OPERATING SYSTEMS LAB

Practice-1

Aim: Implementation of FCFS Scheduling Algorithm in C Language

Description:

The First Come First Serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first. Simplest CPU scheduling algorithm that schedules according to arrival times of processes. It is implemented by using the FIFO queue. When a process enters the ready queue, its PCB is linked to the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. FCFS is a non-pre-emptive scheduling algorithm.

Source Code:

```
//FCFS

#include<stdio.h>

int main()

{

   int n,bt[20],wt[20],tat[20],p[20],i,j,t=0;

   float avwt=0,avtat=0;

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

   scanf("%d",&n);

   printf("Enter Process Burst Time\n");

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

   {

      printf("P%d:",i+1);

      scanf("%d",&bt[i]);

      p[i]=i+1;

   }

   wt[0]=0;    //waiting time for first process is 0

   //calculating waiting time

   for(i=1;i<n;i++)
```

```
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];
    }
    printf("Order of process Execution is\n");
    for(i=0;i<n;i++)
    {
        printf("P%d is executed from %d to %d\n",p[i],t,t+bt[i]);
        t+=bt[i];
    }
    printf("Process\t\tBurst Time\tWaiting Time\tTurnaround Time");
    //calculating turnaround time
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];
        avwt+=wt[i];
        avtat+=tat[i];
        printf("\nP%d\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);
    }
    avwt/=i;
    avtat/=i;
    printf("\nAverage Waiting Time:%.2f",avwt);
    printf("\nAverage Turnaround Time:%.2f",avtat);
    return 0;
}
```

Output:

Enter total number of processes:5

Enter Process Burst Time

P1:8

P2:6

P3:1

P4:9

P5:3

Order of process Execution is

P1 is executed from 0 to 8

P2 is executed from 8 to 14

P3 is executed from 14 to 15

P4 is executed from 15 to 24

P5 is executed from 24 to 27

Process	Burst Time	Waiting Time	Turnaround Time
P1	8	0	8
P2	6	8	14
P3	1	14	15
P4	9	15	24
P5	3	24	27

Average Waiting Time:12.20

Average Turnaround Time:17.60

Practie-2

Aim: To implement SJF Scheduling Algorithm in C language

Description:

The **Shortest Job First (SJF)** scheduling algorithm, also known as **Shortest Job Next (SJN)**, selects the waiting process with the smallest execution time to execute next. In non-pre-emptive SJF, once a process starts executing, it continues until completion without interruption. The process with the **smallest burst time** is selected from the ready queue for execution. It aims to minimize the average waiting time among all scheduling algorithms. However, it may cause **starvation** if shorter processes keep arriving. This can be mitigated using the concept of **ageing**. Non-pre-emptive SJF is suitable when accurate estimates of running time are available.

Source Code:

```
//SJF

#include<stdio.h>

void main()

{

   int bt[20],p[20],wt[20],tat[20],i,j,n,t=0,total=0,pos,temp;

   float avg_wt,avg_tat;

   printf("Enter number of processes: ");

   scanf("%d",&n);

   printf("Enter Burst Time:\n");

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

   {

      printf("P%d:",i+1);

      scanf("%d",&bt[i]);

      p[i]=i+1;          //contains process number

   }

   //sorting burst time in ascending order using selection sort

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

   {
```

```
    pos=i;
    for(j=i+1;j<n;j++)
    {
        if(bt[j]<bt[pos])
            pos=j;
    }

    temp=bt[i];
    bt[i]=bt[pos];
    bt[pos]=temp;
    temp=p[i];
    p[i]=p[pos];
    p[pos]=temp;
}
wt[0]=0;            //waiting time for first process will be zero
//calculate waiting time
for(i=1;i<n;i++)
{
    wt[i]=0;
    for(j=0;j<i;j++)
        wt[i]+=bt[j];
    total+=wt[i];
}
avg_wt=(float)total/n;      //average waiting time
total=0;
printf("Order of process Execution is\n");
for(i=0;i<n;i++)
```

```
    {
        printf("P%d is executed from %d to %d\n",p[i],t,t+bt[i]);
        t+=bt[i];
    }
    printf("Process\t    Burst Time    \tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];     //calculate turnaround time
        total+=tat[i];
        printf("\nP%d\t\t  %d\t\t    %d\t\t\t%d",p[i],bt[i],wt[i],tat[i]);
    }
    avg_tat=(float)total/n;     //average turnaround time
    printf("\nAverage Waiting Time=%.2f",avg_wt);
    printf("\nAverage Turnaround Time=%.2f\n",avg_tat);
}
```

Output:

```
Enter number of processes: 5
Enter Burst Time:
P1:8
P2:6
P3:1
P4:9
P5:3
Order of process Execution is
P3 is executed from 0 to 1
P5 is executed from 1 to 4
P2 is executed from 4 to 10
```

P1 is executed from 10 to 18

P4 is executed from 18 to 27

Process	Burst Time	Waiting Time	Turnaround Time
P3	1	0	1
P5	3	1	4
P2	6	4	10
P1	8	10	18
P4	9	18	27

Average Waiting Time=6.60

Average Turnaround Time=12.00

Practice-3

Aim: To implement non-pre-emptive Priority Scheduling Algorithm in C language

Description:

Priority CPU Scheduling Algorithm is used to schedule the processes as per the priorities assigned to respective processes. In Non-preemptive Priority CPU Scheduling Algorithm, processes are scheduled as per the priorities assigned to respective task and next process is not schedule until and unless current execution of process is not completely finished.

Source Code:

```
//Priority Non-preemptive

#include <stdio.h>

void swap(int *a,int *b)

{

   int temp=*a;

   *a=*b;

   *b=temp;

}

int main()

{

   int n,i,j,t=0,wait_time[20],turn_Around[20],total_wait_time = 0,total_Turn_Around = 0;

   printf("Enter Number of Processes: ");

   scanf("%d",&n);

   int burst[n],priority[n],index[n];

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

   {

      printf("Enter Burst Time and Priority Value for Process %d: ",i+1);

      scanf("%d %d",&burst[i],&priority[i]);

      index[i]=i+1;

   }
```

```
for(i=0;i<n;i++)
{
   int temp=priority[i],m=i;
   for(j=i;j<n;j++)
   {
      if(priority[j] < temp)
      {
         temp=priority[j];
         m=j;
      }
   }
   swap(&priority[i], &priority[m]);
   swap(&burst[i], &burst[m]);
   swap(&index[i],&index[m]);
}
printf("Order of process Execution is\n");
for(i=0;i<n;i++)
{
   printf("P%d is executed from %d to %d\n",index[i],t,t+burst[i]);
   t+=burst[i];
}
printf("Process Id\tBurst Time\tWait Time\\t Turn Around Time\n");
wait_time[0]=0;
for(i=1;i<n;i++)
{
   wait_time[i]=0;
   for (j=0;j<i;j++)
```

```
        wait_time[i] += burst[j];
        total_wait_time += wait_time[i];
    }
    for(i=0; i < n; i++){
        turn_Around[i]= burst[i]+wait_time[i];
        total_Turn_Around+=turn_Around[i];
        printf("P%d\t\t%d\t\t%d\t\t%d\n",index[i],burst[i],wait_time[i],turn_Around[i]);
    }
    float avg_wait_time =(float)total_wait_time / n;
    printf("Average waiting time is %.2f\n", avg_wait_time);
    float avg_Turn_Around =(float)total_Turn_Around / n;
    printf("Average TurnAround Time is %.2f",avg_Turn_Around);
    return 0;
}
```

Output:

```
Enter Number of Processes: 5
Enter Burst Time and Priority Value for Process 1: 8 4
Enter Burst Time and Priority Value for Process 2: 6 1
Enter Burst Time and Priority Value for Process 3: 1 2
Enter Burst Time and Priority Value for Process 4: 9 2
Enter Burst Time and Priority Value for Process 5: 3 3
Order of process Execution is
P2 is executed from 0 to 6
P3 is executed from 6 to 7
P4 is executed from 7 to 16
P5 is executed from 16 to 19
P1 is executed from 19 to 27
```

Process Id	Burst Time	Wait Time	Turn Around Time
P2	6	0	6
P3	1	6	7
P4	9	7	16
P5	3	16	19
P1	8	19	27

Average waiting time is 9.60

Average TurnAround Time is 15.00

Practice-4

Aim: To implement Round Robin Scheduling Algorithm in C language

Description:

One of the CPU scheduling strategies is round-robin. It is **preemptive** in nature that it switches between processes according to the time allotted for each process The round-robin scheduling algorithm equally distributes each resource and processes each division in a circular sequence without giving any consideration to priority. The terms "time quantum" and "time slice" are used to distribute all resources equally. The time slice that is used to switch between the processes is known as **Quantum**. Round Robin scheduling is cyclic in nature and is also known as Time Slicing Scheduling. If a process is finished, it is removed from the ready queue in this round-robin method; otherwise, it will return to the ready queue for the remaining execution. The running queue's processes that have arrived and are awaiting execution are in the ready queue. The processes that originated from the ready queue are carried out using the running queue. Let's look at the round-robin scheduling method right now.

Source Code:

```
// Round Robin scheduling program in c

#include<stdio.h>

void main()

{

   int i, n, sum=0,count=0,y,quant, wt=0, tat=0, at[10], bt[10], temp[10];

   printf("Enter number of process in the system: ");

   scanf("%d", &n);

   y = n;

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

   {

      printf("Enter the Arrival and Burst time of the Process %d : ", i+1);

      scanf("%d %d", &at[i],&bt[i]);

      temp[i] = bt[i];

   }

   printf("Enter the Time Quantum for the process: \t");

   scanf("%d", &quant);
```

```
    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)
        {
            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==n-1)
            i=0;
        else if(at[i+1]<=sum)
            i++;
```

```
        else
            i=0;
    }
    printf("\nAverage Waiting time is %.2f\n", wt*1.0/n);
    printf("Average TurnAround Time is %.2f",tat*1.0/n);
}
```

Output:

Enter number of process in the system: 4

Enter the Arrival and Burst time of the Process 1 : 0 5

Enter the Arrival and Burst time of the Process 2 : 1 4

Enter the Arrival and Burst time of the Process 3 : 2 2

Enter the Arrival and Burst time of the Process 4 : 4 1

Enter the Time Quantum for the process: 2

Process No	Burst Time	TAT	Waiting Time
Process No 3	2	4	2
Process No 4	1	3	2
Process No 2	4	10	6
Process No 1	5	12	7

Average Waiting time is 4.25

Average TurnAround Time is 7.25

Practice-5

Aim: To implement Producer-Consumer Problem in C language

Description:

The producer-consumer problem in C is one of the most famous problems associated with operating systems. In the producer-consumer problem in C, there is a producer which produces products (data) and there is a consumer who consumes the products produced by the producer. Now, in the producer-consumer problem in C, there is a buffer. A buffer is a temporary storage area in the memory that stores data. In the producer-consumer problem in C, we have been provided with a fixed-sized buffer. The buffer in the Producer-Consumer Problem in C contains the produced items by the producer. The consumer consumes the products from the same buffer. In simpler terms, we can say that the buffer is a shared space between the producer and consumer.

- **Producer**: The producer's role is to produce or generate the data and put it in the buffer and start again with the same.
- **Consumer**: The consumer's role is to consume the produced data from the same shared buffer. When the consumer consumes the data, the data is removed from the buffer.

Since the problem is that the producer keeps on producing the data even if the buffer is full and the consumer tries to consume the data even if the buffer is empty. We can either put the producer to sleep or discard the data produced by the producer if the buffer is full. So, whenever the consumer tries to consume the data from the buffer, it will notify the producer that the data is being consumed. This will make the producer produce the data again.

Similarly, we can put the consumer to sleep when there is no data in the buffer (empty buffer case). So, whenever the producer produces the data and put it into the buffer, it will notify the consumer that the data is being produced. This will make the consumer consume the data again.

We should also notice that if there is an inadequate solution is proposed then there may arise a situation when both the producer and consumer are on wait (to be awakened)

The main cause behind the producer-consumer problem is that there is no way that will tell the consumer if the buffer is empty or not. Similarly, the producer cannot know that the buffer is already full.

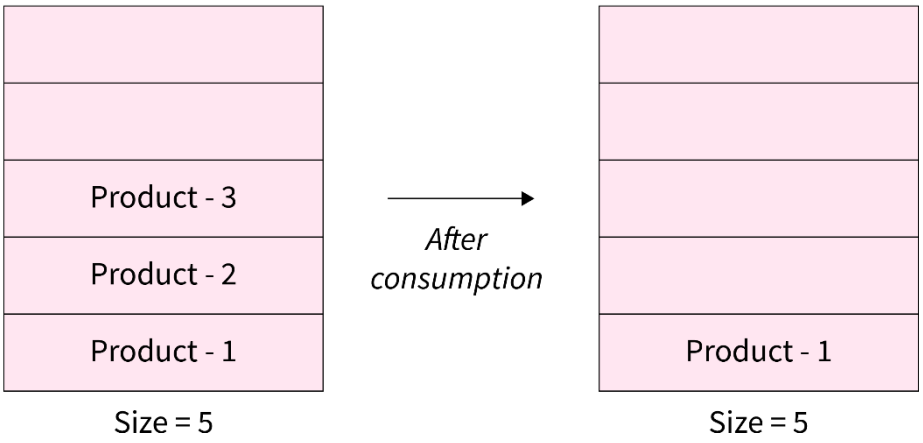
So, as we have discussed above, we can put the consumer to sleep when there is no data in the buffer (empty buffer case). We can similarly put the producer to sleep if the buffer is full. By the term sleep, we mean that the producer should not produce any product when the consumer is consuming the product and vice versa. We can use the sleep() function to put the consumer or the producer to sleep.

Examples of Producer-Consumer Problems in C

Example 1: Suppose we have a buffer of size = 5.

Initially, the buffer is empty and neither producer is producing the data nor the consumer is consuming the data.

Suppose that the producer starts producing the data one at a time. The producer produces 3 data during the production and when the consumer comes he/she consumes two of the three produced data. Now in this scenario, we can see that neither of the two has caused the problem as when the consumer was consuming the data, the buffer was not empty. Similarly, when the producer was producing the data the buffer was not full.



Source Code:

```
// C Program for Producer-Consumer Problem

//The idea is to use the concept of parallel programming and Critical Section to implement the Producer-Consumer problem in C language using OpenMP.

#include <stdio.h>

#include <stdlib.h>

// Initialize a mutex to 1

int mutex = 1;

// Number of full slots as 0

int full = 0;

// Number of empty slots as size of buffer

int empty = 10, x = 0;

// Function to produce an item and add it to the buffer

void producer()

{
```

```
    // Decrease mutex value by 1
    --mutex;
    // Increase the number of full slots by 1
    ++full;
    // Decrease the number of empty slots by 1
    --empty;
    // Item produced
    x++;
    printf("\nProducer produces item %d",x);
    // Increase mutex value by 1
    ++mutex;
}
// Function to consume an item and remove it from buffer
void consumer()
{
    // Decrease mutex value by 1
    --mutex;
    // Decrease the number of full slots by 1
    --full;
    // Increase the number of empty slots by 1
    ++empty;
    printf("\nConsumer consumes item %d", x);
    x--;
    // Increase mutex value by 1
    ++mutex;
}
// Driver Code
```

```
int main()
{
    int n, i;
    printf("\n1. Press 1 for Producer\n2. Press 2 for Consumer\n3. Press 3 for Exit");
// Using '#pragma omp parallel for' can  give wrong value due to synchronization issues. '
//'critical' specifies that 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
    for (i = 1; i > 0; i++) {
        printf("\nEnter your choice:");
        scanf("%d", &n);
        // Switch Cases
        switch (n) {
        case 1:
            // If mutex is 1 and empty is non-zero, then it is possible to produce
            if ((mutex == 1) && (empty != 0))
                producer();
            // Otherwise, print buffer is full
            else
                printf("Buffer is full!");
            break;
        case 2:
            // If mutex is 1 and full is non-zero, then it is possible to consume
            if ((mutex == 1) && (full != 0))
                consumer();
            // Otherwise, print Buffer is empty
            else
```

```
            printf("Buffer is empty!");
            break;
        // Exit Condition
        case 3:
            exit(0);
            break;
        }
    }
}
```

Output:

```
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: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: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:

Producer produces item 5

Enter your choice:1

Producer produces item 6

Enter your choice:1

Producer produces item 7

Enter your choice:1

Producer produces item 8

Enter your choice:

Producer produces item 9

Enter your choice:1

Producer produces item 10

Enter your choice:1

Buffer is full!

Enter your choice:2

Consumer consumes item 10

Enter your choice:2

Consumer consumes item 9

Enter your choice:2

Consumer consumes item 8

Enter your choice:2

Consumer consumes item 7

Enter your choice:2

Consumer consumes item 6

Enter your choice:2

Consumer consumes item 5

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

Practice-6

Aim: To implement Bankers Algorithm in C language

Description:

The Banker's Algorithm, a key deadlock avoidance and detection tool, strategically allocates resources by simulating their distribution to predetermined maximums. This process includes an 's-state' safety check to decide if resource allocation can proceed without leading to a deadlock condition. The Banker's Algorithm is a smart way for computer systems to manage how programs use resources, like memory or CPU time. It helps prevent situations where programs get stuck and can't finish their tasks, which is called deadlock. By keeping track of what resources each program needs and what's available, the algorithm makes sure that programs only get what they need in a safe order. This helps computers run smoothly and efficiently, especially when lots of programs are running at the same time.

Source Code:

```
//Bankers Algorithm in C

#include <stdio.h>

void main() {

	int k=0,a=0,b=0,instance[5],availability[5],allocated[10][5],need[10][5],MAX[10][5],process,P[10],op[10],no_of_resources, cnt=0,i, j;

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

	scanf("%d", &no_of_resources);

	printf("Enter the max instances of each resources\n");

	for (i=0;i<no_of_resources;i++) {

		//availability[i]=0;

		printf("%c= ",(i+65));

		scanf("%d",&instance[i]);

	}

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

	scanf("%d", &process);

	printf("Enter the allocation matrix \n");

	for (i=0;i<no_of_resources;i++)
```

```
printf(" \t%c",(i+65));
printf("\n");
for (i=0;i <process;i++) {
        P[i]=i;
        printf("P[%d]\t",P[i]);
        for (j=0;j<no_of_resources;j++) {
                scanf("%d",&allocated[i][j]);
                //availability[j]+=allocated[i][j];
        }
}
printf("Enter the MAX matrix \n");
for (i=0;i<no_of_resources;i++) {
        printf(" \t%c",(i+65));
        //availability[i]=instance[i]-availability[i];
}
printf("\n");
for (i=0;i <process;i++) {
        printf("P[%d]\t",i);
        for (j=0;j<no_of_resources;j++)
         scanf("%d", &MAX[i][j]);
}
printf("\n");
printf("Availability Matrix \n");
for (i=0;i<no_of_resources;i++)
        printf("%c\t",(i+65));
printf("\n");
for (i=0;i<no_of_resources;i++)
```

```
        scanf("%d", &availability[i]);
printf("\n");
A: a=-1;
for (i=0;i <process;i++) {
        cnt=0;
        b=P[i];
        for (j=0;j<no_of_resources;j++) {
                need[b][j] = MAX[b][j]-allocated[b][j];
                if(need[b][j]<=availability[j])
                 cnt++;
        }
        if(cnt==no_of_resources) {
                op[k++]=P[i];
                for (j=0;j<no_of_resources;j++)
                availability[j]+=allocated[b][j];
        } else
         P[++a]=P[i];
}
if(a!=-1) {
        process=a+1;
        goto A;
}
printf("Order of Processes without Dead Lock\n");
printf("<");
for (i=0;i<k;i++)
printf(" P[%d] ",op[i]);
printf(">");
```

}

Output:

Enter the number of resources : 3

Enter the max instances of each resources

A= 10

B= 5

C= 7

Enter the number of processes : 5

Enter the allocation matrix

	A	B	C
P[0]	0	1	0
P[1]	2	0	0
P[2]	3	0	2
P[3]	2	1	1
P[4]	0	0	2

Enter the MAX matrix

	A	B	C
P[0]	7	5	3
P[1]	3	2	2
P[2]	9	0	2
P[3]	2	2	2
P[4]	4	3	3

Availability Matrix

A	B	C
3	3	2

Order of Processes without Dead Lock

< P[1] P[3] P[4] P[0] P[2] >