UNIT 2: PROCESS MANAGEMENT

Processes and Process Concept

A **process** is a program in execution, which includes the program code and its current activity. The operating system manages processes and ensures they run without interfering with each other. A process is a dynamic entity, having a state (running, waiting, or terminated) and resources (memory, CPU time, etc.).

Process Scheduling

Process scheduling is the method by which the operating system decides which process will run at any given time. The scheduler assigns CPU time to processes based on various factors. The main goal is to optimize system performance and responsiveness.

Operations on Processes

There are several operations that the OS can perform on processes, such as:

- **Creation**: Starting a new process.
- **Termination**: Ending a process.
- **Blocking**: Suspending a process until some condition is met.
- **Unblocking**: Allowing a blocked process to resume execution.

Interprocess Communication (IPC)

Interprocess Communication (IPC) allows processes to communicate and share data with each other. This is essential for systems with multiple processes, as it enables them to work together and exchange information. IPC mechanisms include:

- **Pipes**
- **Message Queues**
- **Shared Memory**
- **Semaphores**

Multithreaded Programming

Multithreaded programming involves dividing a program into smaller threads that can be executed independently. This is useful for parallelism and responsiveness, as threads within a process can run concurrently, especially on multi-core processors.

Multithreading Models

There are several models for multithreading:

1. Many-to-One Model

In this model, many user-level threads are mapped to a single kernel thread.

- **How it works**: The user threads are managed by a user-level thread library, and the kernel sees only one thread, which makes scheduling decisions based on the state of that single kernel thread.
- **Pros**:
  - Simple implementation, since the operating system only needs to manage one thread.
  - Lower overhead for thread management (because the kernel doesn’t need to handle multiple threads).
- **Cons**:
  - If one thread performs a blocking operation (like I/O), all threads are blocked.
  - It doesn't take advantage of multiple processors or cores efficiently because only one thread is being executed at any given time.
- **Example**: The **green threads** in some operating systems are an example of the many-to-one model.

2. One-to-One Model

In this model, each user-level thread is mapped to exactly one kernel thread.

- **How it works**: Each user thread corresponds to a kernel thread, and the kernel manages them independently. The operating system can schedule each thread on a different processor or core, and it handles blocking operations in a way that doesn’t block all threads in the process.
- **Pros**:
  - Improved concurrency, as the kernel can schedule each thread independently, allowing true parallel execution on multi-core systems.
  - Blocking of one thread does not block the entire process.
- **Cons**:
  - There is higher overhead for creating, destroying, and switching between threads, as each thread requires kernel resources.
  - If a system has many user threads, creating an equivalent number of kernel threads can be inefficient and resource-heavy.
- **Example**: **Windows** and **Linux (with pthreads)** typically use the one-to-one model.

3. Many-to-Many Model

In this model, many user-level threads are mapped to many kernel threads. This approach attempts to combine the best aspects of the many-to-one and one-to-one models.

- **How it works**: A set of user-level threads is mapped to a set of kernel threads. The kernel schedules the kernel threads, while the user-level library manages the user threads. This allows more flexibility, as the system can adjust the number of kernel threads based on the number of user threads.
- **Pros**:
  - It allows for better utilization of system resources, as threads can be dynamically mapped to kernel threads.
  - It can take advantage of multi-core processors, improving parallelism and performance.
- **Cons**:
  - More complex to implement than the many-to-one or one-to-one models.
  - Can suffer from inefficiencies if the kernel thread-to-user thread ratio is not well-tuned.
- **Example**: **Solaris** uses the many-to-many model with its **Lightweight Processes (LWP)**.

4. Hybrid Model

The hybrid model is a combination of the previous models. It often uses a mix of one-to-one and many-to-many mapping schemes to maximize efficiency.

- **How it works**: In a hybrid model, some user threads may map directly to kernel threads, while others are multiplexed onto a smaller number of kernel threads. This offers a balance of flexibility and performance.
- **Pros**:
  - Provides the benefits of both one-to-one and many-to-many models.
  - The system can dynamically adjust the number of kernel threads, providing better resource management.
- **Cons**:
  - More complex than either the one-to-one or many-to-many model alone.
  - Still requires efficient management of user and kernel threads.
- **Example**: **Windows 7 and beyond** use a hybrid model, where some threads are mapped one-to-one, and others use many-to-many, depending on the specific thread library and use case.

Summary of the Models:

Model	User Threads	Kernel Threads	Advantages	Disadvantages
Many-to-One	Many	One	Simple to implement, low overhead	Blocks the entire process when one thread is blocked
One-to-One	One	One	True parallelism, better performance on multi-core	Higher overhead due to kernel thread management
Many-to-Many	Many	Many	Flexible, efficient, good for multi-core systems	Complex to implement, requires kernel support
Hybrid	Mix of both	Mix of both	Balances efficiency and flexibility	Complex to manage, requires efficient mapping strategy

---

Thread Libraries

Thread libraries provide the necessary interfaces for creating, managing, and controlling threads within a program. These libraries abstract the underlying details of thread management and provide a set of functions that developers use to implement multithreading in their applications.

Common Thread Libraries:

- **POSIX Threads (pthreads)**:
  - **POSIX Threads** (often called **pthreads**) is a widely used thread library that follows the POSIX standard for thread management. It is available in Unix-like operating systems, including Linux and macOS.
  - It provides functions for thread creation, synchronization, and management.
  - Example of pthreads function: pthread_create() for creating threads and pthread_join() for waiting for thread completion.
- **Windows Threads**:
  - Windows provides a native thread management library as part of its **Windows API**.
  - Windows uses the **CreateThread()** function to create threads and the **WaitForSingleObject()** function to wait for a thread to terminate.
- **Java Threads**:

    - Java provides a built-in mechanism for thread management through its Thread class and Runnable interface.
    - Example: Thread.start() to begin execution of a thread and Thread.join() to wait for a thread to finish.
- **C++11 Threads**:
    - C++ introduced multithreading support in the **C++11 standard** with its std::thread class and related synchronization features like std::mutex, std::lock_guard, and std::condition_variable.
    - Example: std::thread t(func) to create a thread and t.join() to wait for the thread to finish.

Key Thread Library Operations:

- **Thread Creation**: Spawn new threads to run concurrently.
- **Thread Synchronization**: Ensure that threads work in harmony, especially when accessing shared resources.
- **Thread Termination**: Allow threads to safely terminate once their tasks are done.

Threading Issues

Threading issues arise when multiple threads interact and share resources. These issues can lead to unexpected behavior, performance degradation, and even system crashes if not handled properly.

Common Threading Issues:

- **Race Conditions**:
    - A race condition occurs when multiple threads access shared data or resources without proper synchronization, leading to unpredictable results.
    - Example: Two threads incrementing the same variable simultaneously could cause it to be updated incorrectly because the threads interfere with each other.
    - Solution: Use synchronization mechanisms like mutexes or locks to ensure that only one thread can access shared data at a time.
- **Deadlock**:
    - Deadlock happens when two or more threads are blocked forever, waiting for each other to release resources.
    - Example: Thread 1 holds Resource A and waits for Resource B, while Thread 2 holds Resource B and waits for Resource A. Neither thread can proceed.
    - Solution: Deadlock can be avoided by using strategies like lock ordering, timeout mechanisms, or deadlock detection algorithms.

- **Starvation**:
  - Starvation occurs when a thread is perpetually denied access to resources because other threads keep acquiring the resources first.
  - Example: In a priority-based scheduling system, a low-priority thread might never get scheduled to run if high-priority threads constantly preempt it.
  - Solution: Implement fair scheduling algorithms or priority inheritance to ensure that all threads get a chance to run.
- **Priority Inversion**:
  - Priority inversion is a situation where a lower-priority thread holds a resource needed by a higher-priority thread, preventing the high-priority thread from running.
  - Example: A medium-priority thread preempts a low-priority thread, which is holding a resource required by a high-priority thread.
  - Solution: Use priority inheritance or priority ceiling protocols to ensure that low-priority threads cannot block high-priority threads.
- **Concurrency Bugs**:
  - These are subtle bugs that arise from the interaction between threads, leading to unpredictable behavior.
  - Example: A thread might assume that a shared variable is always in a certain state, but concurrent updates by other threads cause the variable to be in an unexpected state.
  - Solution: Thorough testing, code review, and using tools like race detectors can help detect concurrency issues.

Process Scheduling

Process scheduling involves determining which process will run next. There are two types of scheduling:

- **Preemptive Scheduling**: The OS can interrupt a running process to assign CPU time to another process.
- **Non-preemptive Scheduling**: The OS allows the running process to finish before switching to another process.

Scheduling Criteria

The effectiveness of a scheduling algorithm can be judged based on several criteria:

- **CPU Utilization**: Maximizing the use of the CPU.
- **Throughput**: The number of processes completed per unit of time.

- **Turnaround Time**: The total time taken for a process to complete (from arrival to completion).
- **Waiting Time**: The total time a process spends waiting in the ready queue.
- **Response Time**: The time taken from submitting a request to receiving the first response.

Scheduling Algorithms

Process scheduling refers to the method by which the operating system decides which process or thread to execute next. This is important because modern systems often run multiple processes or threads concurrently, and efficient scheduling ensures that the system remains responsive and performs optimally.

Process Scheduling Algorithms:

1. **First-Come, First-Served (FCFS)**:
   - Processes are executed in the order they arrive in the ready queue.
   - **Problem**: This algorithm can lead to poor performance, especially if a long process arrives first (causing a **convoy effect** where shorter processes wait a long time).
2. **Shortest Job First (SJF)**:
   - The process with the shortest execution time is executed first.
   - **Problem**: It requires knowing the length of the next CPU burst, which is often difficult to predict.
   - **Preemptive version**: Shortest Remaining Time First (SRTF), which allows for preemption if a new process with a shorter burst time arrives.
3. **Round Robin (RR)**:
   - Each process is given a fixed time slice (quantum). After this time, it is preempted and placed back in the ready queue.
   - **Problem**: If the time slice is too large, it can behave like FCFS. If it's too small, there's high context-switching overhead.
4. **Priority Scheduling**:
   - Each process is assigned a priority, and the process with the highest priority is scheduled first.
   - **Problem**: Low-priority processes can suffer from **starvation**.
5. **Multilevel Queue Scheduling**:
   - Processes are grouped into multiple queues based on priority or other factors. Each queue may have its own scheduling algorithm.
   - **Problem**: Processes can get stuck in low-priority queues (starvation).

6. **Multilevel Feedback Queue Scheduling**:
   - This is an enhancement of the multilevel queue scheduling. Processes can move between queues based on their behavior (e.g., CPU-bound vs I/O-bound).
   - **Advantage**: It adapts dynamically to different types of processes.
7. **Fair Scheduling**:

- ◦ The goal of fair scheduling algorithms (like **Completely Fair Scheduler (CFS)** used in Linux) is to provide each process with a fair share of the CPU, ensuring no process is starved for CPU time.

Key Concepts in Process Scheduling:

- **Context Switching**: The act of saving the state of a running process and loading the state of another process. Frequent context switching can add overhead.
- **CPU Bound vs I/O Bound**:
  - **CPU-bound processes** require heavy computation and consume a lot of CPU time.
  - **I/O-bound processes** spend more time performing input/output operations than using the CPU.
- **Preemptive vs Non-Preemptive Scheduling**:
  - **Preemptive Scheduling**: The OS can interrupt a running process to give CPU time to another.
  - **Non-preemptive Scheduling**: The OS waits for a process to voluntarily release the CPU.

Multiple-Processor Scheduling

In modern operating systems, **multiple-processor scheduling** is a critical concept for managing systems with more than one processor or core. With multi-core and multiprocessor systems, the OS must decide how to allocate processes or threads across the available processors to maximize performance and system efficiency.

Key Challenges in Multiple-Processor Scheduling

- **Load Balancing**: Ensuring that the workload is evenly distributed across all processors so that no processor is overburdened while others are underutilized.
- **Processor Affinity**: Some systems may prefer or require that certain processes or threads run on the same processor to take advantage of data locality (known as **cache affinity**).
- **Synchronization**: Managing processes and threads that need to communicate or synchronize, as multiple processors can lead to complexities in managing shared resources.

Types of Multiple-Processor Scheduling

1. **Symmetric Multiprocessing (SMP)**
   - In **SMP**, all processors have equal access to memory and I/O resources.
   - Every processor runs the same operating system and has equal access to all processes.

    - The OS can schedule processes on any processor, allowing for **dynamic load balancing** across processors.

   Advantages:

    - Scalability: SMP systems can be easily expanded by adding more processors.
    - Fault tolerance: If one processor fails, others can continue handling the workload.

   Challenges:

    - **Synchronization**: Efficiently managing data consistency between processors can become complex.
    - **Cache Coherence**: Ensuring that the data in each processor's cache is synchronized with the main memory.

2. **Asymmetric Multiprocessing (AMP)**
    - In **AMP**, one processor (often called the **master processor**) controls the system, while the other processors (called **slave processors**) execute tasks as directed by the master.
    - The master processor typically schedules tasks, handles I/O, and manages resources.

   Advantages:

    - Simpler design: With only one processor managing the system, the complexity of the OS scheduling is reduced.
    - Cost-effective for certain workloads that do not need full parallelism.

   Challenges:

    - Limited scalability: Since only one processor is actively controlling the system, adding more processors does not improve performance as significantly as in SMP systems.
    - If the master processor fails, the system becomes non-functional.

3. **Non-Uniform Memory Access (NUMA)**
    - In **NUMA**, memory is divided into regions that are local to each processor, and each processor has faster access to its own local memory.
    - However, access to memory owned by other processors is slower. In this case, the OS must manage memory locality to optimize performance by ensuring that processes that frequently access the same memory are scheduled to run on the same processor.

   Advantages:

- Increased performance: By optimizing memory locality, NUMA systems can reduce memory access latency.
- Scalability: NUMA systems are better suited for large-scale multiprocessor systems.

Challenges:

- Programming complexity: NUMA systems require careful scheduling to ensure that threads access memory regions that are local to the processor they're running on.
- Overhead: Maintaining and managing memory consistency across processors can introduce additional overhead.

4. **Centralized vs. Distributed Scheduling**
   - **Centralized Scheduling**: A single processor or server is responsible for scheduling all processes on the system.
   - **Distributed Scheduling**: Each processor or node in a system is responsible for scheduling its own tasks. This is common in clusters or distributed systems.

   Advantages of Centralized Scheduling:

   - Easier to manage and monitor.
   - Can optimize performance by having full knowledge of all system resources.

   Challenges:

   - Single point of failure: If the central scheduler fails, the entire system can be affected.
   - Scalability: A single scheduler may become a bottleneck as the system grows.

Thread Scheduling

Thread Scheduling is the process by which the operating system decides which thread to execute next. Since threads are the smallest unit of execution within a process, thread scheduling is crucial in managing the concurrency of multithreaded applications. In systems with multiple processors, thread scheduling also involves distributing threads across processors to optimize performance.

Key Concepts in Thread Scheduling

1. **Preemptive vs. Non-Preemptive Thread Scheduling**
   - **Preemptive Scheduling**: The operating system can interrupt a running thread to give CPU time to another thread, typically based on priority or time slices. This allows for better responsiveness in systems with high concurrency, but requires careful synchronization to prevent issues like race conditions.

    - **Non-Preemptive Scheduling**: A thread runs until it voluntarily yields control (e.g., by completing its task or blocking on an I/O operation). This can lead to simpler synchronization but might reduce responsiveness or fairness.
2. **Thread Priority**
    - Threads may have priorities that determine their execution order. Higher-priority threads are scheduled before lower-priority threads.
    - **Priority Inversion**: A problem that arises when a higher-priority thread is blocked by a lower-priority thread holding a shared resource. This can be avoided by using **priority inheritance** or other advanced techniques.
3. **Thread Synchronization**
    - Since threads within a process share memory, they need mechanisms to synchronize their access to shared resources to prevent data corruption or inconsistent states. Common synchronization mechanisms include:
        - **Mutexes**: A mutual exclusion lock to ensure that only one thread can access a critical section at a time.
        - **Semaphores**: Counting mechanisms that control access to a shared resource.
        - **Condition Variables**: Used to allow threads to wait for certain conditions before proceeding.
4. **Thread Affinity**
    - **Thread Affinity** (or **Processor Affinity**) refers to binding a thread to a specific processor. This can improve performance by ensuring that a thread always runs on the same CPU, making better use of the processor's cache and reducing context switching overhead.
    - **Thread Migration**: In some systems, threads may migrate between processors for load balancing. This can help optimize overall system performance but can also introduce overhead due to cache invalidation and synchronization issues.
5. **Multilevel Queue Scheduling (for Threads)**
    - Similar to process scheduling, thread scheduling can use **multilevel queue scheduling**, where threads are divided into multiple queues based on their priority, type (I/O-bound vs. CPU-bound), or other factors. Each queue may use a different scheduling algorithm.
6. **Thread Scheduling in Multiprocessor Systems**
    - In multiprocessor systems, the OS must consider not only the process or thread's priority but also where to assign the thread. The goal is to balance load across processors while minimizing interprocessor communication and synchronization overhead.
    - **Load Balancing**: If one processor is under-utilized while others are overburdened, the scheduler may migrate threads to balance the load.

Scheduling Strategies for Multiple Processors

In systems with multiple processors or cores, the scheduling strategy must address a few key goals:

1. **Load Balancing**:
   - Distribute tasks evenly across all processors.
   - Prevent overloading one processor while leaving others idle.
   - For multiprocessor systems, efficient load balancing is crucial to fully utilizing available resources.
2. **Affinity Scheduling**:
   - For threads with **processor affinity**, ensure that threads are scheduled on processors that have already cached their data to improve performance.
   - This is particularly useful for workloads where threads frequently access the same data.
3. **Fairness**:
   - Ensure that all threads, regardless of the processor they run on, get a fair share of CPU time. This may involve using algorithms like **Round Robin** or **Fair Scheduling**.
4. **Scalability**:
   - Scheduling algorithms should be scalable, meaning they should efficiently handle an increasing number of processors without significant degradation in performance.