CPU scheduling

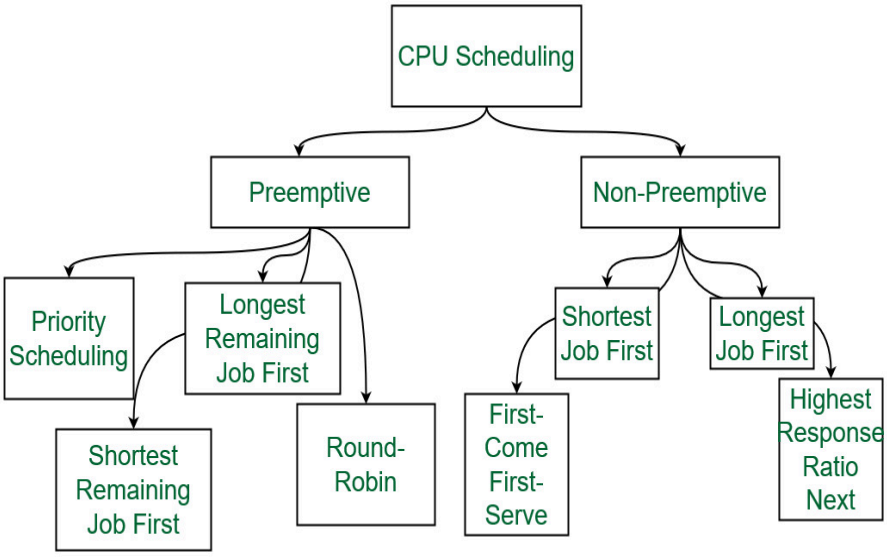
CPU scheduling is the process that allows one process to use the CPU while another process is waiting for a resource like I/O. The goal of CPU scheduling is to make the system efficient, fast, and fair.

Operating systems may have three types of schedulers:

Long-term scheduler: Also known as an admission scheduler or high-level scheduler

Mid-term or medium-term scheduler

Short-term scheduler



Different terminologies to take care of in any CPU Scheduling algorithm?

Arrival Time: Time at which the process arrives in the ready queue.

Completion Time: Time at which process completes its execution.

Burst Time: Time required by a process for CPU 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 turn around time and burst time.

Waiting Time = Turnaround Time – Burst Time

There are two types of CPU scheduling: preemptive and non-preemptive.

In preemptive scheduling, a process is allocated resources for a limited time. A running process can be interrupted by a higher priority process. For example, preemptive scheduling is used when a process moves from a running state to a ready state or from a waiting state to a ready state.

In non-preemptive scheduling, a process retains control of a resource until it is terminated or moves to a waiting state. For example, non-preemptive scheduling is used when a process terminates or moves from running to waiting state.

First Come First Serve:

FCFS is considered to be the simplest of all operating system scheduling algorithms. First come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first and is implemented by using FIFO queue.

Advantages of FCFS:

Easy to implement

First come, first serve method.

CODE:

```
#include<iostream>
using namespace std;
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);

   cout << "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];
        cout << "   " << i+1 << "\t\t" << bt[i] <<"\t    "
            << wt[i] <<"\t\t  " << tat[i] <<endl;
    }

    cout << "Average waiting time = "
         << (float)total_wt / (float)n;
    cout << "\nAverage turn around time = "
         << (float)total_tat / (float)n;
}

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;
}
```

```
Processes   Burst time   Waiting time   Turn around time
   1            10          0             10
   2            5           10            15
   3            8           15            23
Average waiting time = 8.33333
Average turn around time = 16
--------------------------------
Process exited after 0.1821 seconds with return value 0
Press any key to continue . . .
```

Shortest Job First(SJF):

Shortest job first (SJF) is a scheduling process that selects the waiting process with the smallest execution time to execute next. This scheduling method may or may not be preemptive. Significantly reduces the average waiting time for other processes waiting to be executed. The full form of SJF is Shortest Job First.

Advantages of Shortest Job first:

As SJF reduces the average waiting time thus, it is better than the first come first serve scheduling algorithm.

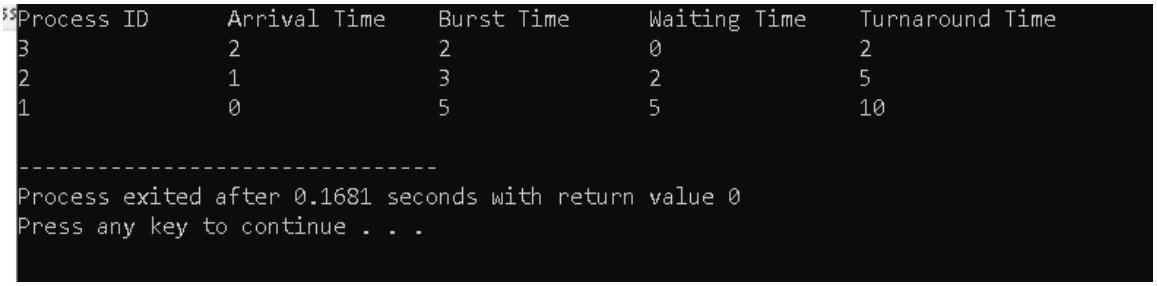
SJF is generally used for long term scheduling

CODE:

```
#include <stdio.h>
// Structure to store process information
struct process {
  int pid; // Process ID
  int burst_time; // Burst time of the process
  int arrival_time; // Arrival time of the process
  int waiting_time; // Waiting time of the process
  int turnaround_time; // Turnaround time of the process
};

// Function to sort the processes by burst time
```

```
void sort_processes_by_burst_time(struct process *processes,
int n) {
  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) {
        struct process temp = processes[j];
        processes[j] = processes[j + 1];
        processes[j + 1] = temp;
      }
    }
  }
}


void calculate_waiting_time(struct process *processes, int n) {
  processes[0].waiting_time = 0;
  for (int i = 1; i < n; i++) {
    processes[i].waiting_time = processes[i - 1].burst_time +
processes[i - 1].waiting_time;
  }
}


void calculate_turnaround_time(struct process *processes, int n)
{
  for (int i = 0; i < n; i++) {
    processes[i].turnaround_time = processes[i].burst_time +
processes[i].waiting_time;
  }
}
```

```
// Function to print the waiting time and turnaround time of each process
void print_waiting_and_turnaround_time(struct process *processes, int n) {
  printf("Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
  for (int i = 0; i < n; i++) {
    printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival_time, processes[i].burst_time, processes[i].waiting_time, processes[i].turnaround_time);
  }
}

int main() {
  struct process processes[] = {{1, 5, 0}, {2, 3, 1}, {3, 2, 2}};
  sort_processes_by_burst_time(processes, sizeof(processes) / sizeof(processes[0]));

  // Calculate the waiting time and turnaround time of each process
  calculate_waiting_time(processes, sizeof(processes) / sizeof(processes[0]));
  calculate_turnaround_time(processes, sizeof(processes) / sizeof(processes[0]));
  print_waiting_and_turnaround_time(processes, sizeof(processes) / sizeof(processes[0]));

  return 0;
}
```



Longest Job First(LJF):

Longest Job First(LJF) scheduling process is just opposite of shortest job first (SJF), as the name suggests this algorithm is based upon the fact that the process with the largest burst time is processed first. Longest Job First is non-preemptive in nature.

Advantages of LJF:

No other task can be scheduled until the longest job or process executes completely.

All the jobs or processes finish at approximately the same time.

CODE:

```
#include <stdio.h>
struct process {
  int pid; // Process ID
  int burst_time; // Burst time of the process
  int arrival_time; // Arrival time of the process
  int waiting_time; // Waiting time of the process
  int turnaround_time; // Turnaround time of the process
```

```
};

// Function to sort the processes by burst time
void sort_processes_by_burst_time(struct process *processes,
int n) {
  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) {
        struct process temp = processes[j];
        processes[j] = processes[j + 1];
        processes[j + 1] = temp;
      }
    }
  }
}

// Function to calculate the waiting time of each process
void calculate_waiting_time(struct process *processes, int n) {
  processes[0].waiting_time = 0;
  for (int i = 1; i < n; i++) {
    processes[i].waiting_time = processes[i - 1].burst_time +
processes[i - 1].waiting_time;
  }
}
// Function to calculate the turnaround time of each process
void calculate_turnaround_time(struct process *processes, int n)
{
  for (int i = 0; i < n; i++) {
    processes[i].turnaround_time = processes[i].burst_time +
processes[i].waiting_time;
  }
}
```

```
// Function to print the waiting time and turnaround time of each process
void print_waiting_and_turnaround_time(struct process *processes, int n) {
  printf("Process ID\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
  for (int i = 0; i < n; i++) {
    printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].arrival_time, processes[i].burst_time, processes[i].waiting_time, processes[i].turnaround_time);
  }
}
int main() {
  struct process processes[] = {{1, 5, 0}, {2, 3, 1}, {3, 2, 2}};
  sort_processes_by_burst_time(processes, sizeof(processes) / sizeof(processes[0]));
  calculate_waiting_time(processes, sizeof(processes) / sizeof(processes[0]));
  calculate_turnaround_time(processes, sizeof(processes) / sizeof(processes[0]));
  print_waiting_and_turnaround_time(processes, sizeof(processes) / sizeof(processes[0]));

  return 0;
}
```

```
Process ID      Arrival Time    Burst Time      Waiting Time    Turnaround Time
3               2               2               0               2
2               1               3               2               5
1               0               5               5               10

--------------------------------
Process exited after 0.1055 seconds with return value 0
Press any key to continue . . .
```

Priority Scheduling:

Preemptive Priority CPU Scheduling Algorithm is a pre-emptive method of CPU scheduling algorithm that works based on the priority of a process.In this algorithm, the editor sets the functions to be as important, meaning that the most important process must be done first.

Advantages of Priority Scheduling:

The average waiting time is less than FCFS

Less complex

CODE:

```
#include <stdio.h>
struct process {
    int arrival_time;
    int burst_time;
    int priority;
    int waiting_time;
    int turnaround_time;
};
void sort_processes(struct process *processes, int n) {
    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) {
                struct process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
```

```
        }
    }
}

void calculate_waiting_time(struct process *processes, int n) {
    processes[0].waiting_time = 0;
    for (int i = 1; i < n; i++) {
        processes[i].waiting_time = processes[i - 1].waiting_time +
processes[i - 1].burst_time;
    }
}
void calculate_turnaround_time(struct process *processes, int n)
{
    for (int i = 0; i < n; i++) {
        processes[i].turnaround_time = processes[i].waiting_time +
processes[i].burst_time;
    }
}
void print_results(struct process *processes, int n) {
    printf("Process\tArrival Time\tBurst Time\tPriority\tWaiting
Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i + 1,
processes[i].arrival_time, processes[i].burst_time,
processes[i].priority, processes[i].waiting_time,
processes[i].turnaround_time);
    }
}
int main() {
    struct process processes[3];

    processes[0].arrival_time = 0;
```

```
    processes[0].burst_time = 10;
    processes[0].priority = 2;

    processes[1].arrival_time = 1;
    processes[1].burst_time = 5;
    processes[1].priority = 0;

    processes[2].arrival_time = 2;
    processes[2].burst_time = 8;
    processes[2].priority = 1;

    sort_processes(processes, 3);
    calculate_waiting_time(processes, 3);
    calculate_turnaround_time(processes, 3);
    print_results(processes, 3);

    return 0;
}
```

```
Process Arrival Time    Burst Time      Priority        Waiting Time    Turnaround Time
1               0               10              2               0               10
2               2               8               1               10              18
3               1               5               0               18              23

--------------------------------
Process exited after 0.425 seconds with return value 0
Press any key to continue . . .
```

Round robin:

Round Robin is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot. It is the preemptive version of First come First Serve CPU Scheduling algorithm. Round Robin CPU Algorithm generally focuses on Time Sharing technique.

Advantages of Round robin:

Round robin seems to be fair as every process gets an equal share of CPU.

The newly created process is added to the end of the ready queue.

CODE:

```
#include<iostream>
using namespace std;
void findWaitingTime(int processes[], int n,
                     int bt[], int wt[], int quantum)
{
        int rem_bt[n];
        for (int i = 0 ; i < n ; i++)
                rem_bt[i] = bt[i];
        int t = 0;
        while (1)
        {
                bool done = true;
                for (int i = 0 ; i < n; i++)
```

```
            {

                if (rem_bt[i] > 0)
                {
                    done = false; // There is a pending process
                    if (rem_bt[i] > quantum)
                    {

                        t += quantum;

                        rem_bt[i] -= quantum;
                    }

                    else
                    {

                        t = t + rem_bt[i];
                        wt[i] = t - bt[i];
                        rem_bt[i] = 0;
                    }
                }
            }

            if (done == true)
            break;
        }
}
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 quantum)
{
        int wt[n], tat[n], total_wt = 0, total_tat = 0;
        findWaitingTime(processes, n, bt, wt, quantum);
        findTurnAroundTime(processes, n, bt, wt, tat);
        cout << "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];
                cout << " " << i+1 << "\t\t" << bt[i] <<"\t "
                        << wt[i] <<"\t\t " << tat[i] <<endl;
        }
        cout << "Average waiting time = "
                << (float)total_wt / (float)n;
        cout << "\nAverage turn around time = "
                << (float)total_tat / (float)n;
}
int main()
{
        int processes[] = { 1, 2, 3};
        int n = sizeof processes / sizeof processes[0];
        int burst_time[] = {10, 5, 8};
```

```
    int quantum = 2;
    findavgTime(processes, n, burst_time, quantum);
    return 0;
}
```