**ASSIGNMENT**

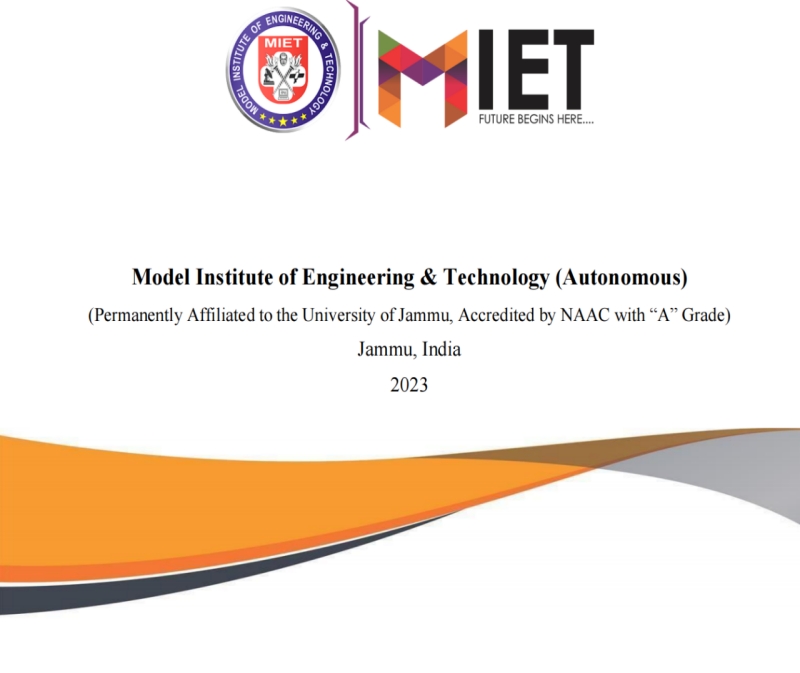
**By**

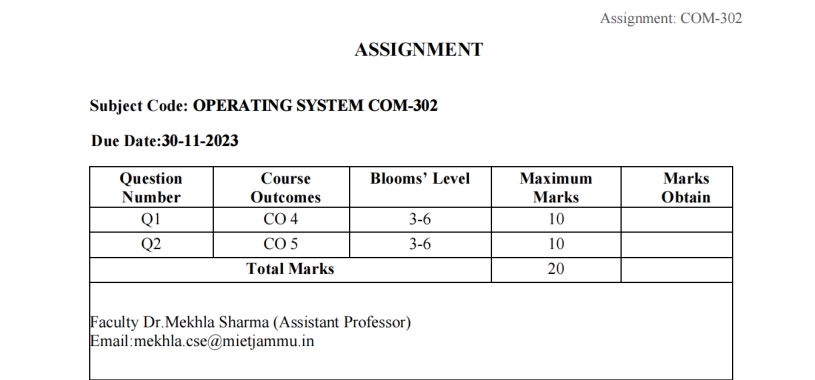
***Gursharan Deep Singh***

**2022A1R042**

**3rd Semester**

**C.S.E**





**INDEX**

|  |  |  |
| --- | --- | --- |
| **S.NO** | **TASK** | **P.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. | **4 - 14** |
| **2** | Design and implement various methods for IPC, such as message passing or shared memory, to facilitate communication between processes in the operating system. | **14 - 21** |

**TASK 1:**

**Round robin scheduling algorithm:**

Round Robin (RR) is a simple and widely used CPU scheduling algorithm in operating systems. It is designed to provide fair access to the CPU for all processes in a system. Here's an explanation of its important terms:

***1. Ready Queue:***

The ready queue is a data structure that holds all the processes that are ready to be executed by the CPU. These processes are in a "ready" state, meaning they are waiting to be scheduled and run.

***2. Time Quantum:***

The time quantum, also known as a time slice or time slot, is a fixed time duration allocated to each process in the ready queue. When a process is scheduled, it is allowed to run for the duration of the time quantum. If the process completes its execution within the time quantum, it is removed from the ready queue. If not, it is moved to the back of the ready queue to wait for its next turn.

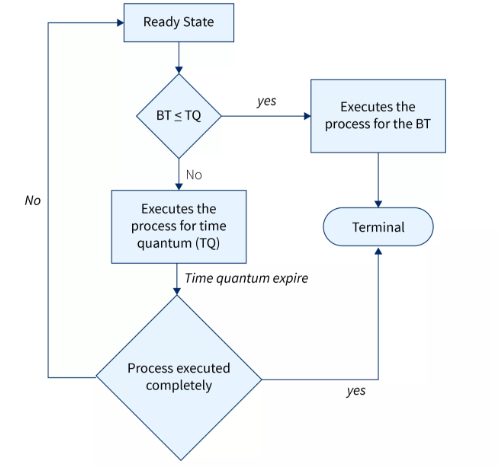
***3. Context Switching:***

Context switching refers to the process of saving and restoring the state of a process so that it can be resumed from where it was paused. In the context of the Round Robin scheduling algorithm, a context switch occurs when the CPU switches from executing one process to another. This involves saving the state of the currently running process, loading the state of the next process from the ready queue, and then transferring control to the new process.

***4. Gantt Chart:***

A Gantt chart is a visual representation of the schedule of processes in a system over time. In the context of Round Robin scheduling, the Gantt chart helps illustrate how each process gets CPU time in a cyclic manner. Each horizontal bar in the chart represents the execution of a process, and the length of the bar corresponds to the time quantum allocated to that process.

**Round Robin Algorithm Execution:**

****

1. Processes in the ready queue are scheduled in a cyclic order.

2. Each process gets a turn to execute for the time quantum.

3. If a process doesn't complete within its time quantum, it is moved to the back of the ready queue.

4. The scheduler moves to the next process in the ready queue.

5. This continues until all processes are completed.

**SOURCE CODE:**

#include<bits/stdc++.h>

using namespace std;

struct Process{

int id;

int arr\_time;

int bursttime;

int rem\_time;

int TAT;

};

void RR(vector<Process>& processes, int QT){

queue<Process> processqueue;

int currenttime=0;

while(!processqueue.empty() || !processes.empty()){

if(!processqueue.empty()){

Process currentprocess = processqueue.front();

processqueue.pop();

int executetime= min(QT, currentprocess.rem\_time);

currentprocess.rem\_time-=executetime;

currenttime +=executetime;

cout<< "Time" << currenttime<< ": Execute " << currentprocess.id<< " [Burst time : " << currentprocess.bursttime

<<" , time remaining: "<<currentprocess.rem\_time<<"]"<<endl;

while(!processes.empty() && processes.front().arr\_time<=currenttime){

processqueue.push(processes.front());

processes.erase(processes.begin());

}

if(currentprocess.rem\_time >0){

processqueue.push(currentprocess);

}

else{

currentprocess.TAT = currenttime-currentprocess.arr\_time;

} }

else{

processqueue.push(processes.front());

processes.erase(processes.begin());

} }}

int main(){

int n;

cout<< "Enter the number of processes; ";

cin>> n;

vector<Process>processes;

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

Process p;

p.id = i + 1;

cout << "Process " << p.id << ": " << endl;

cout << " Arrival Time: ";

cin >> p.arr\_time;

cout << " Burst Time: ";

cin >> p.bursttime;

p.rem\_time = p.bursttime;

processes.push\_back(p);

}

int QT;

cout << "Enter time quantum for Round Robin scheduling: "

cin >> QT;

RR(processes,QT);

}

**CODE ANALYSIS (with example):**

|  |  |  |
| --- | --- | --- |
| PROCESS ID | ARRIVAL TIME | BURST TIME |
| 1 | 0 | 5 |
| 2 | 1 | 6 |
| 3 | 2 | 3 |
| 4 | 3 | 1 |
| 5 | 4 | 5 |
| 6 | 6 | 4 |

Table 1.

***CASE 1: Quantum time = 4 units***

According to the algorithm, we have to maintain the ready queue and the Gantt chart. The structure of both the data structures will be changed after every scheduling.

Ready Queue:

Initially, at time 0, process P1 arrives which will be scheduled for the time slice 4 units. Hence in the ready queue, there will be only one process P1 at starting with CPU burst time 5 units.

|  |
| --- |
| P1 |
| 5 |

GANTT chart: The P1 will be executed for 4 units first.

|  |
| --- |
| P1 |

0 4

Ready Queue: Meanwhile the execution of P1, four more processes P2, P3, P4 and P5 arrives in the ready queue. P1 has not completed yet, it needs another 1 unit of time hence it will also be added back to the ready queue.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P2 | P3 | P4 | P5 | P1 |
| 6 | 3 | 1 | 5 | 1 |

GANTT chart: After P1, P2 will be executed for 4 units of time which is shown in the Gantt chart.

|  |  |
| --- | --- |
| P1 | P2 |

0 4 8

Ready Queue: During the execution of P2, one more process P6 is arrived in the ready queue. Since P2 has not completed yet hence, P2 will also be added back to the ready queue with the remaining burst time 2 units.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| P3 | P4 | P5 | P1 | P6 | P2 |
| 3 | 1 | 5 | 1 | 4 | 2 |

GANTT chart: After P1 and P2, P3 will get executed for 3 units of time since its CPU burst time is only 3 seconds.

|  |  |  |
| --- | --- | --- |
| P1 | P2 | P3 |

0 4 8 11

Ready Queue : Since P3 has been completed, hence it will be terminated and not be added to the ready queue. The next process will be executed is P4.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P4 | P5 | P1 | P6 | P2 |
| 1 | 5 | 1 | 4 | 2 |

GANTT chart: After, P1, P2 and P3, P4 will get executed. Its burst time is only 1 unit which is lesser then the time quantum hence it will be completed.

|  |  |  |  |
| --- | --- | --- | --- |
| P1 | P2 | P3 | P4 |

0 4 8 11 12

Ready Queue: The next process in the ready queue is P5 with 5 units of burst time. Since P4 is completed hence it will not be added back to the queue.

|  |  |  |  |
| --- | --- | --- | --- |
| P5 | P1 | P6 | P2 |
| 5 | 1 | 4 | 2 |

GANTT chart: P5 will be executed for the whole time slice because it requires 5 units of burst time which is higher than the time slice.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P4 | P5 |

0 4 8 11 12 16

Ready Queue: P5 has not been completed yet; it will be added back to the queue with the remaining burst time of 1 unit.

|  |  |  |  |
| --- | --- | --- | --- |
| P1 | P6 | P2 | P5 |
| 1 | 4 | 2 | 1 |

GANTT Chart: The process P1 will be given the next turn to complete its execution. Since it only requires 1 unit of burst time hence it will be completed.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P4 | P5 | P1 |

0 4 8 11 12 16 17

Ready Queue: P1 is completed and will not be added back to the ready queue. The next process P6 requires only 4 units of burst time and it will be executed next.

|  |  |  |
| --- | --- | --- |
| P6 | P2 | P5 |
| 4 | 2 | 1 |

GANTT chart: P6 will be executed for 4 units of time till completion.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P4 | P5 | P1 | P6 |

0 4 8 11 12 16 17 21

Ready Queue: Since P6 is completed, hence it will not be added again to the queue. There are only two processes present in the ready queue. The Next process P2 requires only 2 units of time.

|  |  |
| --- | --- |
| P2 | P5 |
| 2 | 1 |

GANTT Chart: P2 will get executed again, since it only requires only 2 units of time hence this will be completed.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P4 | P5 | P1 | P6 | P2 |

0 4 8 11 12 16 17 21 23

Ready Queue: Now, the only available process in the queue is P5 which requires 1 unit of burst time. Since the time slice is of 4 units hence it will be completed in the next burst.

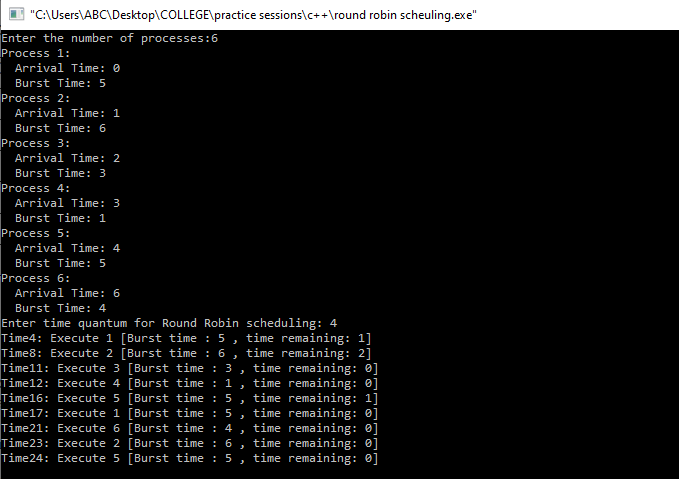
|  |
| --- |
| P5 |
| 1 |

GANTT chart: P5 will get executed till completion.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P4 | P5 | P1 | P6 | P2 | P5 |

0 4 8 11 12 16 17 21 23 24

OUTPUT:



***CASE 2: Quantum time = 2 units***

Similarly, in case 2 the process will run up quantum time 2 and then reschedule itself in the ready queue. After rescheduling the remaining burst time of the process is executed.

|  |  |  |
| --- | --- | --- |
| PROCESS ID | ARRIVAL TIME | BURST TIME |
| 1 | 0 | 5 |
| 2 | 1 | 6 |
| 3 | 2 | 3 |
| 4 | 3 | 1 |
| 5 | 4 | 5 |
| 6 | 6 | 4 |

Table 2

Final GANTT chart for this case is:

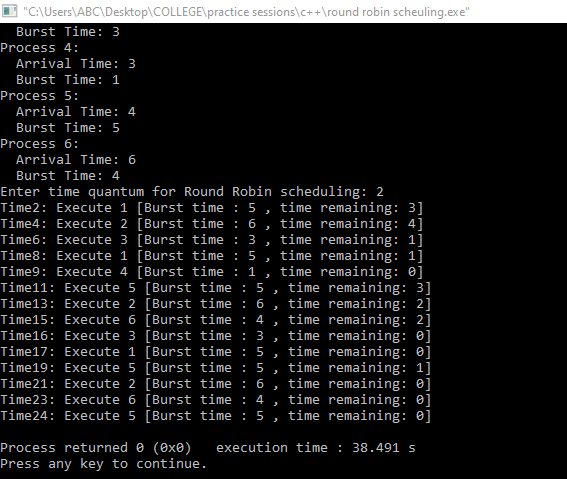
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P1 | P4 | P5 | P2 | P6 | P3 | P1 |

0 2 4 6 8 9 11 13 15 16 17

|  |  |  |  |
| --- | --- | --- | --- |
| P5 | P2 | P6 | P5 |

17 19 21 23 24

OUTPUT:



***CASE 3: Quantum time = 3 units***

Similarly, in case 3 the process will run up quantum time 2 and then reschedule itself in the ready queue. After rescheduling the remaining burst time of the process is executed.

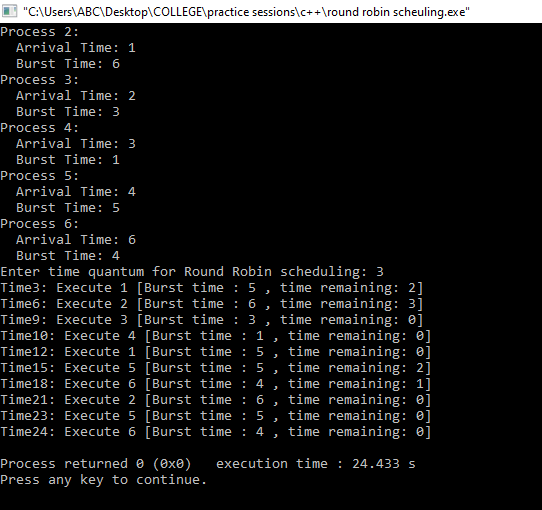
|  |  |  |
| --- | --- | --- |
| PROCESS ID | ARRIVAL TIME | BURST TIME |
| 1 | 0 | 5 |
| 2 | 1 | 6 |
| 3 | 2 | 3 |
| 4 | 3 | 1 |
| 5 | 4 | 5 |
| 6 | 6 | 4 |

Final GANTT chart for this case is:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P4 | P1 | P5 | P6 | P2 | P5 | P6 |

0 3 6 9 10 12 15 18 21 23 24

OUTPUT:



TASK 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:

Inter process communication (IPC) was the transfer of information and interaction between multiple processes in an electronic system. Every operation in a tasking process structure runs on its own, as well as communication between them is required if these processes require to exchange of information or coordination of what they are doing.

IPC is an essential part of contemporary operating systems and can be employed in a variety of applications, which include simple control-line appliances to complicated systems with distributed components. The primary goal of IPC is to make the transmission of knowledge among processes more private and effective.

Here are several IPC methods accessible, each of which has its own set of benefits and drawbacks. Several of the most common IPC techniques are −

* **Semaphores** − Semaphores serve to keep the utilization of resources that are shared synchronized. Their companies serve as responses that limit the number of procedures that may utilize a resource that is shared at any given time. Semaphores are useful for implementing critical sections in which only one process has access to a resource that is shared at a time.
* **Socket** − Sockets constitute an internet-based communications process that enables procedures to interact with one another over a network. Someone can communicate both locally and remotely. In client-server relationships applications, ports are frequently used.
* **Remote Procedure call(RPC)** − RPC is a procedure that enables a single process to call an operation in another. It allows procedures to call treatments in distant systems as though they were actually local, enabling distributed computing. In systems with distributed components, RPC is frequently used.
* **Signals** − Asynchronous IPC signals are employed for informing an operator of an occurrence or interference. The Operating System (OS) sends communication through processes as well as between processes. Programming based on events can be implemented using signals.
* **Message Queues** − Message queues are employed for inter-process interaction when both the sending and getting processes do not need to be present at the same time. The asynchronous communications may be sent and received. A message in a queue possesses a particular final destination and is accessible to multiple processes.

**Pipes** − A pipe is a channel of communication that is one-way that enables a single procedure to transmit data to a different one. Pipes can be identified as or unidentified. The operations running in anonymous pipes have to be associated (i.e., both parent and child processes). Known pipes, on the contrary together, may be utilized by processes that are separate from one another.

**SOURCE CODE(PIPES) :**

#include <bits/stdc++.h>

#include <unistd.h>

#include <string>

int main() {

int pipefd[2]; // Pipe file descriptors: [0] for reading, [1] for writing

// Create a pipe

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

perror("pipe");

return 1;

}

// Fork to create a child process

pid\_t pid = fork();

if (pid == -1) {

perror("fork");

return 1;

}

if (pid == 0) { // Child process (receiver)

// Close the write end since the child is reading

close(pipefd[1]);

// Read the message from the pipe

char buffer[100];

ssize\_t bytesRead = read(pipefd[0], buffer, sizeof(buffer));

// Display the received message

if (bytesRead > 0) {

std::cout << "Child - Received Message: " << buffer << std::endl;

} else {

std::cerr << "Error reading from pipe in child." << std::endl;

}

// Close the read end

close(pipefd[0]);

}

else { // Parent process (sender)

// Close the read end since the parent is writing

close(pipefd[0]);

// Write a message to the pipe

std::string message = "Hello from the parent!";

ssize\_t bytesWritten = write(pipefd[1], message.c\_str(), message.length());

// Display a confirmation message

if (bytesWritten > 0) {

std::cout << "Parent - Sent Message: " << message << std::endl;

} else {

std::cerr << "Error writing to pipe in parent." << std::endl;

}

// Close the write end

close(pipefd[1]);

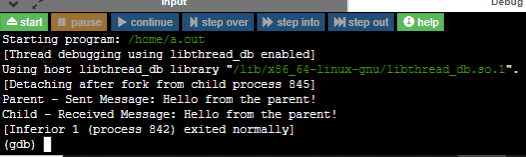
// Wait for the child process to finish

}

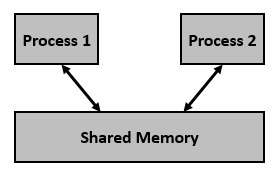
return 0;

}

OUTPUT:



**Shared Memory** − Shared memory is an inter process communication method that enables various programs to make use of a single storage region. This allows them to effectively and effectively share data. Sharing memory is frequently employed in applications that are extremely fast.



Using shared memory for Inter-Process Communication (IPC) involves several steps. Below is a simplified algorithm outlining the process:

Shared Memory IPC Algorithm:

**1. Create or Access a Shared Memory Segment:**

- Generate a unique key for the shared memory segment using `ftok` or another method.

- Use `shmget` to create or access the shared memory segment, specifying its key, size, and permissions.

**2. Attach the Shared Memory Segment:**

- Use `shmat` to attach the shared memory segment to the process's address space.

- Check for errors in the attachment process.

**3. Read/Write Data in Shared Memory:**

- Once attached, the shared memory can be treated as a regular memory array.

- Multiple processes can read from and write to the shared memory.

**4. Detach the Shared Memory Segment:**

- Use `shmdt` to detach the shared memory segment from the process's address space.

- This step is crucial to release resources and ensure proper cleanup.

**5. Remove the Shared Memory Segment:**

- If the shared memory segment is no longer needed, use `shmctl` with the `IPC\_RMID` command to remove it.

- This should be done after all processes have detached the shared memory.

**SOURCE CODE:**

#include <iostream>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <unistd.h>

const int SHARED\_MEMORY\_SIZE = 1024; // Size of the shared memory segment

struct SharedMemory {

char message[SHARED\_MEMORY\_SIZE];

};

int main() {

// Generate a unique key for the shared memory segment

key\_t key = ftok("/tmp", 'X');

// Create or access the shared memory segment

int shmid = shmget(key, sizeof(SharedMemory), IPC\_CREAT | 0666);

if (shmid == -1) {

perror("shmget");

return 1;

}

// Attach the shared memory segment to the process's address space

SharedMemory\* sharedMemory = static\_cast<SharedMemory\*>(shmat(shmid, nullptr, 0));

if (sharedMemory == reinterpret\_cast<SharedMemory\*>(-1)) {

perror("shmat");

return 1;

}

pid\_t pid = fork();

if (pid == -1) {

perror("fork");

return 1;

}

if (pid == 0) { // Child process (consumer)

// Wait for the parent to write a message

while (strlen(sharedMemory->message) == 0) {

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

}

}

else {

std::string message = "Hello from the parent!";

strncpy(sharedMemory->message, message.c\_str(), SHARED\_MEMORY\_SIZE - 1);

wait(nullptr);

shmdt(sharedMemory);

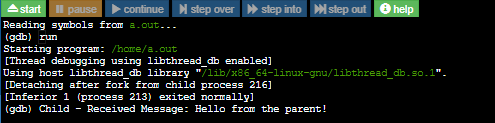
shmctl(shmid, IPC\_RMID, nullptr);

}

return 0;

}

OUTPUT:



End