# Practical-8

**Aim: To show First Come First Serve (FCFS) CPU Scheduling in C.**

FCFS is the simplest scheduling algorithm. It simply queues processes in the order that they arrive in the ready queue. In this, the process that comes first will be executed first and next process starts only after the previous gets fully executed.

Completion Time: Time at which process completes its execution.

Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time

Waiting Time (W.T): Time Difference between turnaround time and burst time.

Waiting Time = Turn Around Time – Burst Time

**Code:**

```
#include<stdio.h>
void findWaitingTime(int processes[], int n, int bt[], int wt[])
{
   wt[0] = 0;
   for (int i = 1; i < n ; i++ )
      wt[i] = bt[i-1] + wt[i-1] ;
}
void findTurnAroundTime( int processes[], int n, int bt[], int wt[], int tat[])
{
   for (int i = 0; i < n ; i++)
      tat[i] = bt[i] + wt[i];
}
void findavgTime( int processes[], int n, int bt[])
{
   int wt[n], tat[n], total_wt = 0, total_tat = 0;
   findWaitingTime(processes, n, bt, wt);
   findTurnAroundTime(processes, n, bt, wt, tat);
   printf("Processes Burst time Waiting time Turn around time\n");
   for (int i=0; i<n; i++)
   {
      total_wt = total_wt + wt[i];
      total_tat = total_tat + tat[i];
```

```
        printf(" %d ",(i+1));
        printf("      %d    ", bt[i] );
        printf("      %d    ",wt[i] );
        printf("      %d    \n",tat[i] );
    }
    int s=(float)total_wt / (float)n;
    int t=(float)total_tat / (float)n;
    printf("Average waiting time = %d",s);
    printf("\n");
    printf("Average turn around time = %d ",t);
}
int main()
{
    int processes[] = { 1, 2, 3};
    int n = sizeof processes / sizeof processes[0];
    int burst_time[] = {10, 5, 8};
    findavgTime(processes, n, burst_time);
    return 0;
}
```

```
C:\Users\plumb\CLionProjects\untitled\cmake-build-debu
Processes Burst time Waiting time Turn around time
 1       10          0          10
 2       5          10          15
 3       8          15          23
Average waiting time = 8
Average turn around time = 16
Process finished with exit code 0
```

# Practical-9

**Aim: To show Shortest Job First and Shortest Remaining Time First CPU Scheduling in C/C++.**

The shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN, also known as Shortest Job Next (SJN), can be pre-emptive or non-pre-emptive. In the Shortest Remaining Time First (SRTF) scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining, and since that time should only reduce as execution progresses, processes will always run until they complete, or a new process is added that requires a smaller amount of time. This is pre-emptive.

**Code:**

```
// Shortest Job First

#include <stdio.h>

int main()

{

   int A[100][4];

   int i, j, n, total = 0, index, temp;

   float avg_wt, avg_tat;

   printf("Enter number of process: ");

   scanf("%d", &n);

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

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

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

      scanf("%d", &A[i][1]);

      A[i][0] = i + 1;

   }

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

      index = i;

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

         if (A[j][1] < A[index][1])

            index = j;

      temp = A[i][1];

      A[i][1] = A[index][1];

      A[index][1] = temp;
```

```
        temp = A[i][0];
        A[i][0] = A[index][0];
        A[index][0] = temp;
    }
    A[0][2] = 0;
    for (i = 1; i < n; i++) {
        A[i][2] = 0;
        for (j = 0; j < i; j++)
            A[i][2] += A[j][1];
        total += A[i][2];
    }
    avg_wt = (float)total / n;
    total = 0;
    printf("P      BT      WT      TAT\n");
    for (i = 0; i < n; i++) {
        A[i][3] = A[i][1] + A[i][2];
        total += A[i][3];
        printf("P%d         %d      %d      %d\n", A[i][0],
                A[i][1], A[i][2], A[i][3]);
    }
    avg_tat = (float)total / n;
    printf("Average Waiting Time= %f", avg_wt);
    printf("\nAverage Turnaround Time= %f", avg_tat);
}
```

```
 P     BT       WT       TAT
P2        6        0        6
P3        8        6        14
P1        15       14       29
Average Waiting Time= 6.666667
Average Turnaround Time= 16.333334
Process finished with exit code 0
```

```
// Shortest Remaining Time First
#include <bits/stdc++.h>
using namespace std;
struct Process {
    int pid; // Process ID
    int bt; // Burst Time
    int art; // Arrival Time
};
void findWaitingTime(Process proc[], int n,
                     int wt[])
{
    int rt[n];
    for (int i = 0; i < n; i++)
        rt[i] = proc[i].bt;
    int complete = 0, t = 0, minm = INT_MAX;
    int shortest = 0, finish_time;
    bool check = false;
    while (complete != n) {
        for (int j = 0; j < n; j++) {
            if ((proc[j].art <= t) &&
                (rt[j] < minm) && rt[j] > 0) {
                minm = rt[j];
                shortest = j;
                check = true;
            }
        }
        if (check == false) {
            t++;
            continue;
        }
        rt[shortest]--;
        minm = rt[shortest];
```

```
        if (minm == 0)
            minm = INT_MAX;
        if (rt[shortest] == 0) {
            complete++;
            check = false;
            finish_time = t + 1;
            wt[shortest] = finish_time -
                        proc[shortest].bt -
                        proc[shortest].art;
            if (wt[shortest] < 0)
                wt[shortest] = 0;
        }
        t++;
    }
}
void findTurnAroundTime(Process proc[], int n,
                        int wt[], int tat[])
{
    for (int i = 0; i < n; i++)
        tat[i] = proc[i].bt + wt[i];
}
void findavgTime(Process proc[], int n)
{
    int wt[n], tat[n], total_wt = 0,
            total_tat = 0;
    findWaitingTime(proc, n, wt);
    findTurnAroundTime(proc, n, wt, tat);
    cout << " P\t\t"
         << "BT\t\t"
         << "WT\t\t"
         << "TAT\t\t\n";
    for (int i = 0; i < n; i++) {
```

```
        total_wt = total_wt + wt[i];
        total_tat = total_tat + tat[i];
        cout << " " << proc[i].pid << "\t\t"
             << proc[i].bt << "\t\t " << wt[i]
             << "\t\t " << tat[i] << endl;
    }
    cout << "\nAverage waiting time = "
         << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
         << (float)total_tat / (float)n;
}

int main()
{
    Process proc[] = { { 1, 6, 2 }, { 2, 2, 5 },
                       { 3, 8, 1 }, { 4, 3, 0}, {5, 4, 4} };
    int n = sizeof(proc) / sizeof(proc[0]);
    findavgTime(proc, n);
    return 0;
}
```

```
C:\Users\plumb\CLionProjects\untitled\cmake-build-debug\unt
 P              BT              WT              TAT
 1              6               7               13
 2              2               0               2
 3              8               14              22
 4              3               0               3
 5              4               2               6

Average waiting time = 4.6
Average turn around time = 9.2
Process finished with exit code 0
```

# Practical-10

**Aim: To implement Bankers Algorithm in C**

Banker's Algorithm is a Deadlock avoidance algorithm and is also used as a Deadlock detection Algorithm. Deadlock condition arises when there is a Mutual Exclusion, Circular wait, no pre-emption, and Circular wait situation. The banker's Algorithm tests for a safe state check the condition for possible activities and decides whether to continue the allocation of resources or not.

**Code:**

```
#include <stdio.h>

#include <conio.h>

int main() {

   int
k=0,a=0,b=0,instance[5],availability[5],allocated[10][5],need[10][5],MAX[10][5],process,P[
10],no_of_resources, cnt=0,I, j, op[300];

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

   scanf("%d", &no_of_resources);

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

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

      availability[i]=0;

      printf("%c= ",(i+97));

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

   }

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

   scanf("%d", &process);

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

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

      printf(" %c",(i+97));

   printf("\n");

   for (i=0;I <process;i++) {

      P[i]=I;

      printf("P[%d]  ",P[i]);

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

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

         availability[j]+=allocated[i][j];
```

```
    }
}
printf("\nEnter the MAX matrix \n     ");
for (i=0;i<no_of_resources;i++) {
    printf(" %c",(i+97));
    availability[i]=instance[i]-availability[i];
}
printf("\n");
for (i=0;I <process;i++) {
    printf("P[%d]  ",i);
    for (j=0;j<no_of_resources;j++)
        scanf("%d", &MAX[i][j]);
}
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("\t <");
  for (i=0;i<k;i++)
    printf(" P[%d] ",op[i]);
  printf(">");
  getch();
}
```

```
C:\Users\plumb\CLionProjects\untitled\cmake-b

 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]4 2 2
  P[4]5 3 3

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

# Practical-11

**Aim: To implement Best fit and Worse fit algorithm for memory management in C++.**

Best fit allocates the process to a partition which is the smallest sufficient partition among the free available partitions. Worst Fit allocates a process to the partition which is largest sufficient among the freely available partitions available in the main memory. If a large process comes at a later stage, then memory will not have space to accommodate it.

**Code:**

```
// Best Fit
#include<iostream>
using namespace std;
void bestFit(int blockSize[], int m, int processSize[], int n)
{
	int allocation[n];
	for (int i = 0; i < n; i++)
		allocation[i] = -1;
	for (int i = 0; i < n; i++)
	{
		int bestIdx = -1;
		for (int j = 0; j < m; j++)
		{
			if (blockSize[j] >= processSize[i])
			{
				if (bestIdx == -1)
					bestIdx = j;
				else if (blockSize[bestIdx] > blockSize[j])
					bestIdx = j;
			}
		}
		if (bestIdx != -1)
		{
			allocation[i] = bestIdx;
			blockSize[bestIdx] -= processSize[i];
		}
```

```
        }
        cout << "\nProcess No.\tProcess Size\tBlock no.\n";
        for (int i = 0; i < n; i++)
        {
                cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";
                if (allocation[i] != -1)
                        cout << allocation[i] + 1;
                else
                        cout << "Not Allocated";
                cout << endl;
        }
}
int main()
{
        int blockSize[] = {100, 500, 200, 300, 600};
        int processSize[] = {212, 417, 112, 426};
        int m = sizeof(blockSize) / sizeof(blockSize[0]);
        int n = sizeof(processSize) / sizeof(processSize[0]);
        bestFit(blockSize, m, processSize, n);
        return 0 ;
}
```

```
C:\Users\plumb\CLionProjects\untitled\cmake

Process No.     Process Size    Block no.
 1              212             4
 2              417             2
 3              112             3
 4              426             5

Process finished with exit code 0
```

```
// Worse Fit
#include<bits/stdc++.h>
using namespace std;
void worstFit(int blockSize[], int m, int processSize[],
          int n)
```

```
{
    int allocation[n];
    memset(allocation, -1, sizeof(allocation));
    for (int i=0; i<n; i++)
    {
        int wstIdx = -1;
        for (int j=0; j<m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
                if (wstIdx == -1)
                    wstIdx = j;
                else if (blockSize[wstIdx] < blockSize[j])
                    wstIdx = j;
            }
        }
        if (wstIdx != -1)
        {
            allocation[i] = wstIdx;

            blockSize[wstIdx] -= processSize[i];
        }
    }
    cout << "\nProcess No.\tProcess Size\tBlock no.\n";
    for (int i = 0; i < n; i++)
    {
        cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
```

```
    }
}
int main()
{
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize)/sizeof(blockSize[0]);
    int n = sizeof(processSize)/sizeof(processSize[0]);
    worstFit(blockSize, m, processSize, n);
    return 0 ;
}
```

```
Process No.     Process Size    Block no.
 1              212             5
 2              417             2
 3              112             5
 4              426             Not Allocated

Process finished with exit code 0
```