**ASSIGNMENT**

By

Name - Adheesh Chopra

Roll no. - 2022A1R043

Semester – III

Department Name – CSE

Group – G (2022A1R039 TO 2022A1R044)

Subject Name – Operating System (COM – 302)



**Model Institute of Engineering & Technology (Autonomous)**

(Permanently Affiliated to the University of Jammu, Accredited by NAAC with “A” Grade)

Jammu, India 2023

Submitted To : Ms. Mekhla Sharma

Email : [mekhla.cse@mietjammu.in](mailto:mekhla.cse@mietjammu.in)

|  |  |
| --- | --- |
| **S.NO** | **CONTENT** |
| 1 | TASK |
| 2 | INTRODUCTION |
| 3 | CODE |
| 4 | PSUEDO CODE |
| 5 | OUTPUT |

**TASK NO. 1:**

Design a program that implements Round Robin scheduling Algorithm. Create a set of processes with specified quantum time and demonstrate how the operating system schedules these processes. Implement and analyze the algorithm with at least 3 different specified quantum time.

**INTRODUCTION:**

Round Robin (RR) is a pre-emptive scheduling algorithm used in operating systems to manage the execution of processes. The primary goal of the Round Robin scheduling algorithm is to provide fair access to the CPU for all processes in the system. It is particularly well-suited for time-sharing systems and environments where multiple users or processes need to share the CPU.

Here's a brief explanation of how the Round Robin scheduling algorithm works:

1. Time Slicing (Quantum): In Round Robin, each process is assigned a fixed time slice or quantum during which it can execute on the CPU. The scheduler allocates a small unit of time to each process in a cyclic manner. This time slice is usually a small, fixed duration.

2. Queue: Processes are organized in a circular queue. The scheduler maintains a ready queue of processes that are waiting to execute. When a process's turn arrives, it is moved from the ready queue to the running state.

3. Execution: A process is allowed to run for its time quantum or until it voluntarily yields the CPU (e.g., due to I/O request or completion). If a process does not complete its execution within the allocated time slice, it is moved to the back of the ready queue, and the next process in the queue gets a chance to execute.

4. Cyclic Execution: The process cycle repeats: a process gets the CPU for a fixed time, and if it doesn't complete its execution in that time, it goes to the back of the queue. The next process in the queue is then given the CPU.

5. Fairness: The Round Robin algorithm ensures fairness among processes. No process is allowed to monopolize the CPU for an extended period, and all processes get a chance to execute in a cyclic manner.

Advantages of Round Robin Scheduling:

* Fairness: Provides fair and equal access to the CPU for all processes.
* Simple Implementation: Relatively simple to implement.
* No Starvation: Since each process gets a turn to execute, there is no indefinite waiting or starvation.

Drawbacks of Round Robin Scheduling:

* Performance: May not be optimal for certain types of workloads, as the fixed time slice might not be well-suited for all types of processes.
* Context Switching Overhead: The frequent context switching between processes can introduce overhead.

Despite its drawbacks, Round Robin is widely used in practice, especially in time-sharing systems and environments where interactive response is essential. It strikes a balance between fairness and simplicity.

**CODE:**

I can provide you with a simple example of a C program that implements the Round Robin scheduling algorithm. In this example, I'll create a set of processes with specified burst times and demonstrate how the operating system schedules these processes using Round Robin with different time quantum values.

#include <stdio.h>

// Process structure

struct Process {

int id; // Process ID

int burst\_time; // Burst time

int remaining\_time; // Remaining burst time

int arrival\_time; // Arrival time

};

// Function to perform Round Robin scheduling

void roundRobin(struct Process processes[], int n, int quantum) {

int time = 0; // Current time

int completed = 0; // Number of processes completed

while (completed < n) {

for (int i = 0; i < n; i++) {

if (processes[i].remaining\_time > 0 && processes[i].arrival\_time <= time) {

// Execute the process for the specified quantum or the remaining time, whichever is smaller

int execute\_time = (processes[i].remaining\_time < quantum) ? processes[i].remaining\_time : quantum;

// Update remaining time for the process

processes[i].remaining\_time -= execute\_time;

// Update current time

time += execute\_time;

// Print the scheduling information

printf("Time %d: Process %d executed for %d units\n", time, processes[i].id, execute\_time);

// Check if the process is completed

if (processes[i].remaining\_time == 0) {

completed++;

// Print completion information

printf("Time %d: Process %d completed\n", time, processes[i].id);

}

}

}

}

}

int main() {

// Number of processes

int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

// Quantum time input

int quantum;

printf("Enter the quantum time: ");

scanf("%d", &quantum);

// Set of processes with specified burst times and arrival times

struct Process processes[n];

for (int i = 0; i < n; i++) {

printf("Enter burst time for Process %d: ", i + 1);

scanf("%d", &processes[i].burst\_time);

printf("Enter arrival time for Process %d: ", i + 1);

scanf("%d", &processes[i].arrival\_time);

processes[i].id = i + 1;

processes[i].remaining\_time = processes[i].burst\_time;

}

// Demonstrate Round Robin scheduling with input values

printf("\nRound Robin Scheduling with Quantum Time = %d\n", quantum);

printf("--------------------------------------------------\n");

roundRobin(processes, n, quantum);

return 0;

}

This program defines a simple Process structure to represent each process with its ID, burst time, and remaining time. The Round Robin function simulates the Round Robin scheduling algorithm. The main function creates a set of processes and demonstrates the scheduling with different quantum time values.

**PSEUDO CODE:**

Here's the pseudocode for the Round Robin scheduling algorithm based on the provided C program.

procedure roundRobin(processes, n, quantum)

time = 0 // Current time

completed = 0 // Number of processes completed

while completed < n do

for i = 0 to n-1 do

if processes[i].remaining\_time > 0 and processes[i].arrival\_time <= time then

// Execute the process for the specified quantum or the remaining time, whichever is smaller

execute\_time = min(processes[i].remaining\_time, quantum)

// Update remaining time for the process

processes[i].remaining\_time -= execute\_time

// Update current time

time += execute\_time

// Print the scheduling information

print("Time", time, ": Process", processes[i].id, "executed for", execute\_time, "units")

// Check if the process is completed

if processes[i].remaining\_time == 0 then

completed++

// Print completion information

print("Time", time, ": Process", processes[i].id, "completed")

end if

end if

end for

end while

end procedure

procedure main()

n = input("Enter the number of processes: ")

// Quantum time input

quantum = input("Enter the quantum time: ")

// Set of processes with specified burst times and arrival times

processes = array of Process

for i = 0 to n-1 do

print("Enter burst time for Process", i + 1, ": ")

burst\_time = input()

print("Enter arrival time for Process", i + 1, ": ")

arrival\_time = input()

processes[i] = {id: i + 1, burst\_time: burst\_time, remaining\_time: burst\_time, arrival\_time: arrival\_time}

end for

// Demonstrate Round Robin scheduling with input values

print("\nRound Robin Scheduling with Quantum Time =", quantum)

print("--------------------------------------------------")

roundRobin(processes, n, quantum)

end procedure

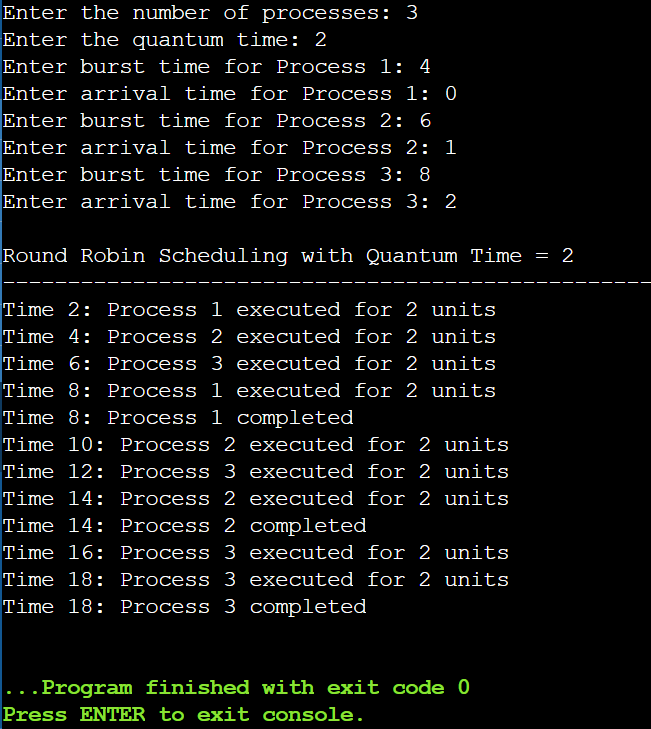
This pseudocode represents the key logic of the Round Robin scheduling algorithm. It uses a while loop to iterate until all processes are completed, and within that loop, it iterates through the processes, executing them for the specified quantum time or the remaining time, whichever is smaller. The algorithm prints scheduling and completion information as the processes are executed.

**OUTPUT:**

Consider the following table of arrival time and burst time for four processes P1, P2 and P3 with three different specified quantum time.

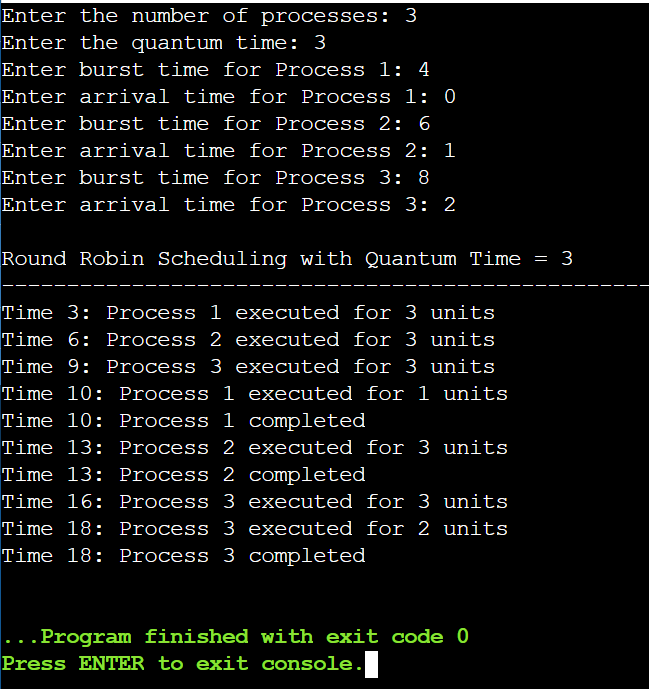
For Quantum Time = 2

|  |  |  |
| --- | --- | --- |
| **PROCESS** | **BURST TIME** | **ARRIVAL TIME** |
| P1 | 4 | 0 |
| P2 | 6 | 1 |
| P3 | 8 | 2 |



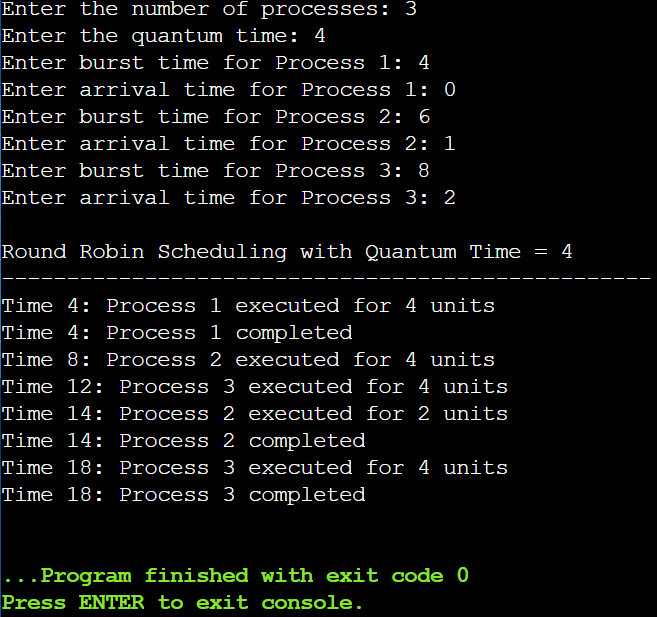
For Quantum Time = 3

|  |  |  |
| --- | --- | --- |
| **PROCESS** | **BURST TIME** | **ARRIVAL TIME** |
| P1 | 4 | 0 |
| P2 | 6 | 1 |
| P3 | 8 | 2 |



For Quantum Time = 4

|  |  |  |
| --- | --- | --- |
| **PROCESS** | **BURST TIME** | **ARRIVAL TIME** |
| P1 | 4 | 0 |
| P2 | 6 | 1 |
| P3 | 8 | 2 |



**TASK NO. 2:**

Design and implement various methods for IPC such as message passing or shared memory to facilitate communication between processes in the operating system.

**INTRODUCTION:**

MESSAGE PASSING

Message passing is a method of Inter-Process Communication (IPC) that allows different processes to communicate and exchange data in an operating system. In message passing, processes communicate by sending and receiving messages. This communication can occur between processes on the same computer or between processes on different computers in a networked environment.

Here are key concepts and characteristics of message passing:

1. Communication Channels:

* Processes communicate through communication channels, which can be implemented using various mechanisms such as pipes, sockets, or message queues.
* Channels provide a means for processes to send and receive messages.

1. Synchronous and Asynchronous Communication:

* In synchronous communication, a sender sends a message and waits for the receiver to acknowledge receipt before continuing.
* In asynchronous communication, a sender sends a message and continues its execution without waiting for an immediate acknowledgment from the receiver.

1. Message Format:

* Messages typically have a well-defined format that both the sender and receiver understand. This format may include a message type, data, and other relevant information.

1. Blocking and Non-Blocking Operations:

* Blocking operations occur when a process is halted until a message is received.
* Non-blocking operations allow a process to continue execution even if the expected message has not yet been received.

1. Point-to-Point and Publish-Subscribe Models:

* Point-to-Point: In this model, a process sends a message to a specific recipient process.
* Publish-Subscribe: In this model, a process (publisher) sends messages to multiple processes (subscribers) without necessarily knowing who the subscribers are.

1. Advantages:

* Simplicity: Message passing provides a straightforward and modular way for processes to communicate without the need for shared memory or complex synchronization mechanisms.
* Isolation: Processes can be isolated from each other, reducing the potential for interference.

1. Disadvantages:

* Overhead: Implementing message passing can introduce overhead due to the need to package, send, and receive messages.
* Synchronization: Ensuring proper synchronization and avoiding deadlocks can be challenging in some cases.

1. Examples of Message Passing Mechanisms:

* Pipes: In Unix-like operating systems, pipes allow one process to send data to another process using the standard input and output streams.
* Sockets: Network sockets enable message passing between processes on different machines over a network.
* Message Queues: Message queues provide a way for processes to send and receive messages in a controlled and orderly manner.

Message passing is widely used in various types of systems, including distributed systems, parallel computing environments, and multi-process operating systems. It allows processes to communicate efficiently while maintaining a level of abstraction that simplifies system design.

SHARED MEMORY

Shared memory is a method of Inter-Process Communication (IPC) that enables multiple processes to share a region of memory, allowing them to read and write data directly to the shared space. This shared region is typically in the address space of the processes involved, allowing them to communicate more efficiently compared to other IPC mechanisms like message passing. Shared memory is widely used for communication in parallel computing, multi-threaded applications, and multi-process systems.

Here are the key concepts and characteristics of shared memory:

1. Shared Memory Segment:

* A shared memory segment is a portion of memory that is created by one process and can be accessed by multiple processes.
* The operating system provides mechanisms for processes to attach to the shared memory segment.

1. Address Space:

* Processes that share memory have a portion of their address space mapped to the shared memory segment.
* This mapping allows processes to read and write data to the shared region as if it were their own local memory.

1. Creation and Attachment:

* The process that creates the shared memory segment does so using system calls provided by the operating system (e.g., shmget in Unix-like systems).
* Other processes can attach to the shared memory segment using the appropriate system calls (e.g., shmat).

1. Read and Write Operations:

* Processes can read and write data directly to the shared memory segment, allowing for fast and efficient communication.
* Synchronization mechanisms (e.g., semaphores or mutexes) are often used to avoid data corruption when multiple processes access shared memory concurrently.

1. Advantages:

* Efficiency: Shared memory communication is often faster than other IPC methods since data can be accessed directly without the need for copying.
* Flexibility: Processes can communicate using shared memory in a flexible and efficient manner, making it suitable for scenarios where frequent data exchange is required.

1. Disadvantages:

* Synchronization: Proper synchronization is essential to prevent race conditions and data inconsistencies when multiple processes access shared memory concurrently.
* Complexity: Managing shared memory requires careful consideration of synchronization mechanisms, leading to potentially more complex code.

1. Examples of Shared Memory Mechanisms:

* Unix Shared Memory: Unix-like operating systems provide functions such as shmget, shmat, and shmdt for creating and managing shared memory segments.
* Windows Shared Memory: In Windows, processes can share memory using functions like CreateFileMapping and MapViewOfFile.

Shared memory is suitable for scenarios where processes need to share large amounts of data frequently and efficiently. It is commonly used in parallel programming, multi-threading, and inter-process communication in various types of applications, such as databases, scientific simulations, and graphics processing.

**CODE:**

MESSAGE PASSING

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

void parentProcess(int pipefd[2]) {

close(pipefd[0]); // Close the read end

char message[] = "Hello Child!";

write(pipefd[1], message, sizeof(message));

close(pipefd[1]); // Close the write end

}

void childProcess(int pipefd[2]) {

close(pipefd[1]); // Close the write end

char buffer[20];

read(pipefd[0], buffer, sizeof(buffer));

printf("Child received: %s\n", buffer);

close(pipefd[0]); // Close the read end

}

int main() {

int pipefd[2];

if (pipe(pipefd) == -1) {

perror("pipe");

exit(EXIT\_FAILURE);

}

pid\_t pid = fork();

if (pid == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid > 0) { // Parent process

parentProcess(pipefd);

} else if (pid == 0) { // Child process

childProcess(pipefd);

}

return 0;

}

SHARED MEMORY

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define SHM\_SIZE 1024

int main() {

key\_t key = ftok("shmfile", 65);

int shmid = shmget(key, SHM\_SIZE, IPC\_CREAT | 0666);

if (shmid == -1) {

perror("shmget");

exit(EXIT\_FAILURE);

}

char \*shared\_memory = (char \*)shmat(shmid, (void \*)0, 0);

if (shared\_memory == (char \*)(-1)) {

perror("shmat");

exit(EXIT\_FAILURE);

}

// Write to shared memory

sprintf(shared\_memory, "Hello, Shared Memory!");

// Detach shared memory

shmdt(shared\_memory);

// Reattach if needed

shared\_memory = (char \*)shmat(shmid, (void \*)0, 0);

// Read from shared memory

printf("Message from shared memory: %s\n", shared\_memory);

// Detach shared memory

shmdt(shared\_memory);

// Remove shared memory

shmctl(shmid, IPC\_RMID, NULL);

return 0;

}

**PSEUDO CODE:**

MESSAGE PASSING

procedure parentProcess(pipefd)

close(pipefd[0]) // Close the read end of the pipe

message = "Hello Child!"

write(pipefd[1], message, sizeof(message))

close(pipefd[1]) // Close the write end of the pipe

end procedure

procedure childProcess(pipefd)

close(pipefd[1]) // Close the write end of the pipe

buffer = createEmptyBuffer(20) // Create a buffer to store the received message

read(pipefd[0], buffer, sizeof(buffer))

print("Child received:", buffer)

close(pipefd[0]) // Close the read end of the pipe

end procedure

procedure main()

pipefd = createPipe() // Create a pipe for communication

pid = fork() // Fork a child process

if pid == -1 then

perror("fork")

exit(EXIT\_FAILURE)

end if

if pid > 0 then // Parent process

parentProcess(pipefd)

else if pid == 0 then // Child process

childProcess(pipefd)

end if

end procedure

This pseudocode represents the key logic of the C program using pipes for communication between parent and child processes. It includes the creation of pipes, forking of processes, and communication through the pipes. Note that the createPipe and createEmptyBuffer functions are assumed to be provided by the underlying system or programming environment.

SHARED MEMORY

procedure main()

key = generateKey("shmfile") // Generate a key for shared memory identification

shmid = createSharedMemory(key, SHM\_SIZE, IPC\_CREAT | 0666) // Create shared memory segment

if shmid == -1 then

perror("shmget")

exit(EXIT\_FAILURE)

end if

shared\_memory = attachSharedMemory(shmid) // Attach to the shared memory segment

if shared\_memory == (char \*)(-1) then

perror("shmat")

exit(EXIT\_FAILURE)

end if

// Write to shared memory

writeMessageToSharedMemory(shared\_memory, "Hello, Shared Memory!")

// Detach from shared memory

detachSharedMemory(shared\_memory)

// Reattach if needed

shared\_memory = attachSharedMemory(shmid)

// Read from shared memory

message = readMessageFromSharedMemory(shared\_memory)

print("Message from shared memory:", message)

// Detach from shared memory

detachSharedMemory(shared\_memory)

// Remove shared memory

removeSharedMemory(shmid)

end procedure

function generateKey(filename)

// Generate a unique key based on the provided filename

// The actual implementation may vary depending on the platform

return key

end function

function createSharedMemory(key, size, flags)

// Create a shared memory segment with the specified key, size, and flags

// The actual implementation may vary depending on the platform

return shmid

end function

function attachSharedMemory(shmid)

// Attach to the shared memory segment identified by shmid

// The actual implementation may vary depending on the platform

return shared\_memory

end function

procedure detachSharedMemory(shared\_memory)

// Detach from the shared memory segment

// The actual implementation may vary depending on the platform

end procedure

procedure writeMessageToSharedMemory(shared\_memory, message)

// Write the specified message to the shared memory

// The actual implementation may vary depending on the platform

end procedure

function readMessageFromSharedMemory(shared\_memory)

// Read the message from the shared memory

// The actual implementation may vary depending on the platform

return message

end function

procedure removeSharedMemory(shmid)

// Remove the shared memory segment identified by shmid

// The actual implementation may vary depending on the platform

end procedure

This pseudocode represents the key logic of the C program using shared memory for communication. It includes the generation of a key, creation of shared memory, attachment to shared memory, writing and reading data to/from shared memory, detachment, and removal of shared memory. Note that the actual implementation may vary depending on the operating system and platform.