COSC450 Overview

Major Topic #1: Overview of Operating System

Slides #1

Slide #	**Short Summary**
1	Preview
2	Elements used in Computer, OS: Protected software provide interface between hardware and software
3	Macroscopic Diagram Breakdown
4	Hardware in Computer System (Physical Devices, Micro-architecture, Machine Language)
5	Macroscopic Diagram Breakdown (Circle- Software, Shell, Kernel, Hardware)
6	Von Newmann Architecture Diagram
7	Overview of Von Newmann
8	Von Newmann Bottleneck: Memory bottleneck since CPU is faster
9	CPU: executes instructions, utilized instruction cycle (Fetch & Execute Cycle)
10	Fetch Cycle: Reads instruction address, loads it, and moves program counter
11	Diagram (Instruction Cycle: Fetch Cycle)
12	Execute Cycle: Contents of IR decoded & executed, potentially involving interaction with memory and ALU.
13	An OS is: an extended machine from the user's perspective, and a resource manager from the developer's perspective.
14	N/A
15	The first gen of 'OS' consisted of vacuum tubes and plugboards (1945~1980)
16	N/A
17	First Gen used plugboards + tapes, took up whole room, 50 multiplication/sec
18	Second Gen of 'OS': Transistors and Batch System (1955~1965)
19	Picture of Second Gen Tape (from IBM)
20	^N/A
21	^N/A
22	N/A
23	N/A
24	N/A
25	N/A
26	Second Gen used Batch System to optimize use of expensive computer
27	Picture demonstrating workflow
28	N/A
29	Third Gen- IC and Multiprogramming (1965~1980)

30	Third gen launched by IBN, moved to electronic computer systems
31	Third gen used integrated circuit and CPU optimization techniques
32	Third gen used Multiprogramming, where processed are loaded into RAM and ran concurrently. Also uses CPU scheduler.
33	Diagram (Multiprogramming System)
34	Third Gen used Spooling, a buffering mechanism where data is temporarily stored as a file to be processed later
35	^ Spooling improved efficiency and multitasking
36	Diagram (Third gen)
37	Diagram (Third gen time sharing system)
38	Fourth Gen: Personal Computer built with LSI (Large Scale IC), VLSI, ULSI (1980~Present)
39	Fifth Gen: Mobile Computers (i.e. Smartphones)
40	Tons of operating system names

Slides #2

Slide #	**Short Summary**
1	Review (Vonn Newman, Generations of Computers)
2	Preview- Computer System Architecture, Operating System Implementation
3	Preview- Multiprocessor System Types, Multiprogramming, Operating System Operation
4	Modern computers consist of CPU + Memory + I/O Devices, with each device controller maintaining local buffer storage and in charge of a I/O device (also having a device driver, which becomes part of OS)
5	Diagram: I/O Devices -> USB controller, video controller, disk controller, etc.
6	CPU: Brain of computer, retrieves instructions (cycle) from memory to execute. Has cache and registers due to Von Newmann Bottleneck
7	CPU composed of: ALU (Arithmetic Logic Unit), Control Unit, Cache, & Registers (General, Program Counter (PC), Stack Pointer, Program Status Word (PSW))
8	CPU: When process stops running, OS saves content of each register in Process Table to finish the job. CPU performance can be improved by using pipelined design for fetch, decode, and execute process
9	CPU: may have multiple cores/calculation units
10	When I/O devices are ready to receive/send data, interrupts OS by sending signal. For I/O operations, instructions (read/write) are sent to device controller's register, then sent to device local buffer, then checks any error, then driver gives control to other parts of OS
11	Interrupts: key part of how OS & hardware interact. Sends interrupt to CPU via system bus, in which immediately sends execution to fix location for service

	routine. CPU resumes once this is finished. For quickness, OS maintains a table of pointers (interrupt vectors) in low memory for this.
12	Hardware has CPU wire called interrupt-request line, of which the CPU reads the interrupt number from and jumps to handler routine corresponding to number in interrupt vector.
13	CPUs have non-maskable interrupt line & maskable interrupt line
14	Diagram: Interrupt vector layout
15	Interrupts are used throughout OS systems to handle asynchronous events, with device controllers and hardware faulters raising interrupts. This allows the most urgent work to be done first (interrupt priorities), and is used heavily for time-sensitive processing
16	CPU can only load instructions from RAM. Secondary Memory -> Main Memory (RAM) -> Cache Memory -> Registers in CPU
17	Large portion of OS is dedicated to managing I/O, though interrupt-driven I/O can produce high overhead when used for bulk data movement. For this, DMA (Direct Access Memory) controllers can be used, independent from CPU
18	OS controls all I/O devices by: Issue commands to devices, Catching interrupts, handling errors. OS also provides interact between devices & the rest of the system
19	Most I/O devices consist of Mechanical Component, Electrical Components, and Device Driver (Software)
20	The bus in PC is the common pathway between CPU & peripheral devices. Parallel buses use slots on the motherboard, while serial buses have external ports
21	Diagram: Parallel & Serial Transmission
22	Parallel Buses: Offers fast data communication, however supports short distance communication due to crosstalk between the parallel line. Costs more and more pins to connect. Can come in form of Peripheral Component Interconnect (PCI) or Accelerated Graphics Port (AGP)
23	Serial Busses: Offers lower data connection, but supports long distance communication and costs less. Can come in form of Universal Serial Bus (USB) or FireWire (IEEE 1394)
24	Earlier Parallel Buses: Industry Standard Architecture (ISA), Extended ISA (EISA), Micro Channel (MCA), & VESA Local Bus (VL)
25	Diagram: Parallel Buses (Earlier Versions)
26	Single Processor Systems: contains one CPU with a single core. The core executes instructions and has registers to store data locally. These systems may have special-purpose processors as well that do not run processes. OS sends information to these special-purpose processors and monitors their status.
28	Multiprocessor systems allow increased throughput (calc. power), with N number of processors having slightly less than an N speed-up ratio. Includes multicore systems, which can be more efficient than multiple chips with single

	cores due to between-chip communication. One-chip structure uses significantly less power than multiple chips.
29	Diagram: Multiprocessor Systems
30	Multiprocessor Systems: Symmetric Multiprocessing (SMP)- All CPUs share global memory
31	Diagram (Symmetric Multiprocessing Architecture)
32	Multiprocessor Systems: Non-Uniform Memory Access (NUMA)- Each CPU has its own local memory
33	Diagram (Non-Uniform Memory Access)
34	Multiprocessor Systems: Clustered Systems
35	Types of Clustered Systems: Asymmetric Clustering (One machine is hot-standby mode) and Symmetric Clustering (Both machines running application & monitoring each other)
36	Multiprogramming: Tasks are stored in RAM and assigned to CPU using OS CPU scheduler
37	Multiprogramming: Several tasks are uploaded to RAM, and OS maintains process table containing essential information to facilitate multiprogramming
38	Multitasking: a logical extension of multiprogramming, allowing for users to perform more than one task at a time and switch back and forth among them.
39	Dual-Mode and Multimode Operation: Must distinguish between the execution of OS (kernel) and user code (user). A mode bit is added to hardware to indicate current mode (kernel 0, user 1). System must change mode from user to kernel when necessary.
40	Dual mode provides a means of protecting OS from errant users or hackers. This is accomplished by designing machine instructions to only be executed in kernel mode.
41	A system call allows a user program to request services from the operating system by triggering a trap to the interrupt vector, switching to kernel mode, where the OS verifies parameters, executes the request, and then returns control to the user program.
42	System Call example- read function
43	Diagram (Dual-Mode and Multimode Operation: System Call)
44	Step-by-step process, showcasing OS changing to kernel mode w/ System Call
45	Timer: OS maintains control over CPU, assigning it to process with timer. The timer may be fixed or variable and can interrupt after specified period. When the period is expired before finishing a job, process must wait for CPU time in ready queue.

Slides #3

Slide #	**Short Summary**
1	Review- Multiprocessor System Types, Dual Mode & Multimode Operation, Multitasking vs. Multiprogramming
2	Preview- OS as resource manager, Operating System Structures
3	OS as Resource Manager- Process, Memory, File, I/O System, Deadlock, and Cache Management
4	Process Management: OS is responsible for creating/deleting, scheduling, and suspending/resuming processes. OS must also provide means for process synchronization and IPC
5	For Program to be executed, CPU must load, read, & write data from memory. When the program terminates, memory space is declared available for next program
6	Memory Management: OS must monitor/manage utilization of memory, dynamically assign/release memory, and determine the processes/data to transfer in/out of RAM
7	OS must provide uniform, logical storage view for user to save a file. OS must implement mass storage, organizing into directories and controlling which user can access a file and how the user accesses it
8	File Management: OS is responsible for creating/deleting files/directories, supporting primitives for manipulating files/directories (ex. read), mapping files onto mass storage (ex. HDD/SSD), Backing up files on stable storage media
9	Mass-Storage Management: OS is responsible for mounting/unmounting, free-space management, storage allocation, disk scheduling (HDD), partitioning, and protection
10	Cache is a hardware or software component that stores data which might be used again soon. Limited size, so cache management is an important OS operation. Can greatly improve performance
11	Movement of information between levels of storage may be controlled by either hardware or OS (cache->CPU/registers is hardware, disk->memory is OS)
12	Deadlocks between processes occur due to limited shared resources, with some processes needing more
13	Deadlock Management: Resource Allocation Graph
14	Deadlock Management: Example- Two processes require same resource, so they stay in blocked state forever
15	Deadlock Management: Four strategies for dealing with deadlock: Ignore, Detection and Recover, Dynamic avoidance by careful allocation, and Prevention by negating one of the four conditions necessary to cause deadlock (Mutual Exclusion, Circular wait, Hold and wait, No preemptive)
16	I/O: OS uses I/O subsystems for managing I/O devices. These subsystems consist of a memory management system (buffering, spooling, caching),

	General device drivers interface, drivers for specific hardware, and processor for specific hardware
17	OS controls all I/O devices by issuing commands, catching interrupts, and handling errors. OS provides interface between devices & rest of system
18	OS Structures: Monolithic, Layered System, Microkernels, Virtual Machine, Client-Server Module, Exokernels
19	Monolithic System: Written as a collection of procedures, with each procedure having a well-defined interface and with each one being free to call any other one
20	Possible structure for a monolithic system: A main program, service functions, and utility functions
21	Diagram- Structure of Monolithic OS
22	Layered System: OS is divided into several layers: process management (layer 0), memory management (1), IPC (2), I/O management (3), Deadlock management (4), user program (5), system operator process (6)
23	With layered system approach, designers can choose kernel-user boundary
24	Microkernels: Achieve high reliability by splitting the OS into small well-defined modules. Only one module runs in kernel mode (rest in user)
25	Diagram: Microkernels
26	Virtual Machine: Virtual Machine Monitor (Hypervisor) runs on bare hardware and does multiprogramming by providing several virtual machines. Each virtual machine are exact copies of bare hardware, with different virtual machines being able to run a different OS
27	Virtual Machine: Virtualization example would be company running several web servers on same machine
28	Diagram: Virtual Machine
29	Diagram: Virtual Machine
30	N/A

Mini-test 1 Review

Question	**Answer**
What happens when a process changes from running state to blocked state?	When a process changes from running to block/ready state, the OS saves all its information in the process table, of which it picks up when the process is resumed
How does a pipelined CPU design improve performance?	A pipelined design improves CPU performance by allowing the CPU to perform multiple stages (fetch, decode, execute) simultaneously
What happens when reading data from a I/O device?	When reading data, the device controller transfers data from the I/O device to its local buffer. After the transfer is complete, the controller checks for any errors, then informs the device driver that the data is really to be processed via interrupt

What happens when the CPU receives an interrupt signal?	It stops the current task and immediately transfers execution to a fixed location where the interrupt service routine is located. After handling the interrupt, CPU resumes
What are two common sources of interrupts?	Device controllers (I/O devices) and hardware faults
How do modern computers handle multiple interrupts efficiently?	Modern computers use a system of interrupt policies to ensure that the most urgent interrupts are handled first
How do CPUs share memory in a multiprocessor system?	Each CPU has a local and shared memory
How does asymmetric clustering work?	One machine is on hot-standby mode, monitoring the active machine and taking over when it fails
What are the four conditions for deadlock to occur?	Mutual exclusion, circular wait, hold and wait, no preemptive
How does microkernels improve reliability?	The OS is split into separate “modules”, with only one module running in kernel mode while the rest run in user mode

Major Topic #2: Process & Thread Management

Slides #4

Slide #	**Short Summary**
1	Review- OS as Resource Manager & Operating System Structures
2	Preview: Process model, creation, termination, states, table, with multiple-threads, scheduling
3	The Process: a program in execution. Status is kept in process table until termination
4	Diagram: Process consists of stack, heap, data, text
5	OS schedules CPU to process based on scheduling algorithm and process state
6	Diagram: Shows processes loaded in RAM
7	N/A
8	We can consider processes from two POV: Real Model (Multiprogramming) and Conceptual (virtual) Model (each process has virtual CPU and RAM)
9	Diagram: Showcases different process models
10	Process Model: When CPU switches between processes, CPU holds process during a non-uniform time term, which is calculated based on priority/number of calculations

11	

Slides #5

Page #	**Short Summary**
1	Review: Processes- Model, Termination, Creation, Scheduling, Hierarchy
2	Preview: IPC, Threads
3	Cooperating processes require IPC mechanism. Process cooperation allows for information sharing, computation speed up, and modularity
4	Two fundamental models of IPC: Shared Memory (OS only involved in creation, not synchronization) and Message Passing (OS involved in creation, synchronization, and mutual exclusion)
5	Shared Memory is faster than message-passing. In shared memory, system calls are required only to establish shared memory regions
6	Shared Memory vs Message Queue Model (msg queue in kernel space)
7	Shared memory created in RAM, with other processes having to attach it using key values. OS only involved in creation, with the OS not controlling the form of data at this location. Processes are responsible for mutual exclusion
8	Shared Memory System Calls: shmget(), shmat(), shmdt(), shmctl()

9	Producer consumer problem with Shared Memory: Both producer and consumer must be written considering mutual exclusion to avoid race condition)
10	Diagram of Producer-Consumer with Shared Memory
11	OS provides means for IPC via Message-Passing. This mechanism allows processes to communicate and synchronize actions without sharing the same address space. Particularly useful in a distributed environment, and has two system call operations (send and receive). To communicate between processes, a communication link must exist between them
12	Message queue is a linked list of message stored within kernel space, identified by a message queue ID. Msgget() opens new queue, msgsnd() adds messages to the queue, msgrcv() removes messages, and msgctl() provides control operations
13	Logical methods for Message-Passing: Direct communication (each process knows end point address) or Indirect communication (messages are sent and received from mailboxes or ports)
14	In Direct communication, each process knows end point address for each message. The link must be established between every pair of processes, being associated with exactly two processes, with peer to peer links
15	Direct communication exhibits: Symmetry in addressing since both processes know end point addresses of each other (Full Duplex), with Asymmetry in addressing since only one processes know end point address (Half Duplex)
16	Indirect Communication: Messages are sent/received through mailboxes or ports
17	Indirect Communication Operations: create mailbox, send message, receive message, remove mailbox. A link can be associated with many processes, only being established if processes share a common mailbox
18	Indirect Communication: Problems can arise when two processes try to receive message at the same time. Can allow link to be associated with at most two processes, allow only one process at a time to execute receive operation, and allow the system to select arbitrarily the receiver
19	Message-Passing Synchronization: Message passing may be either: Blocking, which is considered synchronous (TCP or SCTP with socket) or Non-Blocking, which is considered asynchronous (UDP with socket)
20	Message-Passing Buffering: Queue of messages attached to the link, implemented in one of three ways: Zero capacity (no messages are queued on a link, sender must wait for receiver), Bounded capacity (finite length of n messages, sender must wait of link fill), and Unbounded capacity (infinite length, sender never waits)
21	Overview of Thread: most modern computers are multithreaded, with threads sharing code section, data section, and other OS resources to other threads.
22	Diagram: Multi-threaded vs Single-threaded
23	Multithread software examples

24	Most OS are multithreaded, Linux included. The ps -ef command can be used on Linux machines to see all threads, with pid=2 thread being the parent to all other kernel threads
25	Example of kernel thread
26	Benefits with threads: Resource Sharing (threads share memory and resources), Economy, Responsiveness, Scalability
27	Multithreaded programming with multiple CPU scores provides improved concurrency (some threads can run in parallel)
28	Diagram: Multicore Programming with Threads
29	OS developers' challenges include writing scheduling algorithms for multiple processing cores, while application programmers must modify existing programs and design new ones that are multithreaded
30	Five areas present challenges in programming multicore systems: Identifying Tasks (finding areas to split into separate concurrent tasks), Balance (working load), Data Splitting (data must be divided to run on separate cores), Data Dependency (must consider synchronization to avoid race condition), and Testing & Debugging (more difficult than single threaded programs)
31	Types of Parallelism: *Data parallelism* focuses on distributing subsets of the same data across multiple computing cores and performing the same operation for each core. *Task parallelism* involves distributing tasks (threads) across multiple cores, with each thread performing a unique operation. May be working on the same, or different data
32	Diagram: Data parallelism vs Task parallelism

Slides #6

Page #	**Short Summary**
1	Preview: Thread implementation, Multithreading model, Thread library, Threading issues
2	Thread Implementation: Threads can be provided at either kernel or user level
3	User Level Thread: Kernel is not aware of existence, runtime system controls threads, application starts with single thread and creates more as needed
4	User Level Thread: *Advantages* are thread switching does not require kernel privileges, user level thread can run on any OS, scheduling can be application specific, and user level threads are fast to create and manage. *Disadvantages* are that, in a typical OS, most system calls are blocking. Also, multithreaded applications cannot take advantage of multiprocessing
5	Kernel Level Thread: Thread management done by OS, Kernel maintains context information for the process as a whole and for individual threads within the process, scheduling is done by kernel on a thread basis, and Kernel performs thread creation, scheduling, and management in kernel space
6	Kernel Level Thread: *Advantages* are that kernel can simultaneously schedule multiple threads from the same process on multiple processes, kernel can

	schedule another thread of the same process if it is blocked, and kernel routines themselves can be multithreaded. *Disadvantages* are that kernel threads are generally slower to create and manage when compared to user threads, and transfer of control from one thread to another within the same process requires a mode switch to the kernel
7	Multithreading Models: Many to one relationship, One to one relationship, Many to many relationship
8	Many-to-One Model: Maps many user-level threads to one kernel thread. Thread management is done during library run-time and is efficient, with the thread making a blocking system call. Only one thread can access kernel at a time, so unable to run in parallel on multiprocessor systems
9	Diagram: Many-to-One
10	One-to-One Model: Maps each user thread to a kernel thread. Provides more concurrency than many-to-one, and can run in parallel on multiprocessor systems. Drawback: Since creating a thread requires creating a corresponding kernel thread, many kernel threads may burden system performance
11	Diagram: One-to-One
12	Many-to-Many Model: Multiplexes any number of user threads onto an equal or smaller number of kernel threads. Developers can create many user threads, in which the corresponding kernel threads can run concurrently. This model provides best accuracy for concurrency when there is a blocking system call.
13	Diagram: Many-to-Many
14	Multithreading Models Summary: Many-to-one allows the user to create as many user thread as wished, however it does not run in parallel since the kernel can only schedule one kernel thread at a time. One-to-one model allows greater concurrency, but can greatly suffer with too many threads. Many-to-many model suffers from neither of these shortcomings.
15	Thread Library- Two primary ways of implementing a thread library: User-level Library (entirely in user space with no kernel support) and Kernel-level Library (supported directly by the OS).
16	Implicit Threading
17	Five methods: Thread pools, Fork-Join, OpenMP, Grand Central Dispatch, Intel Threading Building Blocks
18	Fork() and exec() System Calls
19	Signal Handling- May become complicated with multithreading, where multiple processes where receive a signal
20	Signal Handling: Typically caught by first unblocked thread, unless a specific thread is specified (such as in pthread_kill() function)
21	Pthread_killl() function
22	Thread Cancellation- Terminating a thread before it is complete. Can occur in either asynchronous cancellation (immediately terminates target thread) or deferred cancellation (target thread periodically checks)
23	Thread Cancellation- pthread_cancel()

24	Thread Cancellation Points
25	N/A
26	Pthread_cleanup_push()
27	Pthread_cleanup_pop()
28	N/A
29	N/A
30	Thread-Local Storage (TLS): Where each thread needs its own copy of certain data
31	Scheduler Activations: Threads implemented in a process in either user-level library within a single-threaded process, uses kernel threads, or uses Scheduler activations (Hybrid)
32	Scheduler Activations: Many system places and intermediate data structure called Lightweight Process *(LWP)*. To user-thread library, LWP appears to be a virtual processor where the application can schedule user threads to run. Each LWP is attached to a kernel thread, and it is kernel threads that the OS schedules to run on physical processors
33	Different applications/hardwares may require differing numbers of LWPs
34	How Scheduling Activations work

Slides #7

Page #	**Short Summary**
1	**Preview**: Schedulers, Scheduling Queues, Preemptive vs. Non-preemptive scheduling, Scheduling Criteria, CPU Scheduling (Short-term Scheduler0
2	**CPU Scheduling**: Goal of multiprogramming is to have a process always running, maximizing CPU utilization. CPU will select process from ready queue, with the ready queue structure saving pointer to process table for ready state processes
3	**Three Level Scheduler**: Part of OS that chooses next process based on scheduling algorithm is called process scheduler. There are three levels: *Long-Term Scheduler* (selects process from job queue and loads into memory for execution), *Short-Term Scheduler* (Selects process from ready queue and allocates CPU), and *Memory Scheduler* (schedule which process is in memory and in disk).
4	Diagram: Three Level Scheduler
5	**Scheduling Queues**: Once process is allocated in memory and executing on CPU, it could be blocked or more processes could be created for various reasons. Once CPU becomes available, CPU scheduler must select a process from ready queue and run it
6	Diagram: Process of queueing in process scheduling
7	An effective process scheduling algorithm is essential since the process switch is very expensive. It must: save all information for blocked state, save memory map, select a new process, load new process information, and start to run it
8	**Preemptive vs. Nonpreemptive Scheduling:** If a scheduling decision is only made when a process switches from running to blocked state, and when process terminates its job and new memory space becomes available, then it is denoted

	as *nonpreemptive scheduling scheme*. If we consider when the process switches from running to ready state, and when process switched from blocked to ready state, then it is denoted as *preemptive* scheduling scheme. Under *nonpreemptive scheduling*, once CPU has been allocated, process keeps CPU until it releases by either terminating or switching to blocked state.
9	**Preemptive vs. Nonpreemptive Scheduling:** *Preemptive Scheduling* without considering mutual exclusion can result in race conditions with shared data. Preemptive kernel requires mechanisms such as mutex locks to prevent these
10	**Scheduling Criteria:** Criteria to be considered when selecting an algorithm: *CPU Utilization* (want to keep CPU as busy as possible), *Throughput* (number of processes that are completed per time unit), *Turnaround Time* (The interval from time of submission to time of completion), *Waiting Time* (The sum of periods spent waiting in ready queue), and *Response Time* (Time from submission to first response). Want to maximize CPU utilization and Throughput, and minimize Turnaround time, Waiting time, and Response time
11	**CPU Scheduling:** Process migrates from ready queue to various wait queues, with the short-term scheduler needing to select process to be executed. There are various rules that the short-term scheduler can use.
12	**First-Come, First-Served**: Simplest, can be implemented with FIFO queue, drawback is long average waiting times
13	Diagram: First-Come, First-Served (FCFS)
14	**Shortest Job First**: Selects the next process with the smallest “CPU Burst”. Uses FCFS to break ties, and is provably optimal since minimal average waiting time
15	Diagram: Shortest Job First (SJF)
16	**Shortest Job First**: Although optimal, length of CPU bursts cannot be exactly known. However, there are ways to approximate based on previous CPU burst time
17	Shortest Job First Formula
18	**Shortest Remaining Time First**: Preemptive version of the SFJ algorithm is Shortest Time Remaining First algorithm. When a new process arrives at ready queue which previous process is executing, CPU is changed to ready state and short-term scheduler selects a process from ready queue. The newly arrived process may be shorter than what is left in the current process.
19	Diagram: Shortest Remaining Time First
20	**Round-Robin Scheduling**: Familiar with FCFS scheduling, but preemption is added to enable the system to switch between processes. If a process’ execution exceeds 1 time quantum (small unit of time), then it is placed back in the ready queue. Can be implemented with a circular queue, with new/time expired processes pinned to the tail. However, the average waiting time of *RR* is often very long
21	Diagram: Round-Robin Scheduling
22	**Priority Scheduling**: A priority is associated with each process, with the CPU being allocated to the process with the highest priority. Equal-priority processes

	are scheduled in FCFS or RR order. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. There are several variations of Priority Scheduling.
23	**Priority Scheduling**: Priority can be set in two ways: ***Internal Way***- based on measurable quantity or quantities to compute the priority of a process, or ***External Way***- based on the importance of the process
24	**Priority Scheduling**: A major problem of this approach is starvation of lower priority processes. The solution to this is aging, which is a technique where processes that wait longer in the queue gradually increase in priority
25	Diagram: Priority Scheduling
26	Diagram: Priority Scheduling
27	^
28	^
29	**Priority Scheduling**: With a single queue, *O(n)* search may be necessary to determine the highest-priority process. Therefore, multilevel queues can be used to improve search times, with the system keeping different queues of pointers to process tables of the same priority. This works well when combined with round-robin approach.
30	Diagram: Priority Scheduling with Multilevel Queue
31	A multilevel queue scheduling algorithm may be used to partition processes into separate queues based on process type. Additionally, each different level queue may have its own scheduling algorithm
32	**Priority Scheduling**: With multilevel queue, processes are permanently assigned. However, with ***Multilevel Feedback Queue***, processes can move between queues, such as when a process uses too much CPU time
33	**Priority Scheduling**: Parameters for implementing a multilevel feedback queue: The number of queues, the scheduling algorithm for each queue, determining when to upgrade/demote a process to different priority queue, determine which queue a process that needs serviced will enter
34	Example of Priority Scheduling with Multilevel Feedback Queue
35	Diagram: Priority Scheduling with Multilevel Feedback Queue
36	**Guaranteed Scheduling**: Ensures fairness by monitoring the amount of CPU time spent by each user. If there are n processes, each process will receiving 1/n of the CPU power.
36	**Lottery Scheduling**: Processes are selected at random to obtain resources

Slides #10

Page #	**Short Summary**
1	Preview: IPC (Race Condition, Critical Section, Solutions for Mutual Exclusion in a Critical Section)

2	**IPC**: Three issues in IPC: How one process can pass information to another with IPCS, how to make sure two or more processes do not get into the critical section (mutual exclusion), and proper sequencing (synchronization)
3	**Race Condition**: A situation where two or more processes are reading/writing shared data, and the final result depends on who runs precisely when. ***Critical Section***- The part of the program where shared memory is accessed. ***Mutual Exclusion*** in a critical section can avoid race conditions
4	Diagram: Race Conditions
5	Example: Race Condition
6	**Race Condition**: To avoid race condition, ***Mutual Exclusion*** must be used (if one process is using a shared variable/file, the other process should be excluded from doing the same). The choice of algorithm for mutual exclusion is major OS design issue, with the solution for race condition having the following conditions: 1. No two processes may be simultaneously inside their critical regions (mutual exclusion) 2. No process running outside its critical region may block other processes 3. No process should have to wait forever to access critical region 4. No assumptions may be made about speeds on number of CPUs
7	**Race Condition**: There are two approaches for mutual exclusion solutions- ***Busy Wait*** (process will wait until resource become available, or CPU time expires) or ***Sleep and Wakeup*** (A process checks a resource, and sleeps if not available. When the resource becomes available, the process will be waked by system)
8	**Mutual Exclusion with Busy Waiting**: Each process has time term, with process keep checking to get into critical section. Consists of five concepts: Disabling Interrupts (non-preemptive kernel), lock variables, strict alternation, Peterson's solution, and Hardware solutions)
9	**Disabling Interrupt**: Once a process gets into the critical section, interrupts set to disable. Other processes cannot get CPU time until process finishes its job, which leaves the system vulnerable and could cause the end of system
10	Example: Disabling Interrupt causing end of system
11	**Using Lock Variable**: A variable called "lock" (like mutex), where lock=1 means there is a process running in critical section, and process does a busy wait until lock becomes 0
12	Diagram/Example: Using Lock Variable
13	^. Using Lock Variable violates condition #1 of mutual exclusion
14	**Strict Alternation**: Variable turn can be I or j, with process waiting until it is it's turn (P_i can finish job in critical section during i turn)
15	Diagram/Example: Strict Alternation. This violates conditions #2 and #3
16	**Peterson's Solution**: Provides a good algorithmic design to this problem. Is restricted to two processes that alternate execution between their critical sections and remainder sections. Processes are numbered P_0 and P_1
17	Implementation/Example: Peterson's Solution
18	Example: Peterson's Solution

19	Proof: Peterson's Solution
20	**Test and Set Lock – Hardware Solution**: Since TSL instruction is a hardware instruction, the operations of reading the lock and storing into register are guaranteed to be indivisible. This reads/stores a value of LOCK into register RX, with LOCK=1 meaning the critical section is occupied
21	Diagram/Example/Implementation: Test and Set Lock (hardware solution)
22	**Memory Barriers – Hardware Solution**: Two general memory models: Strongly ordered memory (a memory modification on one processor is immediately visible to all other processors), and Weakly ordered memory (a memory modification on one processor may not be immediately visible). With strongly ordered memory, computer architecture provides instructions that can force any changes in memory to be propagated to all other processors, ensuring that memory modifications are visible to threads running on other processors. Such instructions are known as *memory barriers*.
23	**Memory Barrier – Hardware Solution:** A memory barrier is a type of barrier instruction that causes CPU or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction. This typically guarantees that operations prior to the barrier will be performed before operations after the barrier.
24	Example: Memory Barrier
25	Example: Memory Barrier (cont)
26	**Atomic Variables – Hardware Solution:** We can also avoid mutual exclusion by using atomic operations. When a thread or process performs an atomic operation, the other threads or processes see it as happening instantaneously. These are relatively quick compared to locks, and do not suffer from deadlocks and convoying. However, they only do a limited set of operations, which are often not enough to synthesize more complicated operations efficiently. However, this is still vastly more efficient and should be used when possible.
27	**Priority Inversion Problem:** Previous methods waste CPU time through busy waiting. This problem describes that, due to priority scheduling, processes with high priority may never get into critical section, as they are consistently selected as next up to be executed and therefore just constantly re-check the same condition (without other process ever being able to execute to get out of it). This can lead to indefinite waiting.
28	Diagram/Example: Priority Inversion Problem

Slides 11

Page #	**Short Summary**
1	Review (IPC)
2	Preview: Mutual Exclusion in Critical Section (with Sleep and Wake Up)

3	Mutual Exclusion with **Sleep and Wakeup**: A process checks critical section, and goes to sleep if not available. When the process becomes available, it will be waked up by system.
4	**Producer-Consumer Problem**: Occurs when two processes share a common, fixed-size buffer, with the producer putting information in it and the consumer taking it out. Trouble arises when the producer wants to put a new item in a full buffer, or the consumer tries to take an empty out of an empty buffer
5	**Producer-Consumer Problem** (solutions): When producer wants to put an item in a full buffer, producer is to go to sleep and awaked by consumer when item has been removed. Same process for consumer
6	Code Example of Producer-Consumer Problem
7	**Semaphores**: Created by E. W. Dijkstra, an integer variable which essentially tracks the number of wakeups pending (0 for no wakeups saved, +i for I wakeups pending). Can only be accessed by atomic operations down (P) or up (V)
8	**Concept of Semaphores**: Modification of semaphore is executed indivisibly, which means that no other process can simultaneously modify that same semaphore value
9	Code demonstrating Semaphore operation
10	Normal way to implement Semaphores: Operations *up* and *down*, with system briefly disabling all interrupts while testing/updating the semaphore.
11	Code demonstrating how to use *Semaphore* to combat Producer-Consumer problem
12	Code demonstrating how careless use of *Semaphore* can *cause deadlock*
13	^
14	**Dining Philosophers Problem**: Picture
15	**Dining Philosophers Problem**: involves five philosophers seated in a circular arrangement, with each alternating between thinking and eating. To eat, they must acquire two shared chopsticks, which are placed between philosophers and can only be held by one philosopher at a time. Philosophers must wait for both chopsticks to be available before eating and must release them once finished, showcasing the potential concurrency issues of deadlock and resource starvation.
16	**Reader-Writers Problem**: Process reader *R* and writers *W* are sharing resources, with only one resource being able to be accessed at a time. Two processes can read resource at same time, and can use locks to block writing. However, if a writer is waiting for the lock and another reader wants to read it, then *W* would need to get privilege first. If the reader jumped ahead of it enough, *W* would starve
17	Reader-Writers Problem: Diagram/Demo
18	**Mutexes**: When a semaphore's ability to count is not needed, Mutex can be used. It is good for managing mutual exclusion, being one of two states (0 for unlocked, 1 for locked). Essentially a binary semaphore
19	Mutexes example

20	**Monitor**: High level synchronizing primitive. This is a group of procedures, variables, and data structures that are all grouped together in a special kind of module. Only one of these processes can have active at a instant, with the compiler knowing that monitors are special and can handle these calls differently.
21	**Monitor**: Construct for a programming language, so implementation is based on the compiler. Since compiler knows that *monitor* is a special kind of module, compile uses mutex or binary semaphore for mutual exclusion.
22	**Implementation of Monitor**: Conditional variables are used in the monitor, with two operations (*wait*, *signal*). When a monitor procedure discovers that it cannot continue, it waits on some condition variable (such as full). This causes the calling process to *block*, allowing other process to get the monitor. Other processes (ex. Consumer) can *wake* its sleeping partner through a signal on the condition variable the partner is waiting on. System scheduler chooses waiting process if there is more than one.
23	**Implementation of Monitor**: To avoid having two active processes at the same time after a process signals, there are two approaches. The first, by *Hoare*, suggests letting the newly awakened process run and suspending the one that signaled. The second, by *Brinch Hansen*, suggests that the signal statement may appear only as the final statement in a monitor procedure
24	Producer-Consumer Implementation with *Monitor*
25	**Message Passing**: A method of IPC by using two primitive system calls- *send* and *receive*. This is typically used between processes in different systems, as it is slower than semaphore or monitor. When there are no messages available, receiver will be blocked by system until one arrives. If no messages to send, sender will be blocked by system until one becomes available.
26	**Message Passing** (Design Issues): Messages can be lost- to solve, receiver sends acknowledgement message when message is received, and the sender will retransmit the message within a certain time interval if the sender doesn't receive the acknowledgement. However, this creates a new issue; messages could then be sent twice. To solve this, each message is assigned with sequence number (id), with receiver recognizing duplicated messages and discarding one of them
27	Producer-Consumer Implementation with Message Passing

Slides 12

Page #	**Short Summary**
1	**Preview**- Memory Management
2	**Memory Management**- Since perfect memory doesn't exist, most computers have a memory hierarchy: Small, fast, expensive volatile **registers**, Expensive volatile **cache memory**, medium-speed volatile **RAM**, slow, cheap, non-volatile disk storage. Memory management is a part of OS which manages the memory hierarchy

3	**Mono-process Memory Management**: Mono-programming without swapping or paging. Only memory sharing between a user program and OS, with only one program being loaded into memory. When the program finishes its job, the long-term scheduler chooses another job to load into memory
4	Diagram of Mono-Process Memory Management
5	**Multi-process Memory Management**: *Multiprogramming with Fixed Partition*- Memory is divided into n partitions, being able to be done with separate input queue for each partition or a single input queue.
6	Diagram of Multi-process Memory Management
7	**Multi-process with Fixed Partition Memory Management**: Fixed memory partition with *separate input queue* for each partition. When a job arrives, it is put into the queue for the smallest partition large enough to hold it. The disadvantage with this is that, when the large partition queue is empty, but many small jobs are waiting on the small partition queue
8	**Multi-process with Fixed Partition Memory Management**: When a partition becomes free, the closest queue that fits is chosen and loaded into the empty partition (wastes memory) or search the whole queue to find the job for the best fit for the free partition, discriminating against small jobs. The solution for discrimination is one partition for all small jobs, counting the time skipped by scheduler, automatically granted time if a job is skipped for enough of time
9	**Modeling Multiprogramming**: Simple unrealistic model- All processes will never be waiting for I/O at the same time, with CPU being busy 100% of the time
10	**Modeling Multiprogramming**: *Probabilistic Model*- Assuming p is a fraction of time a process is waiting for I/O, and if there are n processes in memory at once, then the probability that all n processes are waiting for I/O is p^n. Therefore, the CPU utilization = 1 - p^n
11	Modeling Multiprogramming- Chart diagram example
12	Modeling Multiprogramming- Chart diagram example explained
13	**Memory Management**: There is usually not enough main memory to hold all active processes, so excess processes must be kept on disk and brought in to run dynamically. There are typically two approaches for this: ***Swapping*** (bringing in each process in it's entirety, running it for a while, then putting back on disk) and ***Virtual Memory*** (allows programs to run while partially loaded in the memory)
14	**Swapping**: Main difference between fixed partition and variable partition- ***Fixed Partition***- The number, location, and size of partitions are fixed. The OS knows the address of each process. Simple to manage the memory, however causes internal fragmentation (wastes space). ***Variable Partition***- The number, location, and size of partitions vary dynamically based on size of process. OS needs to keep track of partition information dynamically for memory allocation and deallocation. This might create multiple holds- Combine them all into one big hole (external fragmentation, needs computationally expensive memory compaction)
15	Swapping with Variable Partition- Diagram

16	**Swapping:** *How much memory should be allocated for a process when it is created or swapped*? Process created with fixed size is simple, however processes usually change size dynamically (heap) (solution: Allocate extra memory for each process and use adjacent hole for managing growing size). If there is not enough memory space for a dynamically changing size, processes is swapped out or killed
17	Swapping: Diagram showing how extra space is allocated for data and stack
18	**Free Memory Space Management**: OS needs to keep memory space information available or occupied. There are two possible ways to keep this memory information: Bitmap or Free List
19	**Free Memory Space Management with Bitmap**: OS keeps a table called Bitmap to keep memory space information, being divided up into allocation units (k bytes). Corresponding to each allocation unit is a bit in the bitmap (0 if unit is free, 1 if unit is occupied)
20	Bitmap: Diagram showing how each bit is allocated to a partition
21	**Free Memory Space Management with Bitmap**: *Size of the allocation unit* an important design issue. A smaller allocation unit size will lead to a larger bitmap, and larger allocation unit sizes will lead to a smaller bitmap, but memory may be wasted in the last unit of the process if the process size is not a multiple of the allocation unit
22	**Free Memory Space Management with Bitmap**: *Advantages*: Simple way to keep track of memory words in a fixed amount of memory since the size of bitmap depends only on the size of memory and size of allocation units. *Disadvantages*: To allocate memory for the process with k unit size, first, the memory manager (part of OS) needs to search for the bitmap
23	**Free Memory Space Management with Free-list**: With free-list, OS maintains a linked list of allocated and free memory segments. Each entry in the list keeps four information: whether it is *hole or process*, *starting address*, *length*, and *a pointer to the next entry*
24	Free list: Diagram showing how memory is under this method
25	**Free Memory Space Management with Free-list**: If the segments list is kept sorted by address, it is simple to update the list when a process terminates or swapped out from memory
26	**Prefixes**: • 1 Byte = 8 bits • 1 KB = 2^{10} Bytes • 1 MB = 2^{20} Bytes • 1 GB = 2^{30} Bytes • 1 TB = 2^{40} Bytes • 1 PB = 2^{50} Bytes • 1 EB = 2^{60} Bytes • 1 ZB = 2^{70} Bytes • 1 YB = 2^{80} Bytes

27 **Free Space Management with Bit Map or Free-list**: Example showing calculations

- The 128-MB memory is allocated in units of 2KB. For the linked list, let's assume that memory is currently consists of an alternating sequence of segment and holes, each 64 KB. Also assume that each node in the linked list needs a 32-bit memory address, a 16-bit length, and a 16-bit next node field.
- Bitmap:
  - #of allocation unit = 128MB/2KB = $(128 \times 2^{20})/(2 \times 2^{10}) = 2^{27}/2^{11} = 2^{16}$ units
  - Size of the bitmap = 2^{16} bits = **2^{13} byte**
- Free list:
  - number of node for linked list= 128 MB /64KB = $2^{27}/2^{16}$ or 2^{11} nodes.
  - size of each node = 32+16+16 = 64 bit = 8 byte =2^3 bytes
  - Total size of linked list = number of node × size of a node = $2^{11} \times 2^3$ bytes = **2^{14} bytes**.

28 **Free Memory Space Management with Free-list**: Algorithms for allocating memory for a process:

- **First Fit**- The memory manager scans the list of segments from the beginning until it finds a hole that is big enough
- **Next Fit**- Works same as first fit, however it starts searching the list from the place it left off last time
- **Best Fit**- It searches entire list and takes the smallest hole that fit for the process (potentially creating small holes, which is not enough for other processes)
- **Worst Fit**- It always takes the largest free hole (potentially creating hole big enough for another process)

29 **Free Memory Space Management with Free-List**: Calculation example

- Consider a swapping system in which memory consists of the following hole sized in memory order: 21KB, 20KB, 4KB, 18KB, 15KB, 14KB, 25KB, 23KB and 35KB.
- Which hole is taken for successive segment request of 9K, 10KB, 15KB and 18KB for first fit? Now repeat the question for best fit, worst fit and next fit.
  - First Fit 9(21) -10(12) -15(20)- 18(18)
  - Best Fit 9(14) – 10(15)-15(18)-18(20)
  - Worst Fit 9 (35) – 10(26) -15 (25) – 18 (23)
  - Next Fit 9(21)-10(20)-15(18) -18(25)

30 **Memory Space Management with Free-list**: Four algorithms can be sped up by maintaining two lists: *one for holes* (can be sorted by size so best fit does not need to search entire list) and another *one for processes*. However, extra effort is needed when updating the two lists based on allocation and deallocation of memory

31 **Memory Space Management with Free-list**: ***Quick Fit-*** Maintains separate lists for holes based on the size of each hole, with each entry being a pointer to the head of a certain size hole. This is quick for finding a hole of required size, however expensive to maintain the separate lists for hole based on deallocation of memory

Slides 13

Page #	**Short Summary**
1	**Review**: Memory Management
2	**Preview**: Memory Management Cont.- Virtual Memory, Virtual Memory with Paging (page tables, Physical Memory Managing (MMU))
3	**Virtual Memory**: Motivation of Virtual Memory- Initial idea of executing program is that the entire logical address space of a program must be in physical memory before the process can access (swapping). However, since the size of the program can be very large and above the limited size of physical memory, this is not possible
4	**Virtual Memory: Overlay**- Split program into pieces (overlays); Overlay 0 starts running, calling another overlay when it finished the job; Overlays are kept on disk and swapped in and out of memory by OS; The work of splitting the program into pieces had to be done by programmer; Splitting up large program into pieces is time consuming
5	**Virtual Memory**: Virtual Memory is a technique that allows the execution of a process to not entirely be done in memory, with the OS needing to keep track of where every part of the program is (main and secondary memory). Virtual memory ideas can be used in multiprogramming system
6	**Virtual Memory**: There are *three* approaches for managing virtual memory- ***Paging***, ***Segmentation***, and ***Segmentation with paging***
7	**Virtual Memory with Paging**: Virtual address space is divided into units called *pages*, with corresponding units in physical memory called *page frames*. A page size must be the same as the page frame size of calculating physical address from virtual address in program counter. Size is usually between 512 Bytes and 64 KB
8	**Virtual Memory with Paging**: How to map Virtual Address to Physical Memory Address- PC (Program Counter in CPU) saves the virtual address of next instruction. When virtual memory is used, a virtual address (generated by program) does not go directly into the memory bus, instead goes to an MMU (Memory Management Unit) that maps the virtual addresses onto the physical memory addresses based on the memory map (page table).
9	Virtual Memory with Paging: Diagram showing how each page in virtual space maps to a page frame in physical space (RAM)
10	Virtual Memory with Paging: Diagram showing how each page in virtual space maps to a page frame in physical space (RAM)
11	**Virtual Memory with Paging**: The CPU sends virtual address on PC to MMU for a fetch cycle, with the MMU calculating the physical address and sends to RAM
12	**Different Size Prefixes**: - 1 Byte = 8 bits - 1 KB (kilo) = 2^{10} Bytes - 1 MB (mega) = 2^{20} Bytes - 1 GB (giga) = 2^{30} Bytes

	- 1 TB (tera) = 2^{40} Bytes - 1 PB (peta) = 2^{50} Bytes - 1 EB (exa) = 2^{60} Bytes - 1 ZB (zetta) = 2^{70} Bytes - 1 YB (yotta) = 2^{80} Bytes
13	Virtual Memory with Paging Example **When solving, must use these equations** Possible Number of Pages = $\frac{Possible\ Virtual\ Space}{Page\ size}$ Number of Page Frames = $\frac{Size\ of\ Memory}{Page\ Size}$
14	Virtual Memory with Paging Example (practice these)
15	Virtual Memory with Paging Example
16	Virtual Memory with Paging Example
17	Virtual Memory with Paging Example
18	Virtual Memory with Paging – Diagram showing how Virtual Memory (pages) is mapped to physical memory, using the same size
19	**Virtual Memory with Paging**: MOV REG, 0;; move content of address 0 to register. Then, MMU checks if virtual address 0 belongs to a virtual page 0, checking memory map such that page 0 is mapped to page frame, and sends the physical address to address bus
20	**Virtual Memory with Paging**: MOV REG, 8900;; Virtual address 8900 is sent to MMU, calculates that this address belongs to virtual page 2, checks memory map such that virtual page 2 is mapped to physical frame 6, sends physical address to address bus
21	**Virtual Memory with Paging**: MOV REG 24660;; Virtual address 24660 is sent to MMU, with MMU calculating this address belongs to virtual page 6. MMU checks memory map such that virtual page 6 is not in physical space yet (page fault). When this occurs, OS will: Pick one of the physical frames, write back to disk, fetch the page to the frame just freed, and change the memory map
22	**Virtual Memory with Paging- Physical Address Mapping**: How to map virtual address into physical address with MMU
23	Diagram
24	Virtual Memory with Paging- Physical Address Mapping example
25	Diagram
26	Virtual Memory with Paging Example
27	**Page Table**: The purpose of a page table is to map virtual pages onto page frame. There are two major issues with this: the page table can be extremely large, and the mapping must be fast.
28	**Page Table**: The page table can be extremely large
29	**Page Table**: The mapping must be fast.

Slides 14

Page	**Short Summary**
1	**Review**: Virtual Memory, Virtual Memory with Paging
2	**Preview**: Virtual Memory with Paging- Page table with Hardware Support, Page Table Structure
3	**Page Table with Hardware Support**: OS maintains a page table per a process. A pointer to the page table is stored in the process table. When short term scheduler selects a process for execution, its page table must be loaded into memory. This can be done on the hardware side with *dedicated high-speed hardware registers* (making the page-address translation efficient but increases context-switch time) and with the *page-table based register*, which is used to save a pointer to a page table which is kept in main memory. Changing the page table requires only change the content of this register.
4	**Page Table with Hardware Support (Translation Look-Aside Buffer)**: Maintaining page table can result in slower memory access times, since the page table entry must be retrieved and then the physical address calculated. This slows memory access by a factor of 2. The solution to this is to use a special, small, fast-lookup hardware cache called a *Translation look-aside buffer (TLB)*, where each entry consists of a key and value.
5	**Page Table with Hardware Support (Translation Look-Side Buffer)**: The TLB contains only a few of the page-table entries. The MMU will check for a TLB hit before searching page table (and then adding to the TLB).
6	TLB Diagram – Hardware functions
7	**Page Table with Hardware Support (Translation Look-Aside Buffer)**: The percentage of times that the page number of interest is found in the TLB is called the *hit ratio*. If this occurs, it will only take the time it takes to access memory to find the point. If it misses, then it will take twice as long.
8	**Page Table with Hardware Support (Translation Look-Aside Buffer)**: CPUs today provide multiple levels of TLBs
9	**Page Table (Shared Pages)**: An advantage of paging is that the possibility of sharing common code, a consideration that is particularly important in an environment with multiple processes.
10	Page Table Diagram – Showing how pages can share frames (one copy for multiple processes)
11	**Structure of the Page Table (Multilevel Page Table)**: The basic idea of multilevel page table method is to avoid keeping all the page tables in memory.
12	Example- Showing how first page table is page table of page tables
13	Multilevel Page Table Diagram: Showcasing page table pointing to other page tables
14	**Structure of the Page Table (Multilevel Page Table)**: Example showing the virtual address, pointer 1 and pointer 2, and offset is stored

15	**Structure of the Page Table (Multilevel Page Table)**: The MMU uses PT1 index into the top-level page table to obtain entry 1. Then the MMU uses PT2 to index into the second-level page table and exact entry 3. If the page is in memory, the page frame number taken from the second-level page table is combined with the offset to construct a physical address
16	**Structure of the Page Table (Hashed Page Table)**: One approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number. *Hashing idea can reduce the number of entries in a page table*. Each entry in the has table contains a linked list of elements that hash to the same location. Each element consists of page number, page frame number, and a pointer to the next element in the linked list.
17	**Structure of the Page Table (Hashed Page Table)**: Describing how the algorithm works
18	Hashed Page Table Diagram: Showcasing how the logical address is mapped from the hash function (to point to the entry in hashed page table) and offset
19	**Structure of the Page Table (Inverted Page Table)**: Inverted page table structure consists of one-page table entry for every frame of the main memory. The number of page table entries in the Inverted Page Table is the number of page frames in the physical memory. A single page table is used to represent the paging information of all the processes. With inverted page table, the overhead of storing an individual page table for every process is eliminated and only a fixed portion of memory is required to store the paging information of all the processes together.
20	**Structure of Page Table (Inverted Page Table)**: Each entry in the page table includes virtual page number, process id, and control bits. Given a process ID and virtual page number, the page table is searched for a match. The index of the page is the same as the page frame, so the physical address can be calculated.
21	Inverted Page Table Diagram: Showcasing functionality, how inverted page table is searched to find physical address
22	**Structure of Page Table (Inverted Page Table)**: Although this scheme decreases the amount of memory needed, it increases the amount of time needed (must search the table). To alleviate this, we can use a hash table. However, each hash table adds a memory reference to the procedure, so one virtual memory reference requires at least two real memory reads (one for the hash-table entry and one for the page table).
23	**Structure of Page Table (Inverted Page Table)**: One issue with inverted page tables involved shared memory. Since there is only one virtual page entry for every physical page, one physical page cannot have two (or more) shared virtual addresses. Therefore, with inverted page tables, only one mapping of a virtual address to the shared physical address may occur at a given time. A reference to another process sharing the memory will result in a page fault and will replace the mapping with a different virtual address.

Slides 15

Page #	**Short Summary**
1	**Review**: Virtual Memory with Paging (TLB, Shared Pages, Page Table Structure (Multilevel Page Table, Hashed Page Table, Inverted Page Table))
2	**Preview**: Demand Page and Page Fault, Free-Frame List, Performance of Demand Page with Page Fault, Swapping with Paging, Copy-on-Write between Parent and Child, Page Table Entries, Page Replacement Algorithms
3	**Demand Page & Page Fault**: Virtual memory and demand paging are memory management techniques used in operating system. Demand paging is a type of swapping done in virtual memory systems. In demand paging, the data is not copied from the secondary memory to the main memory until they are needed or being demanded by some program. While a process is executing, some pages will be in memory while others are in secondary storage. A *Page Fault* is a situation such that process tries to access a page that was not brought into memory.
4	Demand Page & Page Fault Diagram: Showing process
5	**Demand Page & Page Fault**: If the operating system sets the instruction pointer to the first instruction of the process before the process is running, the process immediately faults for the page. Pure demand paging is where the process only brings in pages that it needs, continuously faulting when it needs another page.
6	**Demand Page & Page Fault**: A given instruction could access multiple pages, which could cause multiple page faults.
7	**Free-Frame List**: When a page fault occurs, the operating system must bring the desired page from secondary storage into a free page frame in main memory. If there is no free page frame, than the OS needs to create one. To resolve page faults with no free page frame, most operating systems maintain a free-frame list, a pool of free frames to satisfy such requests
8	**Performance of Demand Paging with Page Fault**: Steps for performance of Demand Paging with page fault
9	**Performance of Demand Paging with Page Fault**: There are three major task components of the page-fault service time: Servicing the page-fault interrupt, reading in the page, and restarting the process.
10	**Performance of Demand Paging with Page Fault**: The effective access time can be modeled with (1-p) * <memory access time> + p * <page switch time>, where p is the Page-fault rate. Effective access time is directly proportional to the page-fault rate p. It is important to keep the page fault rate low in order to prevent slow process execution. An additional aspect of demand paging is the handling and overall use of swap space. I/O to swap space is generally faster than file system because the swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used
11	**Performance of Demand Paging with Page Fault**: There are two options to improve paging throughput with swap pace. Firstly, copying an entire file image into the swap space at process startup, with all demand paging coming from swap space. This, however, increases start-up overhead dramatically. Secondly,

	to initially demand-page from the file system but to write the pages to swap space as they are replaced (what Linux and windows use).
12	**Swapping with Paging**: Process instructions and the data must be in memory to be executed. However, a process, or a portion of a process, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution. Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of a system. Standard swapping involves moving entire processes; however, this is no longer used because of the amount of time required to move entire processes.
13	**Swapping with Paging**: Most systems (Linux and Windows) now use a variation of swapping in which pages of a process are swapped, rather than entire processes. Linux uses virtual memory, so that disk area (swap space) is used as an extension of physical memory for temporary storage when the OS keeps track of processes requiring more physical memory than what is available. When this happens, the swap space is used for both swapping and paging.
14	**Copy-on-Write between Parent and Child**: Traditionally, fork() worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. Copy-on write idea works by allowing the parent and child processes initially the share the same pages. If either processes modifies a shared page, a copy of the shared page is created.
15	Copy-on-Write between Parent and Child: Diagram showing how page copies are made when parent or child writes.
16	**Copy-on-Write between Parent and Child**: When a child is created using vfork(), the parent process is suspended, and the child process uses the address space of the parent. Vfork() does not use copy-on-write, so child modifies page on original address space.
17	Structure of Page Table Entry- Diagram showing the various data in a page table entry
18	**Page Replacement Algorithms**: When a page fault occurs and if there is no free frame, OS needs to choose a victim page currently allocated in a page frame, rewrite to the disk if the page has been modified in memory, the page is allocated into the page frame used by victim table, and the page table is changed. Finding the victim page is dependent on replacement algorithm, and if no frames are free, then two page transfers are required.
19	**Page Replacement Algorithms**: A process's memory access can be characterized by a list of page numbers. This list is called the *reference string* (sequence of page numbers). A paging system can be characterized by three items: The reference string of the executing process, the page replacement algorithm, and the number of page frames available in memory for a process
20	**Page Replacement Algorithms (Optimal Algorithm)**: Replace the page that <u>will not be used for the longest period of time</u>. Optimal algorithm always guarantees the lowest possible page-fault rate for a fixed number of frames. Unfortunately,

	this algorithm is difficult to implement, since it requires future knowledge of the reference string
21	Page Replacement Algorithms – Optimal Algorithm: Diagram showcasing how frames would be replaced
22	**Page Replacement Algorithms (Not Recently Used, NRU)**: When page fault occurs, the OS inspects all pages and classifies into four group base on the page table (modified bit, reference but). The NRU algorithm removes a page at random from the lowest numbered nonempty class.
23	**Page Replacement Algorithms (First In First Out, FIFO)**: Replace the oldest page frame. Very easy to understand, however performance is not always good.
24	Page Replacement Algorithms (FIFO): Diagram demonstrating behavior
25	**Page Replacement Algorithms (Second Chance)**: A simple modification of FIFO that avoids the problem of throwing out a heavily used page is to inspect the R bit of the oldest pages. If the oldest page's R=0, it is old and not references so it can be removed. If the oldest page's R=1, it is old but referenced. So set R=0 and set the page from oldest to newest page
26	Page Replacement Algorithms (Second Chance): Diagram demonstrating functionality, and how the oldest page gets assigned R=0 after moving to oldest page.
27	**Page Replacement Algorithms (The Clock Page Replacement Algorithm)**: Similar with second chance, but it keeps all the page frames on a circular list. When a page fault occurs, the page the hands is pointing to is inspected. Action taken depends on the Reference bit R (R=0, evict the page. R=1, Clear R and advance hand)
28	Page Replacement Algorithms (Clock Page Replacement Algorithms): Diagram showcasing how it is actually like a clock, rotating in a circular fashion, swapping R to 0 and advancing until it finds R=0 process.
29	**Page Replacement Algorithms (The Least Recently Used):** Replace the page that has not been used for the longest period of time. This is the optimal page replacement algorithm looking backward in time, with LRU algorithms working good but potentially requiring substantial hardware assistance to keep track of the information. LRU can be implemented with a counter or stack.
30	Page Replacement Algorithms (Least Recently Used, LRU): Diagram showcasing how it is used

Notes 16

Page #	**Short Summary**
1	**Review:** Demand Page and Page Fault, Free-Frame List, Performance of Demand Page with Page Fault, Swapping with Paging, Copy-on-Write between Parent and Child, Page Table Entries, Page Replacement Algorithms
2	**Preview:** Belady's Anomaly, Design Issues for Paging System, Segmentation, Segmentation with Paging
3	**Modeling Page Replacement Algorithm (Belady's Anomaly)**: Intuitively, it might seem that the more frames the process has, the few page faults a process will get. However, **Laszlo Belady** demonstrated that with the FIFO page replacement algorithm, increasing the number of page frames can sometimes lead to more page faults in a program.
4	About Laszlo Belady (born on April 29th, 1928).
5	Belady's Algorithm: Showcasing how the FIFO algorithm can have more page faults with more pages
6	**Modeling Page Replacement Algorithm (Stack Algorithm)**: A paging system can be characterized by three items: 1. The reference string of the executing process 2. The page replacement algorithm 3. The number of page frames available for a process in memory
7	**Modeling Page Replacement Algorithm (Stack Algorithm)**: Model for Stack Algorithm: Maintains an *internal array M* that keeps track of the state of memory. *M* has as many as *virtual memory pages n*. Top *m* entries contain all the pages currently in memory (*n = # of pages, m = # of page frames*). Bottom *n – m* entries contain all the pages that have been referenced once but have been page out and are not currently in memory
8	**Modeling Page Replacement Algorithm (Model for Stack Algorithm)**: Properties of the model: 1. When a page is referenced, it is always moved to the top entry in *M*. 2. If the page referenced was already in *M*, all pages above it move down one position 3. A transition from within the box to outside of it corresponds to a page being evicted from the memory 4. The pages that were below the referenced page are not moved
9	**Modeling Page Replacement Algorithm (Property for Stack Algorithm)**: If we increase memory size by one page frame and re-execute the process, at every point during the execution, all the pages that were present in the first run are also present in the second run, along with one additional page. Replacement algorithms that are stack algorithms do not suffer from Belady's Anomally.
10	**Modeling Page Replacement Algorithm (Stack Algorithm)**: LRU algorithm with the model: 1. The reference strings: 0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 2 3 4 1 2. The page replacement algorithm: LRU

	3. The number of page frames available in memory: a. Virtual Spaces: 8 pages b. Physical Spaces: 4 page frames or 5 page frames
11	Example showcasing stack algorithm (LRU)
12	Example showcasing stack algorithm (LRU)
13	**Design Issues for Paging System (Local vs Global Allocation)**: With multiple processes competing for frames, we can classify page replacement algorithms into two categories: global replacement and local replacement. Global replacement allows a process to select a replacement frame from the set of all frames, while local replacement requires that each process select from only its own set of allocated frame
14	Diagram showing local allocation, and how the physical space is divided into chunks for each virtual space
15	Diagram showing global allocation, and how the frames in physical space are jumbled into small sections to accommodate virtual spaces
16	**Design Issues for Paging System (Load Control)**: Even when we use best replacement algorithm with global allocation policies, if the combined working sets of all processes exceed the capacity of memory, thrashing can be expected. ***Thrashing***: two or more processes access a shared resource repeatedly such that serious system performance degradation occurs because the system is spending a disproportionate amount of time just *accessing* the shared resource. Possible concern is reducing the degree of multiprogramming- swapping out some process onto the disk
17	**Design Issues for Paging System (Page Size)**: Smaller page size causes smaller internal fragmentation, with larger size causing more unused program to be in memory than a small page size. However, small page size means that program will need many pages with large page tables, wasting time for the seek and rotation. Some system need to load the page table into the hardware registers to execute, with many small pages needing more time to load. The overhead for this causes either a larger, slower page table, or increased internal fragmentation loss
18	**Design Issues for Paging System (Page Size)**: Mathematical Analysis S: average size of process (byte) P: the size of page (byte) E: Each page table entry needs (byte). $\frac{S}{P}$: *Average* number of pages per process $\frac{S}{P} \times E$: Average page table space $\frac{P}{2}$: the wasted memory in the last page of the process
19	**Design Issues for Paging System (Page Size)**: More mathematical analysis

Total overhead by page table and internal fragmentation loss is

$$Overhead(P) = \frac{SE}{P} + \frac{P}{2}$$

$$Overhead'(P) = -\frac{SE}{P^2} + \frac{1}{2} = 0$$

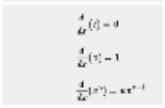
$P = \sqrt{2SE}$: *optimal* page size

$\frac{S}{P}$: *Average* number of pages per process

$\frac{S}{P} \times E$: Average page table space

$\frac{P}{2}$: the wasted memory in the last page of the process

Some Basic Derivatives



20 Segmentation