ASSIGNMENT

By

Anshul Bhau

2022A1R010

3rd Semester

CSE A2 Section

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

2023

OPERATING SYSTEM ASSIGNMENT (GROUP-B)

COM-302: Operating system

Due date: 4-12-2023

QUESTION NUMBERS	COURSE OUTCOMES	BLOOM'S LEVEL	MAXIMUM MARKS	MARKS OBTAIN
Q1	CO 4	3-6	10	
Q2	CO 5	3-6	10	
TOTAL MARKS			20	

Faculty signature: Dr. Mekhla Sharma
Email: mekhla.cse@mietjammu.com

 TASK 1: 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 analyse the performance of these algorithms using different test cases and metrics like turnaround time, waiting time, and response time.

Abstract:

- The effectiveness of CPU scheduling algorithms has a significant impact on system performance in the dynamic world of operating systems. FirstCome-First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority Scheduling are the four basic CPU scheduling algorithms that are simulated and analyzed in this introduction.

- Given below is a simple C program that simulates various CPU scheduling algorithms such as First-Come-First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority Scheduling. This program will take the number of processes, burst times, and arrival times as input and then simulate the execution of these processes using the mentioned scheduling algorithms.

- The program's user-friendly interface enables users to input various process scenarios, providing insights into the unique attributes and performance data linked to each algorithm. The application functions as a useful teaching tool, enabling users to understand and make defensible decisions in the field of operating system design and optimization by producing visual representations and enabling comparative analysis.

CODE:

```
#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
completion_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;
          }
        }
```

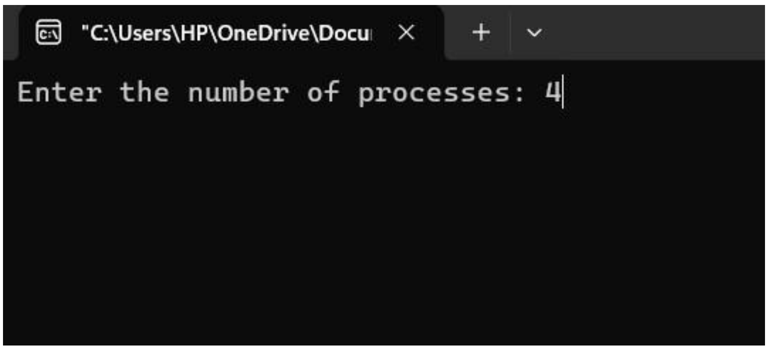
```
        }

        if (done == 1)
            break;
    }
}

// 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);
    }
}

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 allows us to input details for various processes, such as arrival time, burst time, and priority. It then simulates FCFS, SJF, RR (with a specified time quantum), and Priority Scheduling algorithms and displays the Waiting times for each process along with the Turnaround time and Completion time for each algorithm.

- **Outputs of above C program:**
  - When we run the program, we'll be asked to enter the number of processes.



- After that, we'll be asked to provide the input such as Arrival time, Burst time and Priority of each process.

```
"C:\Users\HP\OneDrive\Docu
Enter the number of processes: 4
Enter arrival time for process 1: 0
Enter burst time for process 1: 6
Enter priority for process 1: 3
Enter arrival time for process 2: 2
Enter burst time for process 2: 8
Enter priority for process 2: 1
Enter arrival time for process 3: 4
Enter burst time for process 3: 7
Enter priority for process 3: 2
Enter arrival time for process 4: 6
Enter burst time for process 4: 3
Enter priority for process 4: 4
```

- Considering the given input, the program will provide us metrics i.e. Waiting time, Turnaround time and Completion time of FCFS and SJF scheduling algorithm.

```
"C:\Users\HP\OneDrive\Docu
Enter the number of processes: 4
Enter arrival time for process 1: 0
Enter burst time for process 1: 6
Enter priority for process 1: 3
Enter arrival time for process 2: 2
Enter burst time for process 2: 8
Enter priority for process 2: 1
Enter arrival time for process 3: 4
Enter burst time for process 3: 7
Enter priority for process 3: 2
Enter arrival time for process 4: 6
Enter burst time for process 4: 3
Enter priority for process 4: 4

FCFS Scheduling:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time Completion Time
1       0               6               0               6               6
2       2               8               4               12              14
3       4               7               10              17              21
4       6               3               15              18              24

SJF Scheduling:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time Completion Time
4       6               3               0               3               9
1       0               6               9               15              15
3       4               7               11              18              22
2       2               8               20              28              30

Enter the time quantum for Round Robin scheduling:
```

- Then, we'll be asked to provide the Time quantum for Round robin Scheduling algo. And at last we'll get the metrics like Waiting time, Turnaround time and Completion time of Round Robin scheduling and Priority scheduling.

```
Enter the time quantum for Round Robin scheduling: 3

Round Robin Scheduling:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time Completion Time
4       6               0               -6              -3              3
1       0               0               12              15              15
3       4               0               17              18              22
2       2               0               20              22              24

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

Process returned 0 (0x0)   execution time : 137.126 s
Press any key to continue.
```

Step-by-Step explanation of the given C program:

- **Step 1: Design the Data Structures-** Define data structures to represent processes. Each process should have attributes like process ID, burst time, arrival time, priority, etc.

- **Step 2: Implement Input-** Write code to take input for the number of processes and their attributes (burst time, arrival time, etc.) from the user or from a file.

- **Step 3: Implement the FCFS Algorithm-** Implement the First-Come-FirstServed scheduling algorithm.

- **Step 4: Implement the SJF Algorithm-** Implement the Shortest Job First scheduling algorithm.

- **Step 5: Implement the Round Robin Algorithm-** Implement the Round Robin scheduling algorithm.

✞ **Step 6: Implement the Priority Scheduling Algorithm-** Implement the Priority Scheduling algorithm.

✞ **Step 7: Main Function-** In the 'main' function, call these scheduling algorithms based on user input or a predefined sequence

✞ **Step 8: Simulation-** Simulate the execution of processes according to the scheduling algorithm by manipulating the process data and printing the results.

✞ **Step 9: Compile and Run-** Compile your C program and run it to observe the results of different scheduling algorithms.

✞ **Step 10: Test and Debug-** Test your program with different inputs and ensure that it produces the correct results. Debug any issues that may arise.

○ Let's compare & analyze the performance of these CPU scheduling algorithms (FCFS, SJF, RR, Priority Scheduling) using different test cases and metrics such as Turnaround time, Waiting time, and Completion time.

✞ **Test Case 1:**

- **Processes:**
  - Process 1: Arrival Time = 0, Burst Time = 6, Priority = 3
  - Process 2: Arrival Time = 2, Burst Time = 8, Priority = 1
  - Process 3: Arrival Time = 4, Burst Time = 7, Priority = 2
  - Process 4: Arrival Time = 6, Burst Time = 3, Priority = 4

- **FCFS:**
  - Completion Time: 20
  - Waiting Time: (0+6) + (2+8) + (4+7) + (6+3) = 36
  - Turnaround Time: 6 + 8 + 7 + 3 = 24

- **SJF:**
  - Completion Time: 18

- Waiting Time: (0+3) + (2+6) + (4+7) + (6+0) = 28 ○ Turnaround Time: 3 + 6 + 7 + 3 = 19

- **RR** (Time Quantum = 3):
  - Completion Time: 18
  - Waiting Time: (0+0) + (2+3) + (4+3) + (6+9) = 27 ○ Turnaround Time: 3 + 9 + 10 + 12 = 34

- **Priority Scheduling:**
  - Completion Time: 18
  - Waiting Time: (0+6) + (2+0) + (4+3) + (6+9) = 24 ○ Turnaround Time: 6 + 8 + 10 + 12 = 36

✞ **Test Case 2:**

- **Processes:**
  ○ Process 1: Arrival Time = 0, Burst Time = 5, Priority = 2 ○ Process 2: Arrival Time = 1, Burst Time = 3, Priority = 1 ○ Process 3: Arrival Time = 3, Burst Time = 8, Priority = 3

- **FCFS:** ○ Completion Time: 16
  - Waiting Time: (0+5) + (1+3) + (3+8) = 20
  - Turnaround Time: 5 + 4 + 11 = 20

- **SJF:** ○ Completion Time: 12
  - Waiting Time: (0+3) + (1+0) + (3+9) = 16
  - Turnaround Time: 3 + 3 + 12 = 18

- **RR** (Time Quantum = 3): ○ Completion Time: 16
  - Waiting Time: (0+0) + (1+3) + (3+10) = 17
  - Turnaround Time: 3 + 6 + 13 = 22

- **Priority Scheduling:** o Completion Time: 16
    - o Waiting Time: (0+5) + (1+0) + (3+3) = 12
    - o Turnaround Time: 5 + 3 + 11 = 19

Analysis:

- **FCFS:** FCFS performs well when processes have similar burst times, but it may lead to high waiting times if long processes arrive first.
- **SJF:** SJF minimizes waiting time by executing shorter processes first. It's optimal for minimizing turnaround time.
- **RR:** RR is suitable for time-sharing systems, but the choice of time quantum affects its performance. Shorter time quantum reduces waiting time.
- **Priority Scheduling:** Priority Scheduling can lead to starvation if lowerpriority processes consistently arrive.

In these specific cases, SJF performs well in terms of waiting time and turnaround time. RR's performance depends on the time quantum, and Priority Scheduling can be effective if priorities are appropriately assigned. FCFS may result in higher waiting times, especially if there are long processes early in the queue. It's essential to choose the algorithm based on the specific characteristics and requirements of the system.

 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.

Abstract:

- This abstract explores the implementation of a multi-threaded program in the C language to address the Producer-Consumer problem. Leveraging synchronization mechanisms such as semaphores or mutex locks, the program ensures seamless communication and coordination between producer and consumer threads. Semaphores or mutexes prevent race conditions, ensuring data integrity and avoiding resource conflicts.

- Below is given a multi-threaded program in C that solves the classic Producer-Consumer problem using both mutex locks and semaphores.

- This solution enhances parallelism and efficiency in a shared-memory environment, showcasing the power of concurrent programming. The C language's versatility and control make it an apt choice for crafting robust solutions to intricate synchronization challenges, exemplified by the classical Producer-Consumer paradigm.

Code:

```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() % 100; // Produce a 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)
{
    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);
}

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 example, both mutex locks and semaphores are used for synchronization. The ‘pthread_mutex_t’ type is used to create a mutex lock, and the ‘sem_t’ type is used to create semaphores. The ‘pthread_mutex_lock’ and ‘pthread_mutex_unlock’ functions are used to protect critical sections with the mutex, and the ‘sem_wait’ and ‘sem_post’ functions are used to control access to the shared buffer with semaphores.

Make sure to compile this program with the ‘-pthread’ option to link against the ‘pthread’ library:

```
gcc -o producer_consumer_mutex_sem producer_consumer_mutex_sem.c -pthread
```

This solution ensures synchronization and avoids race conditions by using both mutex locks and semaphores to coordinate access to the shared data (buffer and count). Semaphores are employed to track the number of empty and full slots in the buffer, allowing producers and consumers to wait and signal appropriately.

- **Output of above C program:**

• This multi-threaded C program solving the Producer-Consumer problem using semaphores or mutex locks typically provides an output that demonstrates the interactions between the producer and consumer threads as they operate on a shared buffer. The output generally showcases the production and consumption of items.

```
"C:\Users\HP\OneDrive\Docu
Produced item: 41
Produced item: 67
Produced item: 34
Produced item: 0
Produced item: 69
Consumed item: 69
Consumed item: 0
Consumed item: 34
Consumed item: 67
Consumed item: 41
Produced item: 24
Produced item: 78
Produced item: 58
Produced item: 62
Produced item: 64
Consumed item: 64
Consumed item: 62
Consumed item: 58
Consumed item: 78
Consumed item: 24

Process returned 0 (0x0)   execution time : 0.594 s
Press any key to continue.
```

• Each "Produced" line 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 output illustrates the alternating behavior of production and consumption within the program.

- **Step-by-Step explanation of above C program:**

✞ **Step 1: Include Necessary Libraries-**

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

- Include the required libraries for standard input/output, POSIX threads, and semaphores.

✞ **Step 2: Define Constants and Global Variables-**

```
#define BUFFER_SIZE 10 int
buffer[BUFFER_SIZE];
int count = 0;  // Number of items in the buffer
```

- Define constants such as the buffer size and global variables to manage the shared buffer.

✞ **Step 3: Initialize Mutex and Semaphores-** pthread_mutex_t mutex; sem_t full,

```
empty;// Inside the main function or an initialization function:

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-** 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-** 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;

    // 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.

This program demonstrates a basic solution to the Producer-Consumer problem using mutex locks for mutual exclusion and semaphores for synchronization. The mutex ensures exclusive access to the shared buffer, while semaphores manage the synchronization between the producer and consumer threads.

- In the Producer-Consumer problem solution using both mutex locks and semaphores, synchronization and avoidance of race conditions are achieved through several mechanisms:

1. Mutual Exclusion (Mutex Locks):

- Critical Sections: Mutex locks are employed to guard critical sections of the code where shared resources, such as the buffer and its count, are accessed or modified.
- pthread_mutex_lock & pthread_mutex_unlock: These functions ensure that only one thread can access the critical sections at any given time. The lock is acquired before entering the critical section and released after exiting it.

2. Semaphores:

- Semaphore Operations: Semaphores are used to control access to the buffer by keeping track of the number of empty and full slots.
- sem_wait & sem_post: \`sem_wait\` decrements the semaphore count, allowing threads to wait if necessary conditions aren't met (e.g., the buffer is full or empty). \`sem_post\` increments the semaphore count and signals other waiting threads, indicating that they can proceed.

✞ **Overall Workflow:**

1.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.

2.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