CSC 301 – OPERATING SYSTEMS II (2 UNITS) LECTURE 3

BALOGUN JEREMIAH ADEMOLA

ASSISTANT LECTURER,

DEPARTMENT OF COMPUTER SCIENCE AND MATHEMATICS

MOUNTAIN TOP UNIVERSITY, OGUN STATE, NIGERIA

COURSE OUTLINE

- **Concurrency**
  - States and State Diagrams Structures
  - Dispatching and Context Switching
  - Interrupts
  - Concurrent Execution
  - Mutual Exclusion Problem and Some Solutions
- **Deadlock**
  - Models and Mechanisms (Semaphores, monitors etc.)
  - Producer – Consumer Problems and Synchronization
  - Multiprocessor Issues
- **Scheduling and Dispatching, Memory Management**
  - Overlays, Swapping and Partitions
  - Paging and Segmentations Placement and Replacement Policies
  - Working Sets and Trashing
  - Caching

INTERRUPT

- In digital computers, an interrupt is a response by the processor to an event that needs attention from the software.
  - An interrupt condition alerts the processor and serves as a request for the processor to interrupt the currently executing code when permitted, so that the event can be processed in a timely manner.
  - If the request is accepted, the processor responds by suspending its current activities, saving its state, and executing a function called an interrupt handler (or an interrupt service routine, ISR) to deal with the event.

- This interruption is temporary, and, unless the interrupt indicates a fatal error, the processor resumes normal activities after the interrupt handler finishes.
  - Interrupts are commonly used by hardware devices to indicate electronic or physical state changes that require attention.
  - Interrupts are also commonly used to implement computer multitasking, especially in real-time computing.
  - Systems that use interrupts in these ways are said to be interrupt-driven

INTERRUPT

Vector No.	Mnemonic	Description	Source
0	#DE	Divide Error	DIV and IDIV instructions.
1	#DB	Debug	Any code or data reference.
2		NMI Interrupt	Non-maskable external interrupt.
3	#BP	Breakpoint	INT 3 instruction.
4	#OF	Overflow	INTO instruction.
5	#BR	BOUND Range Exceeded	BOUND instruction.
6	#UD	Invalid Opcode (UnDefined Opcode)	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.
9	#MF	CoProcessor Segment Overrun (reserved)	Floating-point instruction.[2]
10	#TS	Invalid TSS	Task switch or TSS access.
11	#NP	Segment Not Present	Loading segment registers or accessing system segments.
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory.[3]
18	#MC	Machine Check	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction[5]
20-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT *n* instruction.

HARDWARE INTERRUPTS

- A hardware interrupt is a condition related to the state of the hardware that may be signaled by an external hardware device, e.g.,
  - an interrupt request (IRQ) line on a PC,
  - detected by devices embedded in processor logic (e.g., the CPU timer in IBM System/370), to communicate that the device needs attention from the operating system (OS)
  - if there is no OS, from the "bare-metal" program running on the CPU.

- Such external devices may be part of the computer (e.g., disk controller) or they may be external peripherals.
  - For example, pressing a keyboard key or moving a mouse plugged into a PS/2 port triggers hardware interrupts that cause the processor to read the keystroke or mouse position.

- Hardware interrupts can arrive asynchronously with respect to the processor clock, and at any time during instruction execution.
  - Consequently, all hardware interrupt signals are conditioned by synchronizing them to the processor clock, and acted upon only at instruction execution boundaries.

HARDWARE INTERRUPTS - MASKING

- Processors typically have an internal interrupt mask register which allows selective enabling and disabling of hardware interrupts.
- Each interrupt signal is associated with a bit in the mask register;
  - on some systems, the interrupt is enabled when the bit is set and disabled when the bit is clear,
  - while on others, a set bit disables the interrupt.
- When the interrupt is disabled, the associated interrupt signal will be ignored by the processor.
  - Signals which are affected by the mask are called maskable interrupts.
  - Some interrupt signals are not affected by the interrupt mask and therefore cannot be disabled; these are called non-maskable interrupts (NMI).
- NMIs indicate high priority events which cannot be ignored under any circumstances, such as the timeout signal from a watchdog timer.
  - To mask an interrupt is to disable it, while to unmask an interrupt is to enable it.
- A spurious interrupt is an invalid, short-duration signal on an interrupt input.
- These are usually caused by glitches resulting from electrical interference, race conditions, or malfunctioning devices.

SOFTWARE INTERRUPTS

- A software interrupt is requested by the processor itself upon executing particular instructions or when certain conditions are met.
  - Every software interrupt signal is associated with a particular interrupt handler.

- A software interrupt may be intentionally caused by executing a special instruction which, by design, invokes an interrupt when executed.
  - Such instructions function similarly to subroutine calls and are used for a variety of purposes, such as requesting operating system services and interacting with device drivers (e.g., to read or write storage media).

- Software interrupts may also be unexpectedly triggered by program execution errors.
  - These interrupts typically are called traps or exceptions.
  - For example, a divide-by-zero exception will be "thrown" (a software interrupt is requested) if the processor executes a divide instruction with divisor equal to zero.
  - Typically, the operating system will catch and handle this exception.

INTERRUPT – TRIGGERING METHODS

- Each interrupt signal input is designed to be triggered by either a logic signal level or a particular signal edge (level transition).
  - Level-sensitive inputs continuously request processor service so long as a particular (high or low) logic level is applied to the input.
  - Edge-sensitive inputs react to signal edges: a particular (rising or falling) edge will cause a service request to be latched; the processor resets the latch when the interrupt handler executes.

- **LEVEL TRIGGERED**
  - A level-triggered interrupt is requested by holding the interrupt signal at its particular (high or low) active logic level.
  - A device invokes a level-triggered interrupt by driving the signal to and holding it at the active level.
  - Level-triggered inputs allow multiple devices to share a common interrupt signal via wired-OR connections.
  - The processor polls to determine which devices are requesting service.
  - After servicing a device, the processor may again poll and, if necessary, service other devices before exiting the ISR.

- **EDGE-TRIGGERED**
  - An edge-triggered interrupt is an interrupt signaled by a level transition on the interrupt line, either a falling edge (high to low) or a rising edge (low to high).
  - A device wishing to signal an interrupt drives a pulse onto the line and then releases the line to its inactive state.
  - If the pulse is too short to be detected by polled I/O then special hardware may be required to detect it.

INTERRUPT HANDLER

- In computer systems programming, an interrupt handler, also known as an interrupt service routine or ISR, is a special block of code associated with a specific interrupt condition.
  - Interrupt handlers are initiated by hardware interrupts, software interrupt instructions, or software exceptions, and are used for implementing device drivers or transitions between protected modes of operation, such as system calls.
  - The traditional form of interrupt handler is the hardware interrupt handler.

- Hardware interrupts arise from electrical conditions or low-level protocols implemented in digital logic, are usually dispatched via a hard-coded table of interrupt vectors, asynchronously to the normal execution stream (as interrupt masking levels permit), often using a separate stack, and automatically entering into a different execution context (privilege level) for the duration of the interrupt handler's execution.
  - In general, hardware interrupts and their handlers are used to handle high-priority conditions that require the interruption of the current code the processor is executing.

- Interrupt handlers have a multitude of functions, which vary based on what triggered the interrupt and the speed at which the interrupt handler completes its task.
  - For example, pressing a key on a computer keyboard, or moving the mouse, triggers interrupts that call interrupt handlers which read the key, or the mouse's position, and copy the associated information into the computer's memory

INTERRUPT FLAGS

- Unlike other event handlers, interrupt handlers are expected to set interrupt flags to appropriate values as part of their core functionality.

- Even in a CPU which supports nested interrupts, a handler is often reached with all interrupts globally masked by a CPU hardware operation.
  - In this architecture, an interrupt handler would normally save the smallest amount of context necessary, and then reset the global interrupt disable flag at the first opportunity, to permit higher priority interrupts to interrupt the current handler.
  - It is also important for the interrupt handler to quell the current interrupt source by some method (often toggling a flag bit of some kind in a peripheral register) so that the current interrupt isn't immediately repeated on handler exit, resulting in an infinite loop.

- Exiting an interrupt handler with the interrupt system in exactly the right state under every eventuality can sometimes be an arduous and exacting task, and its mishandling is the source of many serious bugs, of the kind that halt the system completely.
  - These bugs are sometimes intermittent, with the mishandled edge case not occurring for weeks or months of continuous operation.
  - Formal validation of interrupt handlers is tremendously difficult, while testing typically identifies only the most frequent failure modes, thus subtle, intermittent bugs in interrupt handlers often ship to end customers.

CONCURRENCY

CONCURRENCY

- Concurrency is the interleaving of processes in time to give the appearance of simultaneous execution.
  - Thus it differs from parallelism, which offers genuine simultaneous execution.

- However the issues and difficulties raised by the two overlap to a large extent:
  - sharing global resources safely is difficult;
  - optimal allocation of resources is difficult;
  - locating programming errors can be difficult, because the contexts in which errors occur cannot always be reproduced easily.

- Parallelism also introduces the issue that different processors may run at different speeds, but again this problem is mirrored in concurrency because different processes progress at different rates.
  - Concurrent processes come into conflict with each other when they are competing for the use of the same resource.
  - They are not necessarily aware of each other, but the execution of one process may affect the behavior of competing processes.

CONCURRENCY – A SIMPLE EXAMPLE

- The fundamental problem in concurrency is processes interfering with each other while accessing a shared global resource.
  - This can be illustrated with a surprisingly simple example:

```
chin = getchar();
chout = chin;
putchar(chout);
```

- Imagine two processes **P1** and **P2** both executing this code at the "same" time, with the following interleaving due to multi-programming.
  - **P1** enters this code, but is interrupted after reading the character *x* into **chin**.
  - **P2** enters this code, and runs it to completion, reading and displaying the character *y*.
  - **P1** is resumed, *but* **chin** now contains the character *y*, so **P1** displays the wrong character.

- The essence of the problem is the shared global variable **chin**.
  - **P1** sets **chin**, but this write is subsequently lost during the execution of **P2**.
  - The general solution is to allow only one process at a time to enter the code that accesses **chin**: such code is often called a critical section.
  - When one process is inside a *critical section* of code, other processes must be prevented from entering that section.
  - This requirement is known as *mutual exclusion*

CONCURRENCY – MUTUAL EXCLUSION

- Mutual exclusion is in many ways the fundamental issue in concurrency.
  - It is the requirement that when a process P is accessing a shared resource R, no other process should be able to access R until P has finished with R.
  - Examples of such resources include files, I/O devices such as printers, and shared data structures.

- There are essentially three approaches to implementing mutual exclusion.
  - Leave the responsibility with the processes themselves: this is the basis of most software approaches.
    - These approaches are usually highly error-prone and carry high overheads.
  - Allow access to shared resources only through special-purpose machine instructions: i.e. a hardware approach.
    - These approaches are faster but still do not offer a complete solution to the problem, e.g. they cannot guarantee the absence of deadlock and starvation.
  - Provide support through the operating system, or through the programming language.
- Three approaches in this category includes: semaphores, monitors, and message passing.

MUTUAL EXCLUSION - SEMAPHORES

- The fundamental idea of semaphores is that processes "communicate" via global counters that are initialised to a positive integer and that can be accessed only through two atomic operations.
  - semSignal(x) increments the value of the semaphore x.
  - semWait(x) tests the value of the semaphore x: if x > 0, the process decrements x and continues; if x = 0, the process is blocked until some other process performs a semSignal, then it proceeds as above.
- A critical code section is then protected by bracketing it between these two operations:

semWait (x);

<critical code section>

semSignal (x);

- In general the number of processes that can execute this critical section simultaneously is determined by the initial value given to x.
  - If more than this number try to enter the critical section, the excess processes will be blocked until some processes exit.
  - Most often, semaphores are initialised to one.

MUTUAL EXCLUSION - MONITORS

- The principal problem with semaphores is that calls to semaphore operations tend to be distributed across a program, and therefore these sorts of programs can be difficult to get correct, and very difficult indeed to prove correct!

- Monitors address this problem by imposing a higher-level structure on accesses to semaphore variables.
  - A monitor is essentially an object (in the Java sense) which has the semaphore variables as internal (private) data and the semaphore operations as (public) operations.
  - Mutual exclusion is provided by allowing only one process to execute the monitor's code at any given time.

- Monitors are significantly easier to validate than "bare" semaphores for at least two reasons:
  - all synchronisation code is confined to the monitor; and
  - once the monitor is correct, any number of processes sharing the resource will operate correctly

MUTUAL EXCLUSION – MESSAGE PASSING

- With an approach based on message passing, processes operate in isolation from each other (i.e. they do not share data), and they exchange information where necessary by the sending and receiving of messages.
- Synchronisation between processes is defined by the blocking policy attached to the sending and receiving of messages.

- The most common combination is
  - **Non-blocking send:** When a process sends a message, it continues executing without waiting for the receiving process.
  - **Blocking receive**: When a process attempts to receive a message, it blocks until the message is available.

- With this blocking policy, mutual exclusion can be achieved for a set of processes that share a mailbox **box.**
- Some number of messages (usually one) is sent to box initially by the system, then each process executes the following code when it wants to enter the critical section:

receive (box);

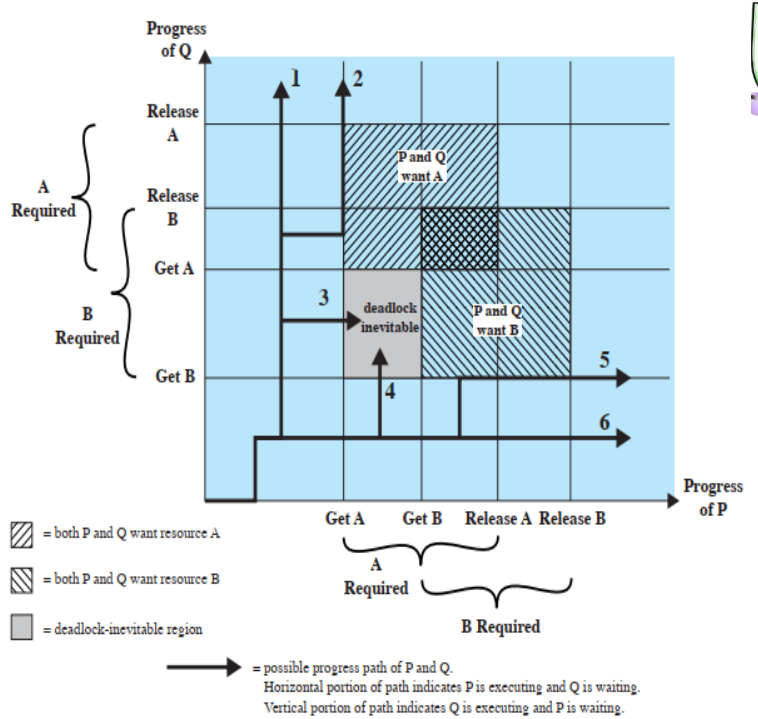
<critical code section>

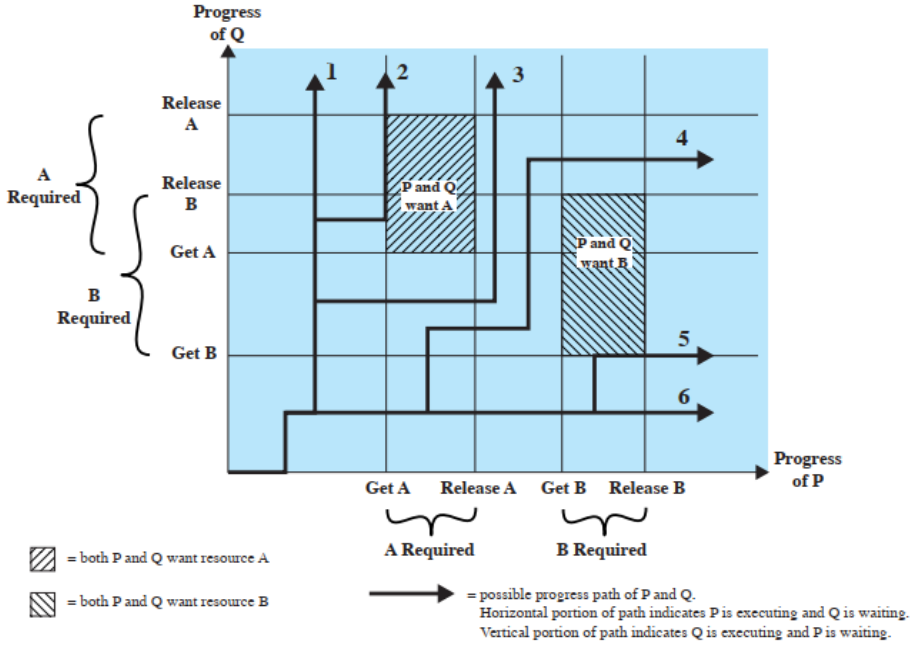
send (box);

DEADLOCK

DEADLOCK

- Deadlock is defined as the permanent blocking of a set of processes that either compete for global resources or communicate with each other.
  - It occurs when each process in the set is blocked awaiting an event that can be triggered only by another blocked process in the set.
- Consider the figure above, in which both processes P and Q need both resources A and B simultaneously to be able to proceed.
  - Thus P has the form get A, ... get B, ..., release A, ..., release B, and Q has the form get B, ... get A, ..., release B, ..., release A.
- *Q acquires both resources, then releases them. P can operate freely later.*
- *Q acquires both resources, then P requests A. P is blocked until the resources are released, but can then operate freely.*
- *Q acquires B, then P acquires A, then each requests the other resource. Deadlock is now inevitable.*
- *P acquires A, then Q acquires B, then each requests the other resource. Deadlock is now inevitable.*
- *P acquires both resources, then Q requests B. Q is blocked until the resources are released, but can then operate freely.*
- *P acquires both resources, then releases them. Q can operate freely later.*





CONDITIONS FOR DEADLOCK

- Three policy conditions are necessary for deadlock to be possible.
  - Mutual exclusion - Only one process may use a resource at one time.
  - Hold and wait - A process may hold some resources while waiting for others.
  - No preemption - No process can be forced to release a resource.
- A fourth condition is required for deadlock to actually occur.
  - Circular wait - A closed chain of processes exists, such that each process is blocked waiting for a resource held by another process in the set.
- Three approaches exist for dealing with deadlock.
  - Prevention involves adopting a static policy that disallows one of the four conditions above.
  - Avoidance involves making dynamic choices that guarantee prevention.
  - Detection and recovery involves recognising when deadlock has occurred, and trying to recover

DEADLOCK PREVENTION – HOLD AND WAIT

- We could force a process to request all of its resources at one time:
  - if successful, it would be able to proceed to completion;
  - if unsuccessful, it would be holding no resources that could block other processes.

- The principal problems are that
  - modular program may not be aware of all of its resource requirements;
  - a process needing several resources may be held up a long time waiting for them all to be available simultaneously;
  - resources allocated to a process may not be used for a long time, leading to inefficient allocations.

DEADLOCK PREVENTION – NO PREEMPTION

- We could force a process that holds R and then unsuccessfully requests R' to release R.

- Or we could force a process that holds R to release it, if it is requested by another process.

- These approaches work only with resources whose state can easily be saved and restored, and even then they lead to obvious inefficiencies.

DEADLOCK PREVENTION – CIRCULAR WAIT

- We could force all processes to request resources in the same order.

- Then two processes both needing A and B would both request A first, guaranteeing that B would be available for the successful process.

- Disallowing circular wait suffers from similar inefficiencies as disallowing hold and wait.

DEADLOCK AVOIDANCE

- Deadlock avoidance is subtly different from deadlock prevention:
  - we allow the three necessary conditions for deadlock,
  - but we dynamically allocate resources in such a way that deadlock never occurs.

- There are two principal approaches.
  - Do not start a process if its demands might lead to deadlock.
    - This strategy is very conservative and thus inefficient.
  - Do not grant a resource request if this allocation might lead to deadlock.
    - The basic idea is that a request is granted only if some allocation of the remaining free resources is sufficient to allow all processes to complete.

DEADLOCK DETECTION AND RECOVERY

- Both prevention and avoidance of deadlock lead to conservative allocation of resources, with corresponding inefficiencies.

- Deadlock detection takes the opposite approach:
  - make allocations liberally, allowing deadlock to occur (on the assumption that it will be rare),
  - apply a detection algorithm periodically to check for deadlock, and
  - apply a recovery algorithm when necessary

- The detection algorithm can be applied at every resource allocation, or less frequently, depending on the trade-off between the likelihood of deadlock occurring and the cost of the algorithm.

- Detection algorithms are broadly similar to the avoidance algorithms discussed previously, and are able to identify which processes are deadlocked under the current resource allocation.

DEADLOCK DETECTION AND RECOVERY…

- Recovery algorithms vary a lot in their severity:
  - Abort all deadlocked processes.
    - Though drastic, this is probably the most common approach!
  - Back-up all deadlocked processes.
    - This requires potentially expensive rollback mechanisms, and of course the original deadlock may recur.
  - Abort deadlocked processes one at a time until the deadlock no longer exists.
  - Preempt resources until the deadlock no longer exists.

- With the last two approaches, processes or resources are chosen to minimise the global "loss" to the set of processes, by minimising the loss of useful processing with respect to relative priorities