Quiz 2 Compressed

1. Processes

- **Process States**: Processes can exist in several states, such as new, running, waiting, ready, or terminated, based on their current activity.
- **Process Control Block (PCB)**: The PCB contains important information about each process, such as its current state, process ID, CPU registers, program counter, memory management data, CPU scheduling details, and I/O status.
- **Scheduling Queues**: Processes are maintained in various queues: the **Job queue** (all processes in the system), **Ready queue** (processes in main memory that are ready to execute), and **Device queue** (processes waiting for I/O).
- **Schedulers**:
  - **Short-term Scheduler (CPU Scheduler)**: Allocates CPU to processes from the ready queue.
  - **Long-term Scheduler (Job Scheduler)**: Controls the degree of multiprogramming by selecting processes from the job pool to be brought into main memory.
  - **Medium-term Scheduler**: Swaps processes in and out of memory to manage the level of multiprogramming.
- **Context Switch**: When the CPU switches from one process to another, it must save the state of the old process and load the state of the new one. This process is called a context switch and introduces an overhead since no useful work is done during the switch.
- **Process Creation**: Processes are created using system calls like `fork()`. A parent process can create child processes, forming a hierarchical tree. The `exec()` system call is used to replace a process's memory space with a new program.
- **Process Termination**: A process can be terminated using `exit()`. The parent process can also wait for the child process to finish using `wait()`. In some cases, a parent process may terminate a child process using `abort()`.

2. Interprocess Communication (IPC)

- **Models**:
  - **Shared Memory**: Processes communicate by sharing a region of memory. Synchronization mechanisms are necessary to prevent race conditions.
  - **Message Passing**: Involves sending and receiving messages between processes. It can be **Direct** (processes explicitly name each other) or **Indirect** (using mailboxes or ports).
- **IPC Scenarios**: Processes can be either independent or cooperating. Cooperating processes often need to communicate for reasons such as modularity, computation speedup, information sharing, and convenience.

- **Direct Communication**: Processes explicitly communicate using `send(P, message)` and `receive(Q, message)`, where links are established automatically.
- **Indirect Communication**: Processes communicate via shared mailboxes. Operations include `send(A, message)` and `receive(A, message)` where A is a mailbox.

3. Threads & Concurrency

- **Thread**: A thread is the smallest unit of CPU utilization, consisting of a thread ID, program counter, register set, and stack. Threads within the same process share code, data, and OS resources.
- **Benefits**: Threads improve **responsiveness**, allow **resource sharing** within a process, provide **economy** in terms of memory and resource usage, and enhance **scalability** by efficiently using multiprocessor architectures.
- **Types**:
  - **User Threads**: Managed at the user level by thread libraries, such as POSIX Pthreads or Java threads.
  - **Kernel Threads**: Managed by the operating system; examples include threads used by Linux and Windows.
- **Multithreading Models**:
  - **Many-to-One**: Maps many user threads to a single kernel thread. It is efficient but cannot run multiple threads in parallel on multicore systems.
  - **One-to-One**: Maps each user thread to a kernel thread, providing more concurrency but with potential overhead due to the large number of kernel threads.
  - **Many-to-Many**: Maps many user threads to an equal or smaller number of kernel threads, allowing for optimal resource usage and improved concurrency.
- **Threading Issues**: Issues include the behavior of `fork()` and `exec()` in multithreaded environments, thread cancellation (immediate or deferred), signal handling in threads, thread pools, and thread-specific data.

4. CPU Scheduling

- **Scheduling Criteria**:
  - **CPU Utilization**: Aim to keep the CPU as busy as possible.
  - **Throughput**: Number of processes that complete their execution in a given time period.
  - **Turnaround Time**: The total time taken for a process to complete, from submission to termination.
  - **Waiting Time**: The total time a process spends waiting in the ready queue.
  - **Response Time**: Time between submission of a request and the first response, important for interactive systems.
- **Scheduling Algorithms**:

  - **First-Come, First-Served (FCFS)**: The simplest scheduling algorithm, but it can lead to the **convoy effect**, where short processes get stuck waiting behind long processes.
  - **Shortest Job First (SJF)**: Selects the process with the shortest next CPU burst. It can be **preemptive** (Shortest Remaining Time First - SRTF) or **non-preemptive**. It is optimal for minimizing waiting time but requires predicting future burst times.
  - **Round Robin (RR)**: Each process is given a fixed time slice (quantum). It is effective for time-sharing systems but may have high overhead due to frequent context switching if the quantum is too small.
  - **Priority Scheduling**: Processes are scheduled based on priority. This can be preemptive or non-preemptive. It may lead to **starvation**, which can be addressed by **aging** (gradually increasing the priority of older processes).
  - **Multilevel Queue Scheduling**: Processes are assigned to different queues based on their type (e.g., system processes, interactive processes). Each queue has its own scheduling algorithm.
  - **Multilevel Feedback Queue**: Allows processes to move between queues based on their behavior and execution history, offering flexibility to optimize system performance.
- **Dispatcher**: The dispatcher module gives control of the CPU to the process selected by the scheduler. It involves context switching, switching to user mode, and jumping to the appropriate point in the user program.

5. Scheduling Examples

- **FCFS**: The average waiting time depends heavily on the order of arrival, which can lead to inefficient CPU usage if a long process is followed by shorter ones.
- **SJF**: This algorithm minimizes average waiting time but requires knowledge of the duration of the next CPU burst, which is not always feasible. The preemptive version is called **Shortest Remaining Time First (SRTF)**.
- **Round Robin (RR)**: Each process gets an equal share of the CPU. The performance depends on the **quantum size**—if too large, it behaves like FCFS; if too small, there is significant overhead from context switching.
- **Priority Scheduling**: Starvation is possible, but **aging** can be used to gradually increase the priority of waiting processes to prevent indefinite blocking.
- **Multilevel Feedback Queue**: Provides a dynamic mechanism to adjust process priorities, thereby optimizing system performance based on process behavior.

6. Additional Concepts

- **Context Switching**: A context switch is required to switch the CPU from one process or thread to another. It incurs overhead, as the CPU does no productive work during the switch.
- **Thread Cancellation**:
  - **Asynchronous Cancellation**: Terminates the target thread immediately, which can lead to resource allocation issues.

  - **Deferred Cancellation**: The target thread periodically checks if it should terminate, providing a more controlled approach.
- **Real-Time Scheduling**: Used for systems that require time-critical task execution. **Hard real-time** guarantees tasks are completed on time, whereas **soft real-time** makes a best-effort approach.
- **Dispatcher Latency**: The time taken by the dispatcher to stop one process and start another. Minimizing this is crucial in real-time systems.
- **Multiple-Processor Scheduling**: Involves load sharing across multiple processors. Complexities arise due to processor affinity, where processes tend to stay on the same processor to improve cache performance.
- **Algorithm Evaluation**: Techniques like **deterministic modeling**, **queueing models**, **simulations**, and **implementations** are used to evaluate and compare scheduling algorithms.