Team 04

**Software Requirement Specification (SRS) 1.1**

Introduction

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. In this project we have used the following algorithms for process scheduling:

* First-Come, First-Served (FCFS) Scheduling
* Shortest-Job-First (SJF) Scheduling
* Round Robin (RR) Scheduling

These algorithms are either non-preemptive or pre-emptive. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be pre-empted until it completes its allotted time, whereas the pre-emptive scheduling is based on priority where a scheduler may pre-empt a low priority running process anytime when a high priority process enters into a ready state.

First come first serve (FCFS)

First Come First Serve is an operating system scheduling algorithm that automatically executes queued requests and processes in order of their arrival. It is the easiest and simplest CPU scheduling algorithm. In this type of algorithm, processes which requests the CPU first get the CPU allocation first. This is managed with a FIFO queue. The full form of FCFS is First Come First Serve.

Algorithm:

1. Input the processes along with their burst time (bt).
2. Find waiting time (wt) for all processes.
3. As first process that comes need not to wait so waiting time for process 1 will be 0 i.e. wt[0] = 0.
4. Find waiting time for all other processes i.e., for process i -> wt[i] = bt[i-1] + wt[i-1].
5. Find turnaround time = waiting\_time + burst\_time for all processes.
6. Find average waiting time = total\_waiting\_time / no\_of\_processes.
7. Similarly, find average turnaround time = total\_turn\_around\_time / no\_of\_processes.

High level design Documentation

1. Function to find the waiting time for all

void findWaitingTime int processes[]

2. Function to calculate turn around time

void findTurnAroundTime int processes[]

3. Function to calculate average time

void findavgTime int processes[]

Low Level Documentation

1.process id's

int processes[]

2.Burst time of all processes

int burst\_time[]

Shortest Job First (SJF)

Shortest Job First is an algorithm in which the process having the smallest execution time is chosen for the next execution. This scheduling method can be pre-emptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution. The full form of SJF is Shortest Job First.

Algorithm: 

1. Sort all the process according to the arrival time.
2. Then select that process which has minimum arrival time and minimum Burst time.
3. After completion of process make a pool of process which after till the completion of previous process and select that process among the pool which is having minimum Burst time.

High Level documentation

1. Using nested for loop

sorting based on arrival time and burst time of the processes

Low Level documentation

1. No of processes

We take input n from the user

1. Arrival time

We take input a[n] from the user

1. Burst time

We take input b[n] from the user

Round Robin Algorithm (RR)

The name of this algorithm comes from the round-robin principle, where each person gets an equal share of something in turns. It is the oldest, simplest scheduling algorithm, which is mostly used for multitasking.

In Round-robin scheduling, each ready task runs turn by turn only in a cyclic queue for a limited time slice. This algorithm also offers starvation free execution of processes

Algorithm for Round Robin

1.      Ask number of processes, arrival time, CPU burst time, time quanta from the user

2.      Sort the processes in order of arrival time in ascending order

3.      Use simple FIFO queue

4.      Push the first process from the sorted list (step 2) into this queue

5.      Use an array to check if the process is in queue or not -> visited [ ]

6.      Keep track of the time using a variable -> current\_time

7.      If the process unit of time to the process that is at front in the queue and pop the process from the queue

8.      Give quantum unit of time to the process that is at the front of the queue and pop this process from the queue

9.      If the burst time of the process becomes 0, calculate CT, TAT, WT and RT for it

10.   If some processes have arrived when this process was executing, insert that arrived processes into the queue

11.   If the current process has burst time remaining, push the process into queue again

12.   If the queue is empty, pick the first process from the list that is not completed

13.   Keep doing this till all processes are completed

There are two ways in which we can group the information related to processes

* Use separate arrays to store Arrival time and CPU burst time

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

at[0]                   at[1]                    at[2]                     at[3]       ……..     at[n-1]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |

bt[0]                   bt[1]                     bt[2]                  bt[3]      ……..    bt[n-1]

* Use of structure

struct process\_struct

{

int pid;

int at;  //arrival time

int bt;  //CPU burst time

int ct, wt, tat, rt, start\_time  //completion, waiting, turnaround, response time

} ps[100]  //array of structure to store information of each process (ps is process)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| pid =  at=  bt  ct= | pid =  at=  bt  ct= | pid =  at=  bt  ct= | pid =  at=  bt  ct= | pid =  at=  bt  ct= |
|  |
|  |
|  |

ps[0]                   ps[1]                    ps[2]                   ps[3]     ………   ps[n-1]

High Level design documentation:

1. struct ( ) { }

contains all the important variables like process ID, Arrival time, Burst time, Waiting time etc.

1. bool ( )

used to compare and finding out the smallest Arrival time for process initialization

1. sort ( )

used to sort the structure on the basis of arrival time in increasing order

1. push ( ), pop ( )

used to push and pop the process indexes in queue

header <queue>

1. max ( )

used to calculate the maximum of current and arrival time  
header <algorithm>

Low level design documentation:

1. visited [ ]

visited array stores the data of processes which visited in the queue

1. Number of processes, n

Takes user input of number of processes

1. Arrival Time, at

Takes user input for arrival time for each process and stores it in a structure

1. Burst Time, bt

Takes user input for burst time for each process and stores it in a structure

1. Time Slice, tq

Takes user input for how long one process should run at a time before context switch

Read Jobs Input from User

Server Side

High level design Documentation

1. socket ( );

Socket creation

1. initializing socaddr\_in structure ,server detail

struct sockaddr\_in sock\_addr\_serv;

1. binding of socket and server

bind();

1. listen() marks the socket referred to by sock\_fd as a passive socket, that is, as a socket that will be used to accept incoming connection requests using accept.

listen();

accept();

1. read the client send data;

read();

1. file open to add the data to it

file.open(); and close the file with file.close();

Low level design Documentation

1. check error in socket creation

int serv\_fd=socket();

if (serv\_fd==-1)

1. set the ip ,port, family

sock\_addr\_serv.sin\_family = AF\_INET;

sock\_addr\_serv.sin\_port = 9988;

sock\_addr\_serv.sin\_addr.s\_addr = inet\_addr("127.0.0.1")

1. accepting client data

struct sockaddr\_in sock\_addr\_cli;

socklen\_t cli\_len = sizeof(sock\_addr\_cli);

int client\_fd=accept(serv\_fd,(struct sockaddr \*)&sock\_addr\_cli,&cli\_len);

read();

1. data enter in the file;
2. close the serv\_fd;

Client Side

High level design Documentation

1. Socket creation

socket();

1. initializing socaddr\_in structure ,server detail

struct sockaddr\_in sock\_addr\_serv;

1. connecting to server

connect();

1. accept i/p
2. write to server

write();

1. Read the acknowledgement from server

read();

Low level design Documentation

1. check error in socket creation

int serv\_fd=socket();

if (serv\_fd==-1)

1. set the ip ,port, family

sock\_addr\_serv.sin\_family = AF\_INET;

sock\_addr\_serv.sin\_port = 9988;

sock\_addr\_serv.sin\_addr.s\_addr = inet\_addr("127.0.0.1")

1. accept i/p (process id ,arrivaL TIME ,burn time) to send to the server
2. close the serv\_fd;

Comparing Scheduling Algorithms

In this task, we need to launch different scheduling algorithms and capture the average wait time and select the best algorithm.

High level Design

1.comparing all 3 algorithm using else if();

Low level Design

1. accept the average waiting time for all 3 algorithms

2. reflect the best algorithm who have less average waiting time.

Read Jobs from stdin and store in file

Read Jobs Details from File and calculate the Turn Around Time, Completion Time and Wait Time using **FCFS**.

Read Jobs Details from File and calculate the Turn Around Time, Completion Time and Wait Time using **SJF**.

Use Ordered Linked List for storing the jobs

Read Jobs Details from File and calculate the Turn Around Time, Completion Time and Wait Time using RR.

Use Circular Linked List for storing the jobs information.

1. Read details of Jobs
2. Create three different child process and launch FCFS, SJF and RR.
3. Display the results of all the three algorithms and also the best algorithm for the given jobs