Concurrency, Mutual Exclusion, Process Synchronization

Atomicity	ensure that no other thread changes while the data is running
Bounded Waiting (No Starvation)	Solution: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
busy-wait mechanism	enforces mutex in the software areas.
Concurrency	a programming design philosophy which is the interleaving of processes in time to give the appearance of simultaneous execution - that are accessing or modifying some shared state
Concurrency	is when two tasks overlap in execution.
Concurrency Benefits	to be able to run multiple applications at about the same time, better resource utilization, better average response time and better performance
Concurrency Control	Allow several transactions to be executing simultaneously such that Collection of manipulated data item is left in a consistent state
Concurrency Controls	are used to make sure each transaction on the database takes place in a particular order rather than at the same time. This keeps the transactions from working at the same time, which could cause data to become incorrect or corrupt the database
Concurrency Drawbacks	Multiple applications need to be protected from one another - 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. Multiple applications may need to coordinate through additional mechanisms Switching among applications requires additional performance overheads and complexities in operating systems (e.g., deciding which application to run next.)
Conditional synchronization	ensure code in different threads will run in correct order
Condition Variables	provide a mechanism to wait for events inside the monitor (a "rendezvous point")
Cooperative Process	Execution of one process affects the execution of other processes.
Critical Section	Each process has a segment of code, in which the process may be changing common variables, updating a table, writing a file, and so on. It is a shared resource.
Critical-section problem	refers to the problem of how to ensure that at most one process is executing its critical section at a given time
Independent Process	Execution of one process does not affects the execution of other processes
Locking	Most used in operating systems usually in the form of semaphore and spinlock
Monitor	fundamental high-level synchronization construct and is a programming language construct that controls access to shared data
Multiprocessor	- Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. - Difficulties in disabling and the re-enabling interrupts on all processors - This message passing delays entry into each critical section, and system efficiency decreases. - Special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically—that is, as one uninterruptible unit
Mutex	reduces latency and busy-waits using queuing and context switches. It can be enforced at both the hardware and software levels
Mutex Locks	Use to protect critical regions and thus prevent race conditions
Mutex Locks	simplest software tool to solve the critical-section problem
Mutual Exclusion	program object that prevents simultaneous access to a shared resource.
Mutual Exclusion	Solution: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
Non-preemptive Kernels	does not allow a process running in kernel mode to be preempted and process will run until it exits kernel mode or voluntarily yields control of the CPU

25.	**Non-preemptive Kernels**	free from race conditions on kernel data structures, as only one process is active in the kernel at a time
26.	**Optimistic concurrency control**	In optimistic concurrency control, users do not lock data when they read it. When a user updates data, the system checks to see if another user changed the data after it was read. If another user updated the data, an error is raised. Typically, the user receiving the error rolls back the transaction and starts over. It is mainly used in environments where there is low contention for data, and where the cost of occasionally rolling back a transaction is lower than the cost of locking data when read.
27.	**Parallel execution**	is when two tasks start at the same time, making it a special case of concurrent execution
28.	**Parallelism**	is the run-time behaviour of executing multiple tasks at the same time
29.	**Pessimistic concurrency control**	- A system of locks prevents users from modifying data in a way that affects other users. - After a user performs an action that causes a lock to be applied, other users cannot perform actions that would conflict with the lock until the owner releases it. - It is mainly used in environments where there is high contention for data, where the cost of protecting data with locks is less than the cost of rolling back transactions if concurrency conflicts occur.
30.	**Preemptive Kernels**	allows a process to be preempted while it is running in kernel mode and process will run until it exits kernel mode or is blocked from its control of the CPU
31.	**Preemptive Kernels**	must be carefully designed to ensure that shared kernel data are free from race conditions
32.	**Process Synchronization**	sharing system resources by processes in a such a way that, Concurrent access to shared data is handled; thereby minimizing the chance of inconsistent data
33.	**Process Synchronization**	categorize into two types: Independent and Cooperative Process
34.	**Progress**	Solution: If no process is in the critical section, then no other process from outside can block it from entering the critical section.
35.	**Race Condition**	Situations where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when.
36.	**Semaphore**	more robust alternative to simple mutexes
37.	**Single Processor**	- Prevent interrupts from occurring while a shared variable was being modified - Current sequence of instructions would be allowed to execute in order without preemption - No other instructions would be run, so no unexpected modifications could be made to the shared variable
38.	**Synchronization mechanism**	allows programs to write rules for concurrency