

|  |  |
| --- | --- |
| **Submitted To :** | Saima Siddique Tashfia Lecturer, Dept. of CSE UITS. |
| **Submitted By :** | Maheru Tabassum Ohana  ID : 2125051015  Dept : CSE, Batch : 50, Section : 7A |
| **Course Code :** | CSE412 |
| **Course Title :** | Operating System Lab |

**Lab Report :** 01

**Date of Submission :** 7 October, 2024.

‘ 

**Experiment No :** 01

**Experiment Name :** CPU SCHEDULING ALGORITHM: First Come First Serve

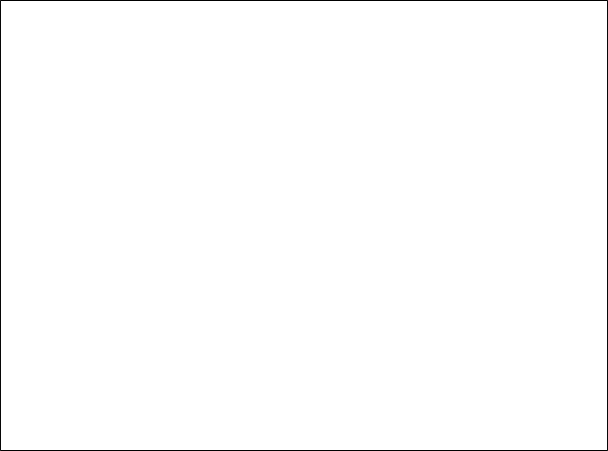
**AIM :** To write a c program to simulate the CPU scheduling algorithm First Come First Serve (FCFS) in C++ and Bash.

**DESCRIPTION :**

The First-Come, First-Served (FCFS) scheduling algorithm is a simple CPU scheduling method where processes are executed in the order of their arrival in the ready queue. It is non-preemptive, meaning once a process starts executing, it runs to completion without interruption. FCFS operates on a First-In, First-Out (FIFO) basis, ensuring fairness by giving each process an equal opportunity. However, it can lead to high average waiting times, particularly if shorter processes are stuck behind longer ones, a phenomenon known as the "convoy effect." While easy to implement, FCFS may be less efficient than more advanced scheduling algorithms.

**In C++ :**

**SOURCE CODE:**



#include

<

iostream

>

#include

<

iomanip

>

#include

<

vector

>

using

namespace

std;

void

generateUniqueID(string

studentID)

{

int

uniqueCode

=

0

;

for

(

char

c

:

studentID)

{

uniqueCode

+=

(

int)c;

}

cout

<<

"Unique

ID

based

on

Student

ID

(

"

<<

studentID

<<

"):

"<<

uniqueCode

<<

endl;

}

struct

Process

{

int

id,

AT,

BT,

CT,

TAT,

WT;

}

;

void

FCFS(vector<Process>&

processes,

int

n)

{

int

total\_TAT

=

0

,

total\_WT

=

0

;

processes[0].CT

=

processes[0].AT

+

processes[0].BT;

//

First

process

processes[0].TAT

=

processes[0].CT

-

processes[0].AT;

//

TAT

processes[0].WT

=

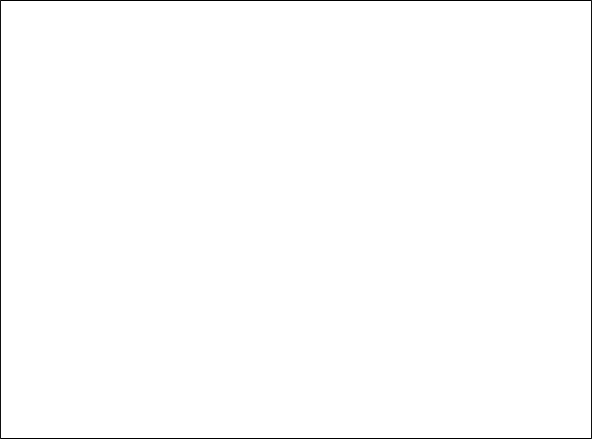
processes[0].TAT

-

processes[0].BT;

//

WT



total\_TAT

+=

processes[0].TAT;

total\_WT

+=

processes[0].WT;

for

(

int

i

=

1

;

i

<

n;

i++)

{

processes[i].CT

=

max(processes[i].AT,

processes[i

-

1]

.CT

)

+

processes[i].BT;

//

CT

processes[i].TAT

=

processes[i].CT

-

processes[i].AT;

//

TAT

processes[i].WT

=

processes[i].TAT

-

processes[i].BT;

//

WT

total\_TAT

+=

processes[i].TAT;

total\_WT

+=

processes[i].WT;

}

cout

<<

"\nProcess\tAT\tBT\tCT\tTAT\tWT\n";

for

(

int

i

=

0

;

i

<

n;

i++)

{

cout

<<

"P"

<<

processes[i].id

<<

"\t"

<<

processes[i].AT

<<

"\t"

<<

processes[i].BT

<<

"\t"

<<

processes[i].CT

<<

"\t"

<<

processes[i].TAT

<<

"\t"

<<

processes[i].WT

<<

"\n";

}

cout

<<

"\nAverage

Turnaround

Time

=

"

<<

(

float)total\_TAT

/

n;

cout

<<

"\nAverage

Waiting

Time

=

"

<<

(

float)total\_WT

/

n

<<

"\n";

}

int

main()

{

string

studentID

=

"2125051015";

generateUniqueID(studentID);

int

n;

cout

<<

"Enter

the

number

of

processes:

";

cin

>>

n;

vector<Process>

processes(n);

for

(

int

i

=

0

;

i

<

n;

i++)

{

processes[i].id

=

i

+

1

;

cout

<<

"Enter

arrival

time

and

burst

time

for

process

P"

<<

i

+

1

<<

":

";

cin

>>

processes[i].AT

>>

processes[i].BT;

}

FCFS(processes,

n);

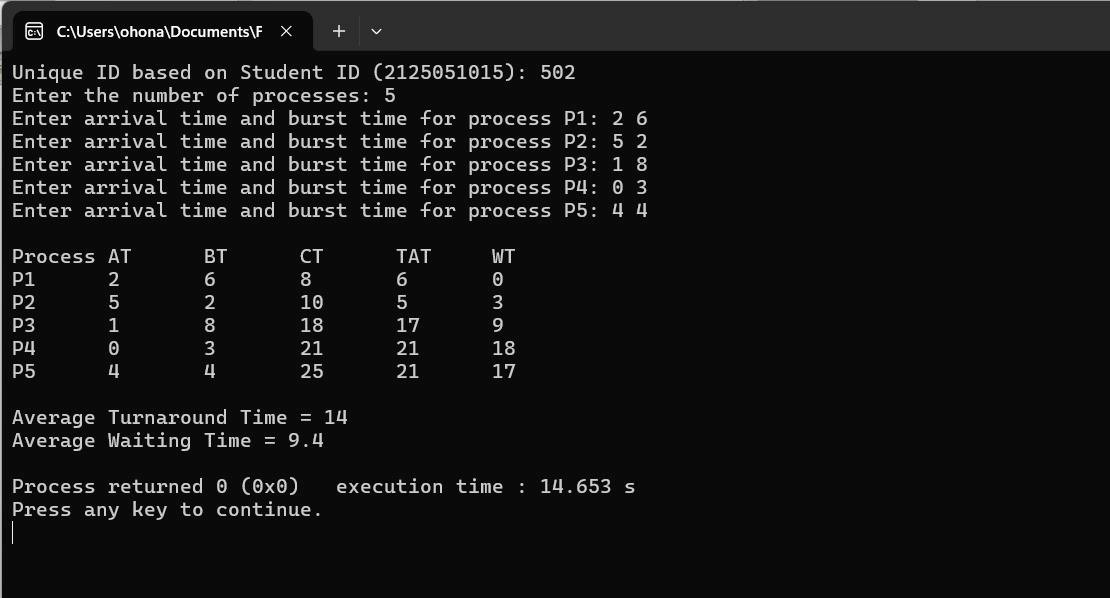
return

0

;

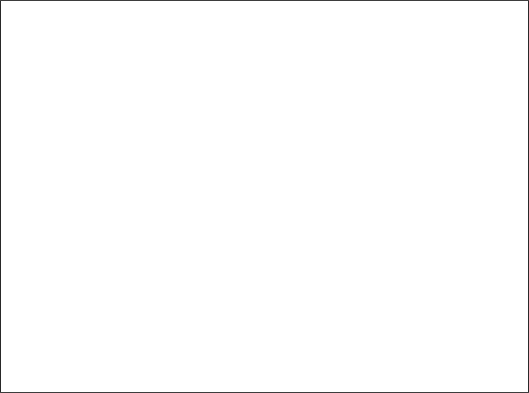
}

**INPUT & OUTPUT -**



**In BashScript :**

**SOURCE CODE:**



#!/bin/bash

generateUniqueID()

{

local

studentID=$1

local

uniqueCode=0

for

((

i=0;

i<${#studentID};

i++

))

;

do

uniqueCode=$((

uniqueCode

+

$(printf

"%d"

"'${studentID:$i:1}")

))

done

echo

"Unique

ID

based

on

Student

ID

(

$studentID

):

$uniqueCode"

}

studentID="2125051015"

generateUniqueID

$studentID

fcfsScheduling()

{

local

n=$1

local

-

a

at\_arr=("${@:2:$n}")

local

-

a

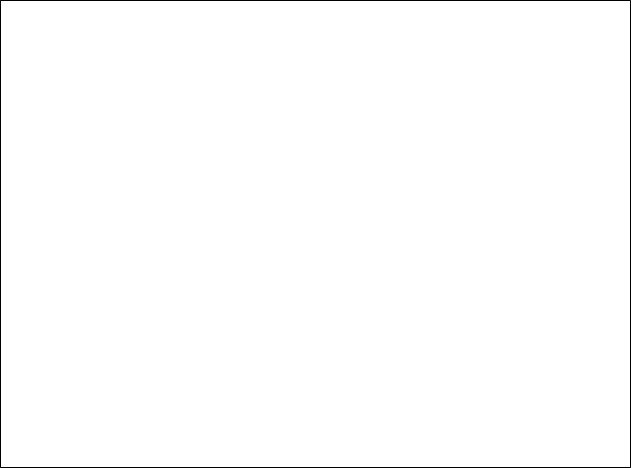
bt\_arr=("${@:$((n+2)):$n}")

#

Burst

time

array



local

-

a

ct\_arr=()

local

-

a

tat\_arr=()

local

-

a

wt\_arr=()

ct\_arr[0]=$((

at\_arr[0]

+

bt\_arr[0]

))

for

((

i=1;

i<$n;

i++

))

;

do

if

((

${at\_arr[i]}

>

${ct\_arr[i-1]}

))

;

then

ct\_arr[i]=$((

at\_arr[i]

+

bt\_arr[i]

))

else

ct\_arr[i]=$((

ct\_arr[i-1]

+

bt\_arr[i]

))

fi

done

local

total\_tat=0

local

total\_wt=0

for

((

i=0;

i<$n;

i++

))

;

do

tat\_arr[i]=$((

ct\_arr[i]

-

at\_arr[i]

))

#

TAT

=

CT

-

AT

wt\_arr[i]=$((

tat\_arr[i]

-

bt\_arr[i]

))

#

WT

=

TAT

-

BT

total\_tat=$((

total\_tat

+

tat\_arr[i]

))

total\_wt=$((

total\_wt

+

wt\_arr[i]

))

Done

echo

-

e

"\nProcesses\tAT\tBT\tCT\tTAT\tWT"

echo

"-----------------------------------------"

for

((

i=0;

i<$n;

i++

))

;

do

echo

-

e

"$i\t\t${at\_arr[i]}\t${bt\_arr[i]}\t${ct\_arr[i]}\t${tat\_arr[i]}\t${wt\_arr[i]}"

done

local

avg\_tat=$(awk

"BEGIN

{

print

$total\_tat/$n}")

local

avg\_wt=$(awk

"BEGIN

{

print

$total\_wt/$n}")

echo

-

e

"\nAverage

Turnaround

Time

(

TAT

)

=

$avg\_tat"

echo

"Average

Waiting

Time

(

WT

)

=

$avg\_wt"

}

main()

{

echo

-

e

"Enter

number

of

processes:

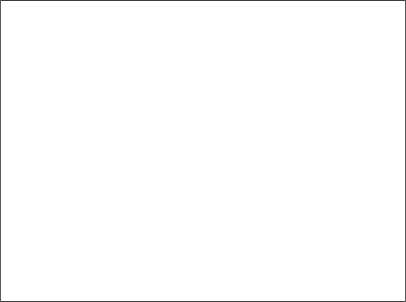
"

read

n

at\_arr=()

bt\_arr=()



for

((

i=0;

i<$n;

i++

))

;

do

echo

-

e

"Enter

Arrival

Time

(

AT

)

for

process

$i:

"

read

at

at\_arr+=($at)

echo

-

e

"Enter

Burst

Time

(

BT

)

for

process

$i:

"

read

bt

bt\_arr+=($bt)

done

fcfsScheduling

$n

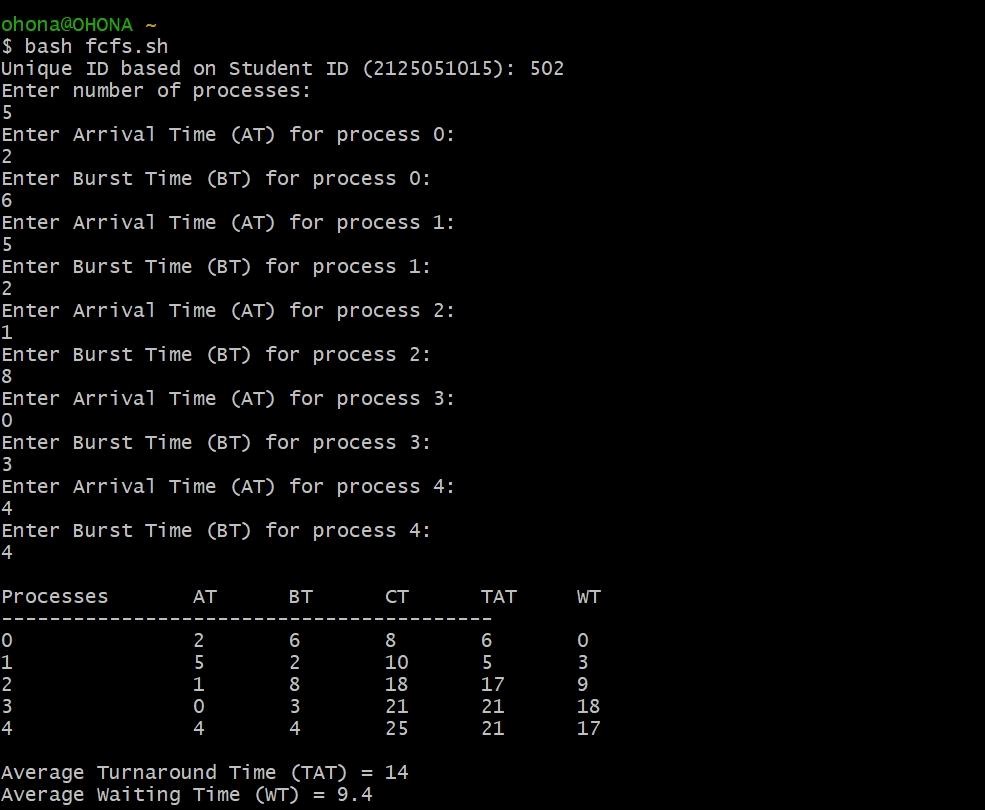
"${at\_arr[@]}"

"${bt\_arr[@]}"

}

main

**INPUT & OUTPUT -**



**Experiment No :** 02

**Experiment Name :** CPU SCHEDULING ALGORITHM: Shortest Job First (Pre-Emptive)

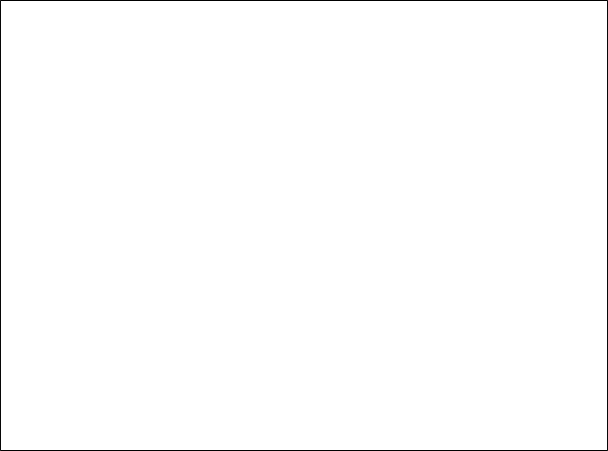
**AIM :** To write a c program to simulate the CPU scheduling algorithm Pre-Emptive Shortest Job First (SJF) in C++ and Bash.

**DESCRIPTION :**

Preemptive Shortest Job First (SJF) is a CPU scheduling algorithm that allows a currently running process to be interrupted if a new process arrives with a shorter remaining burst time. This approach minimizes average waiting and turnaround times by prioritizing shorter tasks. However, it can lead to starvation, where longer processes are delayed indefinitely as shorter jobs arrive. While effective in optimizing performance, it requires accurate knowledge of burst times, making implementation more complex.

**In C++ :**

**SOURCE CODE:**



#include

<

iostream

>

#include

<

iomanip

>

#include

<

vector

>

#include

<

algorithm

>

using

namespace

std;

void

generateUniqueID(string

studentID)

{

int

uniqueCode

=

0

;

for

(

char

c

:

studentID)

{

uniqueCode

+=

(

int)c;

}

cout

<<

"Unique

ID

based

on

Student

ID

(

"

<<

studentID

<<

"):

"<<

uniqueCode

<<

endl;

}

struct

Process

{

int

id,

AT,

BT,

CT,

TAT,

WT,

remaining\_BT;

}

;

bool

arrivalTimeComparator(const

Process

&a,

const

Process

&b)

{

return

a.AT

<

b.AT;

}

void

SJF\_Preemptive(vector<Process>&

processes,

int

n)

{

int

total\_TAT

=

0

,

total\_WT

=

0

;

int

current\_time

=

0

,

completed

=

0

,

min\_index

=

-1

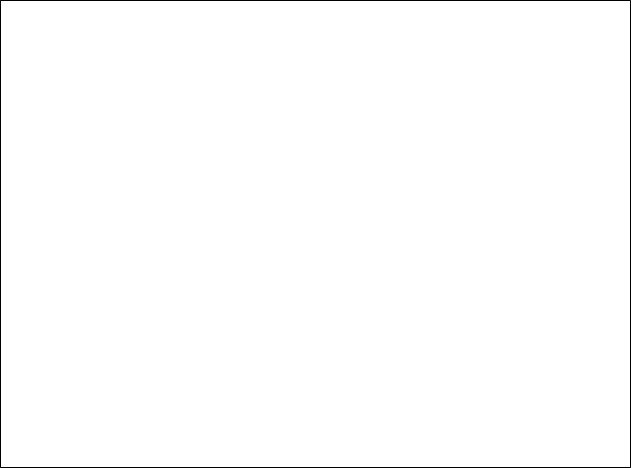
;

bool

process\_in\_execution

=

false;



while

(

completed

<

n)

{

min\_index

=

-1

;

int

min\_BT

=

INT\_MAX;

for

(

int

i

=

0

;

i

<

n;

i++)

{

if

(

processes[i].AT

<=

current\_time

&&

processes[i].remaining\_BT

>

0)

{

if

(

processes[i].remaining\_BT

<

min\_BT)

{

min\_BT

=

processes[i].remaining\_BT;

min\_index

=

i;

}

}

}

if

(

min\_index

==

-1)

{

current\_time++;

//

No

process

is

ready

to

execute,

so

just

increment

time

}

else

{

processes[min\_index].remaining\_BT--;

current\_time++;

if

(

processes[min\_index].remaining\_BT

==

0)

{

completed++;

processes[min\_index].CT

=

current\_time;

processes[min\_index].TAT

=

processes[min\_index].CT

-

processes[min\_index].AT;

//

Turn

Around

Time

processes[min\_index].WT

=

processes[min\_index].TAT

-

processes[min\_index].BT;

//

Waiting

Time

total\_TAT

+=

processes[min\_index].TAT;

total\_WT

+=

processes[min\_index].WT;

}

}

}

cout

<<

"\nProcess\tAT\tBT\tCT\tTAT\tWT\n";

for

(

int

i

=

0

;

i

<

n;

i++)

{

cout

<<

"P"

<<

processes[i].id

<<

"\t"

<<

processes[i].AT

<<

"\t"

<<

processes[i].BT

<<

"\t"

<<

processes[i].CT

<<

"\t"

<<

processes[i].TAT

<<

"\t"

<<

processes[i].WT

<<

"\n";

}

cout

<<

"\nAverage

Turnaround

Time

=

"

<<

(

float)total\_TAT

/

n;

cout

<<

"\nAverage

Waiting

Time

=

"

<<

(

float)total\_WT

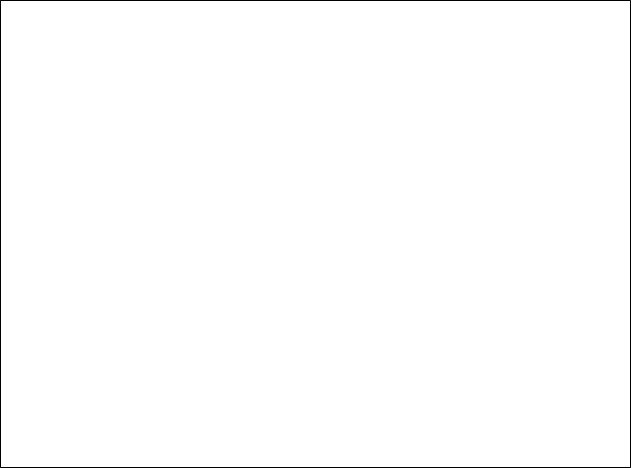
/

n

<<

"\n";

}



int

main()

{

string

studentID

=

"2125051015";

generateUniqueID(studentID);

int

n;

cout

<<

"Enter

the

number

of

processes:

";

cin

>>

n;

vector<Process>

processes(n);

for

(

int

i

=

0

;

i

<

n;

i++)

{

processes[i].id

=

i

+

1

;

cout

<<

"Enter

arrival

time

and

burst

time

for

process

P"

<<

i

+

1

<<

":

";

cin

>>

processes[i].AT

>>

processes[i].BT;

processes[i].remaining\_BT

=

processes[i].BT;

}

sort(processes.begin(),

processes.end(),

arrivalTimeComparator);

SJF\_Preemptive(processes,

n);

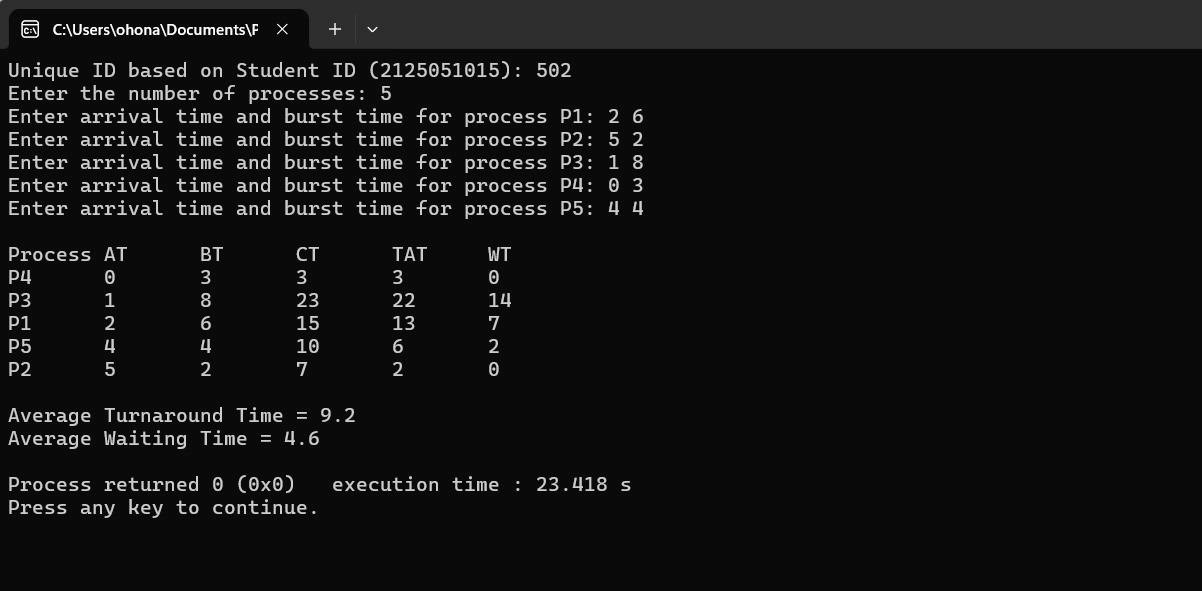
return

0

;

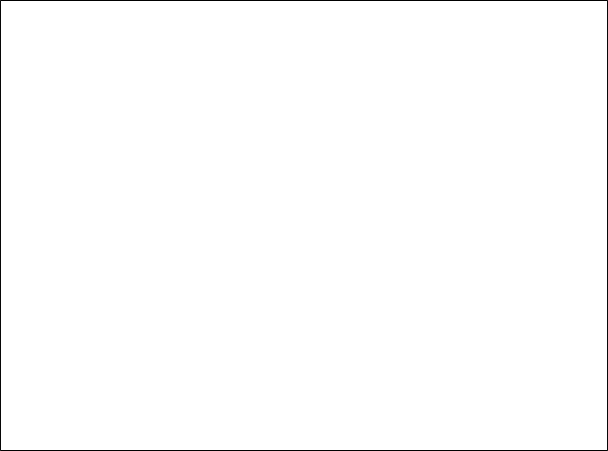
}

**INPUT & OUTPUT -**



**In BashScript :**

**SOURCE CODE:**



#!/bin/bash

generateUniqueID()

{

local

studentID=$1

local

uniqueCode=0

for

((

i=0;

i<${#studentID};

i++

))

;

do

uniqueCode=$((

uniqueCode

+

$(printf

"%d"

"'${studentID:$i:1}")

))

done

echo

"Unique

ID

based

on

Student

ID

(

$studentID

):

$uniqueCode"

}

sjfPreemptiveScheduling()

{

local

n=$1

local

arrival\_time=("${!2}")

local

burst\_time=("${!3}")

local

remaining\_time=("${burst\_time[@]}")

local

ct=()

#

Completion

time

local

tat=()

#

Turnaround

time

local

wt=()

#

Waiting

time

local

is\_completed=()

for

((

i=0;

i<$n;

i++

))

;

do

is\_completed[i]=0

done

local

current\_time=0

local

completed=0

local

total\_tat=0

local

total\_wt=0

local

min\_burst=99999

local

shortest=-1

local

finish\_time=0

local

check=0

while

((

completed

<

n

))

;

do

for

((

i=0;

i<$n;

i++

))

;

do

if

((

arrival\_time[i]

<=

current\_time

&&

is\_completed[i]

==

0

))

;

then

if

((

remaining\_time[i]

<

min\_burst

&&

remaining\_time[i]

>

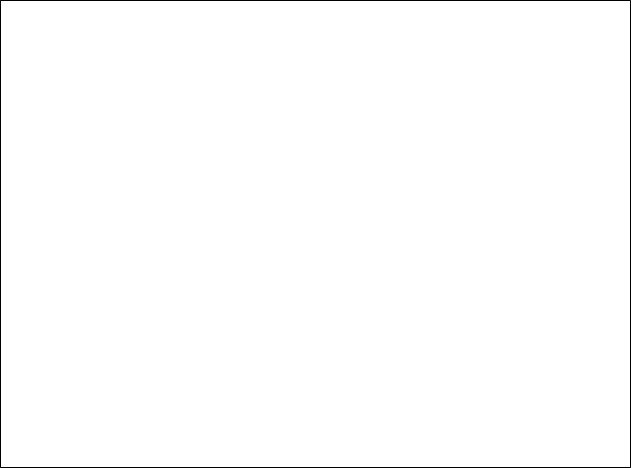
0

))

;

then

min\_burst=${remaining\_time[i]}



shortest=$i

check=1

fi

fi

done

if

((

check

==

0

))

;

then

current\_time=$((

current\_time

+

1

))

continue

Fi

remaining\_time[shortest]=$((

remaining\_time[shortest]

-

1

))

min\_burst=${remaining\_time[shortest]}

if

((

min\_burst

==

0

))

;

then

min\_burst=99999

fi

if

((

remaining\_time[shortest]

==

0

))

;

then

completed=$((

completed

+

1

))

finish\_time=$((

current\_time

+

1

))

ct[shortest]=$finish\_time

tat[shortest]=$((

ct[shortest]

-

arrival\_time[shortest]

))

wt[shortest]=$((

tat[shortest]

-

burst\_time[shortest]

))

total\_tat=$((

total\_tat

+

tat[shortest]

))

total\_wt=$((

total\_wt

+

wt[shortest]

))

is\_completed[shortest]=1

fi

current\_time=$((

current\_time

+

1

))

done

echo

-

e

"\nProcesses\tAT\tBT\tCT\tTAT\tWT"

for

((

i=0;

i<$n;

i++

))

;

do

echo

-

e

"$i\t\t${arrival\_time[i]}\t${burst\_time[i]}\t${ct[i]}\t${tat[i]}\t${wt[i]}"

Done

avg\_tat=$(awk

"BEGIN

{

print

$total\_tat

/

$n}")

avg\_wt=$(awk

"BEGIN

{

print

$total\_wt

/

$n}")

echo

-

e

"\nAverage

Turnaround

Time

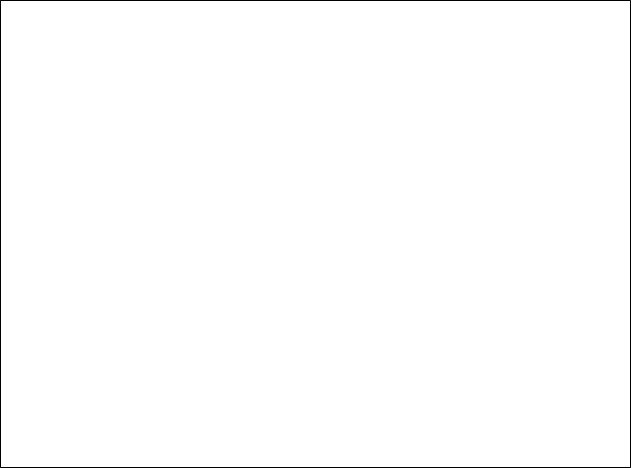
(

TAT

)

=

$avg\_tat"



echo

"Average

Waiting

Time

(

WT

)

=

$avg\_wt"

}

main()

{

echo

-

e

"Enter

number

of

processes:

"

read

n

arrival\_time=()

burst\_time=()

for

((

i=0;

i<$n;

i++

))

;

do

echo

-

e

"Enter

Arrival

Time

(

AT

)

for

process

$i:

"

read

at

arrival\_time+=($at)

echo

-

e

"Enter

Burst

Time

(

BT

)

for

process

$i:

"

read

bt

burst\_time+=($bt)

done

sjfPreemptiveScheduling

$n

arrival\_time[@]

burst\_time[@]

}

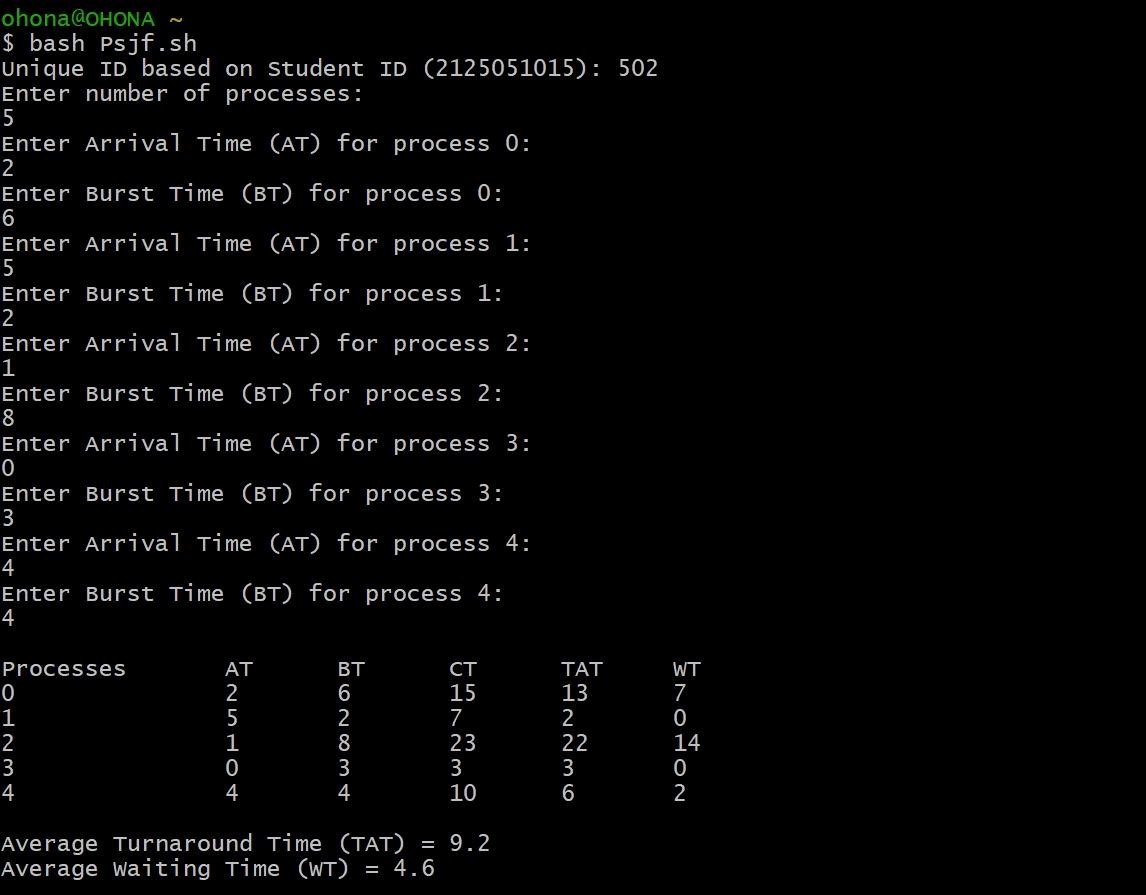
studentID="2125051015"

generateUniqueID

$studentID

main

**INPUT & OUTPUT -**



**Experiment No :** 03

**Experiment Name :** CPU SCHEDULING ALGORITHM: Shortest Job First (Non-preemptive)

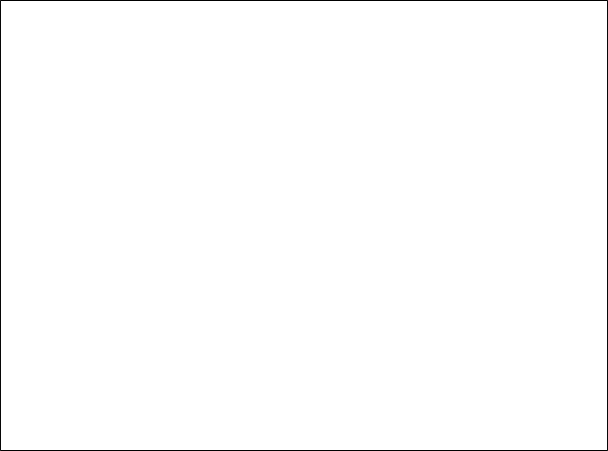
**AIM :** To write a c program to simulate the CPU scheduling algorithm Non-preemptive Shortest Job First (SJF) in C++ and Bash.

**DESCRIPTION :**

Non-Preemptive Shortest Job First (SJF) is a CPU scheduling algorithm where the process with the shortest burst time is selected to run next. Once a process starts, it runs to completion without interruption. This approach minimizes average waiting and turnaround times but can cause longer processes to experience delays if shorter jobs keep arriving. It's simple to implement but requires knowing or estimating process burst times in advance.

**In C++ :**

**SOURCE CODE:**



#include

<

iostream

>

#include

<

iomanip

>

#include

<

string

>

#include

<

algorithm

>

#include

<

vector

>

#include

<

limits

>

using

namespace

std;

void

generateUniqueID(string

studentID)

{

int

uniqueCode

=

0

;

for

(

char

c

:

studentID)

{

uniqueCode

+=

(

int)c;

}

cout

<<

"Unique

ID

based

on

Student

ID

(

"

<<

studentID

<<

"):

"<<

uniqueCode

<<

endl;

}

struct

Process

{

int

id;

int

at;

int

bt;

int

ct;

int

tat;

int

wt;

}

;

bool

arrivalTimeCompare(Process

a,

Process

b)

{

if

(

a.at

==

b.at)

return

a.bt

<

b.bt;

return

a.at

<

b.at;

}

int

findShortestJob(vector<Process>

&processes,

int

currentTime,

vector<bool>

&completed)

{

int

minBT

=

numeric\_limits<int>::max();

int

index

=

-1

;

for

(

int

i

=

0

;

i

<

processes.size();

i++)

{

if

(!

completed[i

]

&&

processes[i].at

<=

currentTime

&&

processes[i].bt

<

minBT)

{

minBT

=

processes[i].bt;

index

=

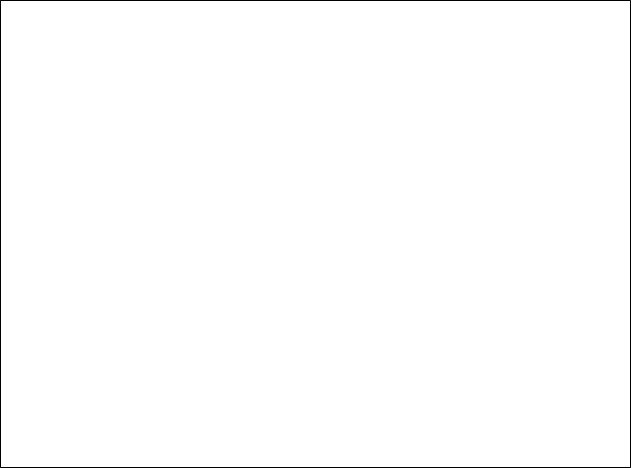
i;

}

}

return

index;



}

int

main()

{

string

studentID

=

"2125051015";

generateUniqueID(studentID);

int

n;

cout

<<

"Enter

the

number

of

processes:

";

cin

>>

n;

vector<Process>

processes(n);

for

(

int

i

=

0

;

i

<

n;

i++)

{

processes[i].id

=

i

+

1

;

//

Process

IDs

start

from

1

cout

<<

"Enter

Arrival

Time

and

Burst

Time

for

Process

"

<<

processes[i].id

<<

":

";

cin

>>

processes[i].at

>>

processes[i].bt;

}

sort(processes.begin(),

processes.end(),

arrivalTimeCompare);

vector<bool>

completed(n,

false);

//

Track

completed

processes

int

currentTime

=

0

;

int

completedProcesses

=

0

;

double

totalTAT

=

0

,

totalWT

=

0

;

while

(

completedProcesses

<

n)

{

int

shortestJob

=

findShortestJob(processes,

currentTime,

completed);

if

(

shortestJob

!=

-1)

{

Process

&p

=

processes[shortestJob];

currentTime

=

max(currentTime,

p.at)

+

p.bt;

p.ct

=

currentTime;

p.tat

=

p.ct

-

p.at;

p.wt

=

p.tat

-

p.bt;

totalTAT

+=

p.tat;

totalWT

+=

p.wt;

completed[shortestJob]

=

true;

//

Mark

the

job

as

completed

completedProcesses++;

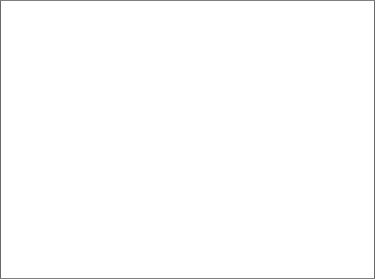
}

else

{

currentTime++;

}



}

cout

<<

"\nProcess\tAT\tBT\tCT\tTAT\tWT\n";

for

(

const

Process

&p

:

processes)

{

cout

<<

"P"

<<

p.id

<<

"\t"

<<

p.at

<<

"\t"

<<

p.bt

<<

"\t"

<<

p.ct

<<

"\t"

<<

p.tat

<<

"\t"

<<

p.wt

<<

"\n";

}

cout

<<

fixed

<<

setprecision(2);

cout

<<

"\nAverage

Turn

Around

Time:

"

<<

totalTAT

/

n;

cout

<<

"\nAverage

Waiting

Time:

"

<<

totalWT

/

n

<<

endl;

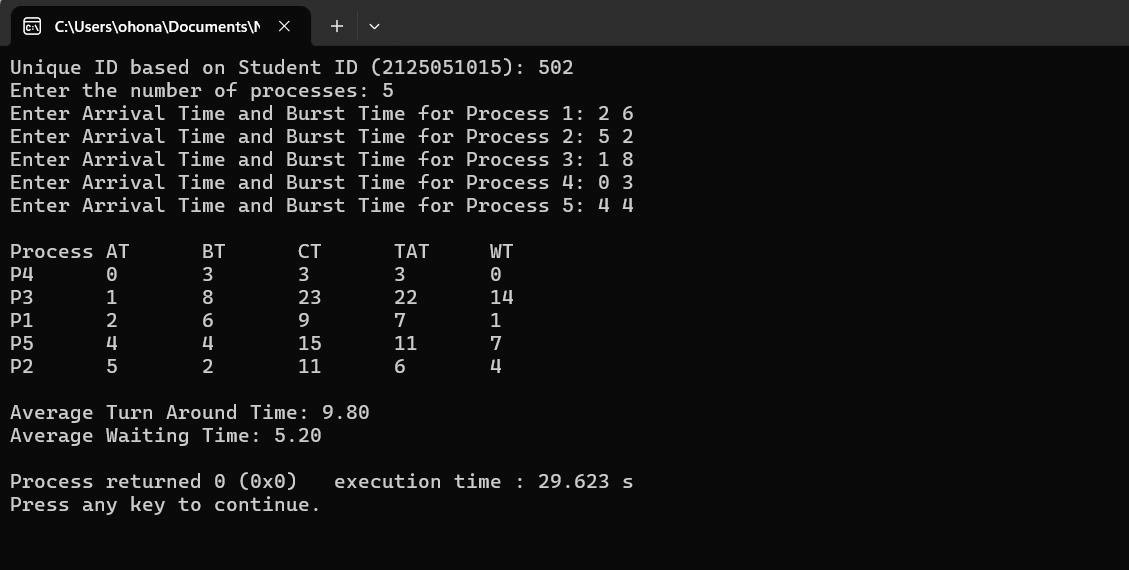
return

0

;

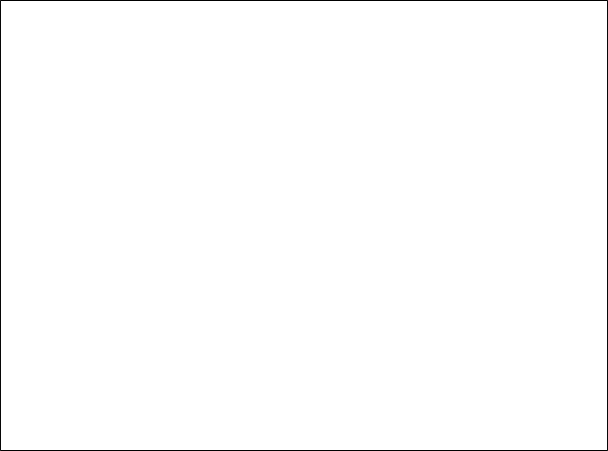
}

**INPUT & OUTPUT -**



**In BashScript :**

**SOURCE CODE:**



#!/bin/bash

generateUniqueID()

{

local

studentID=$1

local

uniqueCode=0

for

((

i=0;

i<${#studentID};

i++

))

;

do

uniqueCode=$((

uniqueCode

+

$(printf

"%d"

"'${studentID:$i:1}")

))

done

echo

"Unique

ID

based

on

Student

ID

(

$studentID

):

$uniqueCode"

}

studentID="2125051015"

generateUniqueID

$studentID

sjfScheduling()

{

local

n=$1

local

-

n

arrival\_time=$2

local

-

n

burst\_time=$3

local

ct=()

local

tat=()

local

wt=()

local

is\_completed=()

for

((

i=0;

i<$n;

i++

))

;

do

is\_completed[i]=0

done

local

current\_time=0

local

completed=0

local

total\_tat=0

local

total\_wt=0

while

((

completed

<

n

))

;

do

local

min\_burst=99999

local

index=-1

for

((

i=0;

i<$n;

i++

))

;

do

if

((

arrival\_time[i]

<=

current\_time

&&

is\_completed[i]

==

0

))

;

then

if

((

burst\_time[i]

<

min\_burst

))

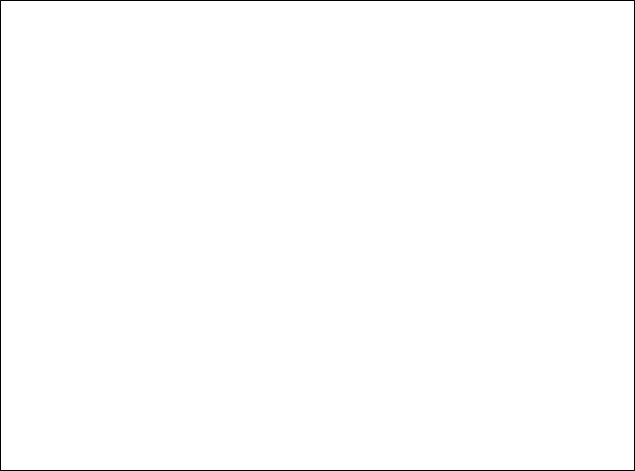
;

then

min\_burst=${burst\_time[i]}

index=$i

fi



fi

done

if

((

index

!=

-1

))

;

then

ct[index]=$((

current\_time

+

burst\_time[index]

))

tat[index]=$((

ct[index]

-

arrival\_time[index]

))

wt[index]=$((

tat[index]

-

burst\_time[index]

))

total\_tat=$((

total\_tat

+

tat[index]

))

total\_wt=$((

total\_wt

+

wt[index]

))

current\_time=${ct[index]}

is\_completed[index]=1

completed=$((

completed

+

1

))

else

current\_time=$((

current\_time

+

1

))

fi

done

echo

-

e

"\nProcesses\tAT\tBT\tCT\tTAT\tWT"

for

((

i=0;

i<$n;

i++

))

;

do

echo

-

e

"$i\t\t${arrival\_time[i]}\t${burst\_time[i]}\t${ct[i]}\t${tat[i]}\t${wt[i]}"

done

avg\_tat=$(bc

-

l

<<<

"scale=2;

$total\_tat/$n")

avg\_wt=$(bc

-

l

<<<

"scale=2;

$total\_wt/$n")

echo

-

e

"\nAverage

Turnaround

Time

(

TAT

)

=

$avg\_tat"

echo

"Average

Waiting

Time

(

WT

)

=

$avg\_wt"

}

main()

{

echo

-

e

"Enter

number

of

processes:

"

read

n

arrival\_time=()

burst\_time=()

for

((

i=0;

i<$n;

i++

))

;

do

echo

-

e

"Enter

Arrival

Time

(

AT

)

for

process

$i:

"

read

at

arrival\_time[$i]=$at

echo

-

e

"Enter

Burst

Time

(

BT

)

for

process

$i:

"

read

bt

burst\_time[$i]=$bt

done



sjfScheduling

$n

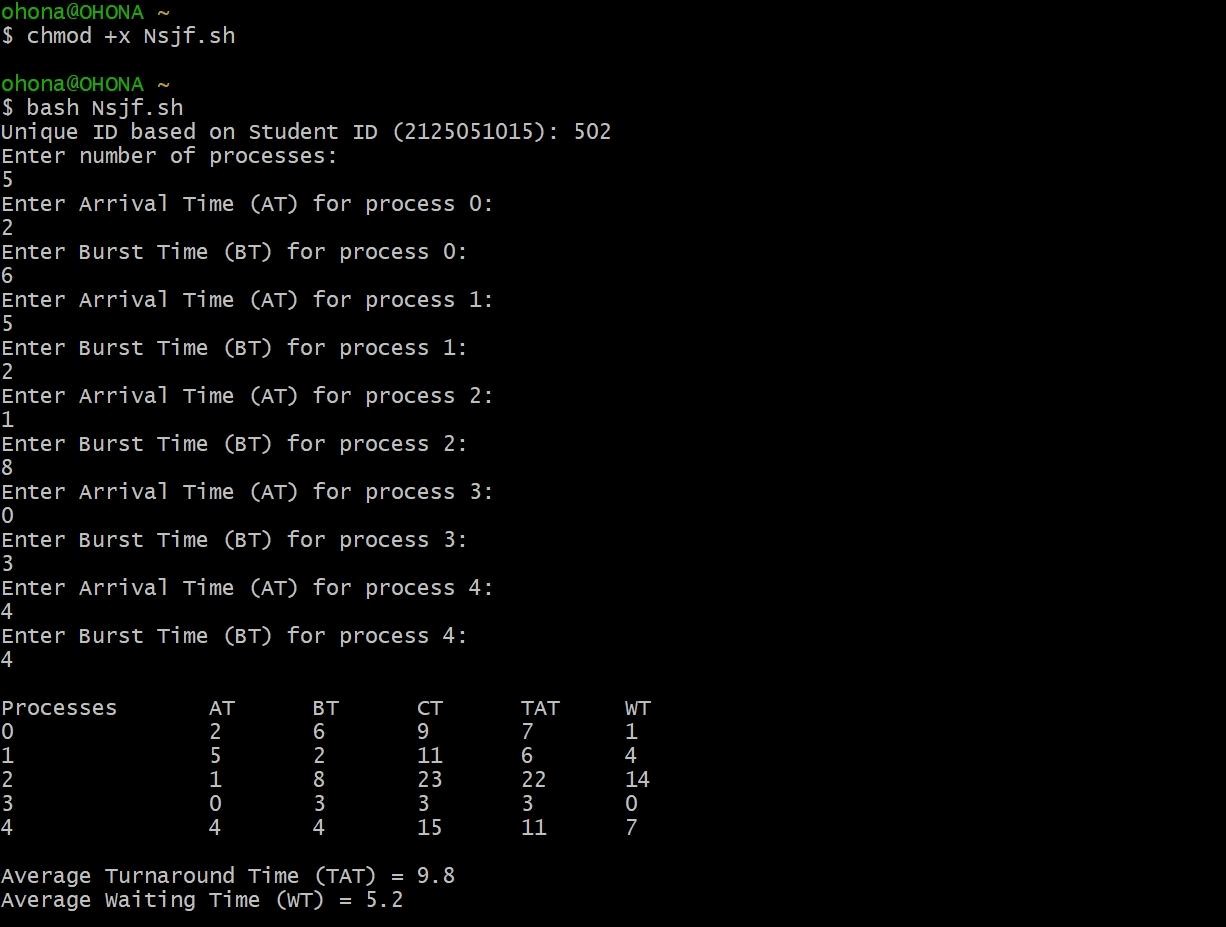
arrival\_time

burst\_time

}

main

**INPUT & OUTPUT -**



**Discussion :**

This lab report demonstrates the implementation of CPU scheduling algorithms—\*\*First-Come,

First-Served (FCFS)\*\*, \*\*Preemptive Shortest Job First (SJF)\*\*, and \*\*Non-Preemptive Shortest Job First (SJF)\*\*—using C++ and Linux Bash scripting. FCFS schedules processes in the order of their arrival without preemption, which is simple but can cause long waiting times due to the "convoy effect." Preemptive SJF improves efficiency by selecting the process with the shortest remaining time, though it may lead to starvation for longer tasks. Non-Preemptive SJF, on the other hand, runs the shortest job to completion without interruptions, offering a balance between simplicity and performance but also facing the issue of potential starvation. Performance metrics like waiting time and turnaround time are evaluated for all algorithms in a Linux environment.

**Conclusion :**

In conclusion, the implementation of FCFS, Preemptive SJF, and Non-Preemptive SJF scheduling algorithms in C++ using Linux Bash scripting highlights the strengths and trade-offs of each approach. FCFS offers simplicity but can result in long waiting times, while Preemptive SJF optimizes CPU efficiency at the risk of starvation for longer processes. Non-Preemptive SJF strikes a balance but still faces the challenge of delaying longer tasks. Overall, these algorithms provide valuable insights into CPU scheduling strategies and their impact on process management.

**Limitations :**

* **Cygwin Requirement :** Running the project on Windows needs Cygwin installation, which adds complexity.
* **Command Complexity :** Implementing FCFS and SJF in Linux involves many commands, challenging for users unfamiliar with Bash.
* **Burst Time Uncertainty :** SJF scheduling requires accurate burst time predictions, which can be hard to estimate.
* **Starvation in SJF :** Both Preemptive and Non-Preemptive SJF may cause longer processes to starve if shorter jobs keep arriving.

**References :**

* 1. **Performance Assessment of CPU Scheduling Algorithms: A Scenario-Based Approach with FCFS, RR, and SJF**. Journal of Computer Science, vol. 20, no. 5, 2024, pp. 972-985. Available: [https://thescipub.com/pdf/jcssp.2024.972.985.pdf](https://thescipub.com/pdf/jcssp.2024.972.985.pdf?formCode=MG0AV3)
  2. **Optimizing Task Scheduling in Heterogeneous Computing Environments: A Comparative Analysis of CPU, GPU, and ASIC Platforms Using E2C Simulator**. arXiv preprint arXiv:2405.08187v1, May 2024. Available:

[https://arxiv.org/html/2405.08187v1](https://arxiv.org/html/2405.08187v1?formCode=MG0AV3)

* 1. **GeeksforGeeks - Shortest Job First (SJF) CPU Scheduling**. GeeksforGeeks.

Available:

[https://www.geeksforgeeks.org/program-for-shortest-job-first-or-sjf-cpu-scheduling-set-1](https://www.geeksforgeeks.org/program-for-shortest-job-first-or-sjf-cpu-scheduling-set-1-non-preemptive/?formCode=MG0AV3)

[-non-preemptive](https://www.geeksforgeeks.org/program-for-shortest-job-first-or-sjf-cpu-scheduling-set-1-non-preemptive/?formCode=MG0AV3)