REPORT

SJF with priority scheduling

**As a project work for Course**

OPERATING SYSTEMS ( CSE 316 )

----------------------------------------------------------------------------------------- Name : Singh Purushottam Manoj

Registration Number : 11802669

Program : CSE B.Tech

Semester : 4th

School : School of Computer Science

University : Lovely Professional University



INDEX

1. Introduction 3
2. Algorithm with time complexity 4 – 5
3. Constraints 6
4. Additional algorithm 7 – 8
5. Code 8 – 12
6. Boundary Conditions 12
7. Test cases and output produced 13 – 15
8. Github repository link 15

Introduction

One of the most important function of the operating system is to manipulate the ordering of the processes which arrives for execution i.e process scheduling. Operating System has to select a process from the waiting queue and put it in the ready queue. It has also to decide whether to pre-empt the process or let the process finish (non-pre-emptive). There are various methods to achieve this scheduling of process. This project is dedicated to solving one of those methods that is non-pre-emptive shortest job first.

In Shortest Job First scheduling, that process is selected to be moved into ready queue which has the lowest burst time among all the processes in the waiting queue. Since, the mode we are observing here is non-pre-emptive, context switch is not possible. All the processes arriving while a process is being executed have to wait. Since, the selection of process from the ready queue happens on the basis of lowest burst time, it can lead to *starvation* for all those processes which have relatively higher burst time. The method devised to overcome this problem is called ageing. It’s like using a priority schedular along with SJF. Every process arriving has a priority with it. In this method, with the passage of time, the priority of the processes increases which are in the waiting queue.

The formula to calculate the priority depends on the area in which a particular OS is desired to be used for. It may be highly desirable in some areas to implement an algorithm which leads to a nearly uniform access to OS for all the processes (serially) or it may be the case that a pre decided fixed amount of waiting time can be ignored. We are going to use in our project priority = 1 + (waiting time / estimated run time).

Algorithm with Time Complexity

1. 1. N Total number of processes **O (1)**
2. at[n], bt[n] arrival time, burst time **O (1)**

// Sort all processes using bubble sort

1. bubble\_sort (n, p [])

{

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

for (j=1; j<n-i; j++) **O (n)**

{

if(at[i-1] > a[i]) **O (1)**

swap(p[i-1], p[i]) // p[i] denotes the process. **O (1)**

}

}

1. priority (int at, int bt, int total\_time\_passed) // to calculate the priority of all the incoming processes

{

waiting time = total\_time\_passed – at; **O (1)**

return 1+ (waiting time/ bt); **O (1)**

}

1. k = 1 **O (1)**
2. for (j=0; j< n; j++) **O (n)**

{

total\_time\_passsed = total\_time\_passed + bt[j]; **O (1)**

min = priority(at[k], bt[k], total\_time\_passsed); **O (1)**

for (i=k; i<n; i++) **O (n)**

{

if (total\_time\_passed > = at[i] && priority (at[i], bt[i], total\_time\_passed )

{

swap (p[k], p[j]) **O (1)**

}

}

k ++; **O (1)**

}

1. w [0] = 0; **O (1)**
2. wsum = 0 **O (1)**
3. for (i=1; i<n; i++) **O (n)**

{

sum = sum + bt[i-1]; **O (1)**

wt[i] = sum – at[i]; **O (1)**

wsum = wsum + wt[i]; **O (1)**

}

1. wavg = wsum / n;  **O (1)**
2. for (i=0; i<n; i++) **O (n)**

{

ta = ta + bt[i]; **O (1)**

tt[i] = ta – at[i]; **O (1)**

tsum = tsum + tt[i]; **O (1)**

}

1. tavg = tsum / n; **O (1)**
2. return;

total complexity=O (1) +O (1) +O(n\*n) +O (1) +O (1)

+ O(n\*n) + O (1) + O (1) + O(n) + O(1)

+ O(n) + O(1)

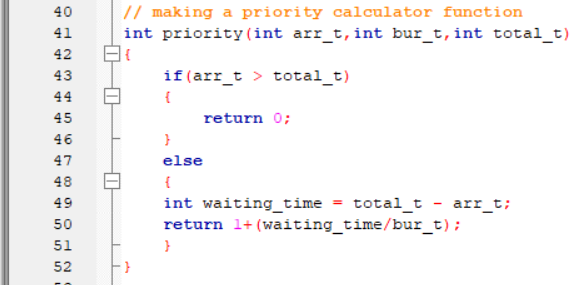
Total complexity = O(n^2)

Constraints

The underlying problem has to deal with the increasing priority with the passage of the time. This functionality has been added in the code. The execution order of the processes don’t just depend on the order of the arrival time combined with lower burst time, but it also depends on the time a process has to wait while other processes are executing.

The formula used for calculating the priority is 1 + (waiting time / estimated run time).

Code snippet:



This function call priority finds out the priority of a process.

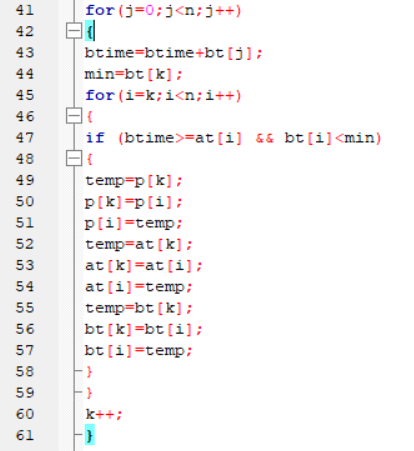
The arguments passed to it are arrival time, burst time and total time passed till now. If a process arrival time is greater than the total time passed, it’s priority is considered to be 0.

Additional Algorithm

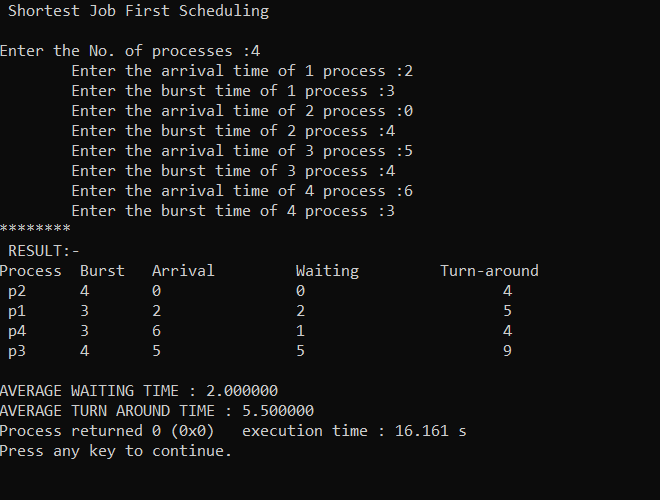
The goal is to show how including priority with a process in terms of waiting time leads to removal of starvation cases. For this to prove that our written algorithm works different than the SJf, we are also going to add the functional code of SJF in the comment section.

The screenshot of output in that case is also attached herewith.

Code snippet:



The output corresponding to this code with input is (1 test case):



Notice that the process p4 is executed before p3 even though the priority of p3 is higher. We are going to tackle this situation in our next code.

CODE

#include<stdio.h>

int main()

{

int i,n,p[10]={1,2,3,4,5,6,7,8,9,10},min,k=1,btime=0;

int burst\_time[10],temp,j,arrival\_time[10],wt[10],tt[10],ta=0,sum=0;

float wavg=0,tavg=0,tsum=0,wsum=0;

printf(" \*\*\*\*\*\*\*Shortest Job First Scheduling\*\*\*\*\*\*\*\n");

printf("\nEnter total No. of processes :");

scanf("%d",&n);

// taking input about the arrival time and the burst time

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

{

printf("\tEnter the arrival time of %d process :",i+1);

scanf(" %d",&arrival\_time[i]);

printf("\tEnter the burst time of %d process :",i+1);

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

}

// Sorting process based on the arrival time

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

{

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

{

if(arrival\_time[i]<arrival\_time[j])

{

temp=p[j];

p[j]=p[i];

p[i]=temp;

temp=arrival\_time[j];

arrival\_time[j]=arrival\_time[i];

arrival\_time[i]=temp;

temp=burst\_time[j];

burst\_time[j]=burst\_time[i];

burst\_time[i]=temp;

}

}

}

int priority(int arr\_t,int bur\_t,int total\_t)

{

if(arr\_t > total\_t)

{

return 0;

}

else

{

int waiting\_time = total\_t - arr\_t;

return 1+(waiting\_time/bur\_t);

}

}

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

{

btime=btime+burst\_timetime[j];

min=priority(arrival\_time[k],burst\_time[k],btime);

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

{

if (btime>=arrival\_time[i] && priority(arrival\_time[i],burst\_time[i], btime) > min)

{

temp=p[k];

p[k]=p[i];

p[i]=temp;

temp=arrival\_time[k];

arrival\_time[k]=arrival\_time[i];

arrival\_time[i]=temp;

temp=burst\_time[k];

burst\_time[k]=burst\_time[i];

burst\_time[i]=temp;

}

}

k++;

}

wt[0]=0;

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

{

sum=sum+burst\_time[i-1];

wt[i]=sum-arrival\_time[i];

wsum=wsum+wt[i];

}

wavg=(wsum/n);

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

{

ta=ta+burst\_time[i];

tt[i]=ta-arrival\_time[i];

tsum=tsum+tt[i];

}

tavg=(tsum/n);

printf("\*\*\*\*\*\*\*\*");

printf("\n RESULT:-");

printf("\nProcess\t Burst\t Arrival\t Waiting\t Turn-around" );

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

{

printf("\n p%d\t %d\t %d\t\t %d\t\t\t%d",p[i],burst\_time[i],arrival\_time[i],wt[i],tt[i]);

}

printf("\n\nAVERAGE WAITING TIME : %f",wavg);

printf("\nAVERAGE TURN AROUND TIME : %f",tavg);

return 0;

}

Boundary Conditions

Note – While specifying the input, following points are to be strictly followed.

1. The maximum numbers of processes that can be checked for is 10.
2. The arrival time of all the processes should be greater than 0.
3. The burst time of all the processes should be greater than 0.
4. All the input should be integer (whole numbers wiz. 0, 1, 2, … n).
5. No two processes are to have same arrival time and same burst time.

**Test cases and output produced**

Sample test case 1.

|  |  |  |
| --- | --- | --- |
| Process | Arrival time | Burst time |
| P1 | 0 | 20 |
| P2 | 5 | 36 |
| P3 | 13 | 19 |
| P4 | 17 | 42 |

The process p1 arrives at time 0. It will continue till time unit 20. Till then processes p2, p3, p4 will be in waiting queue. Calculate the priority of all the processes in waiting queue.

Priority of p2 = 1 + (20 – 5)/36 =1.416

Priority of p3 = 1 + (20 – 13)/19 = 1.36

Priority of p4 = 1 + (20 – 17)/42 = 1.071

Since the priority of p2 is greatest, p2 will be executed next. Now, the total time passed is 20 + 36 = 56. Calculate the priority of all the remaining processes in waiting queue.

Priority of p3 = 1 + (56 – 13)/19 = 3.263

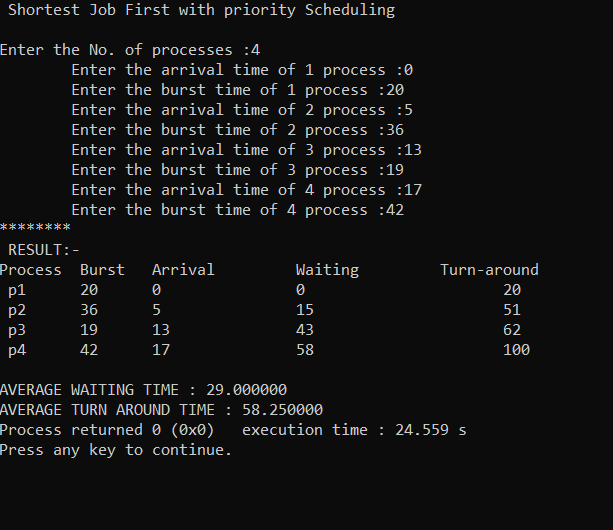
Priority of p4 = 1+ (56 – 17)/42 = 1.928

Since priority of p3 is greater, this will be executed.

After this, the only remaining process is p4 which will be executed.

We can verify this through running the code with above input.

Output snippet :



Sample test case 2:

|  |  |  |
| --- | --- | --- |
| Process | Arrival time | Burst time |
| P1 | 2 | 3 |
| P2 | 0 | 4 |
| P3 | 6 | 5 |
| P4 | 7 | 4 |

Explanation: The process p2 arrives at time 0. So, this process will be executed first.

Total time passed after the processing of p2 = 0 + 4 = 4. After this, the process which are in waiting queue is p2 only. So, this will execute directly. Total time passed after execution of p2 = 4 + 3 = 7.

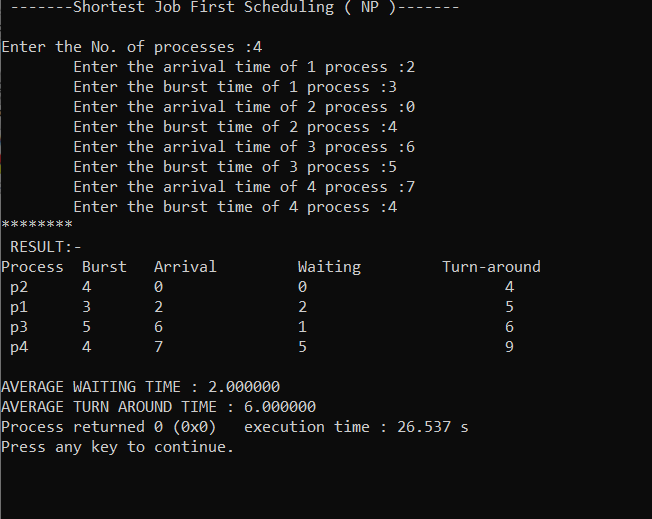
Now, both process p3 and p4 will be in waiting queue as both have their arrival time less than 7.

Calculate the priority of p3 = 1 + (7 – 6)/5 = 1.2

Calculate the priority of p4 = 1 + (7 – 7)/4 = 1

Since, the priority of p3 is higher, p3 will be executed first. After this, the only process left will be p4 which will be executed at the last. It is worth noting that had the algorithm been SJF, process p4 would have executed first, since it has less burst time.

Output snippet:



Github repository link -

*Thank You*