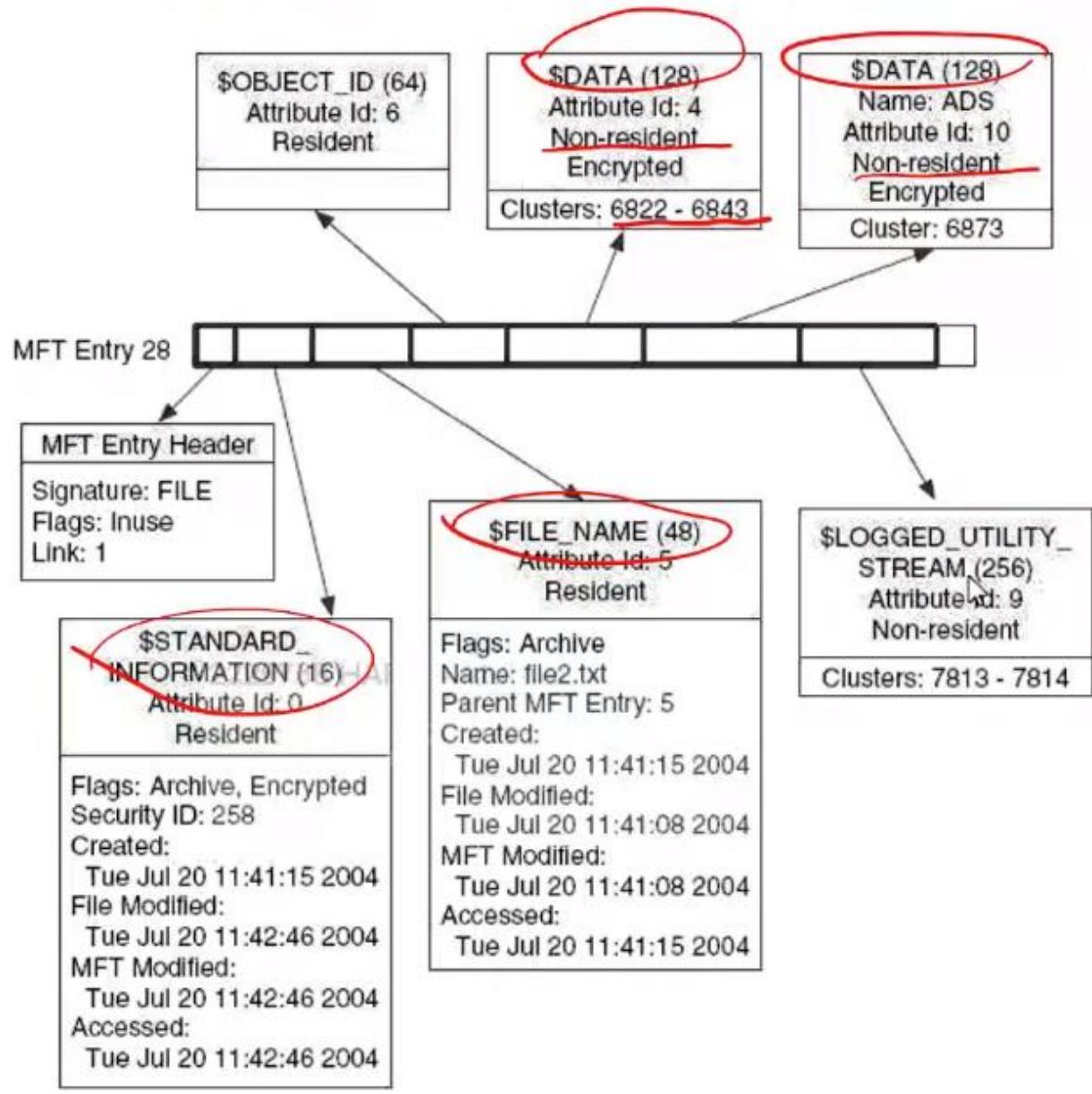
Metadata

- All metadata is stored in one of the attributes, so we will now look at details of the file attributes.
- To review, a typical file has a **\$STANDARD_INFORMATION**, a **\$FILE_NAME**, and a **\$DATA** attribute.
- A typical file that has the standard attributes shown as,



- \$DATA Attribute

\$DATA Attribute



Lecture Video 16 (week14 – 4/11/22): Security

- Protection and Security

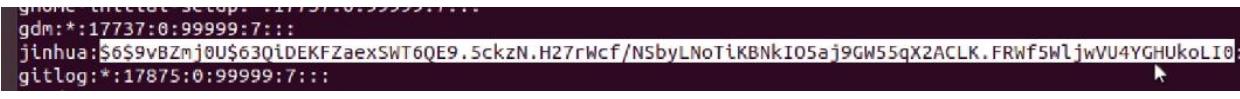
Protection and Security

- Type of misuse
 - Accidental
 - Intentional
- Protection is to prevent either accidental or intentional misuse
- Security is to prevent intentional misuse
 - Three Pieces to Security

Three Pieces to Security

- Authentication – who user is
- Authorization – who is allowed to do what
- Enforcement – make sure people do only what they are supposed to do
 - Authentication

Authentication

- Common approach: **passwords**. Shared secret between two parties. Since only I know password, machine can assume it is me.
- **Problem 1:** System must keep copy of secret, to check against passwords. What if malicious user gains access to this list of passwords?
 - Encryption – transformation that is difficult to reverse without the right key.
 - For example: UNIX /etc/passwd or Linux /etc/shadow file
 - MD5 SHA*
 - passwd -> one way transform -> encrypted passwd
 - System stores only encrypted version, so ok even if someone reads the file!
- 

```
gnome-terminal setup...:17737:0:99999:7:::
gdm:*:17737:0:99999:7:::
jinhua:$6$9vBZmJ0U$63QlDEKFZaexSWT6QE9.5ckzN.H27rWcf/NSbyLNoT1KBNkI05aj9GW55qX2ACK.FRWf5WLjwVU4YGHUkoLI0:gitlog:*:17875:0:99999:7:::
```
- So for example, in the screenshot, the actual encrypted password is stored in the etc/shadow file. So, if someone were to get a copy of the list of passwords, they wouldn't be able to decrypt them; when user inputs their password, it goes through a hash function of some level (256-bit encryption for example), and if it is resolved to be the same hash (as shown above) then user can gain access. If you were to simply input the above encrypted password, then it would be passed into a function first, and then obviously it would not produce the correct result. You need the original value used and passed into the (nearly irreversible) encryption function in order to output what is shown above, which then the OS will compare to the encrypted data base hash output, and if they match, then you gain access.
 - Thus, if passwords are only stored in the encrypted format, then even system administrators cannot know the password (although they may be able to reset it still of course).

Authentication (cont)

- **Problem 2:** Passwords must be long and obscure
 - Paradox: short passwords are easy to crack; long ones, people write down.
 - Technology means we have to use longer passwds; UNIX initially required only lowercase, 5 letter passwords
 - How long for an exhaustive search? $26^5 = 10$ million
 - In 1975, 10 ms to check a passwd -> 1 day
 - In 1992, 0.001ms to check a passwd -> 10 seconds
 - Many people choose even simple passwords, such as English words
 - Problem with using simply an English word is that there are only a few million of them; so it can be easier to crack even with symbols or capitalization (although symbols and capital definitely makes it significantly harder).

Authentication (cont)

- Some Solutions for Problem 2
 - Require more complete passwords. For example: 6 letter, with upper and lower and number and special: $70^6 \sim 600$ billion, or 6 days
 - Make it take a long time to check each password. For example, delay every remote login attempt by 1 second
 - Assign very long password.
- **Problem 3:** Can you trust the encryption algorithm? Recent example: one thought to be safe, has a back door. If there is a back door, means you don't need to do complete exhaustive search.
 - Backdoor means: there could be some kind of special username and password that is always allowed in.
- Security in Distributed Systems

5

telenet
userID, password
ssh

Security in Distributed Systems

- Cryptography functions
 - Symmetric-key (e.g., DES)
 - Public key (e.g., RSA)
 - Message digest (e.g., MD5)
- Security services
 - Confidentiality: preventing unauthorized release of information
 - Integrity: making sure message has not been altered
 - Authentication: verifying identity of the remote participant
- Authorization

Authorization

- Authorization: who can do what
- Access control matrix: formalization of all the permissions in the system

Objects	<u>file1</u>	<u>file2</u>	<u>file3</u>	...
Users				
A	rw	r		
B		rwx		
C	r		rwx	
...				→

- Potentially huge # of users, operations, so impractical to store all of these.

○

Authorization (cont)

1. Access control list – store all permissions for all users with each object
 - Still might be lots of users! UNIX addresses this by having each file stores: r, w, x for owner, group, world.
2. Capability list – each process, stores all objects the process has permission to touch
 - Lots of capability systems built in the past; idea out of favor today. But page tables are an example.

○

- Access Control List: Only need 9 bits for each file in UNIX, 3 bits to set rwx permissions (read write execute) for 3 sets of users (owner, group, and world/everyone else).
- Enforcement

Enforcement

- Enforcer checks passwords, access control lists, etc.
- Any bug in enforcer means: way for malicious user to gain ability to do anything.
 - In UNIX, superuser has all the powers of the UNIX kernel – can do anything. Because of coarse-grained access control, lots of stuff has to run as superuser in order to work. If bug in any one of these programs, you're hosed.
 - Problem: superusers have too much power.
- Enforcement Paradox

sudo

Paradox:

- a. Make enforcer as small as possible
Easier to make correct, but simple-minded protection model
- b. Fancy protection – only minimal privilege necessary, hard to get right
 - State of the World in Security

State of the World in Security

- Authentication – encryption
 - But almost nobody encrypts!
- Authorization – access control
 - But many systems provide only very coarse-grained access control (ex: UNIX – means, need to turn off protection to enable sharing)
- Enforcement – kernel mode
 - Hard to write a million line program without bugs, and any bug is a potential security loophole.
 - - Many security processes must be ran as a super user; thus any bugs can allow for a super user to do serious malicious damage.
- Program Threats
 - Many mobile device apps are free because they contain trojan horse spyware threats.
 - Trap door == backdoor.
 - Logic Bombs → employed by disgruntled employees; they could for example make it so that “if I don’t work any longer in this company, i.e. my user profile is deleted or privileges revoked, then delete these files”.

Program Threats

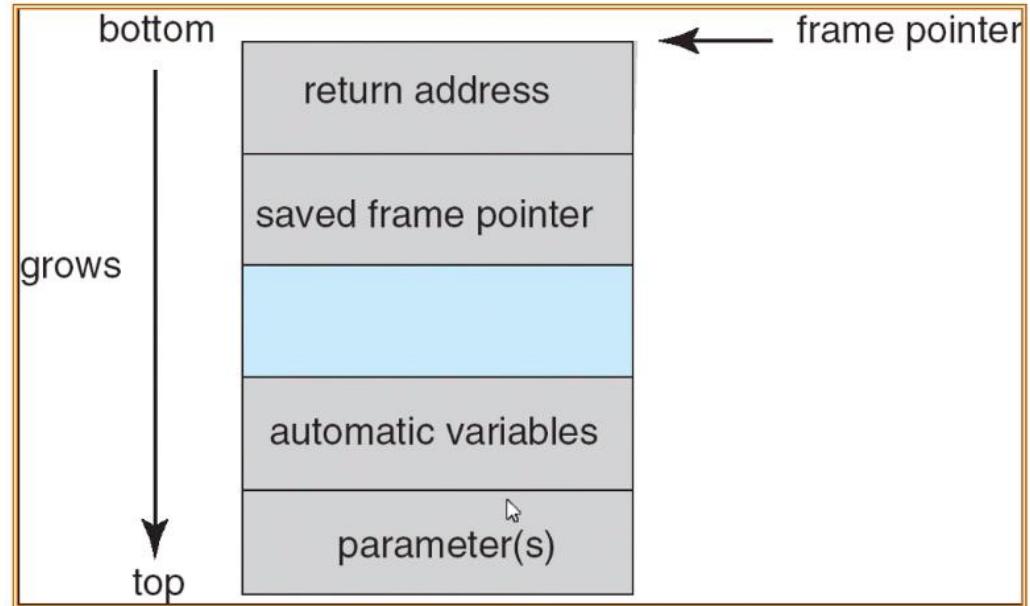
- **Trojan Horse**
 - Code segment that misuses its environment
 - Exploits mechanisms for allowing programs written by users to be executed by other users
 - **Spyware, pop-up browser windows, covert channels**
- **Trap Door**
 - Specific user identifier or password that circumvents normal security procedures
 - Could be included in a compiler
- **Logic Bomb**
 - Program that initiates a security incident under certain circumstances
- **Stack and Buffer Overflow**
 - Exploits a bug in a program (overflow either the stack or memory buffers)
- C Program with Buffer-overflow Condition

C Program with Buffer-overflow Condition

```
#include <stdio.h>
#define BUFFER_SIZE 256
int main(int argc, char *argv[])
{
    char buffer[BUFFER_SIZE];
    if (argc < 2)
        return -1;
    else {
        strcpy(buffer, argv[1]);
        return 0;
    }
}
```

- What happens if a passed in input parameter exceeds 256 bytes?
 - Buffer overflow.
 - Can cause security concerns.

Layout of Typical Stack Frame



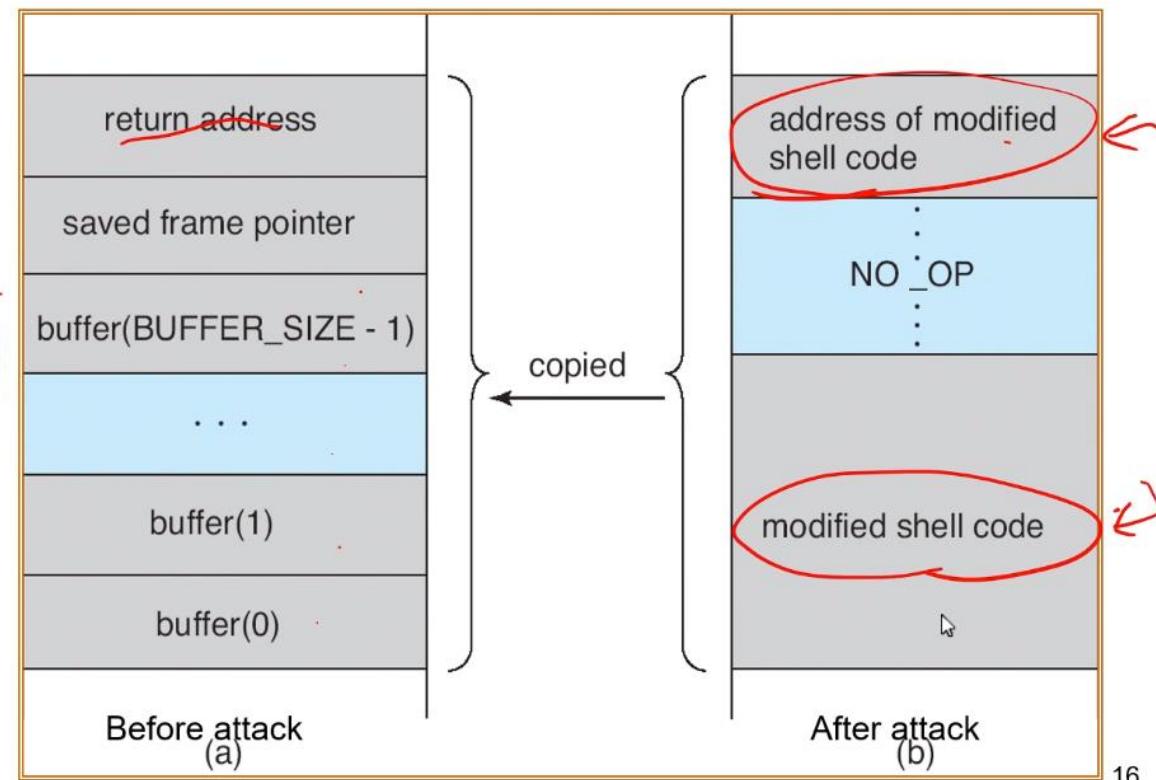
Modified Shell Code

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    execvp(''\bin\sh'', ''\bin\sh'', NULL);
    return 0;
}
```

- Above is a simple shell program. It is a main program, just like the buffer overflow program.
- If the buffer overflows into the return address space of the main function, then essentially the return address will be an invalid address – which can be taken advantage of by making it a valid address by making those overflow values a return address– namely, the address of the main function of the malicious shell program. Now, you can see how a user can manipulate the system by overflowing the buffer of a user program, in order to get their own program to

run. Without buffer overflow, then user could not simply start their program after the current user program returns.

Hypothetical Stack Frame



16

- ○ When main program returns, the modified shell code from the hacker will start to run.
 - This is because the return address of the buffer overflow main program is overwritten by the address of the shell program (when the string copy function was called, but the variable “buffer” was overflowed with the shell code address).
 - This will then allow the shell program to run in the system as a legal user.
- So, it is important to avoid buffer overflow in a program.
- One way to avoid this issue (in case programmer forgets to mitigate any potential buffer overflows) is to not allow executable code (the shell code called a function such that the data parameters would go on the stack, but those parameters are binaries/executable code – this is part of finding a way to hack a system by taking advantage of the buffer overflow program) inside of the stack. Stack should only be allowed to store return address pointers (and other address pointers) and data variables (but don’t allow the variables to be executable code (commands, like load and store, etc.)).
- System and Network Threats

System and Network Threats

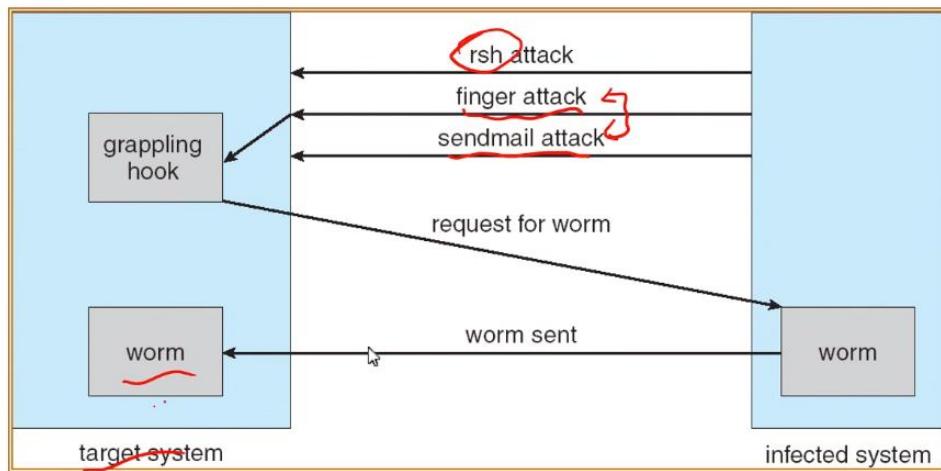
- Worms – use **spawn** mechanism; standalone program
- Internet worm
 - Exploited UNIX networking features (remote access) and bugs in *finger* and *sendmail* programs
 - Grappling hook program uploaded main worm program
- Port scanning
 - Automated attempt to connect to a range of ports on one or a range of IP addresses
- Denial of Service
 - Overload the targeted computer preventing it from doing any useful work
 - Distributed denial-of-service (**DDOS**) come from multiple sites at once
- The Internet Worm
 -

The Internet Worm

- Broke into thousands of computers over internet in 1988.
- Consisted of two programs
 - bootstrap to upload worm
 - the worm itself
- Worm first hid its existence
- Next replicated itself on new machines

- The Morris Internet Worm
 - So it appears that worms, a type of computer virus, can behave much like viruses in the real world; and as with the Morris Internet Worm (which maliciously (but also somewhat accidentally in terms of the extent of damage not thought to occur) got out during testing and got out of control), it can be very dangerous to write and test and manipulate code to make or produce or replicate or improve a computer virus. A very strange but profound paradigm indeed.
 - rsh is a Unix protocol that allowed other systems to remote into the current system, if it is similar to other trusted devices that have gained access, allowing a worm to get access to other machines and spread the virus without needing a password. Then also, via dictionary attack launched, machines that did need a password because a trusted machine or group was not infected in order to gain access were hacked into, now that successfully dictionary attacked machine would be infected, and any other machines that trust it or that is like it and allowed into other machines. So you see he was able to infect thousands of machines this way over time.
 - With enough knowledge and enough missteps and bugs in code, someone even today could perhaps take down the entire (or a very large portion) of the internet and render it useless.

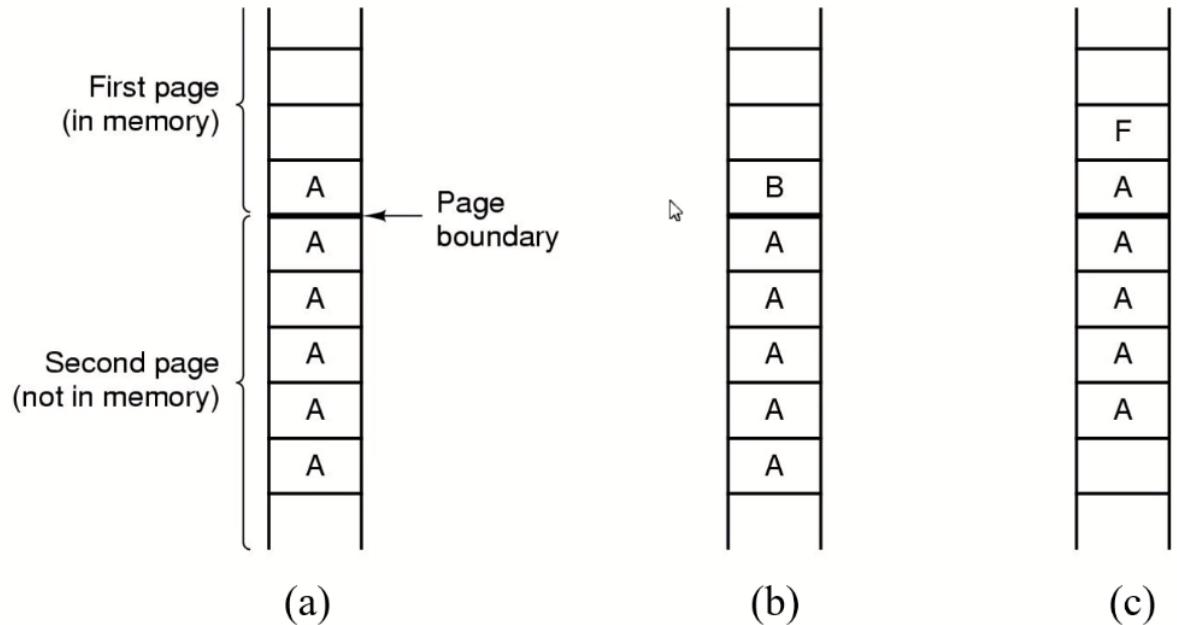
The Morris Internet Worm



- Utilized buffer overflow issues in systems.
- Famous Security Flaws
 - New Tenex OS in the 70's that was supposed to be secure, but within 48hrs of its release, people were able to exploit the system and find all user passwords.

Famous Security Flaws

<https://www.sjoerdlangkemper.nl/2016/11/01/tenex-password-bug/>



The TENEX – password problem

20

o

The TENEX – password problem

- Here's the code for the password check:
`for (i=0; i < 8; i++)
 if (userPassword[i] != realPassword[i])
 go to error;`

256×8
- Looks innocuous, like you'd have to try all combination. 256^8
- But! Tenex also used virtual memory, and it interact badly with the above code.
- Key idea: force page faults at inopportune times:
 - This made it possible to guess user passwords one character at a time

21

- Time difference used: make first letter in memory, next letter on disk; if correct, then page fault will occur, then that letter's page will have to be brought into memory (takes a few milliseconds). If first letter is checked and error returns quickly (a few microseconds), then we know that a guess was wrong. Thus, it would only take 256^8 guesses at most to get the user password.
- Thompson's self-replicating program
 - Ken was one of the inventors of c programming language and Linux OS

Thompson's self-replicating program

- Proposed by Ken Thompson in his Turing award lecture
- Bury trojan horse in binaries, so no evidence in the source
- Replicates itself to every UNIX system in the world, and even to new UNIX's on new platforms. No visible sign
- Gave Ken Thompson the ability to log into any UNIX system in the world.
- Two step:
 1. Make it possible (easy)
 2. Hide it (tricky)
- This is just an idea (but allegedly not deployed).

- Step 1. Modify login.c

A: if (name == "ken")

don't check password

 login as root

Idea is: hide change, so no one can see it.

- Step 2: Modify the C compiler

Instead of having the code in login, put it in the compiler:

B: if see trigger,

insert A into input stream

Whenever the compiler sees a trigger, put A into input stream of the compiler.

Now, don't need A in login.c, just need the trigger.

Need to get rid of the problem in the compiler.

- Step 3: Modify compiler to have:

C: if see trigger2

insert B + C into input stream

This is where self-replicating code comes in!

- Step 4 Compile the compiler with C present – now in the binary for compiler



- Now we trigger 1 in the compiler code is hidden, because the compiler for the compiler has trigger 2 in it, so that trigger 1 is only inserted into the compiler as a binary; which is the ultimate way to hide code, since binary files are unreadable to humans; and perhaps binaries cannot be uncompiled → compilation process is not reversible, thus you could not read the real source code being used for the program (since part of it has a hidden trigger that is inserted only binary and never in the readable human format that is compiled).

- Step 5. Replace code with trigger2
 - Result is – all this stuff is only in the binary for the compiler. Inside the binary there is C, inside that, code for B, inside that code for A. But source code only needs triggers!
 - Every time you recompile login.c, the compiler inserts the backdoor. Every time you recompile the compiler, the compiler re-inserts the backdoor.
 - What happens when you port to a new machine? Need a compiler to generate code; where does that compiler run?
 - On the old machine – C compiler is written in C! So every time you go to a new machine, you infect the new compiler with the old one.
- - Lessons about security threats (Summary)

Lessons

1. Hard to resecure after penetration

What do you need to do remove the backdoor? Remove all the triggers?

What if he left another trigger in the editor – if you ever see anyone removing this trigger, go back and re-insert it!

Re-write the entire OS in assembler? Maybe the assembler is corrupted!

Toggle in everything from scratch every time you log into the computer?

2. Hard to detect when system has been penetrated. Easy to make system forget

3. Any system with bugs has loopholes (and every system has bugs!)

Summary: can't stop loopholes, can't tell if it is happened, can't get rid of it.

- Essentially, security is very hard, and impossible and inevitable.

Lecture Video 17 (week15 – 4/18/22): Distributed Systems and Cloud Computing

- Definition of a Distributed System

Definition of a Distributed System

- A distributed system is a collection of mostly autonomous processors communicating over a communication network and having the following features:
 - **No common physical clock.** The inherent asynchrony amongst the processors
 - **No shared memory.** Requires message-passing for communication.
 - **Autonomy and heterogeneity.** The processors are "loosely coupled" in that they have different speeds and each can be running a different operating system.
 - Why Distributed Systems?

Why Distributed Systems

- Why?
 - to connect physically separate entities
 - to tolerate faults via replication
 - to scale up throughput via parallel CPUs/mem/disk/net
 - to achieve security via isolation

- But:
 - complex: many concurrent parts
 - must cope with partial failure
 - tricky to realize performance potential

- Many machines that are a part of the distributed system; so instead of 1 machine failing every 10 years = 3650 days, if you have 100,000 machines, thus $100k * (1/3650\text{days}) = 20$ to 30 machines will fail per day. So must cope with failure.
- Scalability

Scalability

- Scalability of a system can be measured along at least three different dimensions.
 - First, a system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system.
 - Second, a geographically scalable system is one in which the users and resources may lie far apart.
 - Third, a system can be administratively scalable, meaning that it can still be easy to manage even if it spans many independent administrative organizations.
- Unfortunately, a system that is scalable in one or more of these dimensions often exhibits some loss of performance as the system scales up.
 - There are 3 main scaling techniques: Asynchronous communication, partitioning, and replication.
- Scaling Techniques (1): Asynchronous Communication
 - Avoid blocking a process while waiting for a network response.
 - Example: allow a user to still access and interact with parts of a partially loaded web page, while waiting for the rest of the page to be loaded.

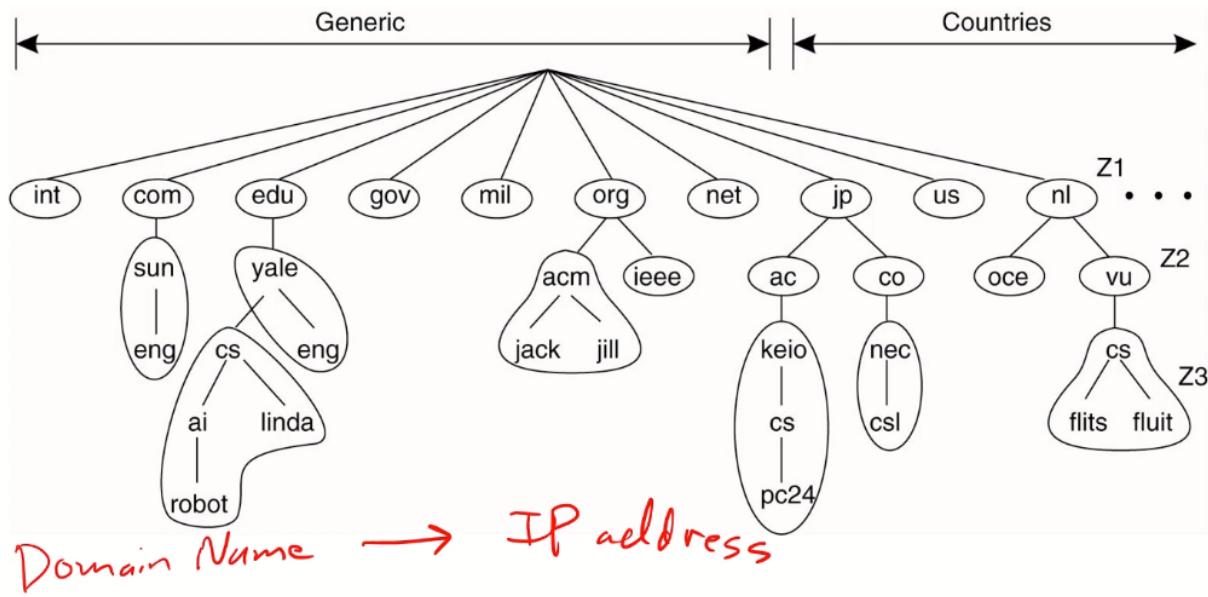
Scaling Techniques (1): Asynchronous Communication AJAX

- Hiding communication latencies is important to achieve geographical scalability
 - Try to avoid waiting for responses to remote (and potentially distant) service requests as much as possible.
 - For example, when a service has been requested at a remote machine, an alternative to waiting for a reply from the server is to do other useful work at the requesters side.
 - Asynchronous communication can often be used in batch-processing systems and parallel applications, in which more or less independent tasks can be scheduled for execution while another task is waiting for communication to complete.
 - Alternatively, a new thread of control can be started to perform the request.
 - Although it blocks waiting for the reply, other threads in the process can continue.
- Scaling Techniques (2): Partitioning

Scaling Techniques (2): Partitioning

- Distribution involves taking a component, splitting it into smaller parts, and subsequently spreading those parts across the system. This is called **partitioning** or **sharding**.
- An excellent example of distribution is the Internet Domain Name System (DNS).
 - The DNS name space is hierarchically organized into a tree of domains, which are divided into nonoverlapping zones

Scaling Techniques (2)



-
- Thousands of servers make up the DNS database to help share the database load in terms of storing all name → ip address information, and serving billions of requests from billions of users.
- Can use a distributed hash table (DHT)
- Support (key, value) store.
 - Ex: key could be facebook user id, and value could be all associated records related to the particular user. Facebook has billions of users, so need to use a distributed system method to store all of the large data tables.
- Scaling Techniques (3): Replication

Scaling Techniques (3): Replication

Single leader and Followers

Multi-leader replication

Leaderless replication

- Overall performance depends on the slowest machine (straggler).

Performance

□ The dream: scalable throughput.

- N servers → N time total throughput via parallel CPU, disk, net.
- So handling more load only requires buying more computers ?

□ Scaling gets harder as N grows:

- Load im-balance, stragglers.
- Non-parallelizable code: initialization, interaction.
- Bottlenecks from shared resources, e.g. network.

○

Fault Tolerance

□ >1000s of servers, complex net → always something broken

- We'd like to hide these failures from the application.

□ We often want:

99. 999 99 %

- Availability -- app can keep using its data despite failures
- Durability -- app's data will come back to life when failures are repaired

99. 999 99 99 %

□ Big idea: replicated servers.

- If one server crashes, client can proceed using the other(s).

○

- When you have so many copies, however, it can get harder and harder to achieve consistency of each replica in the distributed system.

Consistency

❑ Achieving consistency is hard!

- "Replica" servers are hard to keep identical.
- Clients may crash midway through multi-step update.
- Servers crash at awkward moments, e.g. after executing but before replying.
- Network may make live servers look dead; risk of "split brain".

❑ Consistency and performance are enemies.

- "Strong consistency" often leads to slow systems.
- High performance often imposes "weak consistency" on applications.

❑ People have pursued many design points in this spectrum.

- Cloud Computing
 - Cloud computing is the most successful distributed system today.

public

Cloud Computing

❑ Elastic resources

- Expand and contract resources
- Pay-per-use
- Infrastructure on demand



❑ Multi-tenancy

- Multiple independent users
- Security and resource isolation
- Amortize the cost of the (shared) infrastructure

❑ Flexible service management

- Data Centers
 -

Data Centers

- Rented and shared facilities
 - Can be over 300,000 square feet, 100,000 servers, 1 Billion dollars
 - Amazon AWS
 - Microsoft Azure
 - Google AppEngine
 - Cloud computing, Utility computing
- Cloud Service Models

Cloud Service Models

- Software as a Service (SaaS)
 - Provider licenses applications to users as a service
 - E.g., customer relationship management, CRM, e-mail, ...
 - Avoid costs of installation, maintenance, patches...
- Platform as a Service Auto-scaling
 - Provider offers platform for building applications
 - E.g., Google's App-Engine, AWS Elastic Beanstalk
 - Avoid worrying about scalability of platform
- Infrastructure as a service: basically provide customers virtual machines.

Cloud Service Models

Infrastructure as a Service

VM

- Provider offers raw computing, storage, and network
- E.g., Amazon's Elastic Computing Cloud (EC2)
- Avoid buying servers and estimating resource needs

- Four Defining Features

Four Defining Features

Massive scale.

On-demand access: Pay-as-you-go, no upfront commitment. Anyone can access it

Data-intensive nature: What was MBs has now become TBs, PBs and XBs.

- Daily logs, forensics, Web data, etc.
- Humans have data numbness: Wikipedia (large) compress is only about 10 GB!

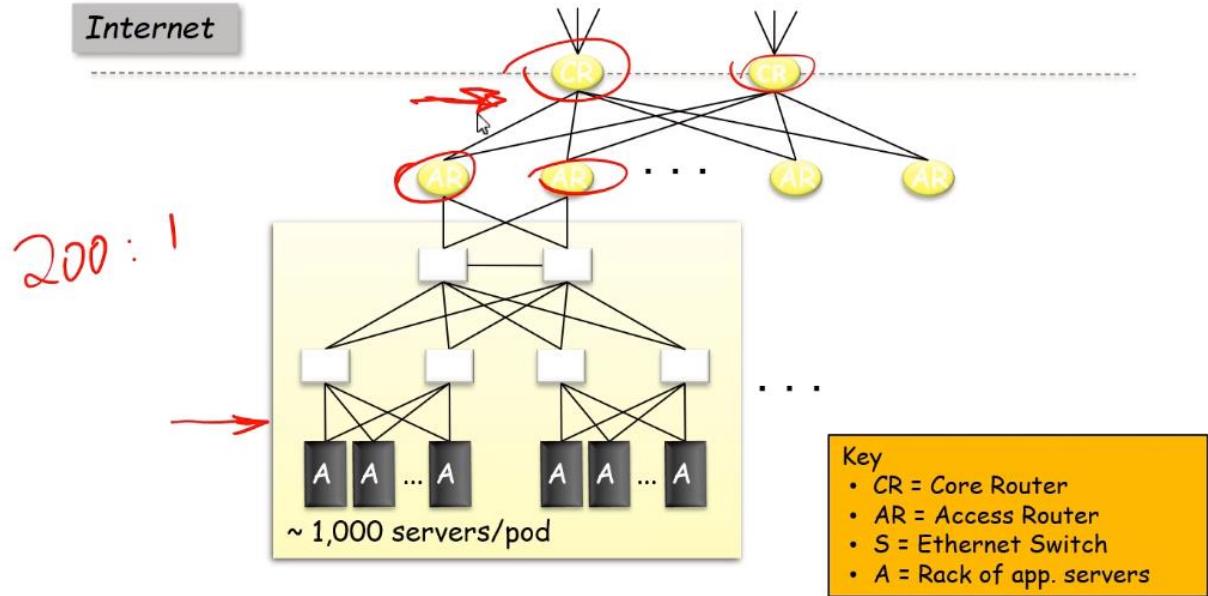
New Cloud Programming Paradigms: MapReduce/Hadoop, NoSQL/Cassandra/MongoDB and many others.

- Massive Scale

Massive Scale

- Facebook [GigaOm, 2012]
 - 30K in 2009 → 60K in 2010 → 180K in 2012
- Microsoft [NYTimes, 2008] 150K machines
 - Growth rate of 10K per month
 - 80K total running Bing
- Yahoo! [2009]: 100K
 - Split into clusters of 4000
- AWS EC2 [Randy Bias, 2009] 40,000 machines
 - 8 cores/machine
- eBay [2012]: 50K machines
- HP [2012]: 380K in 180 DCs
- Google: > 1M machines
 - Data Center Network Topology
 - Need very high-capacity network devices (routers, switches, etc.).

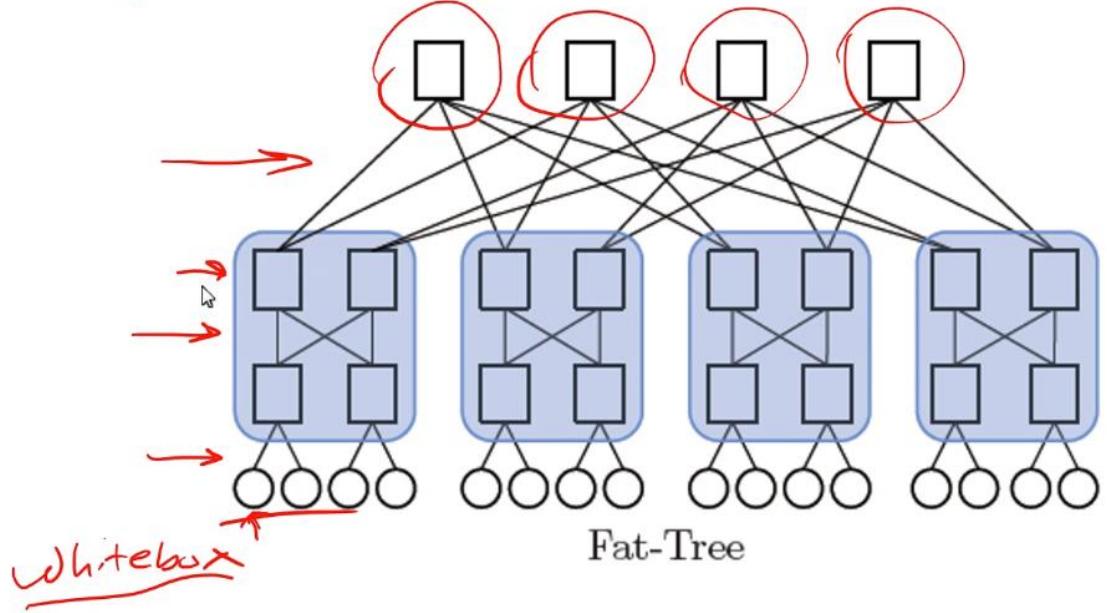
Data Center Network Topology



- Within the same side of a router, you can get 200 times the amount of throughput versus crossing over to another side of the router because when data crosses over even through the very high performance routers, the router can get overloaded and thus the lines get congested and slow down throughput at what becomes a bottleneck.
- Solution is to use Fat Tree Topology.
- Data Center Network: Fat Tree Topology
 - Network capacity at each tree level is around the same; therefore there is almost no over subscriptions/bottlenecks.
 - This will allow for non-blocking since there will be enough throughput between any server in the data centers, since we can connect at full speed without congestion (but still complex to keep a perfect scheduling algorithm to allow for non blocking).
 - Instead of using million dollar routers, we can use regular (few thousand dollars) “white box” switches and servers.
 - Can reduce cost of building data center greatly.

Data Center Network: Fat Tree Topology

No-blocking



- - On Demand Access
 - One example: some services are seasonal: like maybe in summer people use some service more, so you rent more virtual machines, but then in winter you can save a lot of money by only renting a few virtual machines since if demand is much lower during that season.
 - This is why cloud computing is becoming so popular; no large commitment or costs up front; cost is on demand = as needed.

On Demand Access

□ Pay-as-you-go, no upfront commitment. Renting a cab vs. renting a car, or buying one. Ex.:

- AWS Elastic Compute Cloud (EC2): a few cents to a few \$ per CPU hour
- AWS Simple Storage Service (S3): a few cents to a few \$ per GB-month

- Data-intensive Computing

Data-intensive Computing

□ Computation-Intensive Computing ↵

- Example areas: MPI-based, high-performance computing, grids
- Typically run on supercomputers (e.g., NCSA Blue Waters)

□ Data-Intensive

- Typically store data at datacenters
- Use compute nodes nearby
- Compute nodes run computation services

□ In data-intensive computing, the focus shifts from computation to the data: CPU utilization no longer the most important resource metric, instead I/O is (disk and/or network) ↪

- CPU is not the bottleneck for data-intensive computing; network and disk access are. This is because our computing technology is so much greater now, and most of the demand has become for data-intensive computing.

- New Cloud Programming Paradigms

- MySQL databases perform join operations, which is why using that standard is much slower than new cloud computing tools such as Cassandra – which does not use join operations because they are too expensive.

New Cloud Programming Paradigms

- Easy to write and run highly parallel programs in new cloud programming paradigms:
 - Amazon: Elastic MapReduce service (pay-as-you-go)
 - Google (MapReduce)
 - Indexing: a chain of 24 MapReduce jobs
 - ~ 200K jobs processing 50PB/month (in 2006)
 - Yahoo! (Hadoop + Pig)
 - WebMap: a chain of 100 MapReduce jobs
 - 280 TB of data, 2500 nodes, 73 hours
 - Facebook (Hadoop + Hive)
 - ~300TB total, adding 2TB/day (in 2008)
- NoSQL: MySQL is an industry standard, but Cassandra is 2400 times faster! No Join

o