What is the relationship between threads and processes?

Threads and Processes

A process is a program in execution. A thread is a basic unit of CPU utilization that exists within a process. A process can contain one or more threads.

Here's a breakdown of the key concepts and their relationship:

- **Process:** An active entity that represents a program being executed. It has its own address space, memory, resources, and program counter.
- **Thread:** A lightweight execution unit within a process. It shares the process's resources but has its own thread ID, program counter, register set, and stack.

A process with a single thread of control is a traditional or heavyweight process. A process can also have multiple threads of control, allowing it to perform more than one task at a time. For example, a web server can use multiple threads to service multiple client requests simultaneously.

Threads share the code section, data section, and other operating system resources of the process to which they belong. This makes thread creation more efficient than process creation.

The relationship between threads and processes can be explained using different multithreading models:

- **Many-to-One Model:** Many user-level threads are mapped to one kernel thread. Thread management is done in user space, but this model can't take advantage of multiple processing cores.
- **One-to-One Model:** Each user thread is mapped to a kernel thread, allowing multiple threads to run in parallel on multiprocessors. This model is used by Linux and Windows.
- **Many-to-Many Model:** Many user-level threads are multiplexed to a smaller or equal number of kernel threads. This model allows for greater flexibility and concurrency.

Overall, threads provide a way to increase concurrency and efficiency within a process.

What are the advantages of multiple threads over multiple processes? What major disadvantage do they have? Suggest one application that would benefit from the use of threads, and one that would not.

Advantages of Threads

- **Resource Sharing:** Threads share the memory and resources of the process to which they belong by default. This makes communication between threads easier and more efficient.
- **Responsiveness:** Multithreading can increase responsiveness to the user, especially in interactive applications. If one thread is blocked, other threads can continue executing. This is especially beneficial when designing user interfaces.
- **Economy:** Threads are more economical to create and manage than processes. Thread creation and context switching are much faster than the corresponding operations for processes. For instance, creating a process in Solaris is about thirty times slower than creating a thread, and context switching is about five times slower.
- **Scalability:** Multithreaded programs can scale better on multiprocessor and multicore systems, allowing threads to run in parallel on different processing cores.

Disadvantage of Threads

- **Limited Modularity:** Using thread identifiers directly in code can lead to a lack of modularity. Changing thread identifiers might require examining all other thread definitions and modifying references to the old identifier. Techniques involving indirection are more desirable.

Application that Benefits from Threads: Web Server

A multithreaded web server would benefit greatly from the use of threads. A multithreaded web server can handle multiple client requests simultaneously without creating a new process for each request. This improves performance and resource utilization. Each thread can handle a separate client request, allowing the server to continue listening for new requests.

Application that Would Not Benefit from Threads: Single-Tasking System

A single-tasking system, like MS-DOS, would not benefit from threads because it only runs one program at a time. Using multiple threads in such a system wouldn't provide any advantage as there is no concurrency or parallelism to leverage.

What resources are used when a thread is created? How do they differ from those used when a process is created?

Resources Used by Threads and Processes

When a thread is created, it uses fewer resources than when a process is created. This is because threads share the resources of the process to which they belong, while processes have their own independent resources.

Resources used when a thread is created:

- **Thread ID:** A unique identifier for the thread.
- **Program Counter:** Points to the next instruction to be executed.
- **Register Set:** Stores the current values of the CPU registers.
- **Stack:** Stores local variables and function call information.

Threads share the following resources of the process:

- **Code Section:** Contains the program instructions.
- **Data Section:** Contains global variables and static data.
- **Heap:** Dynamically allocated memory.
- **Open Files:** Files that the process has opened.
- **Signals:** Signals that the process can receive.

Resources used when a process is created:

- **Process ID:** A unique identifier for the process.
- **Address Space:** A range of memory addresses that the process can access.
- **Memory:** Physical and virtual memory allocated to the process.

- **CPU Time:** Time allocated to the process for execution.
- **Files:** Files that the process has opened.
- **I/O Devices:** Devices that the process can use, such as the keyboard, mouse, and display.
- **Process Control Block (PCB):** A data structure that stores information about the process.

Key Differences in Resource Usage:

- **Memory:** Processes have their own independent address space and memory, while threads share the memory of their parent process. This is a major difference, as it allows threads to communicate more efficiently.
- **Overhead:** Creating a thread is much less resource-intensive than creating a process. This is because threads share many resources with the process, while a process requires a new address space, memory allocation, and initialization of data structures like the PCB.

Benefits of Thread's Lighter Resource Usage:

The fact that threads use fewer resources than processes offers several benefits:

- **Faster creation and context switching:** This makes multithreaded programs more efficient.
- **Better communication and data sharing:** Threads within a process can easily access shared data, leading to simpler and faster communication.

Conclusion:

Threads are a lightweight execution unit within a process, using fewer resources than processes. They share resources with the parent process, leading to efficient communication and reduced overhead. While processes require significant resources like an independent address space and memory, threads only utilize a small set of resources like a thread ID, program counter, register set, and stack.

What is a thread in OS? How is it different from a process?

Threads in OS

A thread is a basic unit of CPU utilization comprising a thread ID, a program counter, a register set, and a stack. Threads share the same code section, data section, and other operating system resources as the process to which they belong. This means threads can access the same files and signals as the process. **Threads allow a process to perform more than one task at a time.** For example, a web server can use threads to service requests from multiple clients concurrently. Many operating system kernels are now multithreaded, allowing the operating system to perform multiple tasks simultaneously, such as managing devices and memory.

There are several benefits to using threads, including:

- **Responsiveness:** Multithreading an interactive application may allow a program to continue running even if part of it is blocked, increasing responsiveness to the user.
- **Resource Sharing:** Threads share resources by default, while processes must use explicit techniques like shared memory and message passing.
- **Economy:** Creating and context-switching threads is more economical than doing so with processes. For example, creating a process in Solaris is about 30 times slower than creating a thread.
- **Scalability:** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

Threads can be implemented at the user level or by the kernel:

- **User threads** are supported above the kernel and are managed without kernel support.
- **Kernel threads** are supported and managed directly by the operating system.

There are three common models for relating user and kernel threads:

- **Many-to-One Model:** Many user-level threads are mapped to one kernel thread. This model is efficient but cannot take advantage of multiple processing cores.
- **One-to-One Model:** Each user thread is mapped to a kernel thread. This model provides more concurrency than the many-to-one model and allows multiple threads to run in parallel on multiprocessors.
- **Many-to-Many Model:** Many user-level threads are multiplexed to a smaller or equal number of kernel threads.

Threads vs. Processes

A process is a program in execution. A process is a unit of work in a system. A system can consist of a collection of processes, some of which are operating system processes and some of which are user processes. Processes can potentially execute concurrently, even on a single CPU.

A process is more than just program code:

- It includes the process stack.
- It includes a data section, which contains global variables.
- It may also include a heap, which is memory that is dynamically allocated.

A thread exists within a process and shares the process's resources. A process can have multiple threads.

Here is a table summarizing the key differences between threads and processes:

Feature	**Thread**	**Process**
Definition	A basic unit of CPU utilization within a process.	A program in execution, which forms the basis of all computation.
Resources	Shares resources with other threads in the same process.	Has its own independent resources, including memory, files, and I/O devices.

Creation Overhead	Low overhead for creation and management.	High overhead for creation and management.
Communication	Easy and efficient communication through shared memory.	Requires explicit mechanisms like shared memory or message passing for communication.
Concurrency	Can achieve concurrency even on a single-processor system.	Can achieve concurrency only on multiprocessor or multicore systems.
Context Switching	Context switching between threads is faster than between processes.	Context switching between processes is slower due to the larger amount of information that needs to be saved and restored.
Programming Model	Typically used in concurrent programming models.	Typically used in traditional sequential programming models.

Explain multicore programming with an example.

Multicore Programming Explained with an Example

Multicore programming refers to the design and implementation of software that can effectively utilize multiple processing cores within a system. This approach leverages the parallel processing capabilities of multicore systems to enhance application performance, particularly for computationally intensive tasks.

The sources mention that **multicore systems have become increasingly prevalent in recent years**, migrating from servers to desktop and laptop systems and even appearing on mobile devices. This trend has driven the need for effective multicore programming techniques.

The sources state that **multithreading is a crucial mechanism for efficient multicore programming**. By dividing an application into multiple threads of execution, each thread can be assigned to a separate core, enabling parallel execution and potentially significant speedup.

Amdahl's Law is introduced as a formula to estimate the potential performance gains achievable by adding additional cores. It highlights that the speedup is limited by the portion of the application that must be executed serially, emphasizing the importance of maximizing the parallelizable components of an application.

Example: Summing an Array

Consider a program that calculates the sum of elements in an array. In a single-threaded implementation, one thread iterates through the entire array, accumulating the sum. In a multithreaded implementation on a dual-core system, the task can be divided into two threads:

- **Thread A:** Sums elements from index 0 to N/2 - 1 on core 0.
- **Thread B:** Sums elements from index N/2 to N - 1 on core 1.

Each thread executes its portion of the summation in parallel. Once both threads complete, their partial sums are combined to produce the final result.

This example demonstrates the concept of **data parallelism**, where the same operation (summation) is applied to different subsets of data (array elements) on separate cores. **Task parallelism**, in contrast, involves distributing different tasks (potentially operating on the same or different data) across multiple cores.

Assume a scenario with a system having two cores. Each core has two processors. A CPU-intensive application is running on this system. When the program starts, a single file must be opened and it should be read sequentially, i.e., all input is performed as soon as the program starts. Similarly, just before the program terminates, the program results should be written sequentially to a single file. The program is CPU-bound after the startup and before the termination. Explain how will you improve performance by using multithreading.

Also determine how many threads are to be created for taking care of input & output. Substantiate your answer.

For the CPU-bound part of the application, how many threads are to be created? Explain.

Multithreading for CPU-Intensive Application

This scenario presents a CPU-intensive application running on a system with two cores, each containing two processors. The application requires sequential file input at startup and sequential file output before termination. To optimize performance using multithreading, consider the following:

- **Input/Output Threads:** You should create **one thread for input and one thread for output**. This is because the input and output operations are sequential and cannot be parallelized. Having dedicated threads for these tasks allows the CPU-bound portion of the application to proceed concurrently, rather than waiting for I/O operations to complete.
    - For instance, the input thread can handle opening the file and reading the data at the start. The output thread can write the results to the file just before the program terminates. This approach prevents I/O operations from blocking the main processing thread, maximizing CPU utilization.
- **CPU-Bound Threads:** The optimal number of threads for the CPU-bound part of the application should be equal to the number of logical processors available, which is **four** in this case (two cores x two processors per core). Creating more threads than the available

logical processors wouldn't offer additional performance benefits, as the system can only execute a limited number of threads concurrently. Instead, it would lead to increased overhead due to context switching between threads.

  - As the application is CPU-bound, creating four threads allows for parallel execution across all available processors, maximizing the utilization of available computing resources. Each thread can operate on a different portion of the data or execute a different task, contributing to the overall progress of the computation.

Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.

User-Level Threads on Multiprocessor Systems

Yes, a multithreaded solution using multiple user-level threads can achieve better performance on a multiprocessor system than on a single-processor system. Here's why:

- **Parallel Execution:** Multiprocessor systems have multiple processing cores, allowing multiple threads to execute truly in parallel. On a single-processor system, even with multiple user-level threads, only one thread can be executed by the CPU at any given time. The other threads are interleaved, giving the illusion of parallelism, but not achieving true simultaneous execution.
- **Kernel Thread Mapping:** The performance difference depends on the threading model used to map user-level threads to kernel threads.
  - **Many-to-One Model:** In this model, all user-level threads are mapped to a single kernel thread. This means that even on a multiprocessor system, only one user thread can execute at a time, limiting the benefits of multithreading.
  - **One-to-One Model:** Each user-level thread is mapped to a separate kernel thread. This allows for true parallel execution of multiple user threads on a multiprocessor system,

significantly improving performance compared to a single-processor system.

    - **Many-to-Many Model:** This model provides the flexibility to map multiple user-level threads to a smaller or equal number of kernel threads. This model can also achieve better performance on a multiprocessor system, as multiple kernel threads can be scheduled to run in parallel.

- **Example:** Imagine a multithreaded web server using the one-to-one model. On a multiprocessor system, each incoming client request can be handled by a separate user thread running on a different core, allowing the server to process multiple requests concurrently and achieve higher throughput. On a single-processor system, even with multiple threads, the server would still be limited to processing one request at a time, resulting in slower response times and reduced throughput.

Key Takeaway: User-level threads can leverage the parallel execution capabilities of multiprocessor systems to enhance performance significantly, provided the threading model allows for the mapping of multiple user threads to multiple kernel threads.

Explain the different multithreading models.

Understanding Multithreading Models

The sources discuss various aspects of multithreading, including the benefits and challenges of multithreaded programming, the distinction between user-level and kernel-level threads, and the importance of multithreading in the context of multicore programming. They also describe different models for mapping user-level threads to kernel-level threads. These multithreading models determine how user threads are managed and scheduled in relation to kernel threads, affecting the concurrency and performance of multithreaded applications. The three main multithreading models are:

- **Many-to-One Model**
    - This model maps multiple user-level threads to a single kernel thread.

  - Thread management is performed efficiently in user space by the thread library.
  - **Drawback:** If any user thread makes a blocking system call, the entire process blocks because only one kernel thread represents the entire process.
  - **Drawback:** This model can't take advantage of multiple processing cores because only one thread can access the kernel at a time, limiting parallelism.
  - **Example:** Green threads in Solaris and early versions of Java.
- **One-to-One Model**
  - Each user-level thread maps to a distinct kernel thread.
  - This model provides better concurrency than the many-to-one model by allowing other threads to run even if one thread makes a blocking system call.
  - It enables multiple threads to run in parallel on multiprocessor systems.
  - **Drawback:** Creating a user thread requires creating a corresponding kernel thread, which can lead to performance overhead due to the cost of kernel thread creation.
  - **Drawback:** To mitigate performance issues, implementations of this model often limit the number of threads supported by the system.
  - **Example:** Linux, Windows.
- **Many-to-Many Model**
  - This model multiplexes multiple user-level threads onto a smaller or equal number of kernel threads.
  - The number of kernel threads can be specific to an application or machine (more kernel threads might be allocated on a multiprocessor than on a single processor).
  - **Advantage:** Developers can create many user threads as needed without impacting concurrency.

- **Advantage:** The corresponding kernel threads can run in parallel on a multiprocessor, increasing performance.
- **Advantage:** When a thread makes a blocking system call, the kernel can schedule another thread, ensuring efficient resource utilization.
- **Two-Level Model:** A variation of the many-to-many model where user-level threads are multiplexed onto a smaller or equal number of kernel threads, but also allows binding a user-level thread to a kernel thread.
- **Example:** Older Solaris versions (prior to Solaris 9).

Choosing the right multithreading model depends on factors such as the desired level of concurrency, the overhead of kernel thread creation, and the support provided by the underlying operating system.

How user-level threads are different from kernel-level threads? Under what circumstances is one type "better" than the other?

Comparing User-Level and Kernel-Level Threads

User-level threads are threads implemented and managed by a thread library in user space, while **kernel-level threads** are supported and managed directly by the operating system kernel. The sources offer several key distinctions and scenarios where one type of thread proves more advantageous:

Key Differences:

- **Management:** User-level threads are managed by the thread library, requiring no kernel intervention. Kernel-level threads are managed by the operating system, involving system calls and kernel resources.
- **Context Switching:** Context switching between user-level threads is typically faster than between kernel-level threads because it doesn't require a switch to kernel mode.
- **Resource Consumption:** Kernel-level thread creation consumes more resources than user-level thread creation. Each kernel-level

thread requires a separate kernel stack and other kernel resources, while multiple user-level threads can share a single user-level stack.

- **Blocking System Calls:** A blocking system call made by one user-level thread in a many-to-one model will block the entire process. In contrast, with a one-to-one model, other user-level threads mapped to separate kernel threads can continue execution.

When User-Level Threads are "Better":

- **Performance Sensitivity:** User-level threads offer faster creation and context switching, benefiting performance-critical applications. The reduced kernel interaction minimizes overhead.
- **Portability:** User-level threads implemented via a thread library can be more portable across different operating systems than kernel-level threads, which depend on specific kernel support.
- **Fine-grained Control:** User-level threads allow developers finer control over thread scheduling and management policies. The thread library can implement customized scheduling algorithms tailored to the application's needs.

When Kernel-Level Threads are "Better":

- **Concurrency on Multiprocessors:** Kernel-level threads are essential for exploiting true parallelism on multiprocessor systems. Models like one-to-one and many-to-many map user threads to kernel threads, allowing concurrent execution on multiple cores.
- **Blocking Operations:** Kernel-level threads are preferred when an application involves blocking system calls or I/O operations. In a one-to-one model, a blocked kernel thread won't halt other threads in the process.
- **Kernel-Level Resources:** When threads need to interact directly with kernel-level resources or services, such as device drivers or system calls, kernel-level threads are necessary.

A multithreaded application uses only reader-writer locks for synchronization. Applying the four necessary conditions for deadlock, is deadlock still possible if multiple reader-writer locks are used?

Deadlock with Reader-Writer Locks

Yes, even though a multithreaded application uses only reader-writer locks for synchronization, deadlock is still possible if multiple reader-writer locks are used. Applying the four necessary conditions, this can be explained as follows:

- **Mutual Exclusion:** Reader-writer locks inherently enforce mutual exclusion. A writer lock grants exclusive access to a resource, preventing other readers or writers from accessing it concurrently. This satisfies the mutual exclusion condition.
- **Hold and Wait:** If a thread acquires a reader-writer lock (either read or write) and then requests another lock, it fulfills the hold-and-wait condition. The thread holds one lock while waiting for another.
- **No Preemption:** Reader-writer locks, like other synchronization primitives, are typically released voluntarily by the thread holding the lock. They are not preempted by the operating system. This satisfies the no preemption condition.
- **Circular Wait:** Deadlock can occur if multiple threads acquire reader-writer locks in different orders, creating a circular dependency.
    - **Example:** Suppose thread 1 acquires a write lock on resource A and then requests a read lock on resource B. Simultaneously, thread 2 acquires a read lock on resource B and then requests a write lock on resource A. Thread 1 holds a lock on A while waiting for B, and thread 2 holds a lock on B while waiting for A, forming a circular wait that leads to deadlock.

Addressing Deadlock:

Even with reader-writer locks, strategies for preventing or avoiding deadlock are crucial. Techniques mentioned in the sources, like ensuring lock acquisition in a predefined order (total ordering), can be adapted to reader-writer locks.

- **Total Ordering:** Define a strict ordering for all reader-writer locks in the application. Each thread must request locks in ascending order, preventing circular dependencies. This technique, however, requires careful design and implementation to ensure all threads adhere to the predefined order, especially when dealing with dynamically acquired locks.

Additional Considerations:

- While the sources provide a solid foundation for understanding deadlocks and synchronization concepts, they don't directly address deadlock scenarios specific to reader-writer locks. The provided explanation and example are based on general deadlock principles and common practices in multithreaded programming.
- It is advisable to explore additional resources and real-world examples to gain a deeper understanding of deadlock prevention and handling in applications that exclusively utilize reader-writer locks.

Key Takeaway: The possibility of deadlock remains even when using reader-writer locks. Understanding the necessary conditions and applying appropriate deadlock prevention or avoidance techniques are vital for robust multithreaded application development.

What is Process scheduler? What are the types of queues are maintained for process scheduling?

Process Scheduler and Types of Queues

A **process scheduler** is a crucial component of an operating system responsible for determining which process should be executed next and allocating CPU time to that process. It aims to maximize CPU utilization and ensure fair allocation of processing resources among multiple processes in the system.

The sources provide detailed information about the concept of process scheduling and the different types of queues used for managing processes in the system:

- **Job Queue**: When processes enter the system, they are initially placed in a job queue, which holds all the processes in the system. The long-term scheduler, or job scheduler, selects processes from the job queue and loads them into memory for execution, controlling the degree of multiprogramming.
- **Ready Queue**: Processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. The ready queue is typically implemented as a linked list, but it can also be implemented as a FIFO queue, a priority queue, a tree, or an unordered linked list. Each process in the ready queue is represented by its Process Control Block (PCB), which contains information about the process, including its state, program counter, CPU registers, scheduling information, and memory management information. The short-term scheduler, or CPU scheduler, selects a process from the ready queue and allocates the CPU to that process.
- **Device Queues**: When a process requests an I/O operation on a shared device, such as a disk, it may have to wait if the device is busy serving another process. The operating system maintains a separate queue for each I/O device, called a device queue. Processes waiting for a particular I/O device are placed in the corresponding device queue.
- **Other Queues**: Besides the primary queues mentioned above, other queues may exist depending on the specific scheduling algorithms and system design. For instance, a multilevel queue scheduling algorithm partitions the ready queue into multiple separate queues based on process characteristics or priorities. The sources also mention a medium-term scheduler, which can swap out partially executed processes from memory to a backing store to control the degree of multiprogramming, introducing a queue for swapped-out processes.

Queueing Diagram

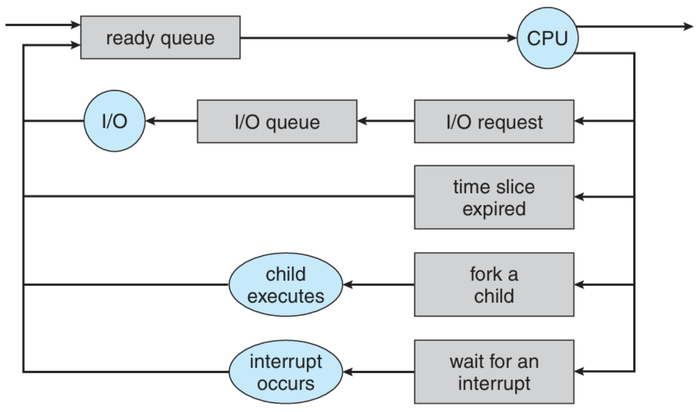

Figure 3.6 Queueing-diagram representation of process scheduling.

What is process scheduling in the operating system? Which feature of the OS makes process scheduling possible? State different types of schedulers?

Process Scheduling in Operating Systems

Process scheduling is a fundamental function of an operating system. It refers to the mechanism by which the operating system determines which process should be executed next and allocates CPU time to that process. Process scheduling aims to maximize CPU utilization, ensure fairness, and provide acceptable response times for all processes in the system. The sources provide insights into how process scheduling operates and its essential role in an operating system.

Feature Enabling Process Scheduling: The ability of the operating system to switch between processes, facilitated by interrupts, is the key feature that makes process scheduling possible. Interrupts allow the operating system to regain control of the CPU when a process finishes executing, requests an I/O operation, or encounters an error. The operating system can then choose another process from the ready queue to run.

Types of Schedulers

The sources describe various types of schedulers, each serving a specific purpose in the process management lifecycle:

- **Long-Term Scheduler (Job Scheduler)**: The long-term scheduler is responsible for selecting processes from the job pool (processes residing on disk) and loading them into memory for execution. This scheduler controls the degree of multiprogramming—the number of processes residing in memory. It considers factors such as the memory requirements of each process and the available memory space. The long-term scheduler doesn't execute as frequently as the short-term scheduler because processes are typically loaded into memory less often.
- **Short-Term Scheduler (CPU Scheduler)**: This scheduler is the heart of process scheduling, selecting a process from the ready queue (processes in memory ready to execute) and allocating the CPU to that process. The short-term scheduler operates far more frequently

than the long-term scheduler and must be very efficient to avoid excessive overhead. It makes its decisions based on various scheduling algorithms, which we'll explore shortly.

- **Medium-Term Scheduler**: Some operating systems, particularly time-sharing systems, implement a medium-term scheduler to manage swapping. Swapping involves removing a process from memory (swapping out) and placing it in a backing store, typically a fast disk, to reduce the degree of multiprogramming or free up memory. Later, the process can be reintroduced into memory and its execution continued (swapping in).

Examples of Schedulers in Real-World Operating Systems

The sources provide examples of schedulers implemented in popular operating systems, demonstrating the practical application of scheduling concepts:

- **Linux Scheduler:** Linux employs a priority-based scheduling algorithm with support for both real-time and non-real-time processes. It uses a system of scheduling classes, each with its own priority. The CFS (Completely Fair Scheduler) algorithm, used by the default scheduling class, aims to provide a fair allocation of CPU time to all processes. Linux also supports real-time scheduling using algorithms like rate-monotonic and earliest-deadline-first.

- **Windows Scheduler**: Windows utilizes a priority-based, preemptive scheduling algorithm. The scheduler prioritizes real-time processes over non-real-time processes. It assigns threads to different priority levels and uses preemption to ensure that the highest-priority thread always runs.

Compare the following types of Schedulers: 1. Short term 2. Long term 3. Medium term.

Comparison of Schedulers: Short-Term, Long-Term, and Medium-Term

The sources discuss three main types of schedulers: **short-term, long-term, and medium-term**, each playing a distinct role in managing processes within an operating system. A comprehensive comparison of these schedulers is provided below:

Feature	Short-Term Scheduler	Long-Term Scheduler	Medium-Term Scheduler
Also Known As	CPU Scheduler	Job Scheduler	Swapper
Purpose	Selects a process from the ready queue and allocates the CPU to it.	Selects processes from the job pool and loads them into memory for execution.	Removes processes from memory (swapping out) and reintroduces them later (swapping in) to regulate the degree of multiprogramming.
Frequency	Executes very frequently, typically every few milliseconds.	Executes much less frequently than the short-term scheduler, sometimes only when a process leaves the system.	Executes at an intermediate frequency between the short-term and long-term schedulers.
Speed	Must be very fast to minimize overhead, as it's invoked during every process switch.	Can afford to take more time to make decisions since it doesn't execute as often.	Speed is less critical compared to the short-term scheduler but still needs to be reasonably efficient.

Key Decisions	Which process from the ready queue should run next? How long should a process run before preemption?	Which processes from the job pool should be loaded into memory? How many processes should be in memory at once?	Which processes should be swapped out? When should a swapped-out process be swapped back in?
Algorithms	Wide variety, including: First-Come, First-Served (FCFS) Shortest Job First (SJF) Round Robin (RR) Priority Multilevel Queue Multilevel Feedback Queue	Typically uses simpler algorithms, such as FCFS or priority-based selection.	May use priority-based or other criteria for selecting processes to swap.

Short-Term Scheduler

The **short-term scheduler** (also known as the **CPU scheduler**) is the most frequently invoked scheduler in the operating system. Its primary function is to select a process from those ready to run (residing in the ready queue) and allocate the CPU to that process. The short-term scheduler is responsible for deciding which process gets to use the CPU next and for how long. This decision-making process involves choosing appropriate scheduling algorithms based on factors like process priority, CPU burst time, and the system's overall workload. The efficiency of the short-term scheduler is crucial as it directly impacts system performance and responsiveness.

Long-Term Scheduler

The **long-term scheduler**, also called the **job scheduler**, has a different role: selecting processes from the job pool (those stored on disk) and loading them into memory for execution. The long-term scheduler controls the degree of multiprogramming, which refers to the number of processes residing in memory at a given time. By carefully selecting the mix of I/O-bound and CPU-bound processes, the long-term scheduler can strive for a balanced system where both the CPU and I/O devices are utilized efficiently. The frequency of execution of the long-term scheduler is much lower than that of the short-term scheduler, as loading new processes into memory is a less frequent event compared to process switching.

Medium-Term Scheduler

The **medium-term scheduler** introduces the concept of **swapping**, which involves removing processes from memory (swapping out) and placing them on a backing store, usually a disk, to reduce the degree of multiprogramming or free up memory. Later, when appropriate, the scheduler can bring these processes back into memory (swapping in) to continue execution. Swapping serves as a mechanism to control the system load and manage memory resources more effectively. The medium-term scheduler operates at an intermediate frequency, less frequent than the short-term scheduler but more frequent than the long-term scheduler.

Briefly explain the criteria for optimizing CPU scheduling algorithms with an example.

Criteria for Optimizing CPU Scheduling Algorithms

The sources emphasize the importance of CPU scheduling algorithms and outline various criteria used to evaluate and optimize their performance. A good CPU scheduling algorithm aims to optimize system resources and provide the best possible performance for the given workload. The criteria mentioned in the sources include:

- **CPU Utilization**: This metric measures how much of the CPU's time is spent doing useful work, as opposed to being idle. Higher CPU utilization is generally desirable, as it indicates that the CPU is being used effectively.
- **Throughput**: Throughput refers to the amount of work completed per unit of time, often measured as the number of processes completed per second or per hour. A higher throughput suggests that the system is processing more work efficiently.

- **Turnaround Time**: From a process's perspective, turnaround time is the time it takes from submission to completion, including waiting time and execution time. A shorter turnaround time means processes are completed faster, leading to better user satisfaction.
- **Waiting Time**: This is the total time a process spends waiting in the ready queue before it gets access to the CPU. Minimizing waiting time is important for keeping processes responsive and preventing unnecessary delays.
- **Response Time**: In interactive systems, response time is the time it takes for a process to respond to a user request. A low response time is crucial for a system to feel interactive and responsive to user actions. The sources suggest that minimizing the variance in response time can be more important than simply minimizing the average response time for interactive systems.