ENDSEM SYSTEM PROGRAMMING OPERATING SYSTEM
UNIT – 4

Q.1] Explain the following types of Schedulers. i) Short Term ii) Long Term iii) Medium Term

ANS: Schedulers in operating systems play a crucial role in managing the execution of processes. They determine which processes should run, when they should run, and for how long. Schedulers are categorized into three main types: short-term, long-term, and medium-term schedulers.

1. Short-Term Scheduler (CPU Scheduler):

- **Function: The short-term scheduler is responsible for selecting which process from the ready queue will be executed next and allocating the CPU to that process.**
- **Frequency: It operates very frequently, typically on the order of milliseconds or microseconds.**
- **Objective: The main goal is to optimize CPU utilization, throughput, and response time. It ensures fairness and prevents starvation by giving each process a fair share of the CPU.**
- **Algorithm: Common algorithms include First-Come-First-Serve (FCFS), Shortest Job Next (SJN) or Shortest Job First (SJF), Round Robin, Priority Scheduling, and Multilevel Queue Scheduling.**

2. Long-Term Scheduler (Job Scheduler):

- **Function: The long-term scheduler is responsible for selecting which processes from the job pool (a pool of processes that are waiting to be brought into memory for execution) will be loaded into memory.**
- **Frequency: It operates much less frequently than the short-term scheduler, potentially only a few times per minute.**
- **Objective: The primary goal is to control the degree of multiprogramming, i.e., the number of processes residing in the main memory. It determines when to admit new processes to the system.**
- **Algorithm: The long-term scheduler doesn't usually involve sophisticated algorithms. Instead, it may use admission control policies based on system resources, memory availability, and other factors.**

3. Medium-Term Scheduler (Swapper):

- **Function: The medium-term scheduler is responsible for deciding which processes should be swapped in and out of the main memory.**
- **Frequency: It operates less frequently than the short-term scheduler but more frequently than the long-term scheduler. It is invoked when processes need to be swapped in or out of the main memory.**
- **Objective: The main goal is to manage the degree of multiprogramming and optimize system performance by deciding which processes should be in the main memory at any given time.**
- **Algorithm: It may use algorithms similar to those of the long-term scheduler but applied to processes that are already in the main memory. For example, it may use page replacement algorithms in virtual memory systems.**

Q.2] Explain seven state process model with diagram? Also explain difference between Five state process model & Seven state process model?

ANS: The seven-state process model is a representation of the various states a process can go through in an operating system. It provides a more detailed view of the lifecycle of a process compared to the simpler five-state process model. The states in the seven-state model are:

1. **New:**
    - **The process is in the new state when it is being created. At this stage, essential data structures are set up, and the process is being initialized.**
2. **Ready:**
    - **After creation, the process moves to the ready state. It means that the process is loaded into the main memory and is waiting to be assigned to a processor for execution.**
3. **Running:**
    - **In the running state, the process is actively being executed on the CPU. Only one process can be in the running state at a time per CPU core.**
4. **Waiting:**
    - **When a process is waiting for an event (such as I/O completion or a signal), it moves to the waiting state. It is temporarily stopped, and another process may be assigned the CPU.**
5. **Blocked:**
    - **Similar to the waiting state, the blocked state represents a process that cannot proceed until a specific event occurs. However, a process in the blocked state is waiting for a resource that is currently held by another process.**
6. **Terminated:**
    - **The process moves to the terminated state when it has finished execution. At this stage, system resources allocated to the process are released.**
7. **Suspended:**
    - **A process may be moved to the suspended state if it needs to be temporarily removed from main memory to free up space. It can later be moved back to the ready or blocked state.**

Seven-State Process Model Diagram:

```
+---------+     +---------+    +------------+
|   New |    |Ready  |    | Running |
+----+----+    +----+----+    +----+------+
       |               |               |
       |               |               |
+----v------+    +--v--+-----+    +--v--------------+
| Waiting |    | Blocked |    | Terminated |
+------------+   +------------+    +-----------------+
       |
+----v-----------+
| Suspended |
+----------------+
```

Difference between Five-State and Seven-State Process Model:

1. **States:**
   - **Five-State Model: It includes the states: New, Ready, Running, Waiting, and Terminated.**
   - **Seven-State Model: It includes the states: New, Ready, Running, Waiting, Blocked, Terminated, and Suspended.**
2. **Detail:**
   - **Five-State Model: It provides a basic representation of the process lifecycle with fewer intermediate states.**
   - **Seven-State Model: It offers a more detailed view by introducing additional states such as Blocked and Suspended.**
3. **Blocked State:**
   - **Five-State Model: It doesn't explicitly have a Blocked state. Waiting encompasses both waiting for resources and waiting for events.**
   - **Seven-State Model: It introduces a separate Blocked state to distinguish between waiting for resources and waiting for events.**
4. **Suspended State:**
   - **Five-State Model: It doesn't have a suspended state.**
   - **Seven-State Model: It introduces a Suspended state, indicating a process that has been temporarily removed from main memory.**

Q.3] Draw Gantt chart and calculate Avg. turnaround time, Avg. waiting time for the following process using SJF non preemptive and round robin with time quantum 0.5 Unit

Process	Burst Time	Arrival Time
P1	2	10
P2	1	10
P3	1	11
P4	1	12

ANS: To calculate the average turnaround time and average waiting time, we'll first create Gantt charts for both Shortest Job First (SJF) non-preemptive scheduling and Round Robin with a time quantum of 0.5 units.

Let's start with SJF:

Gantt Chart for SJF:

Time	P2	P3	P4	P1
10	P2	P3	P4	P1
11	P2	P3	P4	P1
12	P2	P3	P4	P1
13	P2	P3	P4	P1
13.5	P2	P3	P4	
14	P1	P3	P4	
16	P1	P3	P4	

Now, let's calculate the turnaround time and waiting time for each process:

- Turnaround Time (TAT) for each process:
  - TAT(P1) = Completion Time(P1) - Arrival Time(P1) = 16 - 10 = 6
  - TAT(P2) = Completion Time(P2) - Arrival Time(P2) = 13.5 - 10 = 3.5
  - TAT(P3) = Completion Time(P3) - Arrival Time(P3) = 16 - 11 = 5
  - TAT(P4) = Completion Time(P4) - Arrival Time(P4) = 13 - 12 = 1
- Waiting Time (WT) for each process:
  - WT(P1) = TAT(P1) - Burst Time(P1) = 6 - 2 = 4
  - WT(P2) = TAT(P2) - Burst Time(P2) = 3.5 - 1 = 2.5
  - WT(P3) = TAT(P3) - Burst Time(P3) = 5 - 1 = 4
  - WT(P4) = TAT(P4) - Burst Time(P4) = 1 - 1 = 0

Now, let's move on to Round Robin with a time quantum of 0.5 units:

Gantt Chart for Round Robin:

Time	P1	P2	P3	P4	P1	P2	P3	P1
10	P1	P2	P3	P4	P1	P2	P3	P1
10.5	P1	P2	P3	P1	P2	P3	P1	
11	P2	P3	P1	P2	P3	P1		
11.5	P2	P3	P2	P3	P1			
12	P3	P2	P3	P1				
12.5	P3	P2	P1					
13	P3	P1						
13.5	P1							
14								

Now, let's calculate the turnaround time and waiting time for each process in Round Robin:

- **Turnaround Time (TAT) for each process:**
  - **TAT(P1) = Completion Time(P1) - Arrival Time(P1) = 13.5 - 10 = 3.5**
  - **TAT(P2) = Completion Time(P2) - Arrival Time(P2) = 13 - 10 = 3**
  - **TAT(P3) = Completion Time(P3) - Arrival Time(P3) = 14 - 11 = 3**
  - **TAT(P4) = Completion Time(P4) - Arrival Time(P4) = 14 - 12 = 2**
- **Waiting Time (WT) for each process:**
  - **WT(P1) = TAT(P1) - Burst Time(P1) = 3.5 - 2 = 1.5**
  - **WT(P2) = TAT(P2) - Burst Time(P2) = 3 - 1 = 2**
  - **WT(P3) = TAT(P3) - Burst Time(P3) = 3 - 1 = 2**
  - **WT(P4) = TAT(P4) - Burst Time(P4) = 2 - 1 = 1**

Now, we can calculate the average turnaround time and average waiting time:

- **Average Turnaround Time (TAT):**
  - **(3.5 + 3 + 3 + 2) / 4 = 11.5 / 4 = 2.875 units**
- **Average Waiting Time (WT):**
  - **(1.5 + 2 + 2 + 1) / 4 = 6.5 / 4 = 1.625 units**

These are the average turnaround time and average waiting time for the given processes using SJF non-preemptive and Round Robin with a time quantum of 0.5 units.

Q.4] What is mean by Threads, Explain Thread lifecycle with diagram in detail?
ANS: Threads: A thread is the smallest unit of execution within a process. It represents an independent sequence of instructions that can be scheduled and executed by the operating system. Threads share the same resources, such as memory space and file descriptors, within a process, but they have their own program counter, registers, and stack.

Thread Lifecycle:
The lifecycle of a thread consists of several states that represent its progression from creation to termination. The typical thread lifecycle states include:

1. **New:**
   - **The thread is in the new state when it is created but has not yet started executing.**
2. **Runnable (Ready):**
   - **After creation, the thread moves to the runnable state. In this state, the thread is ready to run, but the operating system has not yet scheduled it to execute.**
3. **Blocked (Waiting):**
   - **A thread transitions to the blocked state when it is waiting for a specific event, such as I/O completion or acquiring a lock. In this state, the thread is not eligible to run.**
4. **Timed Waiting:**
   - **This is a variant of the blocked state where the thread is waiting for a specific amount of time before transitioning to the runnable state.**
5. **Terminated:**
   - **The thread enters the terminated state when it has completed its execution or has been explicitly terminated.**

Below is a diagram representing the thread lifecycle:

```
 +--------+
 | New     |
 +----|----+
      v
 +-------------+
 |Runnable     |
 +-----|-------+
       v
 +-----------+
 |Blocked     |
 +-----|-----+
       v
+--------------------+
|Timed Waiting       |
+--------|-----------+
         v
+----------------+
| Terminated     |
+----------------+
```

Explanation of States:

1. **New:**
    - **The thread is created in this state.**
2. **Runnable (Ready):**
    - **The thread is ready to run but has not been scheduled by the operating system yet.**
3. **Blocked:**
    - **The thread is waiting for an event, such as I/O or acquiring a lock. It cannot run until the event occurs.**
4. **Timed Waiting:**
    - **The thread is waiting for a specific amount of time before becoming runnable again.**
5. **Terminated:**
    - **The thread has completed its execution or has been explicitly terminated and has moved to the terminated state.**

Transitions:

- **Threads can transition from the new state to the runnable state when they are scheduled by the operating system.**
- **Threads move from the runnable state to blocked or timed waiting when they encounter a condition that requires them to wait.**
- **Blocked or timed waiting threads become runnable again when the event they were waiting for occurs or when the specified time elapses.**
- **Threads move to the terminated state when they complete their execution or are explicitly terminated by the program.**

Q.5] Compare Compilers and Interpreters.

ANS: Here's a simple and easy comparison of compilers and interpreters:

1. **Definition:**
    - **Compiler: A compiler is a program that translates the entire source code of a program into machine code before execution.**
    - **Interpreter: An interpreter is a program that reads and executes code line by line without prior translation into machine code.**
2. **Execution:**
    - **Compiler: Executes the program after the entire source code is translated into machine code. The resulting executable file can be run multiple times without recompilation.**
    - **Interpreter: Executes the program line by line, translating each line into machine code as it is encountered. This means the program needs to be interpreted each time it runs.**
3. **Performance:**
    - **Compiler: Generally produces faster runtime performance as the entire program is translated before execution, optimizing it for the target machine.**
    - **Interpreter: Often slower than compiled code as it translates and executes code line by line, without the opportunity for extensive optimization.**
4. **Debugging:**
    - **Compiler: Debugging can be more challenging as errors may only be detected after the entire code is compiled, making it harder to pinpoint issues.**
    - **Interpreter: Easier to debug since errors are detected and reported immediately as the code is executed line by line.**
5. **Portability:**
    - **Compiler: Produces platform-dependent executable files, which may require recompilation for different platforms.**
    - **Interpreter: Generally more portable as the interpreter itself can run on different platforms, allowing the same source code to be executed without modification.**
6. **Memory Usage:**
    - **Compiler: Typically requires more memory during compilation due to the need to store intermediate representations and optimized code.**
    - **Interpreter: Often requires less memory as it only needs to store the source code and interpreter program in memory during execution.**
7. **Examples:**
    - **Compiler: Examples include GCC (GNU Compiler Collection), Clang, and Microsoft Visual C++ Compiler.**
    - **Interpreter: Examples include Python Interpreter, Ruby Interpreter, and JavaScript Interpreter (in web browsers).**
8. **Error Handling:**
    - **Compiler: Error messages may be more cryptic as they often relate to issues found during the compilation phase.**
    - **Interpreter: Error messages are usually more informative as they occur during runtime and point directly to the line causing the issue.**

9. **Usage:**
    - **Compiler: Often used for languages where performance is critical, such as system-level programming languages like C and C++.**
    - **Interpreter: Commonly used for scripting languages, rapid prototyping, and scenarios where portability and ease of use are prioritized, like Python and JavaScript.**

Q.6] What is LEX? Explain working of LEX with suitable diagram.
ANS: LEX, which stands for Lexical Analyzer Generator, is a tool used in compiler construction to generate lexical analyzers or scanners for programming languages. Here's a simple explanation of how LEX works:

1. **Input Specification: LEX takes as input a file containing specifications written in a language similar to regular expressions, along with user-defined actions to be taken when certain patterns are matched.**
2. **Tokenization: LEX scans the input file and identifies patterns defined by the user, typically representing tokens in the language being analyzed. These patterns are expressed using regular expressions.**
3. **Token Matching: When LEX encounters text in the input that matches one of the specified patterns, it performs the corresponding action defined by the user. These actions often include tasks such as recording the matched token, updating counters, or performing specific processing tasks.**
4. **Generation of Lexical Analyzer: Based on the specifications provided by the user, LEX generates source code for a lexical analyzer written in a target programming language, such as C or C++. This generated code includes the logic to recognize the specified patterns and execute the corresponding actions.**
5. **Integration with Compiler: The generated lexical analyzer code can be integrated into the larger compiler system. When the compiler processes source code written in the language being analyzed, it passes the input through the lexical analyzer to tokenize it before proceeding with further compilation phases.**

Diagram:

```
+-------------------------+
|       Input File        |
+-------------------------+
             |
             v
+-------------------------+
|   LEX Specification     |
|   (Regular Expressions) |
+-------------------------+
             |
             v
+-------------------------+
|    LEX Processing       |
|   (Tokenization, Action)|
+-------------------------+
             |
             v
+-------------------------+
|   Generated Lexical     |
|    Analyzer Code        |
+-------------------------+
             |
             v
+-------------------------+
|       Compiler          |
|     (Integration)       |
+-------------------------+
```

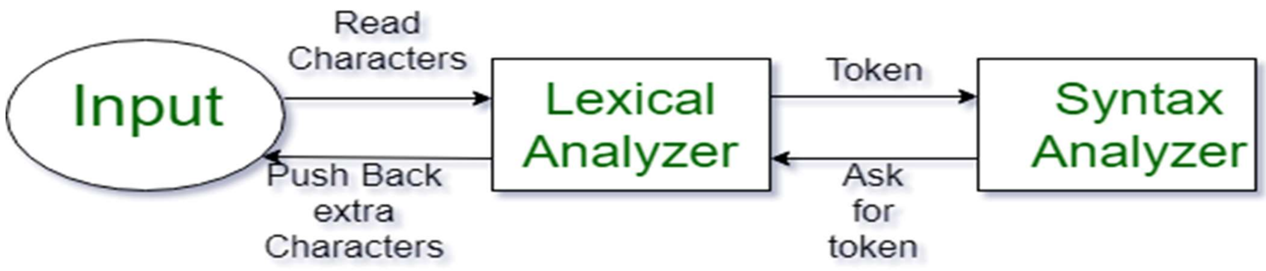
Q.7] Define token, pattern, lexemes & lexical error.
ANS: Here's a simplified explanation:

1. **Token:**
   - A token is a meaningful unit in a programming language. It represents a particular category of symbols that the compiler or interpreter recognizes.
   - It could be a keyword, identifier, operator, constant, or symbol.
   - Tokens are the building blocks of a program and are used by the compiler or interpreter to understand the code's structure and meaning.
2. **Pattern:**
   - A pattern is a rule or template used to describe the structure of valid tokens in a programming language.
   - It defines the syntax or format that tokens must adhere to in order to be recognized by the compiler or interpreter.
   - Patterns are often expressed using regular expressions or similar formalisms to specify the sequence of characters that constitute a valid token.
3. **Lexemes:**
   - Lexemes are the smallest units of meaningful information in the source code of a programming language.
   - They are the actual instances of tokens found in the source code.
   - For example, in the expression "x = 5 + 3", the lexemes include the variable name "x", the assignment operator "=", and the numeric constants "5" and "3".
4. **Lexical Error:**
   - A lexical error, also known as a lexer or scanning error, occurs when the compiler or interpreter encounters an invalid sequence of characters that do not match any of the defined tokens or patterns in the language's grammar.
   - These errors typically occur during the initial scanning or tokenization phase of the compilation process.
   - Lexical errors can include misspelled keywords, unrecognized symbols, or illegal characters.
   - Fixing lexical errors often involves correcting typos or ensuring that the source code conforms to the language's syntax rules.

Q.8] What is a compiler? Explain any two phases of compiler with suitable diagram.

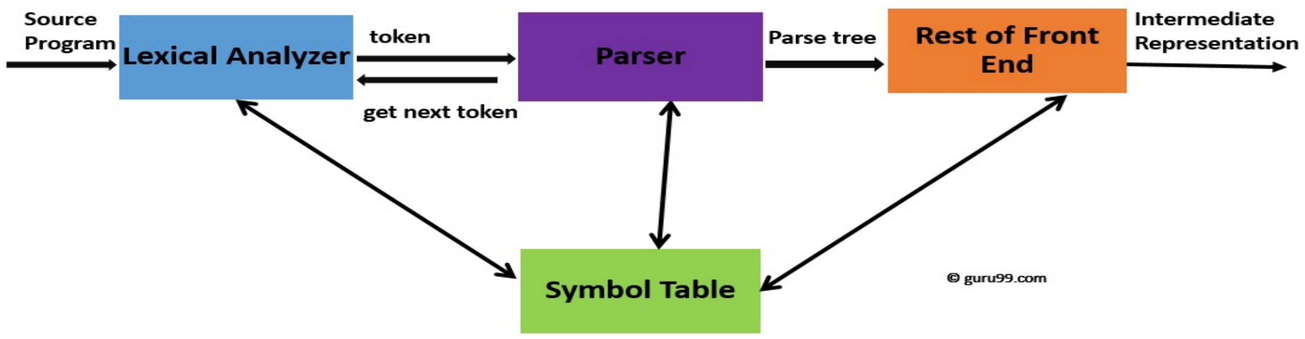
ANS: A compiler is a software tool that translates source code written in a high-level programming language into machine code or bytecode that a computer can understand and execute. Here's a simple explanation of two phases of a compiler:

1. **Lexical Analysis Phase:**
    - **Purpose: This phase breaks the source code into meaningful components called tokens.**
    - **Steps:**
        1. **Scanning: The source code is read character by character, and sequences of characters that form a token (like identifiers, keywords, operators, etc.) are recognized.**
        2. **Tokenization: Tokens are grouped into categories like identifiers, keywords, operators, etc., and represented as tokens.**
    - **Diagram:**



2. **Syntax Analysis Phase (Parsing):**

- **Purpose: This phase verifies that the tokens generated by the lexical analysis phase form grammatically correct statements according to the rules of the programming language.**
- **Steps:**
    1. **Parsing: The tokens generated in the lexical analysis phase are analyzed to form a hierarchical structure according to the grammar rules of the programming language.**
    2. **Syntax Tree Construction: Based on the parsing, a syntax tree or abstract syntax tree (AST) is constructed, representing the syntactic structure of the program.**
- **Diagram:**



Q.9] List different types of Operating Systems? Describe any two of them.
ANS: here are different types of operating systems described in simple and easy points:

1. **Single-user, Single-tasking OS:**
   - Designed to support only one user and one task at a time.
   - Examples include MS-DOS (Microsoft Disk Operating System) and early versions of Apple Macintosh operating system.
2. **Single-user, Multi-tasking OS:**
   - Allows a single user to perform multiple tasks simultaneously.
   - Examples include Microsoft Windows, macOS, and Linux distributions like Ubuntu.
   - Users can run several applications concurrently, switching between them seamlessly.
3. **Multi-user OS:**
   - Supports multiple users accessing the system simultaneously.
   - Each user can have their own set of processes and resources.
   - Examples include Unix, Linux server distributions, and some versions of Windows Server.
4. **Real-time OS:**
   - Designed to process data as it comes in, without buffering delays.
   - Used in applications where timing is critical, like industrial automation, robotics, and aerospace.
   - Examples include VxWorks, QNX, and RTLinux.
5. **Embedded OS:**
   - Optimized for use in embedded systems, which are dedicated to specific tasks or functions.
   - Often found in devices like smartphones, tablets, smart appliances, and automotive systems.
   - Examples include Android (for mobile devices), FreeRTOS, and Embedded Linux.
6. **Distributed OS:**
   - Spreads computation across multiple machines connected by a network.
   - Offers advantages like fault tolerance, load balancing, and resource sharing.
   - Examples include Google's Android (used across multiple devices), Amoeba, and Sprite.
7. **Network OS:**
   - Primarily focused on providing network services such as file sharing, printing, and email.
   - Facilitates communication between different computers and devices on a network.
   - Examples include Novell NetWare, Windows Server, and Linux distributions like CentOS.

8. **Mobile OS:**
    - **Specifically designed for mobile devices like smartphones and tablets.**
    - **Prioritizes features like touch screen support, battery efficiency, and mobile data connectivity.**
    - **Examples include Android, iOS (Apple), and Windows Phone (now discontinued).**

Q.10] Differentiate Preemptive and non preemptive scheduling.
ANS: here's a simple point-wise explanation differentiating between preemptive and non-preemptive scheduling:

Preemptive Scheduling:

1. **Interruptible: Tasks can be paused or interrupted by the operating system.**
2. **Priority-based: Tasks are executed based on their priority levels, with higher priority tasks being executed first.**
3. **Dynamic: Priorities can change dynamically during execution, allowing for flexibility in task management.**
4. **Overhead: Higher overhead due to frequent context switching between tasks.**
5. **Response Time: Generally better response time for high-priority tasks, as they are not blocked by lower-priority tasks.**

Non-preemptive Scheduling:

1. **Non-interruptible: Once a task starts execution, it cannot be interrupted until it completes or voluntarily gives up control.**
2. **FIFO (First In, First Out): Tasks are executed in the order they arrive, without consideration of priority.**
3. **Simple: Less complex compared to preemptive scheduling as there's no need for frequent context switching.**
4. **Predictable: Execution flow is more predictable as tasks run to completion without interruption.**
5. **Resource Utilization: May lead to lower overall resource utilization as higher-priority tasks might have to wait for lower-priority tasks to complete.**

Q.11] What is time quantum and its significance in Round robin scheduling.
ANS: here's a simple explanation of time quantum and its significance in Round Robin scheduling:

1. **Definition: Time quantum is the maximum amount of time a process is allowed to run in a single instance before it is preempted or interrupted.**
2. **Equal Time Slices: In Round Robin scheduling, each process is assigned a small unit of time called a time slice or time quantum.**
3. **Preemption: When a process's time quantum expires, it is preempted, meaning it is temporarily stopped to allow another process to run.**
4. **Fairness: Time quantum ensures fairness by giving each process an equal opportunity to execute, preventing any single process from monopolizing the CPU.**
5. **Cyclic Execution: Processes are executed in a cyclic manner, where each process is given a chance to execute for its allotted time quantum.**
6. **Context Switching: When a process is preempted, the operating system performs a context switch, saving the current state of the process and loading the state of the next process to run.**
7. **Performance: The choice of time quantum affects the performance of the scheduling algorithm. A shorter time quantum results in more frequent context switches but better responsiveness for interactive tasks. A longer time quantum reduces context switch overhead but may lead to longer response times for certain tasks.**
8. **Response Time: Time quantum impacts the response time of processes. Shorter time quanta lead to shorter response times for interactive tasks since they get more frequent chances to execute.**
9. **Tuning: Selecting an appropriate time quantum is crucial for optimizing system performance. It often involves balancing factors like throughput, response time, and overhead.**

Q.12] Explain multithreaded mode and Process Control block in detail.
ANS: here's a breakdown of multithreaded mode and Process Control Block (PCB) in simple points:
Multithreaded Mode:

1. **Definition: Multithreading is a programming concept where multiple threads of execution exist within the same process, sharing resources such as memory and CPU time.**
2. **Concurrency: It allows multiple threads to execute concurrently within the same process. Each thread can perform a different task simultaneously.**
3. **Efficiency: Multithreading improves efficiency by allowing tasks to be executed in parallel, utilizing available CPU resources effectively.**
4. **Responsiveness: Applications with multithreading can remain responsive to user interactions even when performing intensive tasks, as threads can handle background processing.**
5. **Resource Sharing: Threads within the same process share resources such as memory space, file handles, and other system resources.**
6. **Communication: Threads can communicate with each other through shared memory or synchronization mechanisms like mutexes and semaphores.**
7. **Synchronization: Synchronization mechanisms are essential to manage access to shared resources and avoid conflicts between threads accessing the same data concurrently.**

Process Control Block (PCB):

1. **Definition: PCB is a data structure used by the operating system to manage information about each process in the system.**
2. **Storage: Each process in the system has its own PCB, which stores essential information about the process's state and resources.**
3. **Contents: PCB typically contains information such as process ID, process state, program counter, CPU registers, memory allocation, and accounting information.**
4. **Process State: PCB indicates the current state of the process, whether it's running, ready, blocked, or terminated.**
5. **Context Switching: When the operating system switches between processes, it saves the current process's PCB and loads the PCB of the next process to be executed, a process known as context switching.**
6. **Process Scheduling: PCB plays a crucial role in process scheduling, as the operating system uses the information stored in the PCB to prioritize and schedule processes for execution.**
7. **Resource Management: PCB contains information about the resources allocated to the process, such as memory segments, open files, and I/O devices, facilitating efficient resource management by the operating system.**
8. **Process Termination: When a process terminates, its PCB is removed from the system, releasing any allocated resources and freeing up memory for other processes.**
9. **Critical for OS: PCB is fundamental to the functioning of the operating system, providing the necessary infrastructure for process management, scheduling, and resource allocation.**