Tribhuvan University

Institute of Science and Technology

Central Department of Computer Science and Information Technology

Kirtipur, Kathmandu

Advanced Operating System

Submitted by:

Priya Shrestha (Roll no:24)

Suprabha Pokharel(Roll no:57)

Luna Parajuli(Roll no:58)

Pabitra Pantha(Roll no:59)

Submitted to:

Prof.Dr.Binod Adhikari

Assignment 1

1). **Discuss interprocess communication in details**

Introduction:

Interprocess communication (IPC) is a process that allows different processes of a computer system to share information. IPC lets different programs run in parallel, share data, and communicate with each other. It's important for two reasons: First, it speeds up the execution of tasks, and secondly, it ensures that the tasks run correctly and in the order that they were executed.

Types of Process

Let us first talk about types of types of processes.

- **Independent process:** An independent process is not affected by the execution of other processes. Independent processes are processes that do not share any data or resources with other processes. No inter-process communication required here.
- **Co-operating process:** Interact with each other and share data or resources. A co-operating process can be affected by other executing processes. Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of cooperation between them.

Summary:

Inter process communication (IPC) allows different programs or processes running on a computer to share information with each other. IPC allows processes to communicate by using different techniques like sharing memory, sending messages, or using files. It ensures that processes can work together without interfering with each other. Cooperating processes require an Inter Process Communication (IPC) mechanism that will allow them to exchange data and information.

The two fundamental models of Inter Process Communication are:

- Shared Memory
- Message Passing

Message Passing Process Communication Model

Message passing model allows multiple processes to read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

Shared Memory Process Communication Model

The shared memory in the shared memory model is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other. All POSIX systems, as well as Windows operating systems use shared memory.

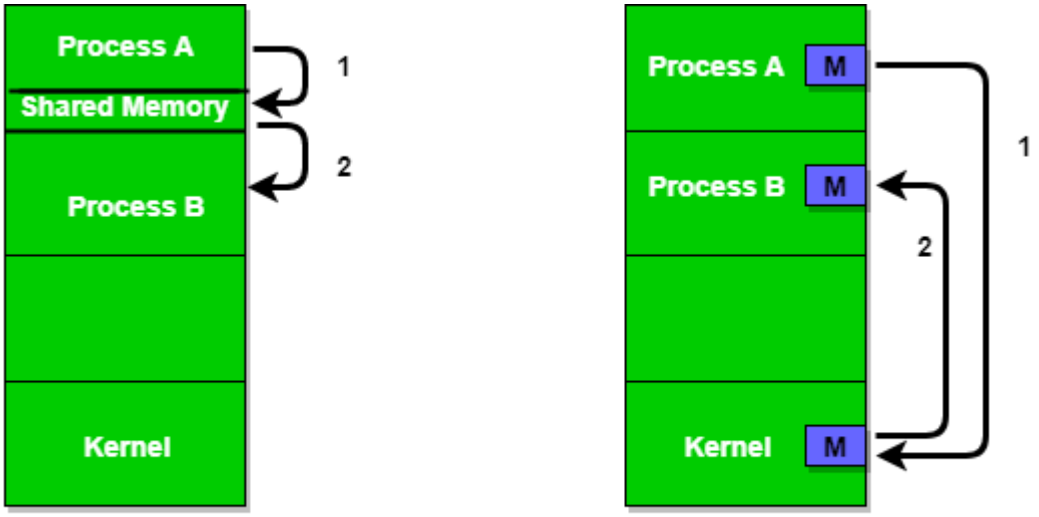


Figure 1 - Shared Memory and Message Passing

Conclusion:

A fundamental component of contemporary operating systems, IPC allows processes to efficiently coordinate operations, share resources, and communicate. IPC is beneficial for developing adaptable and effective systems, despite its complexity and possible security threats.

References:

https://www.geeksforgeeks.org/inter-process-communication-ipc/
https://www.nesoacademy.org/cs/03-operating-system
https://www.tutorialspoint.com/message-passing-vs-shared-memory-process-communication-models

2).What is critical section and critical section problem?

Introduction

The **critical section** is a part of a program where shared resources like memory, data structures, CPU or I/O devices are accessed.

Only one process can execute the critical section at a time to prevent conflicts. The operating system faces challenges in deciding when to allow or block processes from entering the critical section. The **critical section problem** involves creating protocols to ensure that race conditions (where multiple processes interfere with each other) never occur.

The **Critical Section Problem** comes with several challenges that need to be addressed to ensure correct and efficient execution of concurrent processes. Different types of critical section problem are as follows:

a. Race Condition

- When multiple processes access and modify shared data simultaneously, the final outcome depends on the execution order.
- Example: Two threads incrementing the same variable may produce inconsistent results.

b. Deadlock

- When two or more processes are stuck in a circular wait, each waiting for the other to release a resource.
- Example: Process A holds Resource 1 and waits for Resource 2, while Process B holds Resource 2 and waits for Resource 1.

c. Starvation

- A process waits indefinitely because other processes are given higher priority.
- Example: A low-priority thread never gets access to the critical section because higher-priority threads keep executing.

d. Priority Inversion

- A lower-priority process holds a resource needed by a higher-priority process, leading to performance issues.

- Example: A real-time task is blocked because a background task is using a necessary resource.

e. Busy Waiting

- Some solutions use continuous looping to check access to the critical section, wasting CPU resources.

- Example: Using a **while-loop** to check a flag instead of using proper synchronization mechanisms like semaphores.

f. Performance Overhead

- Locking mechanisms (e.g., mutexes, semaphores) introduce delays due to context switching and resource management.

Solutions to critical section problems

To effectively address the Critical Section Problem in operating systems, any solution must meet three key requirements:

A. **Mutual Exclusion**: This means that when one process is executing within its critical section, no other process should be allowed to enter its own critical section. This ensures that shared resources are accessed by only one process at a time, preventing conflicts and data corruption.
B. **Progress:** When no process is currently executing in its critical section, and there is a process that wishes to enter its critical section, it should not be kept waiting indefinitely. The system should enable processes to make progress, ensuring that they eventually get a chance to access their critical sections.
C. **Bounded Waiting:** There must be a limit on the number of times a process can execute in its critical section after another process has requested access to its critical section but before that request is granted. This ensures fairness and prevents any process from being starved of critical section access.

Summary

The **Critical Section Problem** arises in concurrent programming when multiple processes or threads need to access shared resources (e.g., memory, CPU, I/O devices). To prevent conflicts, only one process should enter the critical section at a time. The operating system must ensure proper coordination to avoid issues like **race conditions**, where multiple processes interfere with each other.

Conclusion

The Critical Section Problem is a fundamental challenge in concurrent programming that arises when multiple processes need to access shared resources. If not handled properly, it can lead to serious issues like race conditions, deadlocks, and starvation, which can affect system stability and performance.

To prevent such problems, operating systems implement synchronization mechanisms like locks, semaphores, and monitors, ensuring that processes access shared resources in a controlled manner. By following principles like **mutual exclusion, progress, and bounded waiting**, developers can design efficient and fair systems that prevent conflicts and maintain data integrity.

Ultimately, solving the Critical Section Problem is crucial for building reliable multitasking systems, enabling smooth execution of parallel processes in modern computing environments.

References

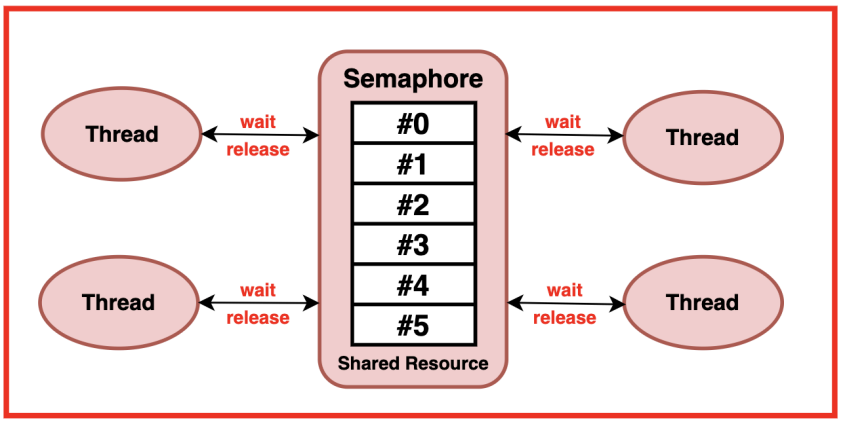
https://www.scaler.com/topics/critical-section-in-os/

https://www.geeksforgeeks.org/solution-to-critical-section-problem/

3).What are semaphores? Explain.

Introduction

In an operating system, multiple processes and threads often compete for shared resources(memory, files, hardware devices etc) and managing proper synchronization of concurrent processes is a fundamental challenge that can lead to race conditions, data corruption, system instability etc. Semaphores introduced by **Edsger Dijkstra** in 1965, are a foundational synchronization primitives/tool that addresses the above challenges via controlling access to shared resources without causing conflicts.

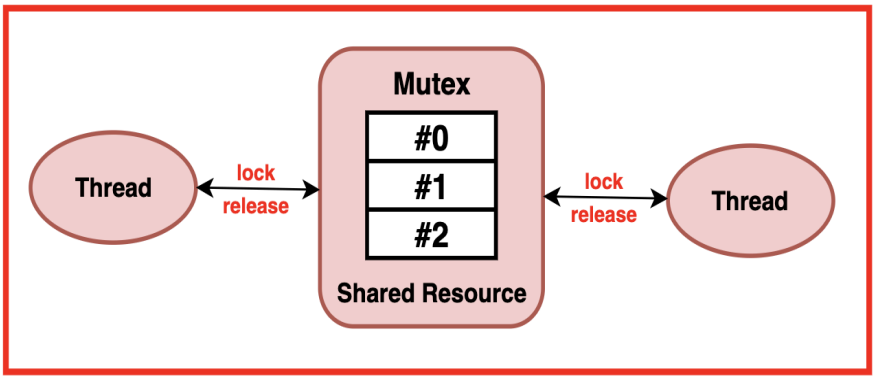


Summary

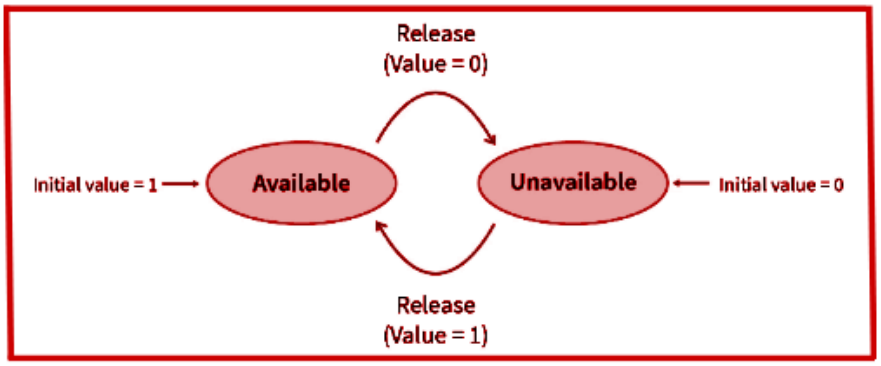
Semaphores are integer variables that are used to solve critical section problems by utilizing two atomic operations: wait and signal. These operations are executed in a mutually exclusive manner to ensure proper synchronization and coordination between concurrent processes. They function by providing a mechanism that signals when a resource is available or when a process should wait for a resource to become available.

Types of Semaphores

A. **Binary Semaphore(Mutex Lock)**

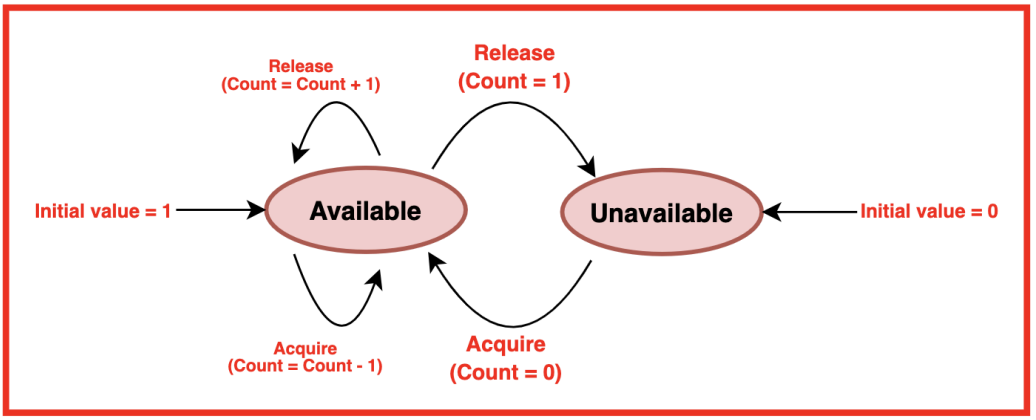


- ➔ Binary Semaphores are used for simple lock/unlock mechanisms where mutual exclusion is required.
- ➔ Mutexes have two values: 0(locked) or 1(released).
- ➔ A resource is accessed by only one thread at a time.



- ➔ When a thread acquires the Semaphore, it sets the value to 0(unavailable), and when it releases the Semaphore, it sets the value back to 1(available).
- ➔ If the value of Semaphore is 1, the process can proceed to the critical section and if 0, then the process cannot continue to the critical section of the code.

B. Counting Semaphore



- ➔ Counting Semaphores are used for managing resources with multiple identical instances.
- ➔ They can take any non-negative integer value.
- ➔ The Semaphore's value is initialised to the number of available resources.
- ➔ Each wait operation decrements the value. If the value is greater than 0, the operation proceeds; if the value is 0, the process is blocked.
- ➔ Each signal operation increments the value and potentially wakes up a blocked process.

Semaphore Operations

Semaphores operate through two main atomic operations:

1. **Wait(P or down operation)**
2. **Signal(V or up operation)**

Wait(P or down operation)

The wait operation, also called the ‘P’ function, sleep, decrease, or down operation, is the semaphore operation that controls the entry of a process into a critical section. If the value of semaphore is positive then we decrease the value of the semaphore and the process enters the critical section. This function is only called before the process enters the critical section.

In pseudocode:

```
wait(S) {
        while(S <=0);
        S - -;
}
```

Signal(V or up operation)

The signal operation, also called the ‘V’ function, increase,or up operation , is the semaphore operation that releases control of the critical section. This operation simply increments the semaphore value, signaling that the process has exited the critical section and allowing other processes to proceed. If there are any processes waiting(blocked) for the semaphore, one of them will be unblocked,allowing it to continue execution.

In pseudocode:

```
signal(S) {
        S ++;
}
```

Applications of Semaphores

Mutual Exclusion: Semaphores are used to enforce mutual exclusion, meaning only one process at a time can access a critical section. This is especially important in scenarios where multiple processes share memory, I/O devices, or other resources.

Producer-Consumer Problem: Semaphores are often used in the producer-consumer problem, where processes (producers) create resources and put them into a buffer, and other processes (consumers) take resources from the buffer. Semaphores ensure that the buffer is managed correctly, avoiding overflow or underflow.

Reader-Writer Problem: In situations where multiple processes need access to a shared resource, semaphores can be employed to solve the reader-writer problem, ensuring that multiple readers can access the resource simultaneously while writers have exclusive access.

Conclusion

To sum up, Semaphores remain a cornerstone of process synchronization in operating systems. By enabling controlled access to shared resources and utilizing the basic operations of wait and signal, they prevent race conditions and ensure system stability. Semaphores are integral in solving problems related to synchronization, such as producer-consumer and reader-writer problems.Hence, Semaphores are essential for system designers and developers working on multi-threaded or distributed systems where proper synchronization is crucial.

References

https://takeuforward.org/operating-system/semaphore-and-its-types
https://www.scaler.com/topics/operating-system/semaphore-in-os/

4).What are monitors? Explains

Introduction

A feature of programming languages called monitors helps control access to shared data. The Monitor is a collection of shared actions, data structures, and synchronization between parallel procedure calls. A monitor is therefore also referred to as a synchronization tool. Some of the languages that support the usage of monitors include Java, C#, Visual Basic, Ada, and concurrent Euclid. Although they can call the monitor's procedures, processes running outside of the monitor are unable to access its internal variables.

For example, the Java programming language provides synchronization mechanisms like the wait() and notify() constructs.

Syntax of monitor in OS

Monitor in os has a simple syntax similar to how we define a class, it is as follows:

```
Monitor monitorName{
variables_declaration;
condition_variables;

procedure p1{ ... };

procedure p2{ ... };

...

procedure pn{ ... };

{

   initializing_code;

}

}
```

In an operating system, a monitor is only a class that includes variable_declarations, condition_variables, different procedures (functions), and an initializing_code block for synchronizing processes.

Summary

Monitors are a high-level synchronization construct used to manage concurrent access to shared resources in an operating system. They encapsulate shared data, synchronization mechanisms, and procedures, ensuring mutual exclusion and structured concurrency control. Programming languages like Java, C#, and Ada support monitors with built-in synchronization mechanisms such as wait() and notify().

Characteristics of Monitors in OS

Monitors in operating systems are a high-level synchronization construct used to manage concurrent access to shared resources. Their key characteristics include:

- **Mutual Exclusion:** Only one thread can be inside the monitor at any time, preventing race conditions and ensuring data consistency.
- **Encapsulation:** Monitors encapsulate shared resources and the procedures that operate on them, keeping synchronization logic centralized and making concurrent programming more manageable.
- **Synchronization Primitives:** Condition variables within monitors allow threads to wait for specific conditions and signal others when required, facilitating efficient thread coordination without busy-waiting.
- **Blocking Mechanism:** If a thread tries to enter a busy monitor, it is placed in a queue (entry queue) and blocked until it gains access, ensuring efficient CPU utilization.
- **Local Data:** Each thread using a monitor has its own local execution stack, preventing unintended interference between threads and ensuring data consistency.
- **Priority Inheritance:** Some implementations include priority inheritance to prevent priority inversion, temporarily elevating the priority of a lower-priority thread holding a resource needed by a higher-priority thread.
- **High-Level Abstraction:** Compared to low-level synchronization tools like semaphores and spinlocks, monitors offer a more structured and easier-to-use approach, reducing complexity and improving maintainability.

Components of a Monitor in an Operating System

A **monitor** is a synchronization construct that ensures safe concurrent access to shared resources by multiple threads or processes. It consists of the following key components:

A. **Shared Resource:**

   a. This is the critical data or resource that multiple threads need to access safely.

   b. Examples include global variables, data structures like queues, or critical code sections that require mutual exclusion.

B. **Entry Queue:**

   a. When a thread attempts to enter a busy monitor, it is placed in the entry queue.

   b. The thread remains blocked until the monitor becomes available, preventing CPU wastage due to busy-waiting.

C. **Entry Procedures (Monitor Procedures):**

   a. Special procedures within the monitor that regulate access to shared resources.

   b. Only one thread can execute an entry procedure at a time, ensuring mutual exclusion.

D. **Condition Variables:**

   a. Used for synchronization between threads within the monitor.

   b. Allow threads to wait for specific conditions to be met before proceeding.

E. **Local Data (Local Variables):**

   a. Each thread executing within the monitor has its own local variables.

   b. This ensures data integrity and prevents interference between threads.

Operations on Condition Variables

Monitors support two key operations on condition variables to facilitate synchronization:

- **y.wait():**
  - If a thread executes y.wait(), it gets suspended and placed in a queue associated with the condition variable.

- It remains blocked until another thread signals it.

- **y.signal():**

  - If a thread calls y.signal(), it wakes up one of the blocked threads waiting on the condition variable.

  - The awakened thread can then proceed with execution

Conclusion

In conclusion, monitors are essential synchronization constructs in operating systems that play a crucial role in managing concurrent access to shared resources. They provide a high-level abstraction for concurrency control, making it easier to develop correct and efficient concurrent programs. Monitors ensure mutual exclusion, encapsulate shared resources and their relevant procedures, and support synchronization primitives like condition variables. By using monitors, developers can avoid data races, prevent deadlocks, and improve the overall reliability and performance of concurrent applications.

References

https://www.prepbytes.com/blog/os-interview-question/what-is-the-monitor-in-os/
https://www.geeksforgeeks.org/monitor-vs-semaphore/
https://www.scaler.com/topics/monitor-in-os/

5).Show the Peterson's algorithm preserve mutual exclusion, indefinite postponement and progress (deadlock)

Introduction

Peterson's Algorithm is a classical solution to the critical section problem in concurrent programming. It is designed to ensure mutual exclusion between two processes that share a common resource without the need for complex hardware support. This algorithm utilizes two key variables: flag and turn, where flag indicates a process's interest in entering the critical section and turn decides which process gets the chance to proceed. Peterson's Algorithm effectively addresses three key requirements of concurrent execution: Mutual Exclusion, Indefinite Postponement (Starvation-Free Execution), and Progress (Deadlock-Free Execution).

Algorithm:

```
P0:      flag[0] = true;
P0_gate: turn = 1;
        while (flag[1] && turn == 1)
        {
           // busy wait
        }
        // critical section
        ...
        turn = 0;
        // end of critical section
        flag[0] = false;
```

```
P1:      flag[1] = true;
P1_gate: turn = 0;
        while (flag[0] && turn == 0)
        {
           // busy wait
        }
        turn = 1;
        // critical section
        ...
        // end of critical section
        flag[1] = false;
```

Mutual Exclusion

Mutual Exclusion ensures that only one process can enter the critical section at a time. In Peterson's Algorithm, a process enters the critical section only if either:
The other process is not interested (flag[1] == false), or
The process has been granted the turn (turn == 0).
This guarantees that two processes will not enter the critical section simultaneously, thereby preserving mutual exclusion.

Indefinite Postponement (Starvation-Free Execution)

Indefinite Postponement occurs when a process is continuously denied entry into the critical section. Peterson's Algorithm prevents this by ensuring a fair alternation of execution through the turn variable. When one process completes execution in the critical section, it sets flag[0] = false, allowing the other process to enter, ensuring no process is starved indefinitely.

Progress (Deadlock-Free Execution) Progress ensures that if multiple processes are waiting to enter the critical section, at least one will eventually proceed. In Peterson's Algorithm:
If a process sets flag[0] = true and the other process also wants to enter (flag[1] = true), then the turnvariable determines which process goes first.
Since a process voluntarily relinquishes control (flag[i] = false) upon exiting, the system never reaches a state where no process can proceed (deadlock-free).

Summary

Peterson's Algorithm is a simple yet effective solution for achieving mutual exclusion in a two-process system. It maintains fairness, ensuring no process is indefinitely postponed, and guarantees progress, preventing deadlock. By carefully using flag and turn, it provides a software-based approach to synchronization in a shared-memory environment.

Conclusion

Peterson's Algorithm is a simple and effective way to achieve mutual exclusion between two processes. It ensures that only one process can access the critical section at a time, preventing conflicts and data inconsistency. While it works well for two processes, it's limitations make it less practical for larger systems.Understanding and implementing Peterson's Algorithm helps in learning the basics of process synchronization, which is crucial for managing shared resources in concurrent programming.

References

https://www.geeksforgeeks.org/petersons-algorithm-in-process-synchronization/

Stallings, W. (2018). Operating Systems: Internals and Design Principles (9th ed.). Pearson

Round Robin Variants

6).Weighted RR

Introduction

Weighted Round Robin (WRR) is a load balancing technique that distributes traffic across multiple servers, but unlike standard Round Robin, it prioritizes servers with higher weights, ensuring those with greater capacity handle more requests.

Summary

WRR assigns weights to each resource and distributes tasks in a cyclic manner. Resources with higher weights receive more tasks per cycle, ensuring that each resource handles a proportionate amount of work based on its capacity.

Weighted Round Robin holds significant importance in the realm of load balancing and network management. Its unique approach of considering the capacity of each server in the network ensures a balanced distribution of client requests. This means that servers with higher capacity handle more requests, while those with lower capacity handle fewer, preventing any single server from being overwhelmed with requests and potentially crashing.

This balanced approach leads to improved network performance as it reduces latency and enhances the overall user experience. It also ensures better resource utilization as no server is left idle or underutilized.

Moreover, WRR provides a level of fault tolerance. In the event of a server failure or unavailability, the algorithm can intelligently redirect client requests to other operational servers. This ensures uninterrupted service and enhances the reliability of the network.

Conclusion

Weighted Round Robin (WRR) is a scheduling algorithm used to distribute workloads across multiple resources based on assigned weights. Resources with higher weights receive more tasks, ensuring balanced and optimized load distribution.

References

https://www.ituonline.com/tech-definitions/what-is-weighted-round-robin-wrr/
https://webhostinggeeks.com/blog/what-is-weighted-round-robin/

7).RR with variable quanta

Introduction

Round Robin (RR) is a preemptive CPU scheduling algorithm where each process gets a fixed time slice (quantum) before the CPU switches to the next process. However, in RR with variable quanta, the quantum time is not fixed; instead, it changes dynamically based on various factors.

Summary

In RR with Variable Quanta, the time quantum is adjusted dynamically rather than being fixed. Various strategies are used to determine the quantum, including:

A. Burst Time-Based Quanta: The quantum is adjusted based on the average burst time of completed processes, ensuring that short processes finish quickly while longer ones receive more CPU time before context switching.
B. Aging-Based Quantum Adjustment: Processes that frequently get preempted receive longer quanta over time, reducing starvation.
C. Priority-Based Quantum: High-priority processes may get smaller quanta for responsiveness, while lower-priority processes receive larger quanta to minimize overhead.
D. Feedback Mechanism (MLFQ-like Approach): The quantum is adapted dynamically based on whether a process is CPU-bound or I/O-bound, ensuring a balanced workload.

These dynamic adjustments improve overall system performance, reducing the trade-offs associated with fixed-quantum RR scheduling.

Steps to Calculate RR with Variable Quanta:

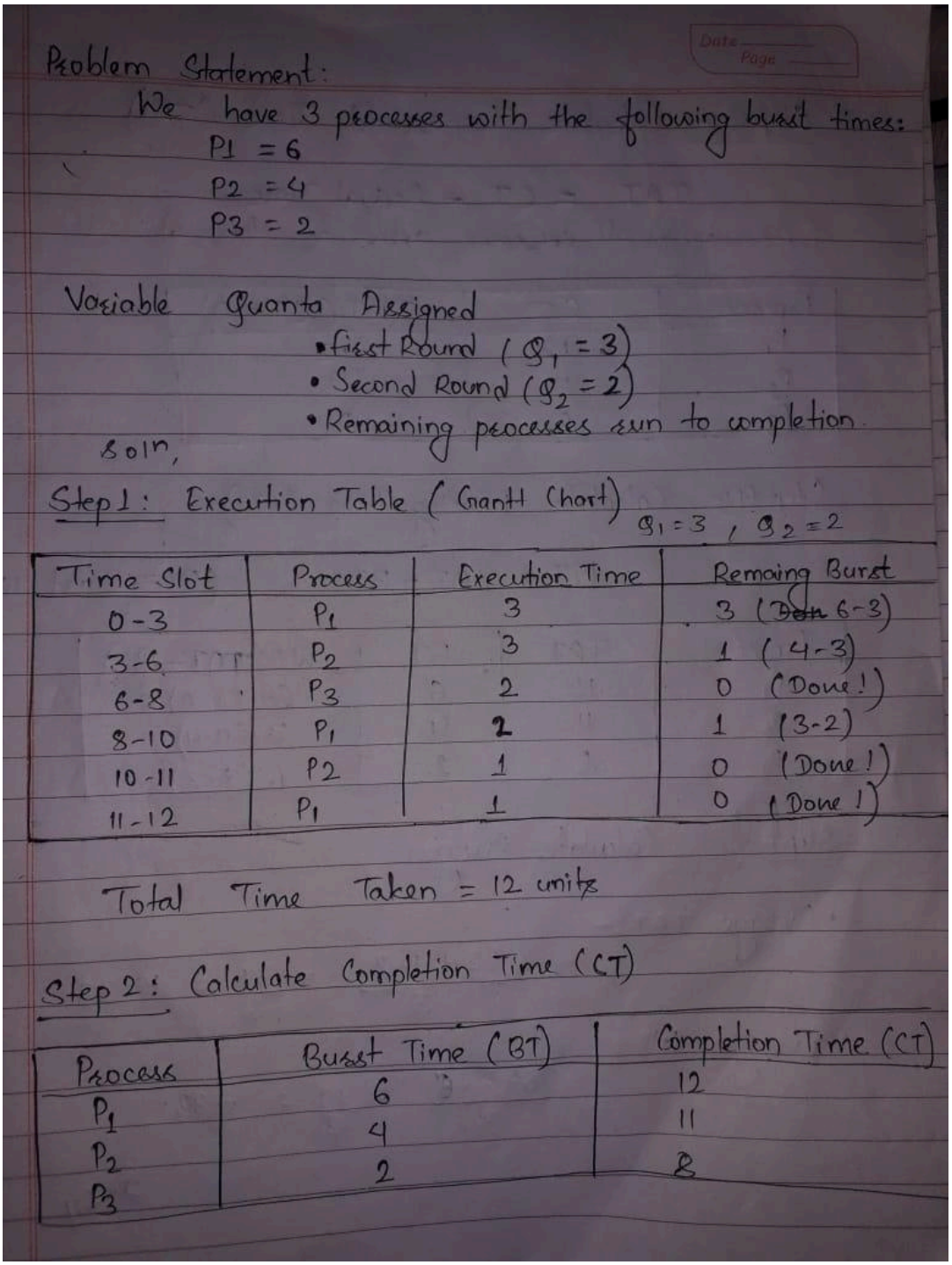
A. **List Processes with Burst Times:**
    a. Gather all processes along with their burst times (execution times).
B. **Define Time Quanta:**
    a. Assign different time quanta (e.g., Q1,Q2,Q3,…) for each round or for different processes.
C. **Execution Steps:**
    a. Execute each process for its assigned quantum.
    b. If a process completes before the quantum expires, move to the next process.
    c. If not, preempt the process and move it to the end of the queue.
D. **Calculate Waiting Time (WT) & Turnaround Time (TAT):**
    a. **Turnaround Time (TAT):** Completion Time – Arrival Time (usually 0 if all arrive at the same time).
    b. **Waiting Time (WT):** TAT – Burst Time.

E. **Compute Average WT and TAT:**
   a. Sum all WT and divide by the number of processes.
   b. Sum all TAT and divide by the number of processes.

Formulas:

★ Turnaround Time (TAT): Completion Time−Arrival Time
★ Waiting Time (WT): TAT−Burst Time
★ Throughput: Total Processes Completed/ Total time

Calculation:



Problem Statement:

We have 3 processes with the following burst times:

P1 = 6
P2 = 4
P3 = 2

Variable Quanta Assigned

- first Round ($Q_1 = 3$)
- Second Round ($Q_2 = 2$)
- Remaining processes run to completion

Soln,

Step 1: Execution Table (Gantt Chart) $Q_1 = 3$, $Q_2 = 2$

Time Slot	Process	Execution Time	Remaing Burst
0-3	P_1	3	3 (6-3)
3-6	P_2	3	1 (4-3)
6-8	P_3	2	0 (Done!)
8-10	P_1	2	1 (3-2)
10-11	P_2	1	0 (Done!)
11-12	P_1	1	0 (Done!)

Total Time Taken = 12 units

Step 2: Calculate Completion Time (CT)

Process	Burst Time (BT)	Completion Time (CT)
P_1	6	12
P_2	4	11
P_3	2	8

Step 3:- Calculate Turnaround Time (TAT)

TAT = CT − Arrival Time

(Assuming all processes arrive at time 0)

Process	CT	TAT (CT − 0)
P_1	12	12
P_2	11	11
P_3	8	8

Step 4:- Calculate Waiting Time (WT)

WT = TAT − BT (BT ⇒ Burst

Process	TAT	BT	WT (TAT − BT)
P_1	12	6	6 → (12 − 6)
P_2	11	4	7 → (11 − 4)
P_3	8	2	6 → (8 − 2)

Step 5:- Calculate Averages

- Average TAT :-

$$\frac{12+11+8}{3} = \frac{31}{3} = 10.33$$

- Average WT :

$$\frac{6+7+6}{3} = \frac{19}{3} = 6.33$$

Ans)

Conclusion

Round Robin with Variable Quanta enhances traditional RR scheduling by dynamically adjusting the quantum based on process characteristics. This approach optimizes CPU utilization, reduces unnecessary context switches, and ensures fairer process execution. By incorporating intelligent quantum selection strategies, this scheduling method strikes a balance between fairness and efficiency, making it a valuable improvement over the conventional RR algorithm.

References

https://ieeexplore.ieee.org/document/7421020