2.1 Processes (Expanded Version)

Process Concept

A **process** is a fundamental concept in operating systems (OS) and is defined as a program in execution. A program itself is a passive entity, such as an application stored on disk, while a process is an active entity that involves the execution of program instructions. When a program is executed, it becomes a process that progresses in a sequential manner. Each process has an associated **program counter**, which tracks the next instruction to be executed, and a **data section**, which includes global variables.

Operating systems like UNIX and Windows handle several types of processes, which may be classified as **batch jobs** in batch systems or **tasks** in time-shared systems. In time-shared systems, multiple processes may be running simultaneously, each requiring CPU time to execute. A process includes various components, including the program code (also known as the **text section**), the **current activity** such as the program counter, CPU registers, and process state, as well as the **stack** (which stores temporary data such as function parameters and local variables), and the **heap** (which stores dynamically allocated memory during runtime).

Processes go through multiple stages during their lifecycle. At any point in time, a process can be in one of the following states:

1. **New**: The process is being created.
2. **Running**: The process is actively being executed by the CPU.
3. **Waiting**: The process is waiting for some event, such as I/O completion.
4. **Ready**: The process is ready to be assigned to a processor.
5. **Terminated**: The process has finished execution.

One of the key distinctions between a **process** and a **program** is that a program is static and passive, whereas a process is dynamic and active. When multiple users execute the same program, each execution creates a distinct process, allowing concurrent execution of different instances of the same program.

Process Representation in Memory

When a process is loaded into memory, it is represented as a series of segments:

1. **Text Segment**: Contains the program code.
2. **Data Segment**: Stores global and static variables initialized before the program starts.
3. **Heap**: Used for dynamic memory allocation during program execution.
4. **Stack**: Stores function calls, return addresses, and local variables.

In addition, a process may also involve inter-process communication (IPC), which allows multiple processes to share data or resources. There are two main models of IPC: **shared memory** and **message passing**. Shared memory enables processes to share a region of memory, while message passing allows processes to exchange messages for synchronization and data exchange.

Process Creation, Suspension, and Termination

Processes are created by the operating system through a variety of system calls, the most common of which is the **fork()** system call in UNIX-like operating systems. When a process creates a new process, it becomes the **parent** of the newly created **child process**. The child process can execute concurrently with the parent process or wait for the parent to terminate.

The key steps involved in process creation include:

1. **Allocating Memory**: Memory space is allocated to the new process for its code, data, stack, and heap segments.
2. **Assigning a Process ID (PID)**: The OS assigns a unique identifier to the process.
3. **Setting Up the Process Control Block (PCB)**: A PCB is created and initialized with details such as the process state, program counter, CPU registers, memory pointers, and scheduling information.

Processes can also be suspended and resumed by the OS to manage system resources more efficiently. **Suspension** occurs when a process is temporarily halted by the OS, often due to the unavailability of resources such as memory or I/O devices. Suspended processes can be resumed once the necessary resources are available. Suspension is common in time-sharing systems where multiple processes are running concurrently, and it allows the OS to allocate CPU time fairly among processes.

**Process Termination** happens when a process completes its execution or is explicitly terminated by the OS or the parent process. A process may terminate for various reasons, including:

* **Normal Completion**: The process has finished its tasks and exits.
* **Errors**: The process encounters an error that prevents it from continuing.
* **Interruption by the Parent Process**: The parent process may terminate its children if the tasks assigned to the children are no longer needed, or if the child has exceeded its allocated resources.
* **Cascading Termination**: If a parent process terminates, the OS may automatically terminate all of its children.

In UNIX systems, the **exit()** system call is used to terminate a process. Once a process is terminated, its resources are deallocated, and it enters the **terminated** state, where the OS reclaims the memory and other resources it had used.

Process States: 2, 5, and 7-State Models

Processes can exist in multiple states during their lifecycle, which are represented in different process models.

1. **2-State Model**: In this simplified model, a process can be either in the **running** state, where it is being executed, or in the **blocked** state, where it is waiting for some event to occur.
2. **5-State Model**: The more common model includes the following states:
   * **New**: The process is being created.
   * **Ready**: The process is in memory and ready to run but is waiting for CPU time.
   * **Running**: The process is currently being executed by the CPU.
   * **Waiting**: The process is waiting for an event like an I/O completion.
   * **Terminated**: The process has finished execution.
3. **7-State Model**: This more detailed model includes additional states to handle processes that are suspended:
   * **New**: The process is being initialized.
   * **Ready**: The process is waiting for CPU time.
   * **Running**: The process is executing.
   * **Waiting**: The process is waiting for I/O or another event.
   * **Suspended-Ready**: The process has been swapped out of memory but is ready to be swapped back in and resume execution.
   * **Suspended-Blocked**: The process is swapped out of memory and is waiting for an event.
   * **Terminated**: The process has completed execution.

Each of these states allows the OS to manage processes more efficiently, ensuring that the CPU is used optimally and processes are executed in a fair and timely manner.

Process Control Block (PCB)

A **Process Control Block (PCB)** is a data structure used by the operating system to store information about a process. It contains all the necessary information to manage a process, including:

* **Process State**: Whether the process is new, ready, running, waiting, or terminated.
* **Program Counter**: The address of the next instruction to be executed.
* **CPU Registers**: The current values of the CPU registers for the process.
* **CPU Scheduling Information**: Priority of the process, pointers to scheduling queues, and other scheduling parameters.
* **Memory Management Information**: Information about memory allocated to the process, such as page tables, segment tables, and base/limit registers.
* **I/O Status Information**: List of I/O devices allocated to the process, as well as information about files being accessed by the process.
* **Accounting Information**: Information related to the process's CPU usage, execution time, and other resource consumption metrics.

The PCB is essential for **context switching**, where the CPU switches from one process to another. During a context switch, the OS saves the current process's state (from its PCB) and loads the state of the next process to be executed. This allows processes to resume execution from where they left off without losing any information.

2.2 Threads (Expanded Version)

Thread Concept

A **thread** is the smallest unit of CPU execution and is a key component in enabling concurrent execution of tasks within a process. Threads are also known as **lightweight processes**, and they allow a single process to perform multiple tasks simultaneously by executing separate threads of control. A thread shares resources such as memory, file descriptors, and other OS resources with other threads in the same process, but has its own stack, program counter, and set of registers.

A single program can contain multiple threads that run in parallel, improving responsiveness and resource sharing. For example, in a web browser, one thread may handle user input, another may handle rendering of web pages, and yet another may handle downloading resources from the internet.

In modern operating systems, **multithreading** is an essential feature for improving the performance of multi-core processors by enabling multiple tasks to execute concurrently. Multithreading also enhances responsiveness by allowing the main thread of an application to continue executing while background tasks, such as file loading or network requests, are handled by other threads.

Benefits of Threads

1. **Responsiveness**: Threads allow programs to continue executing while other tasks are performed in the background. For example, an interactive application can remain responsive to user input while processing data in the background.
2. **Resource Sharing**: Since threads within the same process share the same memory and resources, it is easier to share data between threads without the need for complex communication mechanisms like inter-process communication (IPC).
3. **Economy**: Creating and managing threads is more efficient than creating and managing processes. Threads consume fewer resources and require less overhead because they share the process's memory and other resources.
4. **Scalability**: In multi-core processors, threads allow programs to take advantage of multiple CPU cores by running threads in parallel across different cores, improving overall throughput.

Multithreading Models

Multithreading models define the relationship between user-level threads and kernel-level threads. Different systems implement multithreading in different ways depending on how they handle this relationship. The three primary multithreading models are:

1. **Many-to-One Model**: In this model, multiple user-level threads are mapped to a single kernel thread. This model is efficient because thread management is done at the user level without the need for kernel intervention. However, it suffers from a major limitation: if one thread makes a blocking system call, all threads in the process are blocked since the kernel cannot differentiate between them. As a result, the many-to-one model does not utilize multiple processors effectively in a multiprocessor environment.
2. **One-to-One Model**: In the one-to-one model, each user-level thread maps to its own kernel thread. This allows for better concurrency, as each thread can execute independently on a separate CPU core. Blocking calls made by one thread do not affect other threads in the process. However, the one-to-one model has a higher overhead because the creation of each user thread requires the creation of a corresponding kernel thread. Operating systems like **Windows** and **Linux** use this model.
3. **Many-to-Many Model**: This model maps many user-level threads to a smaller or equal number of kernel threads. It provides a compromise between the many-to-one and one-to-one models by allowing user threads to be managed efficiently while also allowing kernel threads to execute user threads in parallel. The OS dynamically allocates kernel threads based on the number of user threads and available resources. This model is used in operating systems like **Solaris**.

Thread Implementations

Threads can be implemented in two ways: **user-level threads** and **kernel-level threads**.

1. **User-Level Threads**: User-level threads are managed entirely by the user-space thread library. The OS is unaware of the existence of user threads, and all thread management (creation, switching, synchronization) is handled by the thread library. This makes user-level threads more efficient and faster to manage because they do not require kernel-mode privileges. However, since the kernel is unaware of user threads, it treats the entire process as a single thread, which means that blocking system calls block all user threads in the process.
2. **Kernel-Level Threads**: Kernel-level threads are managed directly by the operating system kernel. Each thread is treated as a separate entity, and the OS schedules and executes threads just like processes. Kernel threads can be preempted, and blocking system calls only block the thread making the call, allowing other threads in the process to continue execution. Kernel-level threads offer better concurrency but are slower to create and manage compared to user-level threads.

Symmetric Multiprocessing (SMP)

**Symmetric Multiprocessing (SMP)** is a type of multiprocessing architecture in which multiple processors share a single memory space and I/O system, allowing them to work together to execute multiple tasks in parallel. In an SMP system, each processor runs its own thread of control, but all processors have equal access to system resources.

In a multithreaded SMP system, multiple threads can run concurrently on different processors, improving overall system performance. SMP systems are widely used in modern multi-core processors, where each core acts as an independent processor.

The OS must provide effective thread scheduling and synchronization mechanisms to ensure that threads are distributed efficiently across processors and that resources are shared without conflicts.

2.3 Uniprocessor Scheduling (Expanded Version)

Scheduling Criteria

**Uniprocessor scheduling** refers to the way in which the OS allocates CPU time to various processes on a single processor system. The scheduling criteria determine the goals of the scheduling algorithm and define the measures by which the system's performance can be evaluated. Important scheduling criteria include:

1. **CPU Utilization**: The percentage of time that the CPU is actively executing processes. The goal is to maximize CPU utilization to ensure that the processor is not idle for extended periods.
2. **Throughput**: The number of processes that complete execution in a given period of time. A higher throughput indicates better system performance.
3. **Turnaround Time**: The total time taken to execute a process from the time it is submitted to the time it completes. This includes both waiting time and execution time.
4. **Waiting Time**: The total time a process spends waiting in the ready queue before it gets executed. Reducing waiting time improves responsiveness.
5. **Response Time**: The time taken for the system to respond to a user input or process request. It is especially important in interactive systems where quick feedback is expected.
6. **Fairness**: Ensuring that all processes receive their fair share of CPU time and that no process is starved of resources.

Types of Scheduling

Scheduling can be classified into different types based on when the OS makes scheduling decisions:

1. **Preemptive Scheduling**: In preemptive scheduling, the OS can interrupt a running process to allocate the CPU to another process. This ensures that processes with higher priority or shorter execution times get CPU time when needed. Preemptive scheduling is ideal for systems that require fast response times or prioritize certain tasks over others.
2. **Non-Preemptive Scheduling**: Once a process starts executing in non-preemptive scheduling, it continues running until it either finishes or voluntarily relinquishes the CPU. Non-preemptive scheduling is simple to implement but can lead to issues like long wait times for other processes (e.g., if a long-running process is occupying the CPU).
3. **Long-Term Scheduling**: Long-term schedulers, also known as job schedulers, decide which processes should be admitted into the system for execution. The long-term scheduler controls the degree of **multiprogramming**, which is the number of processes in memory at a given time. Long-term scheduling is invoked infrequently and works to balance the system's workload by selecting appropriate jobs for execution.
4. **Medium-Term Scheduling**: Medium-term schedulers are responsible for swapping processes in and out of memory to manage the **multiprogramming level**. Processes that are not currently needed may be temporarily swapped out of memory to free up space for active processes.
5. **Short-Term Scheduling**: Also known as the CPU scheduler, the short-term scheduler decides which process will be executed next by the CPU. This scheduler operates frequently (e.g., milliseconds) and must make fast decisions to ensure the CPU remains fully utilized.

Scheduling Algorithms

Several scheduling algorithms exist to determine which process gets executed next. These algorithms balance system efficiency, fairness, and responsiveness:

1. **First-Come, First-Served (FCFS)**: Processes are executed in the order they arrive. While FCFS is simple to implement, it can lead to poor performance due to the **convoy effect**, where short processes get stuck behind long ones.
2. **Shortest Job First (SJF)**: Processes with the shortest CPU burst time are executed first. This algorithm minimizes the average waiting time but requires knowledge of future burst times, which may not always be available.
3. **Shortest Remaining Time First (SRTF)**: A preemptive version of SJF, where the process with the shortest remaining time is executed next. SRTF provides better response times than SJF but is more complex to implement.
4. **Round Robin (RR)**: Each process gets a fixed **time slice** (quantum) to execute. When the time slice expires, the process is preempted, and the next process is executed. Round Robin is commonly used in time-sharing systems and ensures fairness by giving each process an equal opportunity to run.
5. **Priority Scheduling**: Processes are assigned priorities, and the process with the highest priority gets the CPU. Priority scheduling can be either preemptive or non-preemptive. However, it can lead to **starvation** of low-priority processes if high-priority processes dominate CPU time. To mitigate this, **aging** techniques may be used, where the priority of a process increases the longer it waits.

2.4 Multiprocessor Scheduling (Expanded Version)

Granularity

**Granularity** refers to the size of tasks in a multiprocessor system. It can be defined as the ratio of computation to communication. Granularity affects the scheduling strategy in multiprocessor systems, with fine-grained parallelism involving frequent communication between tasks and coarse-grained parallelism requiring fewer interactions.

1. **Fine-Grained Parallelism**: In fine-grained systems, tasks are small, and processes need to communicate frequently. This approach increases overhead due to the need for constant coordination between processors.
2. **Coarse-Grained Parallelism**: Coarse-grained systems have fewer, larger tasks with infrequent communication, reducing overhead and making them easier to schedule.

Design Issues in Multiprocessor Scheduling

Multiprocessor scheduling presents unique challenges that do not exist in uniprocessor systems, such as **load balancing**, **processor affinity**, and **communication overhead**. The primary issues include:

1. **Load Balancing**: In multiprocessor systems, the OS must ensure that the workload is evenly distributed across all processors to prevent some processors from being overloaded while others remain idle.
2. **Processor Affinity**: Processor affinity refers to the practice of keeping a process on the same processor to reduce cache misses. This can be **soft affinity**, where the OS tries to keep processes on the same processor, or **hard affinity**, where processes are explicitly bound to a specific processor.
3. **Communication Overhead**: Communication between processes or threads running on different processors introduces overhead due to synchronization mechanisms. The OS must minimize this overhead while ensuring that tasks execute efficiently.

Process Scheduling in Multiprocessor Systems

In multiprocessor systems, the OS must decide how to allocate processes to multiple CPUs. There are two primary approaches to process scheduling:

1. **Asymmetric Multiprocessing**: In asymmetric systems, one processor is dedicated to handling all system-level tasks (e.g., I/O operations, scheduling), while other processors execute user processes. This simplifies scheduling but may limit the system's scalability.
2. **Symmetric Multiprocessing (SMP)**: In SMP systems, all processors are treated equally, and the OS schedules processes across all available CPUs. This allows for better load balancing and scalability, as multiple processors can work together to handle system and user tasks.

Thread Scheduling

In multithreaded multiprocessor systems, thread scheduling becomes more complex because threads must be distributed across processors efficiently. Approaches to thread scheduling include:

1. **Load Sharing**: The OS distributes threads evenly across all processors, ensuring that no processor is idle while others are overloaded.
2. **Gang Scheduling**: In gang scheduling, all threads in a process are scheduled to run together on separate processors. This ensures that threads can communicate and synchronize effectively, reducing communication overhead.
3. **Dedicated Processor Assignment**: In this approach, each thread is assigned to a specific processor, and the OS ensures that the thread always runs on that processor. This minimizes the overhead associated with thread migration.

Real-Time Scheduling in Multiprocessor Systems

Real-time systems have strict timing requirements, and scheduling in multiprocessor real-time systems must ensure that tasks meet their deadlines. There are two types of real-time scheduling:

1. **Hard Real-Time Systems**: In hard real-time systems, missing a deadline can result in catastrophic system failure. The OS must ensure that all critical tasks are completed within their time constraints.
2. **Soft Real-Time Systems**: In soft real-time systems, missing a deadline results in degraded performance but not system failure. The OS prioritizes real-time tasks but allows for some flexibility in meeting deadlines.

2.5 Process Security (Expanded Version)

Process Security Overview

**Process security** is a critical aspect of operating systems that involves protecting processes from unauthorized access, interference, and manipulation. The OS must ensure that processes are isolated from each other and that sensitive data within processes is protected from external threats. In modern computing environments, process security is essential for maintaining system integrity, confidentiality, and availability.

Threats to Process Security

There are several potential threats to process security, including:

1. **Unauthorized Access**: Malicious users or processes may attempt to gain unauthorized access to the data and resources of other processes. This can lead to data breaches, system compromise, and loss of sensitive information.
2. **Inter-Process Interference**: One process may attempt to interfere with the execution of another process, either by modifying its data or by consuming excessive system resources, causing denial of service.
3. **Privilege Escalation**: A process with limited privileges may attempt to exploit system vulnerabilities to gain elevated privileges and access restricted resources.
4. **Code Injection**: Attackers may attempt to inject malicious code into a process's memory space, allowing them to execute unauthorized instructions within the context of the process.

Process Isolation and Access Control

One of the key mechanisms for securing processes is **process isolation**, which ensures that processes cannot directly access each other's memory or resources. Each process operates in its own **address space**, and the OS enforces access controls to prevent unauthorized interactions between processes.

Access control mechanisms include:

1. **User and Group Permissions**: The OS assigns user and group permissions to processes, determining which resources (e.g., files, devices) a process can access. For example, in UNIX-based systems, each file has read, write, and execute permissions that are set for the file owner, group, and others.
2. **Role-Based Access Control (RBAC)**: In RBAC, processes are assigned specific roles based on their function, and access to resources is granted based on the process's role. This provides a more flexible and scalable approach to managing permissions.
3. **Capabilities and Access Control Lists (ACLs)**: In some systems, access to resources is managed through capabilities or ACLs, which define the specific rights a process has over an object (e.g., read, write, execute).

Memory Protection Mechanisms

Modern operating systems use several techniques to protect the memory space of processes from unauthorized access:

1. **Virtual Memory**: The OS uses virtual memory to create a separate address space for each process. This ensures that processes cannot access the memory of other processes directly, providing a layer of isolation.
2. **Memory Segmentation**: Memory segmentation divides a process's memory into segments (e.g., code, data, stack), and the OS controls access to these segments based on the process's privilege level.
3. **Paging**: In systems that use paging, memory is divided into fixed-size pages, and the OS controls access to these pages. The **page table** maps virtual addresses to physical addresses, and access to pages can be restricted based on the process's permissions.
4. **Stack Protection**: To prevent **buffer overflow attacks**, the OS can implement stack protection mechanisms such as **stack canaries**, which detect attempts to overwrite the return address of a function.

Security in Inter-Process Communication (IPC)

Inter-process communication (IPC) mechanisms, such as shared memory, message passing, and pipes, must be secured to prevent unauthorized processes from eavesdropping on or modifying communication between processes. The OS ensures that only authorized processes can establish communication channels and that data exchanged between processes is protected.

Process Monitoring and Auditing

The OS plays a key role in monitoring and auditing process activity to detect and respond to potential security threats. Monitoring involves tracking process behavior, such as system calls, resource usage, and access to sensitive data. If abnormal or suspicious activity is detected, the OS can take corrective actions, such as terminating the process or restricting its access to resources.

Auditing involves maintaining logs of process activity, which can be reviewed by system administrators to identify security incidents and ensure compliance with security policies.

Protection Against Malicious Code

Operating systems implement various security measures to protect processes from **malware** and **exploits**. These include:

1. **Code Signing**: The OS verifies the integrity and authenticity of executable code by checking digital signatures. Only code signed by trusted entities is allowed to execute.
2. **Address Space Layout Randomization (ASLR)**: ASLR randomizes the memory addresses used by processes, making it more difficult for attackers to predict the location of code and data in memory. This helps mitigate memory-based attacks like **buffer overflows**.
3. **Data Execution Prevention (DEP)**: DEP prevents code from being executed in regions of memory that are marked as non-executable, such as the stack or heap. This reduces the risk of code injection attacks.
4. **Sandboxing**: Some operating systems use sandboxing techniques to isolate potentially harmful processes from the rest of the system. A sandboxed process has limited access to system resources and is restricted from interacting with other processes.