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