Process and CPU Scheduling in Operating Systems: A Comprehensive Analysis

1. Introduction: The Core of Operating System Efficiency

Process and CPU scheduling form the bedrock of modern operating systems, orchestrating the execution of applications and ensuring the efficient utilization of computational resources.[1] These mechanisms are fundamental to achieving system responsiveness, fair allocation of CPU time among competing processes, and overall system stability.[4] This report aims to provide a detailed and expert-level analysis of process and CPU scheduling within operating systems, encompassing the theoretical underpinnings and practical implementations found in widely used systems like Unix and Windows. The scope of this report is comprehensive, covering the entire lifecycle of a process, from its creation to termination, including the critical aspects of scheduling, inter-process communication, thread management, synchronization, and deadlock handling. The intended audience for this report comprises individuals with a technical background in computer science, including students, researchers, and engineers seeking a profound understanding of these essential operating system principles. The subsequent sections will delve into the intricacies of process management, the power of threads in harnessing concurrency, the various strategies employed for CPU scheduling, the critical need for process synchronization, and the challenges posed by deadlocks, culminating in case studies that illustrate these concepts in the context of Unix and Windows operating systems.

2. Process Management: Orchestrating Application Execution

2.1. The Extended View: The 7-State Process Model

The lifecycle of a process within an operating system can be described using a state model, which illustrates the various stages a process goes through during its execution. While a 5-state model is common, the 7-state model provides a more detailed view, particularly in how it handles memory management and multitasking.[5] This extended model includes the states: New, Ready, Running, Waiting (Blocked), Ready Suspend, Blocked Suspend, and Terminated (Exit).[6]

The **New** state represents the initial phase when a process is being created. At this stage, the program code is typically located in secondary memory, such as a hard disk, awaiting admission into main memory to become a ready process.[7] This phase involves the allocation of a Process Control Block (PCB), which stores essential information about the process.[7] The 'New' state signifies the period before a program

transitions into an active entity managed by the operating system.

Once a process is admitted into main memory, it enters the **Ready** state.[7] Processes in this state are loaded into main memory and are awaiting execution by the CPU.[8] They reside in a 'Ready Queue', signifying their eligibility for CPU time, managed by the short-term scheduler.[7] The 'Ready' state is crucial for multitasking, as it represents the pool of processes actively competing for processor resources.

The **Running** state indicates that the process is currently being executed by one of the CPU cores.[7] In a single-processor system, only one process can be in the 'Running' state at any given time. The duration a process remains in this state is governed by the scheduling algorithm and the allocated time quantum, if applicable.[7] Interrupts can cause a process to transition from 'Running' back to 'Ready'.[7] This state is where the actual computation of a process occurs.

When a process in the 'Running' state requires an event to occur, such as the completion of an Input/Output (I/O) operation, it transitions to the **Waiting (Blocked)** state.[7] Processes in this state are inactive and do not consume CPU time. They are placed in a 'Device Queue' if waiting for a specific I/O device.[7] Upon completion of the awaited event, the process typically moves back to the 'Ready' state.[7] This state allows the CPU to be utilized by other processes while one is waiting for an external event.

To better manage memory, the 7-state model introduces two suspend states. The **Ready Suspend** state signifies that a process is ready to execute but is currently residing in secondary memory, having been swapped out from main memory due to memory pressure.[5] While in this state, the process is ready for execution as soon as it is loaded back into main memory.[6] The operating system might prefer to suspend a blocked process, but ready processes can be swapped out if a significant amount of memory needs to be freed.[7]

Similarly, the **Blocked Suspend** state indicates that a process is suspended in secondary memory and is waiting for a specific event to occur.[5] This state is reached when a process in the 'Waiting' state is swapped out of main memory, often due to memory constraints.[7] If the wait condition is met for a process in the 'Blocked Suspend' state, it transitions to the 'Ready Suspend' state.[7] Suspending blocked processes is often preferred over suspending ready processes as it does not immediately impact potential CPU utilization.[7]

Finally, the **Terminated (Exit)** state signifies that the process has completed its execution entirely or has been terminated by the operating system or a user.[7] In this

state, the process's resources are typically deallocated, and its PCB may be deleted.[9] The transition to 'Terminated' is the final stage of a process's lifecycle.

The transitions between these seven states are managed by different components of the operating system.[7] The long-term scheduler admits a process from the 'New' state to the 'Ready' state. The short-term scheduler dispatches a process from the 'Ready' state to the 'Running' state. A running process can transition back to 'Ready' due to time slice expiration or preemption, or to 'Waiting' if it requires an I/O operation. Upon completion of an I/O operation, a waiting process moves to the 'Ready' state. A running process transitions to 'Terminated' upon completion. The medium-term scheduler is responsible for transitions involving the suspend states. It moves a process from 'Ready' to 'Ready Suspend' or from 'Waiting' to 'Blocked Suspend' when memory becomes full. Conversely, it moves a process from 'Ready Suspend' back to 'Ready' when memory is available. A blocked suspended process moves to 'Ready Suspend' when the event it was waiting for occurs. A process in 'Ready Suspend' might transition to 'Blocked Suspend' if it needs to wait for an event. Lastly, a process in 'Blocked Suspend' can be terminated, moving it to the 'Exit' state.[7] The 7-state model, by including 'Ready Suspend' and 'Blocked Suspend' states, allows for more effective management of memory and multitasking compared to the 5-state model.[5]

2.2. The Art of Decision-Making: Process Scheduling

Process scheduling is the fundamental activity performed by the operating system to manage the execution of processes on the CPU.[1] In a multitasking environment, where multiple processes compete for the CPU, the scheduler determines which process gets to run and for how long, creating the illusion of concurrency.[4] The primary goal of process scheduling is to optimize system performance according to various objectives.[13]

Several key objectives guide the design of process scheduling algorithms. **Maximizing CPU utilization** aims to keep the CPU as busy as possible, ensuring that computational resources are not wasted.[13] **Maximizing throughput** focuses on completing the highest number of processes within a given time frame, which is particularly important for batch processing systems.[4] **Minimizing turnaround time** seeks to reduce the total time it takes for a process to complete its execution, from submission to termination.[13] **Minimizing waiting time** targets the reduction of the time a process spends in the ready queue, waiting for its turn to use the CPU, which directly impacts system responsiveness.[13] For interactive systems, **minimizing response time**, the delay between a user's request and the system's first response, is crucial for a positive user experience.[13] **Ensuring fairness** dictates that each process

should receive a fair share of CPU time, preventing any process from being indefinitely postponed.[4] Finally, **enforcing priorities** allows the operating system to give preferential treatment to processes deemed more important.[13] It is important to note that these scheduling objectives often involve trade-offs, where optimizing one objective might negatively impact another.[16]

The process scheduler is a core component of the operating system responsible for making these scheduling decisions.[2] There are typically three levels of scheduling involved. The **long-term scheduler**, also known as the job scheduler, controls the degree of multiprogramming by deciding which processes from the job queue on disk are admitted into the ready queue in main memory.[2] It is invoked relatively infrequently and aims to maintain a balanced mix of I/O-bound and CPU-bound processes to optimize resource utilization.[4] The **medium-term scheduler**, or swapper, manages the swapping of processes between main memory and secondary memory.[2] It is invoked occasionally to reduce the degree of multiprogramming, often when memory pressure is high or to handle blocked or inactive processes, thereby improving system responsiveness.[18] The **short-term scheduler**, also known as the CPU scheduler or dispatcher, is responsible for selecting one process from the ready queue to be executed by the CPU next.[2] It is invoked very frequently and its primary goal is to maximize CPU utilization and minimize response time by quickly choosing a ready process for execution.[12]

2.3. Lifecycle Management: Operations on Processes

Throughout their existence, processes undergo several operations that are crucial for managing their execution and the allocation of system resources.[23] **Process creation** is the initial operation, involving the construction of a new process for execution.[23] This can be triggered by various events, including system initialization when the computer boots, user requests to execute a program, or a running process creating new processes (child processes).[23] During creation, the operating system allocates necessary resources such as memory space and assigns a unique Process ID (PID).[28] Child processes can inherit resources from their parent process or have new resources allocated by the OS.[28]

Process termination is the operation that ends the execution of a process.[23] A process can terminate normally after completing its execution, due to errors encountered during execution, or by being explicitly terminated by its parent process or the operating system.[26] Upon termination, the operating system reclaims all resources that were allocated to the process.[28] In some cases, the termination of a parent process can lead to the termination of its child processes, a phenomenon

known as cascading termination.[30]

Scheduling/Dispatching refers to the operation of transitioning a process from the ready state to the running state.[23] This is performed by the short-term scheduler when the CPU becomes free or when a process with a higher priority becomes ready to run.[23]

Blocking occurs when a process in the running state needs to wait for an event to occur, such as the completion of an I/O operation, or when it needs to acquire a resource that is currently unavailable.[23] In this operation, the process voluntarily relinquishes the CPU and moves to the waiting (blocked) state until the event occurs or the resource becomes available.

Preemption is an operation where the operating system interrupts a process that is currently running and moves it back to the ready state.[23] This is typically done in preemptive scheduling systems to allow a higher-priority process to execute or when the running process has exceeded its allocated time slice.[23] Preemption ensures fairness and responsiveness in the system.

Suspension and **Resumption** are operations managed by the medium-term scheduler. Suspension involves moving a process from main memory to secondary memory, either from the ready state (ready suspend) or the blocked state (blocked suspend), to free up main memory resources.[23] Resumption is the reverse operation, bringing a suspended process back into main memory, either to the ready state from ready suspend or to the blocked state from blocked suspend.[23]

Finally, the **priority change** operation allows the operating system or a user to alter the priority associated with a process, which can influence its likelihood of being scheduled for execution.[23] These operations on processes are fundamental to the operating system's ability to manage concurrency, allocate resources efficiently, and ensure the smooth execution of applications.[23]

2.4. Collaboration is Key: Inter-Process Communication (IPC)

Inter-Process Communication (IPC) refers to the mechanisms provided by the operating system that allow different processes to communicate with each other and share data.[31] This is essential for enabling cooperation between processes to achieve complex tasks or to share resources effectively. Various methods exist for IPC, each with its own advantages and disadvantages.

Pipes are a basic form of IPC that provide a unidirectional or bidirectional flow of data

between processes.[31] **Anonymous pipes** are typically used for communication between related processes, such as a parent and its child, while **named pipes** (FIFOs) allow communication between unrelated processes.[31] Pipes are simple to use and are often employed for passing the output of one command as the input to another.

Message queues offer a more structured way for processes to communicate by sending and receiving messages.[31] These messages are stored in a queue, often managed by the kernel, until the receiving process retrieves them. Message queues can facilitate both synchronous and asynchronous communication and allow for message prioritization.

Shared memory provides a highly efficient way for multiple processes to communicate by granting them access to a common region of memory.[31] This method allows for very fast data exchange as processes can directly read from and write to the shared memory area. However, it requires careful synchronization mechanisms, such as semaphores or mutexes, to prevent race conditions and ensure data consistency.

Sockets are primarily used for network communication between processes, which can be running on the same machine or across a network.[31] They provide a standardized interface for communication and support various protocols like TCP (for reliable, connection-oriented communication) and UDP (for faster, connectionless communication). Sockets are fundamental for implementing client-server architectures and distributed systems.

Signals are a form of asynchronous communication where one process can notify another process about the occurrence of a specific event.[35] Signals are typically used for limited data transfer or to send control commands between processes, such as a termination request or a user interrupt.

Semaphores, while primarily used for synchronization, can also facilitate communication between processes by controlling access to shared resources that are used for data exchange.[31] They can enforce order and manage concurrent access to shared memory or files used for IPC.

Remote Procedure Calls (RPC) and Remote Method Invocation (RMI) are mechanisms that allow a process to call a procedure or method in another process, which might be located on a different machine in a distributed system.[31] These methods abstract the underlying network communication details, making it easier for developers to build distributed applications.

Each of these IPC methods offers different trade-offs in terms of efficiency, complexity, and suitability for various communication scenarios.[31] The choice of which method to use depends on factors such as the volume of data being exchanged, the frequency of communication, whether the communicating processes are related or unrelated, and whether the communication needs to span across a network. Performance requirements and the complexity of implementing necessary synchronization also play significant roles in selecting the appropriate IPC mechanism.

3. Harnessing Concurrency: The Power of Threads

3.1. Lightweight Execution Units: Threads Overview

In modern operating systems, the concept of a **thread** has become central to achieving concurrency within a process.[40] A thread can be defined as a basic unit of CPU utilization; it is a single sequential stream of execution within a process.[40] Threads are often referred to as lightweight processes because they exist within the context of a process and share the process's code, data section, and other operating system resources such as open files.[40] Each thread within a process has its own program counter, which tracks the next instruction to be executed, a set of processor registers that hold the current working values, and a stack space that is used for storing local variables and the history of function calls.[40] Additionally, each thread has a unique identifier, known as the thread ID.

3.2. Unlocking Potential: Benefits of Threads

Employing threads within a program offers several significant advantages over using multiple separate processes for achieving concurrency.[46] One key benefit is **enhanced responsiveness**; if a process is divided into multiple threads, one thread can continue to execute even if another part of the process is blocked or is performing a lengthy operation.[46] This is particularly important for interactive applications where the user interface should remain responsive even during resource-intensive tasks. Another significant advantage is **resource sharing**; threads within the same process share the same address space, making it straightforward and efficient for them to communicate and share data without the need for complex inter-process communication mechanisms.[46] **Improved economy** is also a major factor, as creating and managing threads is generally less resource-intensive compared to processes, in terms of both time and memory.[46] Furthermore, context switching between threads within the same process is typically faster than context switching between different processes because threads share the same address

space. Finally, **increased concurrency** can be achieved through the use of threads, especially on multi-core or multi-processor systems, where multiple threads from the same process can execute in parallel on different cores, potentially leading to substantial performance improvements for applications that can be parallelized.[46] Threads are particularly beneficial for applications with graphical user interfaces, for performing background processing tasks, and for handling I/O-bound operations without blocking the entire process.[48]

3.3. Kernel vs. User: Types of Threads

Threads in operating systems can be broadly categorized into two main types: user-level threads and kernel-level threads.[40] **User-level threads** are implemented and managed by a user-level thread library, without direct involvement from the operating system kernel.[40] Because thread management is done in user space, operations like thread creation and context switching are very fast as they do not require kernel intervention.[55] However, a significant drawback of user-level threads is that if one user-level thread within a process performs a blocking system call, the entire process might block, as the kernel is unaware of the individual user-level threads.[56] Additionally, user-level threads cannot inherently take advantage of true parallelism on multi-core systems, as they are all mapped to a single kernel process from the kernel's perspective.[55] They offer efficiency and portability across different operating systems, provided the corresponding thread library is available.[55]

In contrast, **kernel-level threads** are directly supported and managed by the operating system kernel.[40] The kernel maintains a thread table that keeps track of all kernel-level threads in the system.[58] Because the kernel is aware of and schedules each kernel-level thread independently, they can achieve true parallelism on multi-core systems, allowing multiple threads from the same process to run concurrently on different processors.[55] Furthermore, if one kernel-level thread blocks, other threads within the same process can continue to execute.[56] However, the overhead associated with creating and managing kernel-level threads, as well as the context switching between them, is higher compared to user-level threads because these operations require kernel intervention.[55] Kernel-level threads are generally more robust and can better utilize the capabilities of the underlying hardware and operating system.[55]

The relationship between user-level threads and kernel-level threads can be defined by different threading models. The **many-to-one model** maps multiple user threads to a single kernel thread, which is efficient but does not allow for true parallelism and can lead to process blocking. The **one-to-one model** maps each user thread to a

separate kernel thread, providing better concurrency and parallelism but with more overhead. The **many-to-many model** multiplexes multiple user threads onto an equal or smaller number of kernel threads, aiming to balance efficiency and concurrency. Modern operating systems, such as Linux and Windows, primarily use the one-to-one model to leverage the benefits of kernel-level threads for better concurrency and responsiveness.[42]

4. CPU Scheduling: Optimizing Processor Utilization

4.1. Measuring Efficiency: CPU Scheduling Criteria

The effectiveness of CPU scheduling algorithms is evaluated based on several key criteria that reflect different aspects of system performance and fairness.[13] **CPU utilization** measures the percentage of time the CPU is busy executing processes. A high CPU utilization indicates that the processor is being used efficiently, although 100% utilization might not always be optimal, especially in interactive systems where responsiveness is crucial.[13] **Throughput** is the number of processes that complete their execution per unit of time. A higher throughput signifies that the system is accomplishing more work.[4] **Turnaround time** is the total time taken by a process from its submission until its completion, encompassing waiting time, execution time, and I/O operations.[13] Minimizing turnaround time is important for overall system efficiency and user satisfaction. **Waiting time** is the total duration a process spends in the ready queue, waiting for its turn to be executed by the CPU.[13] Lower waiting times generally lead to better system responsiveness. **Response time** is the time elapsed from the submission of a request until the first response is produced, which is particularly critical for interactive systems to provide a good user experience.[13] **Fairness** ensures that each process receives a fair share of CPU time, preventing any process from being indefinitely delayed.[4] Finally, **predictability** refers to the consistency in the execution time of a process under similar system load, which is important for real-time systems where timing requirements are strict.[13] Different CPU scheduling algorithms prioritize these criteria differently, making certain algorithms more suitable for specific types of operating systems and workloads.[13]

4.2. Taking Turns or Interrupting: Preemptive vs. Non-Preemptive Scheduling

CPU scheduling algorithms can be broadly classified into two categories: preemptive and non-preemptive.[65] **Preemptive scheduling** allows the operating system to interrupt a currently running process and allocate the CPU to another process, typically based on priority or the expiration of a time quantum.[66] This approach is crucial for time-sharing systems as it ensures that no single process can monopolize

the CPU, and it allows for better responsiveness, especially for high-priority processes.[66] Preemption is essential for interactive systems to provide timely responses to user actions.[66] However, preemptive scheduling introduces overhead due to the need for context switching, which involves saving the state of the interrupted process and loading the state of the next process to be executed. Examples of preemptive scheduling algorithms include Round Robin, Priority (preemptive), and Shortest Remaining Time First (SRTF).[66]

In **non-preemptive scheduling**, once a process is allocated the CPU, it continues to run until it completes its CPU burst or voluntarily releases the CPU, for instance, by requesting an I/O operation or by terminating.[66] This method is simpler to implement and has less overhead associated with context switching compared to preemptive scheduling.[66] However, non-preemptive scheduling can lead to poor response times, especially if a long-running process occupies the CPU, potentially causing shorter processes to wait for an extended period, leading to starvation.[66] Non-preemptive scheduling is often used in batch processing systems where the overall throughput is more important than immediate responsiveness to individual tasks. Examples of non-preemptive scheduling algorithms include First-Come, First-Served (FCFS), Shortest Job First (SJF - in its non-preemptive form), and Priority (non-preemptive).[66]

4.3. A Toolkit of Strategies: CPU Scheduling Algorithms

Operating systems employ a variety of CPU scheduling algorithms to manage the execution of processes. Each algorithm has its own characteristics, advantages, and disadvantages, making it suitable for different scenarios.

First-Come, First-Served (FCFS) is one of the simplest scheduling algorithms, where processes are executed in the order they arrive in the ready queue.[73] It is a non-preemptive algorithm, meaning once a process starts executing, it continues until it finishes or blocks.[75] FCFS is easy to understand and implement.[73] However, it can suffer from the convoy effect, where a long-running process at the front of the queue can delay all subsequent shorter processes, leading to a high average waiting time.[73]

Shortest Job First (SJF) is a scheduling algorithm that selects the process with the shortest estimated burst time to execute next.[73] SJF can be either preemptive (known as Shortest Remaining Time First - SRTF) or non-preemptive.[81] It is known for minimizing the average waiting time of processes.[73] A significant disadvantage of SJF is the potential for starvation of longer processes if shorter jobs keep arriving.[73] Additionally, it requires knowing or accurately predicting the burst time of processes,

which is not always feasible.[73]

Round Robin (RR) is a preemptive scheduling algorithm designed for time-sharing systems.[86] Each process is allocated a fixed amount of CPU time, called a time quantum or time slice.[86] If a process does not complete its execution within its time quantum, it is preempted and moved to the end of the ready queue.[89] RR ensures fairness by giving an equal share of CPU time to each process and provides better response times compared to FCFS.[87] However, the performance of RR depends heavily on the size of the time quantum; a small quantum leads to frequent context switches, increasing overhead, while a large quantum can result in poor response times for interactive tasks.[87]

Priority Scheduling assigns a priority level to each process, and the process with the highest priority is executed first.[93] Processes with the same priority are typically executed in FCFS order.[93] Priority scheduling can be either preemptive or non-preemptive.[95] While it allows for prioritizing important tasks, a significant disadvantage is the potential for starvation of low-priority processes if high-priority processes continuously arrive.[94] To mitigate this, aging techniques can be used to gradually increase the priority of processes that have been waiting for a long time.[98] Priorities can be assigned statically or dynamically adjusted based on process behavior.[97]

Multi-Level Queue Scheduling (MLQ) partitions the ready queue into several queues, each with its own priority level and scheduling algorithm.[96] Processes are assigned to a queue based on their properties, such as priority, process type (e.g., interactive, batch), or memory requirements.[100] Higher-priority queues are typically processed before lower-priority queues.[101] Processes are usually permanently assigned to a queue.[100] While MLQ allows for different scheduling policies for different types of processes, it can lead to starvation of processes in lower-priority queues if higher-priority queues are always non-empty.[103]

Multi-Level Feedback Queue Scheduling (MLFQ) is an evolution of MLQ that allows processes to move between queues based on their CPU usage and behavior.[96] The algorithm aims to favor short jobs and I/O-bound processes.[107] Processes typically start in a high-priority queue, and if they use up their entire time quantum, they are moved to a lower-priority queue.[107] Conversely, processes that frequently block for I/O and thus do not use their full time slice might stay in higher-priority queues or even be moved up.[107] MLFQ can prevent starvation by employing aging, where the priority of a process that has been waiting for a long time in a lower-priority queue is increased.[107] This makes MLFQ more flexible and adaptable to different process needs compared

to MLQ.[106]

Algorithm	Preemptive?	Priority?	Starvation?	Favors?	Advantages	Disadvantages
FCFS	No	No	Yes	Long jobs	Simple to implement, fair in arrival order	High waiting time, convoy effect
SJF (Non-Preemptive)	No	Yes (burst time)	Yes	Short jobs	Minimum average waiting time	Requires burst time knowledge, potential starvation of long jobs
SRTF (Preemptive SJF)	Yes	Yes (remaining burst)	Yes	Short remaining time jobs	Minimum average waiting time, better response time than non-preemptive	Requires burst time knowledge, potential starvation of long jobs, overhead
Round Robin	Yes	No	No	All jobs equally	Fair, good for time-sharing, better response time than FCFS	Higher average waiting time, performance depends on time quantum
Priority (Non-Preemptive)	No	Yes (external/internal)	Yes	High-priority jobs	Simple, gives preference to	Starvation of low-priority jobs

					important jobs	
Priority (Preemptive)	Yes	Yes (external/internal)	Yes	High-priority jobs	Responsive to high-priority jobs	Starvation of low-priority jobs, overhead of preemption
Multi-Level Queue	Yes/No	Yes (queue)	Yes	Depends on queue algorithm	Flexible for different process types	Inflexibility of process assignment to queues, potential starvation
Multi-Level Feedback Queue	Yes	Yes (dynamic)	No	Short/Interactive jobs	Adaptable to process behavior, can prevent starvation	Complex to implement and tune

4.4. Harnessing Parallelism: Multi-Processor Scheduling

In systems with multiple processors (or CPU cores), the complexity of scheduling increases as the operating system must decide not only which process to run but also on which processor.[113] There are two main approaches to multi-processor scheduling: Symmetric Multiprocessing (SMP) and Asymmetric Multiprocessing.[113]

In **Symmetric Multiprocessing (SMP)**, each processor is self-scheduling, meaning that the scheduler for each processor independently examines the ready queue and selects a process (or thread) to execute.[113] All processors in an SMP system typically work on the same copy of the operating system and share the same main memory and I/O resources.[113] This approach inherently handles load balancing as any idle processor can pick up a process from the common ready queue. SMP also offers fault tolerance; if one processor fails, the rest of the system can continue to operate. By allowing multiple processors to work in parallel, SMP systems can achieve higher

overall performance and throughput. However, SMP requires a locking system to ensure safe access to shared resources, and there is a potential for cache incoherence issues when multiple processors access the same data. The scheduling process in SMP is also more complex than in single-processor systems.

Asymmetric Multiprocessing employs a master-slave architecture, where one processor is designated as the master server responsible for all scheduling decisions and I/O processing, while the other processors act as slave servers that only execute user code.[113] This design is simpler to implement compared to SMP as it reduces the need for complex load balancing and synchronization across multiple processors. However, asymmetric multiprocessing suffers from the critical disadvantage of having a single point of failure; if the master processor fails, the entire system halts. The master processor can also become a performance bottleneck, especially under heavy workloads. Furthermore, there is a potential for uneven workload distribution among the slave processors depending on the master's scheduling algorithm.

Load balancing is a critical aspect of multi-processor scheduling, aiming to distribute the workload evenly across all available processors to maximize system throughput and minimize response times.[113] Techniques for load balancing include **push migration**, where tasks are moved from overloaded processors to idle or less busy ones, and **pull migration**, where an idle processor takes tasks from the queue of a busy processor.[113] Load balancing can be **static**, where the workload is distributed before execution begins, or **dynamic**, where the workload is redistributed during execution based on the current load of the processors.[123] It can also be **centralized**, with a single scheduler managing all processors, or **distributed**, where each processor participates in making scheduling decisions.[123]

Processor affinity is another important consideration in multi-processor scheduling. It refers to the tendency of a process to continue running on the same processor it was previously running on.[113] This is beneficial because the processor's cache might still contain data and instructions related to that process, leading to improved performance by reducing memory access latency.[121] Processor affinity can be **soft**, where the operating system attempts to keep a process on the same processor but does not guarantee it, or **hard**, where a process is restricted to running only on a specific set of processors.[117] While load balancing aims to distribute the workload, processor affinity can enhance the performance of individual processes by exploiting cache locality, and there is often a trade-off between these two goals.

5. Process Synchronization: Ensuring Data Integrity in Concurrency

5.1. The Challenge of Concurrency: Background and Significance

In concurrent systems, where multiple processes or threads execute and access shared resources, ensuring data integrity and consistent program behavior is a significant challenge.[127] A **race condition** arises when two or more processes access and manipulate shared data concurrently, and the final outcome of the execution depends on the particular order in which these accesses occur.[127] This non-deterministic behavior can lead to data corruption and inconsistent states, making it crucial to implement mechanisms for **process synchronization**.[127] Process synchronization refers to the coordination of the execution of multiple processes to ensure that they access shared resources in a controlled and predictable manner, thereby preventing race conditions and maintaining data consistency.[127]

5.2. Protecting Shared Resources: The Critical Section Problem

The **critical section problem** is a fundamental issue in concurrent programming that deals with how to ensure that when one process is executing a critical section of code that accesses shared resources, no other process is allowed to execute in its critical section simultaneously.[127] A critical section is a segment of code where a process accesses shared resources such as variables, data structures, files, or I/O devices.[137] To ensure proper synchronization and avoid race conditions, any solution to the critical section problem must satisfy three essential requirements [132]:

- **Mutual Exclusion:** Only one process can be inside the critical section at any given time. If another process attempts to enter its critical section while one is already executing, it must wait until the first process has exited.[132]
- **Progress:** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely. Only processes that are not executing in their remainder sections can participate in the decision, and this decision must be made in a finite amount of time.[132]
- **Bounded Waiting:** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. This condition ensures that no process is starved and eventually gets access to the critical section.[132]

5.3. Software-Based Harmony: Peterson and Bakery Algorithms

Operating systems employ both software and hardware solutions to address the

critical section problem. Two classic software-based algorithms are Peterson's algorithm and the Bakery algorithm.

Peterson's Algorithm is a concurrent programming algorithm that provides a mutual exclusion solution for two processes that share a single resource.[132] It uses only shared memory for communication between the two processes. The algorithm relies on two shared variables: a boolean array flag of size two, where flag[i] being true indicates that process *i* wants to enter the critical section, and an integer variable turn that indicates whose turn it is to enter the critical section.[132] Before entering the critical section, a process sets its flag to true and sets the turn to the other process's index. It then waits in a loop as long as the other process's flag is true and it is not its turn. Upon exiting the critical section, the process resets its flag to false.[132] Peterson's algorithm satisfies all three requirements of the critical section problem: mutual exclusion, progress, and bounded waiting.[156] However, it is limited to only two processes and relies on busy waiting.[158] Moreover, due to issues with memory ordering and caching in modern architectures, Peterson's algorithm might not work correctly without specific memory barrier instructions.[158]

The **Bakery Algorithm**, developed by Leslie Lamport, is a generalization of the mutual exclusion problem for *n* processes.[135] Before entering the critical section, each process takes a "number" (a ticket) which is intended to be greater than the numbers of all other processes currently trying to enter the critical section.[164] The process with the smallest number enters the critical section first. In case two processes receive the same number, the process with the lower process ID is given priority.[164] The algorithm uses two shared data structures: a boolean array choosing to indicate if a process is in the process of choosing a number, and an integer array number to store the number chosen by each process.[167] The Bakery algorithm satisfies mutual exclusion, progress, and starvation freedom (a stronger form of bounded waiting).[164] Notably, the original algorithm does not require atomic read or write operations on memory registers.[164] However, similar to Peterson's algorithm, it involves busy waiting.[171]

5.4. Hardware Assistance: Synchronization Hardware

Many modern operating systems and hardware architectures provide built-in support for synchronization through specialized hardware instructions that can be used to implement synchronization primitives more efficiently than purely software-based approaches.[127] These instructions typically perform operations atomically, meaning they complete in a single, uninterruptible step.

The **Test-and-Set** instruction is a common hardware mechanism that atomically tests

the value of a boolean variable and sets it to true.[133] It returns the original value of the variable. This instruction can be used to implement a simple binary lock (spin lock) where a process continuously loops, testing and setting the lock until it finds it to be false, at which point it sets it to true and enters the critical section. Upon exiting, it resets the lock to false.

The **Swap** instruction atomically swaps the contents of two memory locations.[133] It can be used for mutual exclusion by using a shared lock variable and a process-local key variable. A process wanting to enter the critical section sets its key to true and then atomically swaps its key with the lock. If the lock was initially false, the key will become false, and the process can enter. The lock is set back to false upon exit.

The **Compare-and-Swap** instruction is a more powerful atomic operation that compares the content of a memory location with a given expected value. If the values are equal, it replaces the content of that memory location with a new provided value.[138] This instruction can be used to implement various lock-free and wait-free synchronization primitives, offering more flexibility and potentially better performance than traditional lock-based mechanisms.

These hardware synchronization mechanisms provide the foundation for building more complex synchronization tools and are crucial for efficient concurrency control in modern operating systems.

5.5. Classical Challenges: Problems of Synchronization

Several classical problems in concurrency illustrate the challenges of process synchronization and serve as benchmarks for evaluating synchronization mechanisms.

The **Producer-Consumer Problem** involves two types of processes, producers and consumers, that share a common, fixed-size buffer.[127] Producers generate data and place it into the buffer, while consumers retrieve data from the buffer. The core challenge is to ensure that a producer does not add data to a full buffer and a consumer does not try to remove data from an empty buffer. Additionally, access to the buffer must be synchronized to prevent race conditions when multiple producers or consumers are involved.[186] A common solution to this problem involves using semaphores to manage the state of the buffer: a counting semaphore to track the number of empty slots, another to track the number of filled slots, and a binary semaphore (mutex) to provide mutual exclusion when accessing the buffer.[186]

The **Readers-Writers Problem** deals with a shared resource (such as a database or a file) that can be accessed by multiple processes, some of which only read the data

(readers) and some of which both read and modify the data (writers).[129] The main constraints are that multiple readers can access the resource simultaneously as long as no writer is present, but when a writer is accessing the resource, no other reader or writer can access it at the same time.[194] Solutions to this problem often involve using semaphores to control access to the shared resource, with variations that might prioritize readers over writers or vice versa.[194]

The **Dining Philosophers Problem** is a classic problem that illustrates the challenges of resource sharing and the potential for deadlock in concurrent systems.[129] It involves a set of philosophers sitting around a circular table, with one chopstick placed between each pair of philosophers. To eat, a philosopher needs to pick up both the chopstick to their left and the chopstick to their right. The problem arises when multiple philosophers try to pick up their chopsticks simultaneously, potentially leading to a situation where each philosopher holds one chopstick and is waiting for the other, resulting in a deadlock.[206] Solutions often involve using semaphores to represent the chopsticks and implementing protocols that prevent deadlock, such as limiting the number of philosophers who can try to pick up chopsticks at the same time or imposing an order in which the chopsticks must be picked up.[204]

6. Deadlocks: Navigating the Perils of Resource Contention

6.1. Understanding the Landscape: System Model

In operating systems, a **deadlock** is a critical situation that occurs when two or more processes become blocked indefinitely because each process is holding onto a resource that another process needs, and none of them are willing to release their resource until they obtain the resource they are waiting for.[214] This creates a standstill where no involved process can proceed with its execution. The system model for deadlocks involves understanding the interaction between processes and resources. Processes require various resources, such as CPU time, memory, files, and I/O devices, to execute. These resources can be either shareable (e.g., read-only files) or non-shareable (e.g., printers, exclusive file access).[215] Processes typically follow a sequence of requesting a resource, using the resource, and then releasing it. However, in a multi-process environment, the competition for limited non-shareable resources can lead to deadlock if certain conditions are met simultaneously.

6.2. Identifying the Culprits: Deadlock Characterization

A deadlock situation can only arise if four necessary conditions are simultaneously true in a system [214]:

- **Mutual Exclusion:** At least one resource must be held in a non-shareable mode, meaning that only one process can use the resource at any given time. If another process requests that resource, it must wait until the resource is released.[214]
- **Hold and Wait:** A process must be holding at least one resource while it is waiting to acquire other resources that are currently held by other processes. The process does not release the resources it is holding while waiting.[214]
- **No Preemption:** Resources cannot be forcibly taken away from a process that is holding them. A resource can only be released voluntarily by the process that is holding it, typically after the process has completed its use of the resource.[214]
- **Circular Wait:** There exists a set of two or more processes where each process in the set is waiting for a resource that is held by another process in the same set, forming a circular chain of dependencies.[214] For example, process A is waiting for a resource held by process B, and process B is waiting for a resource held by process A.

If all four of these conditions are present in a system at the same time, a deadlock can occur, preventing the affected processes from making any further progress.

6.3. Strategies for Resolution: Methods for Handling Deadlocks

Operating systems employ various methods to handle deadlocks, which can be broadly categorized into four approaches: deadlock prevention, deadlock avoidance, deadlock detection and recovery, and ignoring the problem.[215]

6.4. Proactive Measures: Deadlock Prevention

Deadlock prevention aims to ensure that the system will never enter a deadlock state by preventing at least one of the four necessary conditions from occurring.[215] **Eliminating mutual exclusion** is not always feasible, as some resources are inherently non-shareable. However, for certain resources like printers, the technique of spooling can be used, where print jobs are queued in memory, allowing processes to proceed without waiting for the actual printer.[215] To **eliminate hold and wait**, one approach is to require a process to request all the resources it will need before starting its execution. If all resources are available, they are allocated to the process; otherwise, the process waits. Another method is to require a process to release all resources it is currently holding before making a new request.[215] To **eliminate no preemption**, if a process holding some resources requests another resource that cannot be immediately allocated, all the resources currently held by the process can be preempted (forcibly taken away) and released. The process will then need to request all its resources again, including the additional one.[215] This is only feasible for

resources whose state can be easily saved and restored. **Eliminating circular wait** can be achieved by imposing a total ordering on all resource types. Processes are then required to request resources in an increasing order of their enumeration. This prevents the formation of a circular chain of processes waiting for resources held by each other.[215] Among these techniques, eliminating circular wait by ordering resources is often considered the most practical.

6.5. Cautious Allocation: Deadlock Avoidance

Deadlock avoidance is a strategy where the operating system allows the possibility of deadlock but makes choices during resource allocation to avoid ever reaching a deadlock state.[214] This approach requires the system to have information about the maximum resources that a process might need during its execution.[218] The system then analyzes each resource request to determine if granting it would lead to a "safe state" or an "unsafe state".[214] A **safe state** is one in which there exists a sequence of resource allocations that can satisfy the maximum needs of all processes without leading to a deadlock.[214] If granting a request would result in an unsafe state, the request is denied, even if the resource is currently available.[217] The **Banker's algorithm** is a well-known deadlock avoidance algorithm that operates on this principle.[215] It simulates the allocation of resources to processes based on their declared maximum needs and checks if the system remains in a safe state after each allocation.[223]

6.6. Identifying the Impasse: Deadlock Detection

Deadlock detection involves allowing deadlocks to occur and then employing algorithms to identify if a deadlock has indeed taken place.[214] Once a deadlock is detected, the system can then take steps to recover from it. Two common algorithms for deadlock detection are based on graphs. The **Resource Allocation Graph (RAG)** can be used in systems where each resource has a single instance.[214] A cycle in the RAG, which shows processes and resources along with request and allocation edges, indicates the presence of a deadlock.[214] The **Wait-For Graph (WFG)** is used in systems where resources can have multiple instances.[214] The WFG is constructed by removing the resource nodes from the RAG and collapsing the edges. An edge from process P1 to P2 in the WFG signifies that P1 is waiting for a resource held by P2. A cycle in the WFG indicates a deadlock.[214] An adaptation of the Banker's algorithm can also be used for deadlock detection in systems with multiple resource instances.[229]

6.7. Breaking the Cycle: Recovery from Deadlock

Once a deadlock is detected, the operating system needs to employ **recovery from deadlock** techniques to break the deadlock and allow the system to return to a normal operating state.[215] There are several common approaches to deadlock recovery. **Process termination** involves aborting one or more of the processes involved in the deadlock.[218] One option is to abort all deadlocked processes, which guarantees to break the deadlock but can result in a significant loss of work. Another approach is to abort one process at a time until the deadlock cycle is eliminated. **Resource preemption** involves forcibly taking resources away from one or more of the deadlocked processes and allocating them to other processes so that the deadlock cycle is broken.[218] This method raises issues such as selecting which resources and processes to preempt (victim selection), what to do with the preempted process (rollback, restart), and ensuring that starvation does not occur (where the same process is repeatedly chosen as a victim). **Rollback** is a technique where the state of the deadlocked processes is rolled back to a previous safe state, and they are restarted from that point. This requires the system to maintain information about the past states of processes.[235] Another technique used in real-time systems is **priority inversion**, where the priority of a low-priority process holding a needed resource is temporarily increased to allow it to complete and release the resource.[235]

7. Case Study: Process Scheduling in the Unix Ecosystem

The traditional Unix operating system employs a process scheduling algorithm that falls under the general category of **round robin with multilevel feedback**.[240] This means that when the kernel schedules a process and its allocated **time quantum** (or time slice) expires, the process is preempted and added to one of several **priority queues**.[240] The scheduler uses the relative time of execution and the priority associated with each process to determine which process to schedule next.[241]

The scheduling algorithm, often referred to as schedule_process, is executed at the conclusion of a context switch. It selects the highest priority process from those that are "ready to run" and "loaded in memory," as well as those that have been "preempted".[241] If multiple processes have the same highest priority, the scheduler typically picks the one that has been "ready to run" for the longest duration, following a round robin policy.[241]

Each process in Unix has a priority value stored in its process table entry. This priority is dynamically calculated based on the recent CPU usage of the process; a lower numerical priority value indicates a higher scheduling priority (meaning the process is more likely to be scheduled).[241] The range of process priorities is divided into two

classes: user priorities and kernel priorities. User priorities are for processes that were preempted upon returning from kernel mode to user mode, while kernel priorities are typically achieved in the sleep algorithm.[241] User priorities are numerically lower than a certain threshold, and kernel priorities are numerically higher.[241]

The kernel calculates process priorities in several situations. When a process is about to go to sleep, it is assigned a priority based on the reason for the sleep; processes sleeping in lower-level algorithms that might cause more system bottlenecks tend to receive a higher priority upon waking.[242] When a process returns from kernel mode to user mode, its priority is typically lowered to a user-level priority to penalize it for using kernel resources and to ensure fairness to other processes.[241] Additionally, the clock handler in Unix adjusts the priorities of all processes in user mode at regular intervals (e.g., every second on System V) and triggers the scheduling algorithm to prevent a single process from monopolizing the CPU.[241] The recent CPU usage of a running process is incremented with every clock tick, and periodically (e.g., once a second), the clock handler applies a decay function to these usage values and then recalculates the priorities of all user-mode processes.[241]

User processes in Unix can influence their scheduling priority using the nice system call.[241] The nice value provided by the user is added to the base priority of the process in the priority calculation, allowing users to voluntarily decrease the priority of their processes to be "nicer" to others.[241] Only the superuser has the privilege to supply negative nice values, which would increase the priority of a process.[241] Child processes created using the fork system call inherit the nice value of their parent process.[241]

Modern Unix-based systems, such as Linux, have evolved beyond the traditional Unix scheduler. The Linux kernel, for example, implements the process scheduler as a core part of the kernel itself, within the __schedule() function.[244] Each processor in a Linux system has its own runqueue, which is a list of processes (more accurately, threads) waiting to run on that CPU.[244] The scheduler decides which thread on the runqueue should run next.[244] Linux utilizes the Completely Fair Scheduler (CFS) as a common scheduling algorithm, which aims to provide fair CPU time to all runnable processes.[73] Tools like perf, SystemTap, dtrace, and sysdig can be used to obtain in-depth information about process scheduling in Linux.[245] The /proc filesystem also provides a wealth of scheduling-related data, and the top command offers a real-time view of process scheduling information.[245]

8. Case Study: Process Scheduling in the Windows Environment

Windows operating systems, including Windows 11, employ a priority-driven, preemptive scheduling algorithm to manage the execution of threads.[1] The system scheduler, often referred to as the dispatcher, ensures that the highest-priority runnable thread always receives the next processor time slice.[247] This is a preemptive system, meaning that a higher-priority thread that becomes ready to run can interrupt a currently running thread, even if the latter has not finished its allocated time.[248]

Windows utilizes a 32-level priority scheme for thread scheduling, ranging from 0 to 31.[248] These priority levels are divided into three classes: **sixteen real-time levels** (16 through 31), which have the highest priority and are never adjusted by the system; **fifteen variable levels** (1 through 15), which are dynamic and subject to temporary priority boosts; and **one system level** (0), reserved for memory management tasks.[248]

From the Windows API perspective, processes are organized by their **priority class**, assigned at creation time. These classes include Real-time, High, Above Normal, Normal, Below Normal, and Idle.[248] Within each process, individual threads have a **relative priority** within their class, such as Time-critical, Highest, Above-normal, Normal, Below-normal, Lowest, and Idle.[248] Each thread has a **base priority** derived from its process priority class and its relative thread priority. However, scheduling decisions are made based on the **current priority** of the thread, which can be temporarily increased by the system through a mechanism called priority boosting.[248]

The Windows scheduler maintains a **dispatcher database** to keep track of all ready threads and the state of each processor.[248] On multi-processor systems, there are per-processor ready queues to enhance scalability.[248] Windows uses a constant-time algorithm to find the highest-priority ready thread by examining a bitmask called the **ready summary**, which indicates which priority levels have threads ready to run.[248]

Windows employs **priority boosting** to enhance the responsiveness of interactive applications and to prevent certain threads from being starved of CPU time.[246] Priority boosts are typically applied to threads in the dynamic priority range (0 through 15) under various circumstances, such as upon completion of I/O operations, after a thread has been waiting for an executive event or semaphore, when a thread in the foreground process completes a wait operation, when GUI threads wake up due to windowing activity, and when a ready-to-run thread has not been running for an extended period. Real-time priority threads (16-31) are never boosted. Windows also features the MultiMedia Class Scheduler Service (MMCSS) to ensure smooth playback of multimedia applications by boosting the priority of their threads.

CPU affinity in Windows allows restricting a thread or a process to run only on a

specific set of processors in a multi-processor system.[246] While by default, threads can run on any available processor, applications can set an affinity mask to improve performance by leveraging cache locality or to isolate workloads. Affinity can be set per thread, per process, or even per job object. Windows attempts to schedule threads on the most optimal processor, considering the thread's ideal processor (a preferred processor) and the last processor it ran on.

Users and administrators can observe and manage process scheduling in Windows using tools like **Task Manager** and **Process Explorer**, which allow viewing and sometimes modifying process and thread priorities and affinities.[248] The start command can be used to launch processes with a specific priority class. For automated task scheduling, Windows provides the **Task Scheduler**, which allows running scripts or programs at specific times or in response to certain events.[250] The at and schtasks commands can also be used for scheduling tasks from the command line.[251]

9. Conclusion: Towards Efficient and Reliable Concurrent Systems

This report has provided a comprehensive analysis of process and CPU scheduling within operating systems, highlighting the fundamental principles and practical implementations in Unix and Windows. The intricate dance of managing process states, scheduling CPU time, facilitating inter-process communication, and harnessing the power of threads is crucial for achieving efficient resource utilization, system responsiveness, and fairness in modern computing environments. The various scheduling algorithms, from the simplicity of FCFS to the adaptability of MLFQ, each offer different trade-offs in optimizing system performance based on specific criteria and workload characteristics. The challenges of process synchronization, particularly the critical section problem, necessitate robust mechanisms like Peterson's and Bakery algorithms, hardware synchronization primitives, and semaphores to ensure data integrity in concurrent execution. Finally, understanding and addressing the complexities of deadlocks through prevention, avoidance, detection, and recovery techniques is vital for maintaining system stability and reliability. The case studies on Unix and Windows reveal the distinct approaches taken by these operating systems in managing their computational resources, reflecting their respective design philosophies and target user bases. As computing continues to evolve with multi-core architectures, real-time requirements, and increasing demands for energy efficiency, the principles and techniques of process and CPU scheduling remain at the forefront of operating system design and development, driving ongoing innovation in concurrency management.

Works cited

1. www.tutorialspoint.com, accessed on April 20, 2025, https://www.tutorialspoint.com/operating_system/os_process_scheduling.htm#:~:text=Definition,of%20a%20Multiprogramming%20operating%20systems.
2. Process Scheduler in Operating System | GATE Notes - BYJU'S, accessed on April 20, 2025, https://byjus.com/gate/process-scheduler-in-operating-system-notes/
3. Process Scheduling in Operating Systems - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/operating_system/os_process_scheduling.htm
4. Scheduling (computing) - Wikipedia, accessed on April 20, 2025, https://en.wikipedia.org/wiki/Scheduling_(computing)
5. brainly.com, accessed on April 20, 2025, https://brainly.com/question/39230363#:~:text=The%20five%2Dstate%20model%20of,management%20of%20memory%20and%20multitasking.
6. Differentiate Between 5 State and 7 State Process Models - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/differentiate-between-5-state-and-7-state-process-models
7. 7 State Process Model, accessed on April 20, 2025, https://avcton.github.io/Literature/Operating-Systems/7-State-Process-Model
8. 3.2: Process States - Engineering LibreTexts, accessed on April 20, 2025, https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics/03%3A__Processes_Concepts/3.2%3A_Process_States
9. States of a Process in Operating Systems | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/states-of-a-process-in-operating-systems/
10. Seven-state transition diagram The NEW, TERMINATED, READY, RUNNING, and... - ResearchGate, accessed on April 20, 2025, https://www.researchgate.net/figure/Seven-state-transition-diagram-The-NEW-TERMINATED-READY-RUNNING-and-BLOCKED-states_fig9_268347340
11. process State Models | PPT - SlideShare, accessed on April 20, 2025, https://www.slideshare.net/slideshow/operating-systems-254856024/254856024
12. Process Schedulers in Operating System | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/process-schedulers-in-operating-system/
13. CPU Scheduling Criteria - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/cpu-scheduling-criteria/
14. Process and Thread Scheduling - Computer Science | UC Davis Engineering, accessed on April 20, 2025, https://www.cs.ucdavis.edu/~pandey/Teaching/ECS150/Lects/05scheduling.pdf
15. Operating Systems: Lecture 8: Process Scheduling (1), accessed on April 20, 2025, http://staff.um.edu.mt/csta1/courses/lectures/csm202/os8.html
16. Q. what is Process Scheduling? Define its objectives. Ans., accessed on April 20, 2025, https://dspmuranchi.ac.in/pdf/Blog/process%20scheduling.pdf
17. What is a scheduler in OS? - Design Gurus, accessed on April 20, 2025,

https://www.designgurus.io/answers/detail/what-is-a-scheduler-in-os
18. Process Schedulers In Operating System: Types, Functions & Algorithms - PW Skills, accessed on April 20, 2025, https://pwskills.com/blog/process-schedulers-in-operating-system/
19. Process Scheduler: Job and Process States | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/process-scheduler-job-and-process-status/
20. Understanding Process Schedulers in Operating Systems - DigiiMento, accessed on April 20, 2025, https://digiimento.com/understanding-process-schedulers-in-operating-systems/
21. CSC 553 Operating Systems Types of Scheduling, accessed on April 20, 2025, https://home.adelphi.edu/~siegfried/cs553/553l9.pdf
22. Fundamentals of Operating Systems: Process Scheduling Cheatsheet - Codecademy, accessed on April 20, 2025, https://www.codecademy.com/learn/fundamentals-of-operating-systems/modules/os-process-scheduling/cheatsheet
23. Operations on Processes | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/operations-on-processes/
24. Operations on Process in OS- Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/operations-on-process-in-os/
25. Introduction of Process Management | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/introduction-of-process-management/
26. Operations on Processes - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/operating_system/os_operations_on_processes.htm
27. Different Operations on Processes - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/different-operations-on-processes
28. Operations on Process in OS - Dextutor Operating System, accessed on April 20, 2025, https://dextutor.com/operations-on-process-in-os/
29. Operations on Processes, accessed on April 20, 2025, https://blogs.30dayscoding.com/blogs/os/process-management/processes/operations-on-processes/
30. Operating Systems-Unit-2-20A05402T-Process Creation-Process Termination - YouTube, accessed on April 20, 2025, https://m.youtube.com/watch?v=axGx4M2OpmE&pp=ygURI29wZXJhdGlvbnByb2Nlc3M%3D
31. Methods in Inter process Communication | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/methods-in-interprocess-communication/
32. 5.4: Interprocess Communication - Engineering LibreTexts, accessed on April 20, 2025, https://eng.libretexts.org/Courses/Delta_College/Operating_System%3A_The_Basics/05%3A_Process_Synchronization/5.4%3A_Interprocess_Communication
33. Inter Process Communication (IPC) | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/inter-process-communication-ipc/

34. Interprocess communications - Win32 apps | Microsoft Learn, accessed on April 20, 2025, https://learn.microsoft.com/en-us/windows/win32/ipc/interprocess-communications
35. Interprocess communication (IPC) and synchronization | Operating Systems Class Notes, accessed on April 20, 2025, https://library.fiveable.me/operating-systems/unit-2/interprocess-communication-ipc-synchronization/study-guide/MytfOfYD1RcgAzlV
36. Inter-Process Communication in Operating Systems - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/operating_system/os_inter_process_communication.htm
37. Inter-process communication - Wikipedia, accessed on April 20, 2025, https://en.wikipedia.org/wiki/Inter-process_communication
38. Chapter 5 Interprocess Communication Mechanisms, accessed on April 20, 2025, https://tldp.org/LDP/tlk/ipc/ipc.html
39. Interprocess Communication in Operating System - Educators - Digilent Forum, accessed on April 20, 2025, https://forum.digilent.com/topic/21747-interprocess-communication-in-operating-system/
40. Threads in Operating System (OS) - Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/operating-system/threads-in-operating-system/
41. Threads and its Types in Operating System | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/threads-and-its-types-in-operating-system/
42. Operating Systems: Threads, accessed on April 20, 2025, https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html
43. Thread in Operating System | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/thread-in-operating-system/
44. Thread (computing) - Wikipedia, accessed on April 20, 2025, https://en.wikipedia.org/wiki/Thread_(computing)
45. Threads And Concurrency - Operating Systems - OMSCS Notes, accessed on April 20, 2025, https://www.omscs-notes.com/operating-systems/threads-and-concurrency/
46. Benefits of threads - IBM, accessed on April 20, 2025, https://www.ibm.com/docs/ssw_aix_71/com.ibm.aix.genprogc/benefits_threads.htm
47. What Are the Advantages of Using Threads? | Lenovo US, accessed on April 20, 2025, https://www.lenovo.com/us/en/glossary/thread/
48. Why should I use a thread vs. using a process? - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/617787/why-should-i-use-a-thread-vs-using-a-process
49. Advantages of Thread Over Process | PDF - Scribd, accessed on April 20, 2025, https://www.scribd.com/document/517335232/Notes

50. Threads vs. Processes: How They Work Within Your Program - Backblaze, accessed on April 20, 2025, https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/
51. Difference between Process and Thread | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/difference-between-process-and-thread/
52. What are the importance of processes and threads : r/computerscience - Reddit, accessed on April 20, 2025, https://www.reddit.com/r/computerscience/comments/1b1y7jv/what_are_the_importance_of_processes_and_threads/
53. Concurrency: Processes vs Threads - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/4315292/concurrency-processes-vs-threads
54. Operating System: Introduction to Thread. User-Level and kernel Level threads - YouTube, accessed on April 20, 2025, https://www.youtube.com/watch?v=53VP9V8-PwU
55. Difference between User Level Threads and Kernel Level Threads - Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/user-level-threads-and-kernel-level-threads/
56. Difference between User Level thread and Kernel Level thread - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/difference-between-user-level-thread-and-kernel-level-thread/
57. User Level Threads and Kernel Level Threads - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/user-level-threads-and-kernel-level-threads
58. User and kernel level threads, accessed on April 20, 2025, http://www.cs.iit.edu/~cs561/cs450/ChilkuriDineshThreads/dinesh's%20files/User%20and%20Kernel%20Level%20Threads.html
59. Difference between user-level and kernel-supported threads? - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/15983872/difference-between-user-level-and-kernel-supported-threads
60. Processor Management (Part 1: Threads)! User-Level vs. Kernel-Level Threads!, accessed on April 20, 2025, https://people.engr.tamu.edu/bettati/Courses/410/2014A/Slides/threads.pdf
61. User/Kernel level threads : r/compsci - Reddit, accessed on April 20, 2025, https://www.reddit.com/r/compsci/comments/b4mswk/userkernel_level_threads/
62. Why are user level threads faster than kernel level threads? - Super User, accessed on April 20, 2025, https://superuser.com/questions/669883/why-are-user-level-threads-faster-than-kernel-level-threads
63. linux - How user level thread talk with kernel level thread - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/73675665/how-user-level-thread-talk-with-kernel-level-thread

64. Questions about user and kernel level threads, and Unix and Windows implementations. : r/compsci - Reddit, accessed on April 20, 2025, https://www.reddit.com/r/compsci/comments/tumgfo/questions_about_user_and_kernel_level_threads_and/
65. Different Types of Non-Preemptive CPU Scheduling Algorithms - Turing, accessed on April 20, 2025, https://www.turing.com/kb/different-types-of-non-preemptive-cpu-scheduling-algorithms
66. Preemptive and Non-Preemptive Scheduling | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/preemptive-and-non-preemptive-scheduling/
67. Difficulty understanding pre-emptive vs non-preemptive CPU scheduling, accessed on April 20, 2025, https://cs.stackexchange.com/questions/47926/difficulty-understanding-pre-emptive-vs-non-preemptive-cpu-scheduling
68. Preemptive and Non-Preemptive Scheduling in Operating Systems - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/operating_system/os_preemptive_and_non_preemptive_scheduling.htm
69. Operating Systems: CPU Scheduling, accessed on April 20, 2025, https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_CPU_Scheduling.html
70. Different Types of Non Preemptive CPU Scheduling Algorithms - Sourcebae, accessed on April 20, 2025, https://sourcebae.com/blog/different-types-of-non-preemptive-cpu-scheduling-algorithms/
71. What is the difference between preemptive and non-preemptive scheduling?, accessed on April 20, 2025, https://stackoverflow.com/questions/56928159/what-is-the-difference-between-preemptive-and-non-preemptive-scheduling
72. Difference Between Preemptive and Non-Preemptive Scheduling - YouTube, accessed on April 20, 2025, https://www.youtube.com/watch?v=UlpgVptO5Gk
73. Job Scheduling Algorithms: Which Is Best For Your Workflow? - Redwood Software, accessed on April 20, 2025, https://www.redwood.com/article/job-scheduling-algorithms/
74. First Come First Serve (FCFS) Scheduling - Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/first-come-first-serve/
75. FCFS – First Come First Serve CPU Scheduling - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/first-come-first-serve-cpu-scheduling-non-preemptive/
76. Program for FCFS CPU Scheduling | Set 1 - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/program-for-fcfs-cpu-scheduling-set-1/
77. FCFS Scheduling in OS - DataFlair, accessed on April 20, 2025, https://data-flair.training/blogs/fcfs-scheduling-in-os/

78. FCFS Scheduling Algorithm in Operating Systems - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/operating_system/os_fcfs_scheduling_algorithm.htm
79. FCFS Scheduling - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/fcfs-scheduling
80. SJF Scheduling : Explained - Tutorial - takeUforward, accessed on April 20, 2025, https://takeuforward.org/operating-system/sjf-scheduling-explained/
81. Shortest Job First or SJF CPU Scheduling - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/shortest-job-first-or-sjf-cpu-scheduling/
82. C Program for Shortest Job First (SJF) Scheduling Algorithm | Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/sjf-scheduling-in-c/
83. Shortest Job First Scheduling in Operating Systems - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/operating_system/os_shortest_job_first_scheduling.htm
84. Shortest job next - Wikipedia, accessed on April 20, 2025, https://en.wikipedia.org/wiki/Shortest_job_next
85. Program for Shortest Job First (or SJF) CPU Scheduling | Set 1 (Non- preemptive), accessed on April 20, 2025, https://www.geeksforgeeks.org/program-for-shortest-job-first-or-sjf-cpu-scheduling-set-1-non-preemptive/
86. www.scaler.com, accessed on April 20, 2025, https://www.scaler.com/topics/round-robin-scheduling-in-os/#:~:text=The%20Round%20robin%20scheduling%20algorithm,every%20process%20gets%20executed%20cyclically.
87. Round Robin Program in C - NxtWave, accessed on April 20, 2025, https://www.ccbp.in/blog/articles/round-robin-program-in-c
88. Round-robin scheduling - Wikipedia, accessed on April 20, 2025, https://en.wikipedia.org/wiki/Round-robin_scheduling
89. What is Round Robin Scheduling in OS? - Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/round-robin-scheduling-in-os/
90. Scheduling Algorithms - OSDev Wiki, accessed on April 20, 2025, https://wiki.osdev.org/Scheduling_Algorithms
91. Program for Round Robin Scheduling for the Same Arrival Time - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/program-for-round-robin-scheduling-for-the-same-arrival-time/
92. Round Robin Scheduling in Operating System | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/round-robin-scheduling-in-operating-system/
93. Priority CPU Scheduling with different arrival time – Set 2 ..., accessed on April 20, 2025, https://www.geeksforgeeks.org/priority-cpu-scheduling-with-different-arrival-time-set-2/

94. Priority Scheduling Algorithm in OS (Operating System) - Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/operating-system/priority-scheduling-algorithm/
95. Priority Scheduling in Operating System - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/priority-scheduling-in-operating-system/
96. Process Scheduling Algorithms in Operating Systems - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm
97. Priority Scheduling Algorithm in Operating Systems - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/operating_system/os_priority_scheduling_algorithm.htm
98. Priority Scheduling Algorithm in Operating System - DataFlair, accessed on April 20, 2025, https://data-flair.training/blogs/priority-scheduling-algorithm-in-operating-system/
99. Dynamic priority scheduling - Wikipedia, accessed on April 20, 2025, https://en.wikipedia.org/wiki/Dynamic_priority_scheduling
100. Multilevel queue - Wikipedia, accessed on April 20, 2025, https://en.wikipedia.org/wiki/Multilevel_queue
101. Multilevel Queue Scheduling | MLQ CPU Scheduling - PrepBytes, accessed on April 20, 2025, https://www.prepbytes.com/blog/queues/multilevel-queue-mlq-cpu-scheduling/
102. Multilevel Queue (MLQ) CPU Scheduling - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/multilevel-queue-mlq-cpu-scheduling/
103. Multilevel Queue Scheduling in Operating Systems - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/operating_system/os_multilevel_queue_scheduling.htm
104. Multilevel Queue Scheduling in Operating System - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/multilevel-queue-scheduling-in-operating-system
105. Difference between Multi Level Queue Scheduling (MLQ) and Priority Scheduling, accessed on April 20, 2025, https://www.geeksforgeeks.org/difference-between-multi-level-queue-scheduling-mlq-and-priority-scheduling/
106. Difference between Multilevel Queue (MLQ) and Multi Level Feedback Queue (MLFQ) CPU Scheduling Algorithms | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/difference-between-multilevel-queue-mlq-and-multi-level-feedback-queue-mlfq-cpu-scheduling-algorithms/
107. Multilevel feedback queue - Wikipedia, accessed on April 20, 2025, https://en.wikipedia.org/wiki/Multilevel_feedback_queue
108. Multilevel Feedback Queue Scheduling in C, C++ and Python - GitHub,

accessed on April 20, 2025, https://github.com/iabdullah215/Multilevel-Feedback-Queue-Scheduling
109. Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling ..., accessed on April 20, 2025, https://www.geeksforgeeks.org/multilevel-feedback-queue-scheduling-mlfq-cpu-scheduling/
110. 8: Scheduling: The Multi-Level Feedback Queue - GitHub Pages, accessed on April 20, 2025, https://ceunican.github.io/aos/08.Scheduling_The_Multi-level_Feedback_queue.pdf
111. Multi-Level Feedback Queues - Computer Science, accessed on April 20, 2025, https://www.cs.usfca.edu/~mmalensek/cs326/schedule/lectures/326-L11.pdf
112. OS19b - Solved example | Multilevel Feedback Queue Scheduling - YouTube, accessed on April 20, 2025, https://www.youtube.com/watch?v=y7LrZms1azA
113. Multi-Processor Scheduling | Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/operating-system/multi-processor-scheduling/
114. Multiple processor scheduling - Wisemonkeys, accessed on April 20, 2025, https://wisemonkeys.info/blogs/Multiple-processor-scheduling
115. Multiprocessor and Real Time Scheduling - Florida State University, accessed on April 20, 2025, https://www.cs.fsu.edu/~baker/opsys/notes/mprtsched.html
116. Multiple Processors Scheduling in Operating System - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/multiple-processors-scheduling-in-operating-system
117. Multiple-Processor Scheduling in Operating System - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/multiple-processor-scheduling-in-operating-system/
118. 15.2: Multiprocessor Scheduling - Engineering LibreTexts, accessed on April 20, 2025, https://eng.libretexts.org/Courses/Delta_College/Introduction_to_Operating_Systems/15%3A_Multiprocessor_Scheduling/15.02%3A_Multiprocessor_Scheduling
119. Multiprocessor Scheduling - OSDev Wiki, accessed on April 20, 2025, https://wiki.osdev.org/Multiprocessor_Scheduling
120. CPU Scheduling in Operating Systems - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/cpu-scheduling-in-operating-systems/
121. Multiprocessor Scheduling (Advanced) - cs.wisc.edu, accessed on April 20, 2025, https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-multi.pdf
122. Scheduling on multiple cores with each list in each processor vs one list that all processes share - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/61738710/scheduling-on-multiple-cores-with-each-list-in-each-processor-vs-one-list-that-a
123. Scheduling and Load Balancing in Distributed System ..., accessed on April 20, 2025, https://www.geeksforgeeks.org/scheduling-and-load-balancing-in-distributed-sy

stem/
124. An Efficient Algorithm for Load Balancing in Multiprocessor Systems - The Science and Information (SAI) Organization, accessed on April 20, 2025, https://thesai.org/Downloads/Volume9No3/Paper_24-An_Efficient_Algorithm_for_Load_Balancing.pdf
125. 1 Load Balancing / MultiProcessor Scheduling, accessed on April 20, 2025, https://courses.grainger.illinois.edu/cs598csc/sp2009/lectures/lecture_5.pdf
126. Chapter 1 Introduction to Scheduling and Load Balancing, accessed on April 20, 2025, http://www.diag.uniroma1.it/~ciciani/DIDATTICA/ARCHITETTURE/SchedLB.pdf
127. Introduction of Process Synchronization | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/introduction-of-process-synchronization/
128. Process Synchronisation in OS - Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/operating-system/process-synchronization-in-os/
129. Process Synchronization in Operating Systems: A Comprehensive Guide - Bito AI, accessed on April 20, 2025, https://bito.ai/resources/process-synchronization-in-operating-systems-a-comprehensive-guide/
130. Process Synchronization in Operating Systems - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/operating_system/os_process_synchronization.htm
131. PROCESS SYNCHRONIZATION IN OPERATING SYSTEM A SURVEY OF TECHNIQUES - IRJMETS, accessed on April 20, 2025, https://www.irjmets.com/uploadedfiles/paper//issue_11_november_2024/64484/final/fin_irjmets1732690175.pdf
132. Operating Systems: Process Synchronization, accessed on April 20, 2025, https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_Synchronization.html
133. Chapter 7: Process Synchronization Background, accessed on April 20, 2025, https://www.cs.kent.edu/~farrell/osf03/lectures/ch7-2up.pdf
134. Chapter 7: Process Synchronization Background - GMU CS Department, accessed on April 20, 2025, https://cs.gmu.edu/~setia/cs571-F01/slides/conc-prog.pdf
135. Module 7: Process Synchronization Background, accessed on April 20, 2025, https://i.cs.hku.hk/~fcmlau/POS99/lectures/mod7.2.pdf
136. Chapter 7: Process Synchronization Background, accessed on April 20, 2025, https://www.cs.umbc.edu/courses/undergraduate/421/spring03/slides/ch7-2.pdf
137. Critical Section Problem in OS (Operating System) - Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/critical-section-in-os/
138. What is the critical section problem in operating systems?, accessed on April 20, 2025, https://how.dev/answers/what-is-the-critical-section-problem-in-operating-systems

139. Critical Section Problem in OS (Operating System) | Hero Vired, accessed on April 20, 2025, https://herovired.com/learning-hub/topics/critical-section-problem-in-os/
140. Critical Section in Synchronization | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/g-fact-70/
141. Critical Section in OS (Operating System) - PrepBytes, accessed on April 20, 2025, https://www.prepbytes.com/blog/operating-system/critical-section-in-os/
142. Critical Section Problem - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/critical-section-problem
143. Critical section - Wikipedia, accessed on April 20, 2025, https://en.wikipedia.org/wiki/Critical_section
144. Operating Systems - Lecture #8: Critical Sections - the University of Warwick, accessed on April 20, 2025, https://warwick.ac.uk/fac/sci/physics/research/condensedmatt/imr_cdt/students/david_goodwin/teaching/operating_systems/l8_criticalsection2013.pdf
145. Critical Regions in Operating System - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/critical-regions-in-operating-system/
146. Is the definition of Critical Section wrong or being misused? - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/70126330/is-the-definition-of-critical-section-wrong-or-being-misused
147. Mutual Exclusion in Synchronization - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/mutual-exclusion-in-synchronization/
148. What is progress and bounded waiting in critical section? - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/33143779/what-is-progress-and-bounded-waiting-in-critical-section
149. What is progress and bounded waiting in 'critical section algorithm'? - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/25962097/what-is-progress-and-bounded-waiting-in-critical-section-algorithm
150. Bounded waiting and progress requirements of critical section problem solution based on exchange instruction, accessed on April 20, 2025, https://cs.stackexchange.com/questions/60652/bounded-waiting-and-progress-requirements-of-critical-section-problem-solution-b
151. 2.3 Critical Section Problem | Mutual Exclusion | Progress | Bounded Waiting - YouTube, accessed on April 20, 2025, https://www.youtube.com/watch?v=GN6UfA03oiU
152. CRITICAL SECTION Problem || Mutual Exclusion || Bounded Waiting || Progress || Operating System - YouTube, accessed on April 20, 2025, https://www.youtube.com/watch?v=O61sGf62VKo
153. CS5460: Operating Systems, accessed on April 20, 2025, https://my.eng.utah.edu/~cs5460/slides/Lecture08.pdf
154. en.wikipedia.org, accessed on April 20, 2025, https://en.wikipedia.org/wiki/Peterson%27s_algorithm#:~:text=Peterson's%20algor

ithm%20(or%20Peterson's%20solution,only%20shared%20memory%20for%20communication.
155. Peterson's Algorithm in Process Synchronization | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/petersons-algorithm-in-process-synchronization/
156. Peterson's algorithm - Wikipedia, accessed on April 20, 2025, https://en.wikipedia.org/wiki/Peterson%27s_algorithm
157. Peterson's Solution - Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/petersons-solution/
158. Peterson's Solution to the Critical Section Problem - Jyotiprakash's Blog, accessed on April 20, 2025, https://blog.jyotiprakash.org/petersons-solution-to-the-critical-section-problem
159. Peterson's Algorithm in Process Synchronization - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/petersons-algorithm-in-process-synchronization
160. N process Peterson algorithm | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/n-process-peterson-algorithm/
161. Where is Peterson's algorithm used in the real world? - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/76869258/where-is-petersons-algorithm-used-in-the-real-world
162. Peterson's Solution - YouTube, accessed on April 20, 2025, https://www.youtube.com/watch?v=gYCiTtgGR5Q
163. Peterson's Solution-Critical Section Problem-Operating Systems-20A05402T - YouTube, accessed on April 20, 2025, https://www.youtube.com/watch?v=7H4TNsxAlf4
164. Deconstructing the Bakery to Build a Distributed State Machine, accessed on April 20, 2025, https://cacm.acm.org/research/deconstructing-the-bakery-to-build-a-distributed-state-machine/
165. Bakery Algorithm in OS | Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/operating-system/bakery-algorithm-in-os/
166. Bakery Algorithm in Process Synchronization - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/bakery-algorithm-in-process-synchronization
167. Bakery Algorithm in Process Synchronization - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/bakery-algorithm-in-process-synchronization/
168. Bakery Algorithm - nob.cs.ucdavis.edu!, accessed on April 20, 2025, https://nob.cs.ucdavis.edu/classes/ecs150-2022-02/handouts/sync-bakery.pdf
169. Deconstructing the Bakery to Build a Distributed State Machine - Leslie Lamport, accessed on April 20, 2025, https://lamport.azurewebsites.net/pubs/bakery/dbakery-complete.pdf
170. Lamport's bakery algorithm - Wikipedia, accessed on April 20, 2025,

https://en.wikipedia.org/wiki/Lamport%27s_bakery_algorithm
171. Lamport's Bakery Algorithm - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/lamport-s-bakery-algorithm
172. The Black-White Bakery Algorithm - TAU, accessed on April 20, 2025, https://www.cs.tau.ac.il/~afek/gadi.pdf
173. Operating System #26 Bakery Algorithm - YouTube, accessed on April 20, 2025, https://www.youtube.com/watch?v=YHQxp-XduS0
174. Synchronization Hardware in OS | Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/operating-system/synchronization-hardware-in-os/
175. Hardware Synchronization Algorithms : Unlock and Lock, Test and Set, Swap, accessed on April 20, 2025, https://www.geeksforgeeks.org/hardware-synchronization-algorithms-unlock-and-lock-test-and-set-swap/
176. OPERATING SYSTEMS PROCESS SYNCHRONIZATION, accessed on April 20, 2025, https://web.cs.wpi.edu/~cs3013/c07/lectures/Section06-Sync.pdf
177. Synchronization Hardware in Operating System - YouTube, accessed on April 20, 2025, https://www.youtube.com/watch?v=h4COK43R40U
178. CSC 256/456: Operating Systems - Synchronization Principles I, accessed on April 20, 2025, https://www.cs.rochester.edu/courses/456/fall2014/slides/05-Synchronization.pdf
179. CS 134: Operating Systems - Computer Hardware Synchronization, accessed on April 20, 2025, https://www.cs.hmc.edu/~geoff/classes/hmc.cs134.201209/slides/class03_hardware_beamer.pdf
180. How to explain thread synchronization mechanism at OS level conceptually?, accessed on April 20, 2025, https://stackoverflow.com/questions/56721648/how-to-explain-thread-synchronization-mechanism-at-os-level-conceptually
181. hardware mechanisms - operating system - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/4066585/hardware-mechanisms
182. Semaphores in Process Synchronization - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/semaphores-in-process-synchronization/
183. Semaphores, mutexes, and monitors | Operating Systems Class Notes - Fiveable, accessed on April 20, 2025, https://library.fiveable.me/operating-systems/unit-6/semaphores-mutexes-monitors/study-guide/S4jiSv0NbbMFRCtj
184. Using Semaphores for Producer-Consumer Problems - DEV Community, accessed on April 20, 2025, https://dev.to/kalkwst/using-semaphores-for-producer-consumer-problems-3d0p
185. Producer–consumer problem - Wikipedia, accessed on April 20, 2025, https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem

186. Producer Consumer Problem using Semaphores | Set 1 - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/producer-consumer-problem-using-semaphores-set-1/
187. Semaphores, Producer-Consumer, Readers-Writers, accessed on April 20, 2025, https://cseweb.ucsd.edu/classes/sp16/cse120-a/applications/ln/lecture78.html
188. Producer Consumer Problem Using Semaphores - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/producer-consumer-problem-using-semaphores
189. How to solve the producer-consumer using semaphores? - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/8288479/how-to-solve-the-producer-consumer-using-semaphores
190. The Producer/Consumer Problem, Using Semaphores (Multithreaded Programming Guide), accessed on April 20, 2025, https://docs.oracle.com/cd/E19455-01/806-5257/sync-15907/index.html
191. Producer Consumer synchronization with only 1 semaphore - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/57496477/producer-consumer-synchronization-with-only-1-semaphore
192. ELI5 - Semaphores and the producer/consumer problem : r/compsci - Reddit, accessed on April 20, 2025, https://www.reddit.com/r/compsci/comments/5e08s5/eli5_semaphores_and_the_producerconsumer_problem/
193. Producer Consumer Problem in C using Semaphore and Mutex | Operating System, accessed on April 20, 2025, https://m.youtube.com/watch?v=caFjPdWsJDU&pp=ygUQI3Byb2JsZW1jb25zdW1lcg%3D%3D
194. Readers–writers problem - Wikipedia, accessed on April 20, 2025, https://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem
195. Readers-Writers Problem - Jyotiprakash's Blog, accessed on April 20, 2025, https://blog.jyotiprakash.org/readers-writers-problem
196. Solution of Readers/Writers Problem using Semaphores, accessed on April 20, 2025, https://cs.gordon.edu/courses/cs322/lectures/transparencies/readers_writers.html
197. Readers/Writers w/ Readers Priority (Using Semaphores), accessed on April 20, 2025, https://www.cs.kent.edu/~farrell/osf99/lectures/L15.pdf
198. Solution to readers/writers problem using Semaphores - the denning institute, accessed on April 20, 2025, http://denninginstitute.com/modules/ipc/orange/readsem.html
199. Readers-Writers Problem | Set 1 (Introduction and Readers Preference Solution) | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/readers-writers-problem-set-1-introduction-and-readers-preference-solution/

200. Readers-Writers Problem | Writers Preference Solution - GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/readers-writers-problem-writers-preference-solution/?ref=rp
201. Readers-Writers Problem in C using pthreads and semaphores - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/58083898/readers-writers-problem-in-c-using-pthreads-and-semaphores
202. Semaphore solution to reader-writer: order between updating reader count and waiting or signaling on read/write binary semaphore? - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/46881402/semaphore-solution-to-reader-writer-order-between-updating-reader-count-and-wai
203. The Readers Writers Problem - YouTube, accessed on April 20, 2025, https://www.youtube.com/watch?v=p2XDhW5INOo&pp=0gcJCdgAo7VqN5tD
204. Dining Philosopher Problem Using Semaphores | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/dining-philosopher-problem-using-semaphores/
205. The Dining Philosophers Problem - YouTube, accessed on April 20, 2025, https://www.youtube.com/watch?v=FYUi-u7UWgw
206. Dining Philosophers Problem in OS | Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/operating-system/dining-philosophers-problem-in-os/
207. Struggles with the Dining Philosophers Problem and Semaphores : r/cprogramming - Reddit, accessed on April 20, 2025, https://www.reddit.com/r/cprogramming/comments/1hzlgo7/struggles_with_the_dining_philosophers_problem/
208. Dining Philosophers Problem | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/dining-philosophers-problem/
209. Is this the right way to code Dining philosopher using semaphore without any deadlock at all? - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/71385647/is-this-the-right-way-to-code-dining-philosopher-using-semaphore-without-any-dea
210. Dining philosophers - deadlock problem : r/learnprogramming - Reddit, accessed on April 20, 2025, https://www.reddit.com/r/learnprogramming/comments/1ce8wyc/dining_philosophers_deadlock_problem/
211. How to use Parallel Semaphores for Dining Philosophers Problem, accessed on April 20, 2025, https://cs.stackexchange.com/questions/136407/how-to-use-parallel-semaphores-for-dining-philosophers-problem
212. Dining Philosophers using semaphores - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/27084579/dining-philosophers-using-semaphores

213. Lecture 12: Semaphores - CS 110: Principles of Computer Systems, accessed on April 20, 2025, https://web.stanford.edu/class/cs110/summer-2021/lecture-notes/lecture-12/
214. Operating Systems: Deadlocks, accessed on April 20, 2025, https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7_Deadlocks.html
215. Introduction of Deadlock in Operating System | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/introduction-of-deadlock-in-operating-system/
216. Deadlock Prevention And Avoidance | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/deadlock-prevention/
217. CS 453 Operating Systems, accessed on April 20, 2025, https://home.adelphi.edu/~siegfried/cs453/453l7.pdf
218. Chapter 7: Deadlocks, accessed on April 20, 2025, http://www.cs.fsu.edu/~lacher/courses/COP4610/lectures_9e/ch07.pdf
219. Deadlock Prevention in OS (Operating System) - Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/operating-system/deadlock-prevention-in-operating-system/
220. 11.2: Deadlock Detection and Prevention - Engineering LibreTexts, accessed on April 20, 2025, https://eng.libretexts.org/Courses/Delta_College/Introduction_to_Operating_Systems/11%3A_Concurrency-__Deadlock_and_Starvation/11.02%3A_Deadlock_Detection_and_Prevention
221. Deadlock Prevention-Operating Systems-unit-4-20A05402T - YouTube, accessed on April 20, 2025, https://www.youtube.com/watch?v=9N6x-jrTVgw
222. www.geeksforgeeks.org, accessed on April 20, 2025, https://www.geeksforgeeks.org/deadlock-prevention/#:~:text=Deadlock%20avoidance%20ensures%20that%20a,of%20resources%20it%20may%20need.
223. Deadlock Avoidance in OS - Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/operating-system/deadlock-avoidance-in-os/
224. What is Deadlock Avoidance in OS (Operating System)? - Hero Vired, accessed on April 20, 2025, https://herovired.com/learning-hub/blogs/deadlock-avoidance-in-os/
225. Deadlock Avoidance in Operating System - Studytonight, accessed on April 20, 2025, https://www.studytonight.com/operating-system/deadlock-avoidance-in-operating-system
226. Deadlock Avoidance - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/deadlock-avoidance
227. Operating Systems: Deadlock, Deadlock Prevention and Avoidance - YouTube, accessed on April 20, 2025, https://www.youtube.com/watch?v=miQ55SsU1YA
228. Day 14: Deadlock Detection Algorithms | Exploring Operating Systems - Mohit Mishra, accessed on April 20, 2025, https://mohitmishra786.github.io/exploring-os/src/day-14-deadlock-detection-algorithms.html

229. Deadlock Detection And Recovery | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/deadlock-detection-recovery/
230. Deadlock Detection Algorithm in Operating System - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/deadlock-detection-algorithm-in-operating-system
231. Deadlocks: detection, prevention, and avoidance | Operating Systems Class Notes, accessed on April 20, 2025, https://library.fiveable.me/operating-systems/unit-2/deadlocks-detection-prevention-avoidance/study-guide/Anv5z5nulHbDgaQv
232. OPERATING SYSTEMS DEADLOCKS, accessed on April 20, 2025, https://web.cs.wpi.edu/~cs3013/c07/lectures/Section07-Deadlocks.pdf
233. Do operating systems have deadlock detection? : r/AskProgramming - Reddit, accessed on April 20, 2025, https://www.reddit.com/r/AskProgramming/comments/v5dy8o/do_operating_systems_have_deadlock_detection/
234. MODERN OPERATING SYSTEMS Third Edition ANDREW S. TANENBAUM Chapter 6 Deadlocks, accessed on April 20, 2025, https://www.ece.stonybrook.edu/~yang/333slides-2010/MOS-Ch06-e3+Security.pdf
235. Recovery from Deadlock in Operating System | GeeksforGeeks, accessed on April 20, 2025, https://www.geeksforgeeks.org/recovery-from-deadlock-in-operating-system/
236. Deadlock Detection and Recovery - Tutorialspoint, accessed on April 20, 2025, https://www.tutorialspoint.com/deadlock-detection-and-recovery
237. Deadlock Detection and Recovery in Operating System - Scaler Topics, accessed on April 20, 2025, https://www.scaler.com/topics/deadlock-detection-in-os/
238. Recovery from Deadlock - YouTube, accessed on April 20, 2025, https://m.youtube.com/watch?v=tEfqOPKgTo8
239. Is Deadlock recovery possible in MultiThread programming? - Stack Overflow, accessed on April 20, 2025, https://stackoverflow.com/questions/3850995/is-deadlock-recovery-possible-in-multithread-programming
240. github.com, accessed on April 20, 2025, https://github.com/suvratapte/Maurice-Bach-Notes/blob/master/8-Process-Scheduling-and-Time.md#:~:text=The%20scheduler%20on%20the%20UNIX,of%20the%20several%20priority%20queues.
241. PROCESS SCHEDULING AND TIME:, accessed on April 20, 2025, https://jkmaterials.yolasite.com/resources/materials/UNIX/UNIX_INTERNALS/UNIT-VII.pdf
242. Maurice-Bach-Notes/8-Process-Scheduling-and-Time.md at master ..., accessed on April 20, 2025, https://github.com/suvratapte/Maurice-Bach-Notes/blob/master/8-Process-Scheduling-and-Time.md

243. UNIX process scheduling - BME, accessed on April 20, 2025, https://www.mit.bme.hu/eng/system/files/oktatas/targyak/8776/unix_3_process_scheduling.pdf
244. What's the process scheduler in Linux? - Unix & Linux Stack Exchange, accessed on April 20, 2025, https://unix.stackexchange.com/questions/486182/whats-the-process-scheduler-in-linux
245. Process scheduling data on linux - Unix & Linux Stack Exchange, accessed on April 20, 2025, https://unix.stackexchange.com/questions/260485/process-scheduling-data-on-linux
246. How do Modern Operating Systems Handle Process Scheduling? - Bigly Sales, accessed on April 20, 2025, https://biglysales.com/how-do-modern-operating-systems-handle-process-scheduling/
247. Scheduling - Win32 apps | Microsoft Learn, accessed on April 20, 2025, https://learn.microsoft.com/en-us/windows/win32/procthread/scheduling
248. Processes, Threads, and Jobs in the Windows Operating System ..., accessed on April 20, 2025, https://www.microsoftpressstore.com/articles/article.aspx?p=2233328&seqNum=7
249. Windows11 scheduling algorithms - Microsoft Q&A, accessed on April 20, 2025, https://learn.microsoft.com/en-us/answers/questions/1327360/windows11-scheduling-algorithms
250. MS Windows 10 CPU Scheduling - Prof. Dr. Hasan Hüseyin BALIK, accessed on April 20, 2025, http://www.hasanbalik.com/LectureNotes/OpSys/Assignments/MS%20Windows%2010%20CPU%20Scheduling.pdf
251. Using Operating System Scheduling on Windows Systems - SAS documentation, accessed on April 20, 2025, https://documentation.sas.com/doc/en/scheduleug/9.4/n05akp7m9jxhzfn1coz5nepy7zxf.htm