LAB 6 : Process Scheduling, Communication and Synchronization in OS

Nirajan Bekoju
PUL076BCT039
076bct039.nirajan@pcampus.edu.np

Nishant Luitel
PUL076BCT041
076bct041.nishant@pcampus.edu.np

Nabin Da Shrestha
PUL076BCT037
076bct037.nabin@pcampus.edu.np

Prakash Chaulagain
PUL076BCT045
076bct045.prakash@pcampus.edu.np

I. Introduction

Process scheduling, communication, and synchronization are essential concepts in operating systems that deal with managing multiple processes and ensuring that they run efficiently and effectively.

Process scheduling is the process of allocating CPU time to the various processes in a computer system. In modern operating systems, multiple processes may run simultaneously, and the operating system must schedule and manage their execution to ensure that the system runs efficiently. The scheduler determines which process should run next based on various factors, such as priority, the amount of time a process has already used, and the available resources.

Communication between processes is necessary for a variety of reasons, including inter-process communication (IPC), where different processes need to exchange data or signals to achieve some common goal. Processes can communicate with each other through shared memory, message passing, or pipe mechanisms, depending on the operating system and the specific needs of the processes.

Synchronization is the process of coordinating the execution of multiple processes to ensure that they do not interfere with each other or create conflicts while accessing shared resources. Synchronization is essential to avoid race conditions, deadlock, and other issues that can arise when multiple processes attempt to access shared resources simultaneously. Techniques like semaphores, mutex, and conditional variables are used to achieve synchronization among processes.

In summary, process scheduling, communication, and synchronization are all critical components of operating system design, as they ensure that the various processes in a system can work together efficiently, without conflicts or interference, while utilizing the available resources optimally.

II. First Come First Serve

Aim : To write a C program to implement the CPU scheduling algorithm for First Come First Serve

Problem Description : Cpu scheduler will decide which process should be given the CPU for its execution.For this it use different algorithm to choose among the process. one among that algorithm is fcfs algorithm.

In this algorithm the process which arrive first is given the cpu after finishing its request only it will allow cpu to execute other process.

Algorithm :

Step 1 : Create the number of process.
Step 2 : Get the ID and Service time for each process.
Step 3 : Initially, Waiting time of first process is zero and Total time for the first process is the starting time of that process.
Step 4 : Calculate the Total time and Processing time for the remaining processes.
Step 5 : Waiting time of one process is the Total time of the previous process.
Step 6 : Total time of process is calculated by adding Waiting time and Service time.
Step 7 : Total waiting time is calculated by adding the waiting time for lack process.
Step 8 : Total turn around time is calculated by adding all total time of each process.
Step 9 : Calculate Average waiting time by dividing the total waiting time by total number of process.
Step 10 : Calculate Average turn around time by dividing the total time by the number of process.
Step 11 : Display the result.

Program :

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

```
// Turnaround Time = completion of a process - submission of a process
// Waiting Time = turnaround time - burst time

// Function to find the waiting time for all processes
int waitingTime(int proc[], int n, int burst_time[], int wait_time[]){
    wait_time[0] = 0;
    for (int i = 1; i < n; i++){
        wait_time[i] = wait_time[i - 1] + burst_time[i - 1];
    }
    return 0;
}

// calculating the turn around time
int turnAroundTime(int proc[], int n, int burst_time[], int wait_time[], int tat[]){
    for (int i = 0; i < n; i++){
        tat[i] = wait_time[i] + burst_time[i];
    }
    return 0;
}

// calculate the average time
int avgTime(int proc[], int n, int burst_time[]){
    int wait_time[n], tat[n], total_wt = 0, total_tat = 0;
    int i;

    waitingTime(proc, n, burst_time, wait_time);
    turnAroundTime(proc, n, burst_time, wait_time, tat);

    printf("Processes  Burst   Waiting Turn around \n");
    // calculate total waiting time and total turn around time
    for(i = 0; i < n; i++){
        total_wt += wait_time[i];
        total_tat += tat[i];
        printf("%d\t  %d\t\t %d \t%d\n", i+1, burst_time[i], wait_time[i], tat[i]);
    }

    printf("Average waiting time = %f\n", (float)total_wt / (float)n);
    printf("Average turn around time = %f\n", (float)total_tat / (float)n);
    return 0;
}

int main(){
    int proc[] = {1, 2, 3};
    int n = sizeof proc / sizeof proc[0];
    int burst_time[] = {5, 8, 12};
    avgTime(proc, n, burst_time);
    return 0;
}
```

Ouput :

```
Processes  Burst   Waiting Turn around
1         5              0      5
2         8              5      13
3         12             13     25
Average waiting time = 6.000000
Average turn around time = 14.333333
```

III. SHORTEST JOB FIRST

Aim : To write a C program to implement the CPU scheduling algorithm for Shortest job first.

Problem Description : Cpu scheduler will decide which process should be given the CPU for its execution. For this it use different algorithm to choose among the process. one among that algorithm is sjf algorithm.

In this algorithm the process which has less service time given the cpu after finishing its request only it will allow cpu to execute next other process.

Algorithm :

Step 1 : Get the number of process.
Step 2 : Get the id and service time for each process.
Step 3 : Initially the waiting time of first short process as 0 and total time of first short is process the service time of that process.
Step 4 : Calculate the total time and waiting time of remaining process.
Step 5 : Waiting time of one process is the total time of the previous process.
Step 6 : Total time of process is calculated by adding the waiting time and service time of each process.
Step 7 : Total waiting time calculated by adding the waiting time of each process.
Step 8 : Total turn around time calculated by adding all total time of each process.
Step 9 : Calculate average waiting time by dividing the total waiting time by total number of process.
Step 10 : Calculate average turn around time by dividing the total waiting time by total number of process.
Step 11 : Display the result.

Program :

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

// Turnaround Time = completion of a process { submission of a process
// Waiting Time = turnaround time { burst time

// Function to find the waiting time for all processes
int waitingTime(int proc[], int n, int burst_time[], int wait_time[]){
    wait_time[0] = 0;
    for (int i = 1; i < n; i++){
        wait_time[i] = wait_time[i - 1] + burst_time[i - 1];
    }
    return 0;
}

// calculating the turn around time
int turnAroundTime(int proc[], int n, int burst_time[], int wait_time[], int tat[]){
    for (int i = 0; i < n; i++){
        tat[i] = wait_time[i] + burst_time[i];
    }
    return 0;
}

// calculate the average time
int avgTime(int proc[], int n, int burst_time[]){
    int wait_time[n], tat[n], total_wt = 0, total_tat = 0;
    int i;

    waitingTime(proc, n, burst_time, wait_time);
    turnAroundTime(proc, n, burst_time, wait_time, tat);

    printf("ProcessesId  Burst   Waiting Turn around \n");
    // calculate total waiting time and total turn around time
    for(i = 0; i < n; i++){
        total_wt += wait_time[i];
        total_tat += tat[i];
        printf("%d\t\t  %d\t %d \t%d\n", proc[i], burst_time[i], wait_time[i], tat[i]);
```

```
    }

    printf("Average waiting time = %f\n", (float)total_wt / (float)n);
    printf("Average turn around time = %f\n", (float)total_tat / (float)n);
    return 0;
}

void swap(int* xp, int* yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

int shortestJobFirst(int proc[], int n, int burst_time[]){
    int shortest_process_queue[n];
    for (int i = 0; i < n - 1; i++){
        // last i elements are already in place
        for (int j = 0; j < n - i - 1; j++){
            if (burst_time[j] > burst_time[j + 1]){
                swap(&burst_time[j], &burst_time[j + 1]);
                swap(&proc[j], &proc[j + 1]);
            }
        }
    }
    avgTime(proc, n, burst_time);
    return 0;
}

int main(){
    int proc[] = {5, 2, 3, 4, 100, 457};
    int n = sizeof proc / sizeof proc[0];
    int burst_time[] = {8, 5, 12, 5, 100, 1};
    shortestJobFirst(proc, n, burst_time);
    return 0;
}
```

Ouput :

```
ProcessesId  Burst   Waiting Turn around
457               1      0      1
2                 5      1      6
4                 5      6      11
5                 8      11     19
3                 12     19     31
100               100    31     131
Average waiting time = 11.333333
Average turn around time = 33.166668
```

IV. Round Robin

Aim : To write a C program to simulate the CPU scheduling algorithm for round robin.

Problem Description : CPU scheduler will decide which process should be given the CPU for its execution .For this it use different algorithm to choose among the process .one among that algorithm is Round robin algorithm. In this algorithm we are assigning some time slice .The process is allocated according to the time slice ,if the process service time is less than the time slice then process itself will release the CPU voluntarily .The scheduler will then proceed to the next process in the ready queue .If the CPU burst of the currently running process is longer than time quantum ,the timer will go off and will cause an interrupt to the operating system .A context switch will be executed and the process will be put at the tail of the ready queue.

Algorithm :

```
Step 1: Initialize all the structure elements
Step 2: Receive inputs from the user to fill process id,burst time and arrival time.
Step 3: Calculate the waiting time for all the process id.
    i) The waiting time for first instance of a process is calculated as:

        a[i].waittime=count + a[i].arrivt

    ii) The waiting time for the rest of the instances of the process is
        calculated as:

        a) If the time quantum is greater than the remaining burst time
        then waiting time is calculated as:a[i].waittime=count + tq

        b) Else if the time quantum is greater than the remaining burst time
        then waiting time is calculated as:

            a[i].waittime=count - remaining burst time

Step 4: Calculate the average waiting time and average turnaround time
Step 5: Print the results of the step 4.
```

Program :

```
#include <stdio.h>

void main()
{
    // initlialize the variable name
    int i, NOP, sum = 0, count = 0, y, quant, wt = 0, tat = 0;
    int at[10], bt[10], temp[10];
    float avg_wt, avg_tat;
    printf(" Total number of process in the system: ");
    scanf("%d", &NOP);
    y = NOP;

    // Input arrival time, burst time and time quantum
    // Use for loop to enter the details of the process
    // like Arrival time and the Burst Time
    for (i = 0; i < NOP; i++)
    {
        printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i + 1);
        printf(" Arrival time is: ");
        scanf("%d", &at[i]);
        printf(" \nBurst time is: ");
        scanf("%d", &bt[i]);
        // store the burst time in temp array
        temp[i] = bt[i];
    }

    // Accept the Time quantum
    printf("\nEnter the Time Quantum for the process: ");
    scanf("%d", &quant);

    // Display the process No, burst time, Turn Around Time and the waiting time
    printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");
    for (sum = 0, i = 0; y != 0;)
    {
```

```
        if (temp[i] <= quant && temp[i] > 0) // define the conditions
        {
            sum = sum + temp[i];
            temp[i] = 0;
            count = 1;
        }
        else if (temp[i] > 0)
        {
            temp[i] = temp[i] - quant;
            sum = sum + quant;
        }
        if (temp[i] == 0 && count == 1)
        {
            y--;
            printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i + 1, bt[i],
                    sum - at[i], sum - at[i] - bt[i]);
            wt = wt + sum - at[i] - bt[i];
            tat = tat + sum - at[i];
            count = 0;
        }

        if (i == NOP - 1)
        {
            i = 0;
        }
        else if (at[i + 1] <= sum)
        {
            i++;
        }
        else
        {
            i = 0;
        }
    }
    // represents the average waiting time and Turn Around time
    avg_wt = (float)wt / NOP;
    avg_tat = (float)tat / NOP;
    printf("\n Average Turn Around Time: \t%f", avg_wt);
    printf("\n Average Waiting Time: \t%f", avg_tat);
}
```

Ouput :

```
Total number of process in the system: 4

Enter the Arrival and Burst time of the Process[1]
Arrival time is: 5

Burst time is: 12

Enter the Arrival and Burst time of the Process[2]
Arrival time is: 0

Burst time is: 8

Enter the Arrival and Burst time of the Process[3]
Arrival time is: 3

Burst time is: 15
```

```
 Enter the Arrival and Burst time of the Process[4]
 Arrival time is: 13

 Burst time is: 3

Enter the Time Quantum for the process: 5

 Process No              Burst Time              TAT             Waiting Time
Process No[4]            3                               5                       2
Process No[2]            8                               26                      18
Process No[1]            12                              28                      16
Process No[3]            15                              35                      20
```

V. PRIORITY SCHEDULING

Aim : To write a C program to implement CPU scheduling algorithm for priority scheduling.

Problem Description : Cpu scheduler will decide which process should be given the CPU for its execution.For this it use different algorithm to choose among the process. one among that algorithm is fcfs algorithm.

In this algorithm the process which arrive first is given the cpu after finishing its request only it will allow cpu to execute other process.

Algorithm :

Step 1 : Get the number of process,burst time and priority.
Step 2 : Using for loopi=0 to n-1 do step 1 to 6.
Step 3 : If i=0,wait time=0,T[0]=b[0];
Step 4 : T[i]=T[i-1]+b[i] and wt[i]=T[i] - b[i].
Step 5 : Total waiting time is calculated by adding the waiting time for lack process.
Step 6 : Total turn around time is calculated by adding all total time of each process.
Step 7 : Calculate Average waiting time by dividing the total waiting time by total number of process.
Step 8 : Calculate Average turn around time by dividing the total time by the number of process.
Step 9 : Display the result.

Program :

```
#include <stdio.h>
// Turnaround Time = completion of a process { submission of a process
// Waiting Time = turnaround time { burst time

void swap(int* xp, int* yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

int main(){
    int NUM = 4;
    int process[] = {1, 2, 3, 4};
    int burst_time[] = {5, 7, 2, 3};
    int priority[] = {1, 6, 4, 5};

    // get the sorted array based on the priority
    for (int i = 0; i < NUM - 1; i++){
        for (int j = 0; j < NUM - i - 1; j++){
            if(priority[j] < priority[j + 1]){
                swap(&process[j], &process[j + 1]);
                swap(&burst_time[j], &burst_time[j + 1]);
                swap(&priority[j], &priority[j + 1]);
            }
        }
```

```
    }

    // T stores the starting time of the process
    int t = 0;
    printf("Executing the process in required order\n");
    for (int i = 0; i < NUM; i++){
        printf("P%d is executed from %d to %d\n", process[i], t, t + burst_time[i]);
    }

    printf("\n");
    int wait_time = 0;
    int total_waiting_time = 0;
    int total_TAT_time = 0;
    printf("Process ID\t Burst Time\t Wait Time \t Turn Around Time\n");
    for (int i = 0; i < NUM; i++){
        printf("P%d \t\t\t %d\t\t %d \t\t %d\n", process[i], burst_time[i],
                wait_time, wait_time + burst_time[i]);
        total_waiting_time += wait_time;
        total_TAT_time += wait_time + burst_time[i];
        wait_time += burst_time[i];
    }

    printf("Average Waiting Time is %f \n", (float)total_waiting_time / (float)NUM);
    printf("Average Turn Around Time is %f \n", (float)total_TAT_time / (float)NUM);

}
```

Ouput :

```
Executing the process in required order
P2 is executed from 0 to 7
P4 is executed from 0 to 3
P3 is executed from 0 to 2
P1 is executed from 0 to 5

Process ID       Burst Time      Wait Time       Turn Around Time
P2                       7              0               7
P4                       3              7               10
P3                       2              10              12
P1                       5              12              17
Average Waiting Time is 7.250000
Average Turn Around Time is 11.500000
```

VI. Producer Consumer Problem Using Semaphore

Aim : To write a C program to implement the Producer & consumer Problem.

Algorithm :

Step 1: The Semaphore mutex, full & empty are initialized.

Step 2: In the case of producer process

i. Produce an item in to temporary variable.

ii. If there is empty space in the buffer check the mutex value for enter into

iii. the critical section. If the mutex value is 0, allow the producer to add value in the temporary variable to the buffer.

Step 3: In the case of consumer process

i. It should wait if the buffer is empty

ii. If there is any item in the buffer check for mutex value, if the mutex==0, remove item from buffer.

iii. Signal the mutex value and reduce the empty value by 1.

iv. Consume the item.

Step 4: Print the result

Program :

```
#include <stdio.h>

int mutex = 0;
int num_full_slot = 0;
int num_empty_slot = 4;
int x = 0;

void producer(){
    // increase the mutex value so that can't interfere
    ++mutex;
    // increase the number of full slot
    ++num_full_slot;
    // decrease the number of empty slot
    --num_empty_slot;

    // Produce Item
    x++;
    printf("\nProducer produces item %d", x);

    // signal that the producer has completed the process and
    // no longer requires the processor
    --mutex;
}

void consumer(){
    ++mutex;
    --num_full_slot;
    ++num_empty_slot;
    printf("\nConsumer consumes item %d", x);
    x--;
    --mutex;
}

int main(){
    int choice;
    printf("\n1. Press 1 for Producer"
           "\n2. Press 2 for Consumer"
           "\n3. Press 3 for Exit");

    // "critical" specifies that the code is executed by only one thread
    // at a time i.e only one thread enters the critical section at a given time
    #pragma omp critical

    while (1)
    {
        printf("\nEnter your choice :  ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            if (mutex == 0 && num_empty_slot != 0){
                producer();
            }
            else{
                printf("Buffer is full");
            }
            break;
```

```
            case 2:
                if(mutex == 0 && num_full_slot != 0){
                    consumer();
                }
                else{
                    printf("Buffer is empty");
                }
                break;
            case 3:
                exit(0);
                break;
            default:
                break;
            }
        }
    }
```

Ouput :

```
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
Enter your choice :  1

Producer produces item 1
Enter your choice :  1

Producer produces item 2
Enter your choice :  1

Producer produces item 3
Enter your choice :  1

Producer produces item 4
Enter your choice :  1
Buffer is full
Enter your choice :  2

Consumer consumes item 4
Enter your choice :  2

Consumer consumes item 3
Enter your choice :  2

Consumer consumes item 2
Enter your choice :  2

Consumer consumes item 1
Enter your choice :  2
Buffer is empty
Enter your choice :  2
Buffer is empty
Enter your choice :  3
```

VII. Conclusion

In conclusion, this lab report presented the implementation of several important concepts in operating systems, including process scheduling algorithms such as FCFS, SJF, round robin, and priority scheduling, and the synchronization mechanism of semaphores in the Producer-Consumer problem. These implementations were done using the C programming language,

which allowed for a detailed understanding of the underlying concepts and algorithms. Through the process of implementing and testing these algorithms and synchronization mechanisms, we gained a deeper understanding of the challenges involved in managing and coordinating multiple processes in an operating system. Overall, this lab report provides a valuable contribution to the study of operating systems, and serves as a useful resource for those interested in learning more about these essential concepts in computer science.