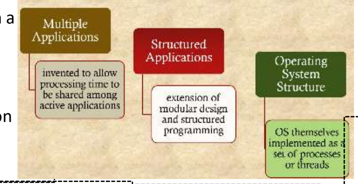
OS design->management of processes and threads: multiprogramming, multiprocessing, distributed Processing

-**multiprogramming**: manage multiple processes within a uniprocessor system
-**multiprocessing:** manage multiple processes within a multiprocessor
-**Distributed processing:** manage processes executing on multiple, distributed computer systems. Ex: The recent proliferation of lusters



interleaving and overlapping:
-viewed as examples of concurrent processing
-both present the same problems

Uniprocessor-the relative speed of execution of processes cannot be predicted:
-depends on activities of other processes
-the way OS handles interrupts
-scheduling policies of the OS

concurrency encompasses a host of design issues: communication among processes; sharing of and competing for resources(memory, files, I/O access); synchronization of the activities of multiple processes; allocation of processor time to process

critical section: a section of code within a process that: require access to shared resources+ must not be executed while another process is in a corresponding section of code

deadlock: a situation in which two or more processes are unable to proceed because each is waiting for one of the other to ***do something***

livelock: a situation in which two or more processes continuously change their states in response to changes in the other process(es) ***without doing*** any useful work

starvation: a situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceeded, it is never chosen

Difficulties of Concurrency: -sharing of global resources -Difficult for OS to manage the allocation of the resources optimally -Difficult to locate programming errors as results are not deterministic and reproducible

atomic operation: a function implemented-sequence of >=1 instructions that indivisible->no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes

mutual exclusion: the requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources

race condition: a situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution: the "loser" of the race is the process that updates last and will determine the final value of the variable

What design and management issues are raised by the existence of concurrency?
1. The OS must be able to keep track of the various processes. This is done with the use of process control blocks .
2. The OS must allocate and de-allocate various resources for each active process. At times, multiple processes want access to the same resource. These resources include
• Processor time: This is the scheduling function. Memory: Most operating systems use a virtual memory scheme. Files: I/O devices: Discussed in Chapter 11 .
3. The OS must protect the data and physical resources of each process against unintended interference by other processes..
4. The functioning of a process, and the output it produces, must be independent
of the processing speed

resource competing causes deadlocks, starvation, need for mutual exclusion

-**main objective of deadlock prevention techniques:** adopt a policy at the design level to eliminate one of the conditions for deadlock. -**Explain why in the deadlock detection algorithm, u must mark each process that has a row in the allocation matrix of all zeros**? Because a process without resources cannot be deadlocked**. -Does disabling the interrupts guarantee mutual exclusion in multiprocessors?** Disabling interrupts will not prevent other processes from executing on a different processor and accessing the critical section at the same time. ***-How does a monitor guarantee mutual exclusion?*** The data of the monitor is only accessible through its methods and only one process can call these methods at a particular time. **What is the major disadvantage of the special machine instructions**? Busy Wait.

Degree of Awareness	Relationship	Influence that one process has on the other	potential control problems
processes unaware of each other	Competition	-Results of one process independent of **action** of others -timing of process may be affected	-mutual exclusion -deadlock (renewable resource) -Starvation
processes indirectly aware of each other	cooperation by sharing	Result of one process may depend on **information** obtained from others	-mutual exclusion'-deadlock (renewable resource)'-Starvation -data coherence
process directly aware of each other (have communication primitives available to them	Cooperation by communication	Result of one process may depend on **info** obtained from others timing of process may be affected	-deadlock (consumable resource) -Starvation

. Select the resource that is **NOT** reusable:

a) Semaphores b) CPU c) Memory **d) Messages** e) None of the above

. Select the condition for a deadlock that guarantees that no resource can be forcibly removed from a process holding it:

a) Mutual Exclusion b) Hold and Wait **c) No Preemption** d) Circular Wait

. Select the instruction from the correct solution of Peterson's algorithm that represents a busy wait:

a) flag[0]=TRUE; b) turn=1; **c) while(flag[1] && turn==1);** d) flag[0]=FALSE;

'. Select the policy used by a weak semaphore to release the processes in the queue after a signalSem:

a) FIFO; b) LRU; c) FCFS **d) Not specified;**

'. Choose the most useful combination when selecting the primitives in the message passing solution:

a) Blocking send, Blocking receive b) Blocking send, Nonblocking receive
c) Nonblocking send, Blocking receive d) Nonblocking send, Nonblocking receive

. It is the condition in which multiple processes try to get access to a shared resource at the same time:

a) Starvation b) Deadlock **c) Racing condition** d) Mutual Exclusion

. Select the type of semaphore that is normally used to create a critical section:

a) Mutex b) Counting c) Strong d) Weak e) None of the above

. The OS needs to be concerned about **competition for resources** when the processes are:

a) Indirectly aware of each other **b) Unaware of each other**
c) Directly aware of each other d) None of the above

.. Select the method that must be part of a program based on monitors to solve for the producer-consumer problem:

a) take_from_buffer(); b) produce(); c) consume(); d) None of the above

.. Select the primitive that is **NOT** an atomic operation:

a) waitSem(); b) signalSem();
c) Special Machine Instructions **d) None of the above**

A restaurant has a single employee taking orders and has three seats for its custon The employee can only serve one customer at a time and each seat can accommodate one customer at a time. Complete the following function template in a that guarantees that customers will never have to wait for a seat while holding the they have just purchased.

```
semaphore seats = 3;
semaphore employee = 1;
void customer () {
        semWait(&seats);
        semWait(&employee);
        order_food();
        semSignal(&employee);
```

a) Set the initial values of the semapho **(5 points).**
b) Select the missing instructions after **order_food()** and **eat()** from the following list **(15 points)**:

1. semSignal(&seats);

mutual exclusion: requirements: -must be enforced -a process that halts must do so without interfering with other processes- no deadlock or starvation- A process must not be denied access to a critical section when there is no other process using it-No Assumptions are made about relative process speeds or number of processes- A process remains inside its critical section for a finite time only (13)

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
     while (compare_and_swap(bolt, 0, 1) == 1)
         /* do nothing */;
       /* critical section */;
       bolt = 0;
       /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), ... ,P(n));
}
```

(a) Compare and swap instruction

busy wait: A loop and do nothing(while(); for();)
strong semaphore: FIFO, FCFS
In producer-customer problem: buffer is a shared resource

Concurrency arises in three different contexts:
• **Multiple applications:** Multiprogramming was invented to allow processing time to be dynamically shared among a number of active applications.
• **Structured applications:** As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of concurrent processes.
Operating system structure: The same structuring advantages apply to systems programs, and we have seen that operating systems are themselves often implemented as a set of processes or threads.

Banker's Algorithm

KEY:
A = Allocation
C = MAX
C – A = NEED
R = RESOURCES
AVAILABLE
V = R – A

EXAMPLE:
R = [3 1 2 2 1]
A = [3 1 1 2 0]
V = [0 0 1 0 1]

C – A: (Need)
0 1 0 0 1
0 0 1 0 1
0 0 0 0 1
1 0 1 0 1

AVAILABLE (V)/(Work)
0 0 1 0 1
1 1 1 0 1
2 1 2 1 1
3 1 2 2 1

$$C = \begin{array}{c c} & \begin{array}{ccccc} R_1 & R_2 & R_3 & R_4 & R_5 \end{array} \\ \begin{array}{c} P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} & \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix} \end{array} \qquad A = \begin{array}{c c} & \begin{array}{ccccc} R_1 & R_2 & R_3 & R_4 & R_5 \end{array} \\ \begin{array}{c} P_1 \\ P_2 \\ P_3 \\ P_4 \end{array} & \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \qquad R = [3 \quad 1 \quad 2 \quad 2 \quad 1]$$

V = [0 0 1 0 1] *SAFE STATE*

A:
1 0 1 1 0
1 1 0 0 0
1 0 0 1 0
0 0 0 0 0
\+ --------------
3 1 1 2 0

Solution: Need <= Work, if so - > Work = Work + Allocation

P1 = 0 1 0 0 1 <= 0 0 1 0 1, False
P2 = 0 0 1 0 1 <= 0 0 1 0 1, True -> Work = 0 0 1 0 1 + 1 1 0 0 0 = 1 1 1 0 1
P1 = 0 1 0 0 1 <= 1 1 1 0 1, True -> Work = 1 1 1 0 1 + 1 0 1 1 0 = 2 1 2 1 1
P3 = 0 0 0 0 1 <= 2 1 2 1 1, True -> Work = 2 1 2 1 1 + 1 0 0 1 0 = 3 1 2 2 1
P4 = 1 0 1 0 1 <= 3 1 2 2 1, True

Safe State = <P2, P1, P3, P4> or <P2, P3, P4, P1>