- **Model Institute of Engineering & Technology(Autonomous)**
  (Permanently Affiliated to the University of Jammu, Accredited by NAAC with "A" Grade) Jammu, India

By :- Karan Sharma
Roll Number - 2022a1r009
Section A2
Sem :- 3rd

ASSIGNMENT

Subject Code: COM-302 (Operating System)

Due Date: 04/12/2023

Question Number	Course Outcomes	Blooms' Level	Maximum Marks	Marks Obtain
Q1	**CO1, CO2 & CO5**	3-4	10	
Q2	**CO3, CO4**	3-4	10	
Total Marks			20	

Submitted to: Mrs. Mekhla Sharma

Faculty Signature:
Email:

TASK 1

-><u>Write a program in a language of your choice to simulate various CPU scheduling algorithms such as First-Come-First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority Scheduling. Compare and analyze the performance of these algorithms using different test cases and metrics like turnaround time, waiting time, and response time.</u>

:->The efficiency of CPU scheduling algorithms plays a crucial role in shaping the overall performance of operating systems in their dynamic environments. Within this introduction, we explore and analyze four fundamental CPU scheduling algorithms: First-Come-First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority Scheduling.

Provided below is a straightforward C program that replicates diverse CPU scheduling algorithms, including First-Come-First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority Scheduling. This program prompts users to input the number of processes, burst times, and arrival times, subsequently simulating the execution of these processes using the specified scheduling algorithms.

Featuring a user-friendly interface, the program empowers users to input various process scenarios, offering insights into the distinct attributes and performance metrics associated with each algorithm. Serving as an educational tool, this application facilitates a deeper understanding and informed decision-making in the domain of operating system design and optimization. Through the generation of visual representations and support for comparative analysis, users can gain a comprehensive grasp of CPU scheduling intricacies.

```c
#include<stdio.h>
#include<stdlib.h>

// Process structure
struct Process
{
    int process_id;
    int arrival_time;
    int burst_time;
    int waiting_time;
    int turnaround_time;
    int int Process::turnaround_time
    int priority; // Used for Priority Scheduling
};

// Function to swap two processes
void swap(struct Process *xp, struct Process *yp)
{
    struct Process temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// Function to perform First-Come-First-Served (FCFS) scheduling
void fcfs(struct Process processes[], int n)
{
    int currentTime = 0;
    for (int i = 0; i < n; i++)
    {
        processes[i].waiting_time = currentTime - processes[i].arrival_time;
```

```
        if (processes[i].waiting_time < 0)
        {
            processes[i].waiting_time = 0;
            currentTime = processes[i].arrival_time;
        }
        processes[i].completion_time = currentTime + processes[i].burst_time;
        processes[i].turnaround_time = processes[i].completion_time - processes[i].arrival_time;
        currentTime = processes[i].completion_time;
    }
}

// Function to perform Shortest Job First (SJF) scheduling
void sjf(struct Process processes[], int n)
{
    // Sort processes based on burst time
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (processes[j].burst_time > processes[j + 1].burst_time)
            {
                swap(&processes[j], &processes[j + 1]);
            }
        }
    }
    fcfs(processes, n);
}

// Function to perform Round Robin (RR) scheduling
void roundRobin(struct Process processes[], int n, int timeQuantum)
```

```
{
    int currentTime = 0;
    while (1)
    {
        int done = 1;
        for (int i = 0; i < n; i++)
        {
            if (processes[i].burst_time > 0)
            {
                done = 0;
                if (processes[i].burst_time > timeQuantum)
                {
                    currentTime += timeQuantum;
                    processes[i].burst_time -= timeQuantum;
                }
                else
                {
                    currentTime += processes[i].burst_time;
                    processes[i].waiting_time = currentTime - processes[i].arrival_time - processes[i].burst_time;
                    processes[i].burst_time = 0;
                    processes[i].completion_time = currentTime;
                    processes[i].turnaround_time = processes[i].completion_time - processes[i].arrival_time;
                }
            }
        }
        if (done == 1)
            break;
    }
}

```

```c
// Function to perform Priority Scheduling
void priorityScheduling(struct Process processes[], int n)
{
    // Sort processes based on priority
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (processes[j].priority > processes[j + 1].priority)
            {
                swap(&processes[j], &processes[j + 1]);
            }
        }
    }
    fcfs(processes, n);
}

// Function to display the details of processes
void displayProcesses(struct Process processes[], int n)
{
    printf("Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\tCompletion Time\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].process_id,
               processes[i].arrival_time, processes[i].burst_time,
               processes[i].waiting_time, processes[i].turnaround_time,
               processes[i].completion_time);
    }
}
```

```c
int main()
{
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process processes[n];

    // Input process details
    for (int i = 0; i < n; i++)
    {
        processes[i].process_id = i + 1;
        printf("Enter arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrival_time);
        printf("Enter burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
        printf("Enter priority for process %d: ", i + 1);
        scanf("%d", &processes[i].priority);
    }

    // Perform FCFS scheduling
    printf("\nFCFS Scheduling:\n");
    fcfs(processes, n);
    displayProcesses(processes, n);

    // Reset process details for SJF scheduling
    for (int i = 0; i < n; i++)
    {
        processes[i].waiting_time = 0;
        processes[i].turnaround_time = 0;
        processes[i].completion_time = 0;
```

```
    }

    // Perform SJF scheduling
    printf("\nSJF Scheduling:\n");
    sjf(processes, n);
    displayProcesses(processes, n);

    // Reset process details for Round Robin scheduling
    for (int i = 0; i < n; i++)
    {
        processes[i].waiting_time = 0;
        processes[i].turnaround_time = 0;
        processes[i].completion_time = 0;
    }

    // Perform Round Robin scheduling
    int timeQuantum;
    printf("\nEnter the time quantum for Round Robin scheduling: ");
    scanf("%d", &timeQuantum);
    printf("\nRound Robin Scheduling:\n");
    roundRobin(processes, n, timeQuantum);
    displayProcesses(processes, n);

    // Reset process details for Priority Scheduling
    for (int i = 0; i < n; i++)
    {
        processes[i].waiting_time = 0;
        processes[i].turnaround_time = 0;
        processes[i].completion_time = 0;
    }

    // Perform Priority Scheduling
    printf("\nPriority Scheduling:\n");
    priorityScheduling(processes, n);
    displayProcesses(processes, n);

    return 0;
}
```

This code facilitates the input of details for different processes, including arrival time, burst time, and priority. Subsequently, it simulates four scheduling algorithms: FCFS, SJF, RR (with a user-specified time quantum), and Priority Scheduling. The output includes the waiting times for each process, as well as the turnaround time and completion time for each algorithm.

OUTPUT

```
Enter the number of processes: 4
Enter arrival time for process 1: 1
Enter burst time for process 1: 2
Enter priority for process 1: 3
Enter arrival time for process 2: 4
Enter burst time for process 2: 5
Enter priority for process 2: 6
Enter arrival time for process 3: 6
Enter burst time for process 3: 5
Enter priority for process 3: 4
Enter arrival time for process 4: 3
Enter burst time for process 4: 2
Enter priority for process 4: 1

FCFS Scheduling:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time Completion Time
1       1               2               0               2               3
2       4               5               0               5               9
3       6               5               3               8               14
4       3               2               11              13              16

SJF Scheduling:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time Completion Time
1       1               2               0               2               3
4       3               2               0               2               5
2       4               5               1               6               10
3       6               5               4               9               15

Enter the time quantum for Round Robin scheduling: 3

Round Robin Scheduling:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time Completion Time
1       1               0               -1              1               2
4       3               0               -1              1               4
2       4               0               6               8               12
3       6               0               6               8               14

Priority Scheduling:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time Completion Time
4       3               0               0               0               3
1       1               0               2               2               3
3       6               0               0               0               6
2       4               0               2               2               6

Process returned 0 (0x0)   execution time : 150.087 s
Press any key to continue.
```

EXPLANATION

:->**Code Explanation:**

->Struct Definition:

->Defines a structure named Process to hold information about each process, including process ID, arrival time, burst time, waiting time, turnaround time, completion time, and priority.

->Function swap:

A utility function to swap two Process structures. Used for sorting processes based on burst time and priority.

->FCFS Scheduling (FCFS function):

Simulates First-Come-First-Served scheduling.

Calculates waiting time, completion time, and turnaround time for each process.

->SJF Scheduling (SJF function):

Sorts processes based on burst time.

Calls FCFS to simulate Shortest Job First scheduling.

->Round Robin Scheduling (Round Robin function):

Simulates Round Robin scheduling with a specified time quantum.

Calculates waiting time, completion time, and turnaround time for each process.

->Priority Scheduling (Priority Scheduling function):

Sorts processes based on priority.

Calls FCFS to simulate Priority Scheduling.

->Display Processes (Display Processes function):

Prints a table with details of each process, including process ID, arrival time, burst time, waiting time, turnaround time, and completion time.

->Main Function (main):

Takes user input for the number of processes and details of each process (arrival time, burst time, and priority).

Performs FCFS, SJF, Round Robin, and Priority Scheduling.

Displays the results for each scheduling algorithm.

:->Output Explanation:

The output displays a table for each scheduling algorithm, showing process details and relevant metrics (waiting time, turnaround time, completion time).

```
FCFS Scheduling:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time Completion Time
1       1               2               0               2               3
2       4               5               0               5               9
3       6               5               3               8               14
4       3               2               11              13              16

SJF Scheduling:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time Completion Time
1       1               2               0               2               3
4       3               2               0               2               5
2       4               5               1               6               10
3       6               5               4               9               15

Enter the time quantum for Round Robin scheduling: 3

Round Robin Scheduling:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time Completion Time
1       1               0               -1              1               2
4       3               0               -1              1               4
2       4               0               6               8               12
3       6               0               6               8               14

Priority Scheduling:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time Completion Time
4       3               0               0               0               3
1       1               0               2               2               3
3       6               0               0               0               6
2       4               0               2               2               6

Process returned 0 (0x0)   execution time : 150.087 s
Press any key to continue.
```

You'll observe the impact of different scheduling algorithms on the overall performance of the system, with variations in waiting times and turnaround times for each process.

TASK 2

:->Write a multi-threaded program in C or another suitable language to solve the classic Producer Consumer problem using semaphores or mutex locks. Describe how you ensure synchronization and avoid race conditions in your solution.

->This abstract delves into the realm of multi-threaded programming in the C language, addressing the timeless Producer-Consumer problem. Through the adept use of synchronization mechanisms like semaphores or mutex locks, the program orchestrates seamless communication and coordination between producer and consumer threads. These mechanisms act as guardians, preventing race conditions, maintaining data integrity, and averting resource conflicts.

• Presented below is a C program that tackles the Producer-Consumer problem with a dual approach—utilizing both mutex locks and semaphores.

• This solution not only ensures data consistency but also amplifies parallelism and efficiency in a shared-memory environment. It highlights the prowess of concurrent programming, showcasing how C's versatility and control are instrumental in crafting robust solutions for intricate synchronization challenges inherent in the classical Producer-Consumer paradigm.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int count = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t full, empty;

void* producer(void* arg)
{
    for (int i = 0; i < 10; ++i)
    {
        int item = rand() % 50 + 1; // Produce a different random item
        sem_wait(&empty);            // Wait for an empty slot
        pthread_mutex_lock(&mutex);
        buffer[count++] = item;
        printf("Produced item: %d\n", item);
        pthread_mutex_unlock(&mutex);
        sem_post(&full); // Signal that a slot is now full
    }
    pthread_exit(NULL);
}

void* consumer(void* arg)
{
```

```c
    for (int i = 0; i < 10; ++i)
    {
        sem_wait(&full); // Wait for a full slot
        pthread_mutex_lock(&mutex);
        int item = buffer[--count];
        printf("Consumed item: %d\n", item);
        pthread_mutex_unlock(&mutex);
        int pthread_mutex_unlock(pthread_mutex_t* m) ow empty
    }
    pthread_exit(NULL);
}

int main()
{
    pthread_t producer_thread, consumer_thread;

    // Initialize semaphores
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);

    // Create producer and consumer threads
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    // Wait for threads to finish
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    // Clean up
    pthread_mutex_destroy(&mutex);
    sem_destroy(&full);
    sem_destroy(&empty);

    return 0;
}
```

In this illustration, synchronization is achieved through the utilization of both mutex locks and semaphores. The pthread_mutex_t type is employed to instantiate a mutex lock, while the sem_t type is employed for creating semaphores. Critical sections are safeguarded using the pthread_mutex_lock and pthread_mutex_unlock functions, ensuring exclusive access. Additionally, the sem_wait and sem_post functions regulate access to the shared buffer, employing semaphores to manage synchronization.

OUTPUT

The output of the provided C program, addressing the Producer-Consumer problem with semaphores and mutex locks, typically reveals the dynamic interactions between producer and consumer threads. It showcases the coordinated operations on a shared buffer, illustrating the production and consumption of items. The specifics of the output will include details such as the produced and consumed item values, offering insights into the synchronization achieved through the implemented mechanisms.

```
Produced item: 42
Produced item: 18
Produced item: 35
Produced item: 1
Produced item: 20
Consumed item: 20
Consumed item: 1
Consumed item: 35
Consumed item: 18
Consumed item: 42
Produced item: 25
Produced item: 29
Produced item: 9
Produced item: 13
Produced item: 15
Consumed item: 15
Consumed item: 13
Consumed item: 9
Consumed item: 29
Consumed item: 25

Process returned 0 (0x0)   execution time : 0.033 s
Press any key to continue.
```

Each "Produced" line in the output signifies an item added to the buffer by the producer thread, while each "Consumed" line indicates the consumption of an item by the consumer thread. This alternating pattern in the output illustrates the synchronized behavior of production and consumption within the program. The coordination between the producer and consumer threads ensures a seamless flow of items in and out of the shared buffer, demonstrating the effective implementation of synchronization mechanisms.

Step-by-Step explanation of the C code:

Step 1 : Include Necessary Libraries-

```
#include <stdio.h>
```

```
#include <pthread.h>

#include <semaphore.h>
```

Step 2: Define Constants and Global Variables

```
#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];

int count = 0; // Number of items in the buffer
```

Step 3: Initialize Mutex and Semaphores

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

sem_t full, empty;

pthread_mutex_init(&mutex, NULL);

sem_init(&full, 0, 0);

sem_init(&empty, 0, BUFFER_SIZE);
```

Initialize a mutex and two semaphores: full to track the number of items in the buffer, and empty to track the number of empty slots.

Step 4: Producer Function

```
void* producer(void* arg)

{

   for (int i = 0; i < 10; ++i)

   {

      int item = rand() % 50 + 1; // Produce a different random item

      sem_wait(&empty);           // Wait for an empty slot

      pthread_mutex_lock(&mutex);

      buffer[count++] = item;

      printf("Produced item: %d\n", item);

      pthread_mutex_unlock(&mutex);
```

```
        sem_post(&full); // Signal that a slot is now full
    }
    pthread_exit(NULL);
}
```

->In the producer function:

-Produce an item.

-Wait on the empty semaphore if the buffer is full.

-Acquire the mutex lock to ensure exclusive access to the buffer.

-Add the item to the buffer, update the count, and print the produced item & release the mutex lock.

-Signal that the buffer is not empty by posting to the full semaphore.

Step 5: Consumer Function

```
void* consumer(void* arg)
{
    for (int i = 0; i < 10; ++i)
    {
        sem_wait(&full); // Wait for a full slot
        pthread_mutex_lock(&mutex);
        int item = buffer[--count];
        printf("Consumed item: %d\n", item);
        pthread_mutex_unlock(&mutex);
        sem_post(&empty); // Signal that a slot is now empty
    }
    pthread_exit(NULL);
}
```

->In the consumer function:

-Wait on the full semaphore if the buffer is empty.

-Acquire the mutex lock.

-Consume an item from the buffer, update the count, and print the consumed item.

-Release the mutex lock.

-Signal that the buffer is not full by posting to the empty semaphore.

Step 6: Main Function

```
int main()
{
    pthread_t producer_thread, consumer_thread;

    // Initialize semaphores
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);

    // Create producer and consumer threads
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    // Wait for threads to finish (this will never happen in this example)
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    // Clean up
```

```
    pthread_mutex_destroy(&mutex);

    sem_destroy(&full);

    sem_destroy(&empty);


    return 0;

}
```

->In the main function:

-Create the producer and consumer threads.

-Wait for threads to finish (Note: In this example, the threads run indefinitely, so the pthread_join calls are not reached).

-Clean up by destroying the mutex and semaphores.

Overall Workflow:

->Producer Workflow:

-The producer waits on the empty semaphore, which signifies the number of empty slots in the buffer.

-Once there's an empty slot available (sem_wait passes), it acquires the mutex lock to modify the buffer, adds an item, and increments the count.

-It releases the mutex lock and signals the full semaphore to notify consumers that an item is available.

->Consumer Workflow:

-The consumer waits on the full semaphore, indicating the number of full slots in the buffer.

-When there's an item to consume (sem_wait passes), it acquires the mutex lock to access the buffer, consumes an item, decrements the count, and releases the mutex lock.

-It signals the empty semaphore to inform producers that there is an empty slot available.

Race Condition Avoidance:

->Mutex Locks: Ensure exclusive access to critical sections, preventing multiple threads from simultaneously modifying shared data, avoiding race conditions.

->Semaphores: Control access to the buffer, ensuring that producers and consumers wait or proceed based on the availability of resources (empty and full slots), thereby preventing conflicts in accessing shared resources.

->By combining mutex locks to protect critical sections and semaphores to control access to the buffer, this solution ensures synchronization between the producer and consumer threads and effectively avoids race conditions when accessing and modifying shared resources.

GROUP PICTURE:


WROGN