**Chapter-3**

**Process Concept**

1. \*\*Process\*\*: In the context of operating systems, a process is a program in execution. It's not just the program code but also includes the current state, like the value of the program counter and processor registers. A process also includes various sections like the stack, data, and heap, which store different types of data. Multiple processes can run concurrently on a computer, and each process is an independent entity with its own resources.

2. \*\*Process State\*\*: A process can be in various states during its lifetime, such as "New" (when it's being created), "Running" (when it's actively executing instructions), "Waiting" (when it's waiting for an event like I/O), "Ready" (when it's ready to run but waiting for CPU time), and "Terminated" (when it has finished execution). The specific states and their names may vary across operating systems.

3. \*\*Process Control Block (PCB)\*\*: A PCB is a data structure that the operating system uses to manage and keep track of information related to each process. It contains details like the process state, program counter, CPU registers, scheduling information, memory management information, accounting information, and I/O status. The PCB serves as a repository for all information that may vary from one process to another.

4. \*\*Threads\*\*: A thread is the smallest unit of execution within a process. While a process traditionally represents a single program with its own memory space, threads within a process share the same memory space. This allows multiple threads within the same process to run concurrently. Threads are beneficial for tasks that can be parallelized, and they are particularly useful on multi-core processors. Modern operating systems support multithreading, and the PCB may include information for each thread within a process.

5. \*\*Linux Process Representation\*\*: In Linux, the process control block is represented by a C structure called `task\_struct`. This structure contains information about the process, including its state, scheduling information, parent-child relationships, open files, and memory management details. The kernel maintains a pointer called `current` to the process currently executing on the system.

In summary, processes in operating systems are the execution units of programs, and each process has its own state, resources, and control block. Processes can have multiple threads for concurrent execution, and Linux represents processes using the `task\_struct` structure, with the `current` pointer pointing to the currently executing process.

**Scheduling**

Understanding the concepts of scheduler, scheduling queues, and context switching is crucial to grasp how processes are managed in an operating system. Let's break down these concepts step by step:

1. \*\*Scheduler\*\*:

- The scheduler is a crucial component of the operating system responsible for determining which process gets access to the CPU (Central Processing Unit) and for how long.

- Its primary goal is to maximize CPU utilization, ensure fairness, and minimize response time and waiting time for processes.

- There are various scheduling algorithms, such as First-Come-First-Serve (FCFS), Round Robin, Priority Scheduling, and more. Each has its advantages and disadvantages, making them suitable for different use cases.

2. \*\*Scheduling Queues\*\*:

- Scheduling queues are data structures used by the scheduler to organize and manage processes based on their current state and priority.

- Commonly, you'll find the following types of scheduling queues:

- \*\*Ready Queue\*\*: Processes that are ready to execute but waiting for CPU time are placed in the ready queue.

- \*\*Waiting Queue\*\*: Processes that are waiting for some event, such as I/O completion or a timer, are placed in the waiting queue.

- \*\*Priority Queue\*\*: Processes can also be organized based on their priority levels, where higher-priority processes get preference in execution.

3. \*\*Context Switching\*\*:

- Context switching is the process of saving the state of a currently executing process (saving its program counter, registers, and other necessary information) and loading the saved state of another process into the CPU.

- It occurs when the scheduler decides to switch the CPU from one process to another.

- Context switching is essential for multitasking and ensuring that multiple processes can run on a single CPU.

- It introduces some overhead due to the time required to save and restore process states, but it's necessary for process isolation and fairness.

**Operations on Processes**

\*\*Process Creation:\*\*

1. \*\*Parent and Child Processes:\*\* In an operating system, a process can create new processes. The creating process is called the \*\*parent process\*\*, and the new processes it creates are called \*\*child processes\*\*. Each child process may, in turn, create more processes, forming a tree-like structure of processes.

2. \*\*Unique Identifiers (PID):\*\* Most operating systems, including UNIX, Linux, and Windows, identify processes using a unique process identifier (PID), typically an integer. The PID uniquely identifies each process in the system and can be used to access various attributes of a process within the kernel.

3. \*\*Process Tree:\*\* Processes are organized in a hierarchical manner. For example, in Linux, the `init` process (with PID 1) serves as the root parent process for all user processes. It can create various user processes, which can further create child processes, forming a process tree.

4. \*\*Process Creation Mechanisms:\*\* When a process creates a new process, it may provide certain resources and initialization data to the child process. This includes CPU time, memory, open files, and more. There are two primary mechanisms for process creation:

- \*\*Fork-Exec Model (UNIX/Linux):\*\* In this model, a new process is created using the `fork()` system call. The child process is initially a copy of the parent process, including its code and data. Then, the child can use the `exec()` system call to load a new program into its address space, replacing the parent's code with a new one.

- \*\*CreateProcess Model (Windows):\*\* In Windows, processes are created using the `CreateProcess()` function. This function allows a parent process to specify various attributes of the child process, such as the application to run and its parameters. The child process starts with a clean slate, loading the specified program.

**\*\*Process Termination:\*\***

1. \*\*Normal Termination:\*\* A process can terminate when it finishes executing its final statement and invokes the `exit()` system call. At this point, the process may return a status value to its parent process via the `wait()` system call. All the resources allocated to the process, such as memory and open files, are deallocated by the operating system.

2. \*\*Forced Termination:\*\* In some cases, a process may be forcibly terminated by another process, typically its parent. This can occur when a child process misbehaves or exceeds resource limits. The parent can use appropriate system calls (e.g., `kill` in UNIX/Linux or `TerminateProcess()` in Windows) to terminate the child.

3. \*\*Orphan Processes:\*\* If a parent process terminates without waiting for its child processes to finish, these child processes become orphaned. Orphan processes are assigned a new parent, usually the `init` process (or similar), which will wait for them to complete and collect their exit statuses.

In summary, process creation involves the creation of new processes by parent processes, while process termination refers to the end of a process's execution. Processes can terminate normally or be forcefully terminated by their parent or other processes. Orphan processes are handled by assigning them a new parent process. These mechanisms are fundamental in managing the execution of programs in an operating system.

Here's a simplified overview of how these concepts work together:

- Processes in the "Ready Queue" are those that are eligible to run and are waiting for CPU time.

- The scheduler selects a process from the "Ready Queue" based on its scheduling algorithm and dispatches it for execution on the CPU.

- When it's time to switch to another process (due to time quantum expiration, priority change, or any other reason), a context switch occurs.

- During a context switch, the state of the currently running process is saved, and the next process to run is selected.

- The saved state of the new process is then restored, allowing it to continue execution from where it left off.

This cycle repeats, with processes moving between the ready queue, CPU execution, and waiting queue as needed, all managed by the scheduler and facilitated by context switching.

The specific implementation details and data structures used for these components can vary depending on the operating system and its design.

**Interprocess communication (IPC)**

\*\*Interprocess Communication (IPC):\*\* IPC refers to the mechanisms and techniques used by processes in an operating system to communicate and synchronize with each other. Processes may need to cooperate and exchange information for various reasons, such as information sharing, computation speedup, modularity, and convenience.

\*\*Independent vs. Cooperating Processes:\*\*

- Independent processes are those that do not affect or be affected by other processes in the system. They do not share data with other processes.

- Cooperating processes are those that can affect or be affected by other processes in the system. They share data or communicate with other processes.

\*\*Reasons for Cooperating Processes:\*\*

1. \*\*Information Sharing:\*\* Multiple users or processes may need to access the same data or resources simultaneously.

2. \*\*Computation Speedup:\*\* Dividing a task into subtasks that run in parallel can speed up computation, especially on multi-core systems.

3. \*\*Modularity:\*\* System functions can be divided into separate processes or threads, enhancing modularity and maintainability.

4. \*\*Convenience:\*\* Even individual users may work on multiple tasks simultaneously.

\*\*Shared-Memory Model vs. Message-Passing Model:\*\*

- \*\*Shared-Memory Model:\*\* In this model, cooperating processes share a region of memory. They can exchange information by reading and writing data to this shared memory. This model is suitable for larger data exchanges and requires synchronization mechanisms to avoid conflicts when multiple processes access shared memory concurrently.

- \*\*Message-Passing Model:\*\* In this model, processes communicate by sending and receiving messages. Each process has its own address space, and communication happens through messages exchanged between processes. This model is well-suited for smaller data exchanges and simplifies the synchronization process.

\*\*Shared-Memory Systems:\*\*

- Shared memory allows processes to share a portion of memory, enabling them to read and write data in that shared region.

- Processes using shared memory must be careful about synchronization to prevent conflicts when multiple processes access shared data simultaneously.

- It can be suitable for tasks like producer-consumer problems, where multiple processes need to exchange data.

\*\*Message-Passing Systems:\*\*

- In message passing, processes communicate by sending and receiving messages. Each process has its own memory space.

- Message-passing systems are often used in distributed environments where processes can be on different computers.

- This approach simplifies synchronization, as messages provide clear boundaries for communication.

\*\*Synchronization:\*\*

- Synchronization is crucial in both shared-memory and message-passing systems to ensure that processes communicate effectively without data corruption or race conditions.

- It involves mechanisms like locks, semaphores, and barriers to coordinate the execution of processes.

- Proper synchronization ensures that processes behave correctly and safely when accessing shared resources or communicating via messages.

Overall, the choice between shared memory and message passing depends on the specific requirements of the application and the characteristics of the system. Both models have their advantages and trade-offs, and the selection should be based on factors like data size, communication patterns, and system architecture.

The passage you provided explains how the Windows operating system uses a message-passing mechanism called the Advanced Local Procedure Call (ALPC) facility for interprocess communication (IPC) between processes running on the same machine. ALPC is designed to provide efficient and secure communication between different subsystems within the Windows OS.

Here are the key points to understand about ALPC and how it works:

1. \*\*Modularity in Windows:\*\* Windows is designed with modularity in mind, allowing it to support multiple operating environments or subsystems. Application programs interact with these subsystems using a message-passing mechanism.

**Communication in Client–Server Systems**

\*\*Sockets:\*\*

- A socket is defined as an endpoint for communication between two processes over a network.

- In general, sockets follow a client-server architecture where a server listens on a specific port for incoming client requests.

- Servers providing specific services, like telnet, FTP, or HTTP, listen on well-known ports (e.g., telnet server on port 23, FTP server on port 21, HTTP server on port 80).

- When a client initiates a connection, it is assigned an arbitrary port number greater than 1024 by its host computer.

- A connection consists of a pair of sockets, one on the client side and one on the server side, identified by IP addresses and port numbers.

- All connections must be unique, ensuring that each connection consists of a unique pair of sockets.

\*\*Remote Procedure Calls (RPCs):\*\*

- RPC is a communication paradigm used for remote services between processes on different systems.

- It hides the complexity of communication and allows clients to invoke procedures on a remote server as if they were local.

- Messages in RPC are well-structured and contain an identifier specifying the function to execute and parameters to pass.

- RPCs involve the use of stubs on both the client and server sides. Stubs help marshal and unmarshal parameters and invoke remote procedures.

- To address differences in data representation between systems, RPC systems often use machine-independent representations like XDR (external data representation).

- RPCs can encounter issues like network errors, duplication of calls, and ensuring that messages are acted on exactly once.

- To guarantee at-most-once semantics, messages may include timestamps, and the server keeps a history of processed messages.

- For exactly-once semantics, acknowledgment (ACK) messages are used, and the client resends RPC calls until it receives an ACK.

- RPC systems also need a way to bind client and server processes dynamically. This can be done through fixed port numbers or a rendezvous (matchmaker) mechanism, where a central daemon helps clients find the correct port for a specific RPC.

In summary, sockets provide a low-level mechanism for communication between processes over a network, while RPCs offer a higher-level, more structured way to invoke remote procedures on other systems, making distributed computing more accessible and abstracting many of the complexities of network communication.

**Remote Procedure Calls (RPCs)**

1. \*\*RPC Paradigm\*\*: RPC is a mechanism that abstracts the procedure-call mechanism for use between systems with network connections. It allows a client to invoke a procedure on a remote server as if it were a local procedure call.

2. \*\*Structured Messages\*\*: Unlike low-level IPC (Inter-Process Communication) messages, RPC messages are well-structured. Each message is addressed to an RPC daemon listening on a specific port on the remote system. The message contains an identifier specifying the function to execute and the parameters to pass to that function.

3. \*\*Ports\*\*: Ports are used to differentiate between different network services on a system. A server may have multiple ports, each associated with a specific RPC service. Clients send RPC messages to the appropriate port to access the desired service.

4. \*\*Stub\*\*: To abstract the details of communication, RPC systems provide a stub on the client side for each remote procedure. The stub marshals the parameters, converts them into a format suitable for transmission over the network, and sends the message to the server. A similar stub on the server side receives the message and invokes the procedure.

5. \*\*Data Representation\*\*: RPC systems address differences in data representation between client and server machines. They often use machine-independent representations like External Data Representation (XDR) to ensure data consistency during marshalling and unmarshalling.

6. \*\*Semantic Guarantees\*\*: RPCs can fail, or they can be duplicated and executed multiple times due to network errors. RPC systems implement semantic guarantees like "at most once" and "exactly once" to ensure the reliability of remote procedure calls.

- \*\*At Most Once\*\*: Messages are timestamped, and the server keeps a history to detect and ignore duplicate messages, ensuring a call is executed at most once.

- \*\*Exactly Once\*\*: The server acknowledges receipt and execution of an RPC call, and the client resends the call until it receives the acknowledgment.

7. \*\*Dynamic Binding\*\*: RPC calls can use predetermined fixed port addresses or dynamic binding through a rendezvous mechanism. A rendezvous daemon on a fixed RPC port helps clients discover the port number of the RPC they need to execute dynamically.

8. \*\*Distributed File Systems\*\*: RPCs are useful in implementing distributed file systems. Clients can send RPC messages to a server's distributed file system port to perform file-related operations like read, write, rename, delete, or obtain status.

In summary, RPC is a high-level communication paradigm that allows remote procedure invocation in a client-server architecture. It uses structured messages, supports dynamic binding, handles data representation differences, and provides semantic guarantees to ensure reliable communication between distributed systems. It's commonly used in various distributed systems, including distributed file systems.