CHITTAGONG UNIVERSITY OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

NAME OF THE ASSIGNMENT

Implement existing CPU scheduling algorithms and design a new algorithm

Course Code : CSE-336

Course Title : Operating System (Sessional)

Date of submission: 25/06/24

REMARK

Submitted by:

1904119 : Shahriar Rashid Tonmoy

2004096 : Istiaque Ahmed Nizami

2004097 : Atikia Faiza Aska

2004098 : Md. Nur Siddik Ruman

2004099 : Md. Safayat Bin Nasir

Level – 3 Term – I

Section - B Group - B1

Experiment Name:

A program by implementing a novel CPU scheduling algorithm and conduct a comparative study with the existing CPU scheduling algorithms.

Problem Description:

In this lab assignment, we have to implement all the CPU scheduling algorithms discussed during your theory class. Then, we have to propose and implement our own CPU scheduling algorithm. Finally, we have to conduct a comparative study among different scheduling algorithms.

List of scheduling algorithms we need to implement-

1. First Come First Serve (FCFS) Scheduling Algorithm

2. Non-Preemptive Shortest Job First (SJF) Scheduling Algorithm

3. Preemptive Shortest Job First (SJF) Scheduling Algorithm

4. Non-Preemptive Priority Scheduling Algorithm

5. Preemptive Priority Scheduling Algorithm

6. Round Robin Scheduling Algorithm

7. Our Own Scheduling Algorithm

To conduct a comparative study among different scheduling algorithms, we have to measure the following evaluation metrics for each of the algorithms.

1. Average Response Time

2. Average Waiting Time

3. Average Turnaround Time

Objectives:

1. To know about various CPU scheduling algorithms.

2. To propose and implement our own CPU scheduling algorithm.

3. To compare different CPU scheduling algorithms.

Theory:

CPU scheduling is the basis of multi-programmed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

Scheduling of is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

The success of CPU scheduling depends on an observed property of processes: process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the CPU scheduler, which selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Preemptive and Non-preemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances:

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)

2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)

3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)

4. When a process terminates

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is non preemptive or cooperative. Otherwise, it is preemptive. Under non preemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state. Virtually all modern operating systems including Windows, macOS, Linux, and UNIX use preemptive scheduling algorithms.

Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes.

Another component involved in the CPU-scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU's core to the process selected by the CPU scheduler.

The dispatcher should be as fast as possible, since it is invoked during every context switch. The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

• **CPU utilization:** We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system). (CPU utilization can be obtained by using the top command on Linux, macOS, and UNIX systems.)

• **Throughput:** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process over several seconds; for short transactions, it may be tens of processes per second.

• **Turnaround time:** From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process

to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O.

• **Waiting time**: The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

• **Response time:** In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure. However, under some circumstances, we prefer to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

Investigators have suggested that, for interactive systems (such as a PC desktop or laptop system), it is more important to minimize the variance in the response time than to minimize the average response time. A system with reasonable and predictable response time may be considered more desirable than a system that is faster on the average but is highly variable. However, little work has been done on CPU-scheduling algorithms that minimize variance.

Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU's core. There are many different CPU scheduling algorithms.

List of scheduling algorithms we implemented:

1. First Come First Serve (FCFS) Scheduling Algorithm:

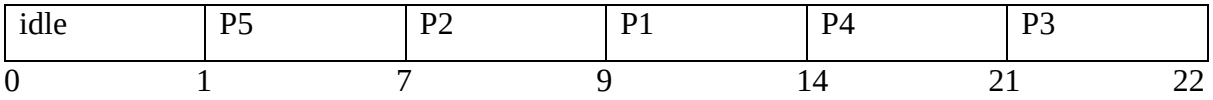
By far the simplest CPU-scheduling algorithm is the first-come first-serve (FCFS) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is

allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand. On the negative side, the average waiting time under the FCFS policy is often quite long.

Example: We have used the following data table to execute:

Process	Burst Time	Arrival Time
P1	5	3
P2	2	2
P3	1	5
P4	7	4
P5	6	1

Gantt chart representation of FCFS CPU scheduling algorithm is given below:



Process	Start Time	End Time	Response Time	Waiting Time	Turnaround Time
P1	9	14	6	6	11
P2	7	9	5	5	7
P3	21	22	16	16	17
P4	14	21	10	10	17
P5	1	7	0	0	6

Average Response Time: 7.4

Average Waiting Time: 7.4

Average Turnaround Time: 11.6

2. Non-Preemptive Shortest Job First (SJF) Scheduling Algorithm:

The Non-Preemptive Shortest Job First (SJF) algorithm is a CPU scheduling algorithm that selects the process with the shortest burst time for execution. The non-preemptive SJF algorithm allows a process to run until completion without interruption or preemption. It's important to note that accurately predicting the burst time of processes in practice can be challenging, and inaccurate predictions may lead to suboptimal scheduling decisions. Additionally, the non-preemptive nature of this algorithm can lead to inefficiencies if shorter processes are followed by longer processes, causing the longer processes to wait longer.

Example: We have used the following data table to execute:

Process	Burst Time	Arrival Time
P1	5	3
P2	2	2
P3	1	5
P4	7	4
P5	6	1

Gantt chart representation of non-preemptive SJF CPU scheduling algorithm is given below:

idle	P5	P3	P2	P1	P4

0 1 7 8 10 15 22

Process	Start Time	End Time	Response Time	Waiting Time	Turnaround Time
P1	10	15	7	7	12
P2	8	10	6	6	8
P3	7	8	2	2	3
P4	15	22	11	11	18
P5	1	7	0	0	6

Average Response Time: 5.2

Average Waiting Time: 5.2

Average Turnaround Time: 9.4

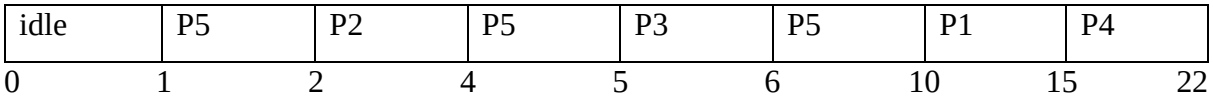
3. Preemptive Shortest Job First (SJF) Scheduling Algorithm:

The Preemptive Shortest Job First (SJF) algorithm, also known as Shortest Remaining Time First (SRTF), is a CPU scheduling algorithm that selects the process with the shortest remaining burst time for execution. Unlike the non-preemptive SJF algorithm, the preemptive SJF algorithm allows processes to be interrupted or preempted if a new process with a shorter burst time arrives. It's important to note that accurately estimating the remaining burst time in practice can be challenging, and inaccurate estimates may lead to inefficient scheduling decisions. Additionally, the Preemptive SJF algorithm may suffer from starvation if very long processes continuously arrive, as shorter processes may never get a chance to execute.

Example: We have used the following data table to execute:

Process	Burst Time	Arrival Time
P1	5	3
P2	2	2
P3	1	5
P4	7	4
P5	6	1

Gantt chart representation of preemptive SJF CPU scheduling algorithm is given below:



Process	Start Time	End Time	Response Time	Waiting Time	Turnaround Time
P1	10	15	7	7	12
P2	2	4	0	0	2
P3	5	6	0	0	1
P4	15	22	11	11	18
P5	1	10	0	3	9

Average Response Time: 3.6

Average Waiting Time: 4.2

Average Turnaround Time: 8.4

4. Non-Preemptive Priority Scheduling Algorithm:

Preemptive Priority Scheduling is a CPU scheduling algorithm where each process is assigned a priority value, and the process with the highest priority is selected for execution. However, unlike non-preemptive priority scheduling, in the preemptive version, a running process can be preempted or interrupted by a higher-priority process. It's important to note that careful priority management is essential to avoid starvation, where lower-priority processes are consistently neglected. Priorities can be adjusted dynamically to maintain fairness and ensure that processes with lower priorities eventually get a chance to execute.

Example: We have used the following data table to execute:

Process	Burst Time	Arrival Time	Priority
P1	5	3	3
P2	2	2	1
P3	1	5	2
P4	7	4	1
P5	6	1	3

Gantt chart representation of non-preemptive priority CPU scheduling algorithm is given below:

idle	P5	P2	P4	P3	P1

0 1 7 9 16 17 22

Process	Start Time	End Time	Response Time	Waiting Time	Turnaround Time
P1	17	22	14	14	19
P2	7	9	5	5	7
P3	16	17	11	11	12
P4	9	16	5	5	12
P5	1	7	0	0	6

Average Response Time: 7

Average Waiting Time: 7

Average Turnaround Time: 11.2

5. Preemptive Priority Scheduling Algorithm:

Non-Preemptive Priority Scheduling is a CPU scheduling algorithm where each process is assigned a priority value, and the process with the highest priority is selected for execution. In the non-preemptive version, once a process starts running, it continues until completion without being interrupted or preempted by other processes. It's important to note that non-preemptive priority scheduling can suffer from the problem of indefinite blocking or starvation, where lower-priority processes may never get a chance to execute if higher-priority processes keep arriving. Techniques like aging or priority aging can be employed to gradually increase the priority of waiting processes, ensuring fairness in CPU allocation over time.

Example: We have used the following data table to execute:

Process	Burst Time	Arrival Time	Priority
P1	5	3	3
P2	2	2	1
P3	1	5	2
P4	7	4	1
P5	6	1	3

Gantt chart representation of preemptive priority CPU scheduling algorithm is given below:

idle	P5	P2	P4	P3	P5	P1

0 1 2 4 11 12 17 22

Process	Start Time	End Time	Response Time	Waiting Time	Turnaround Time
P1	17	22	14	14	19
P2	2	4	0	0	2
P3	11	12	6	6	7
P4	4	11	0	0	7
P5	1	17	0	10	16

Average Response Time: 4

Average Waiting Time: 6

Average Turnaround Time: 10.2

6. Round Robin Scheduling Algorithm:

The round-robin (RR) scheduling algorithm is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

Example: We have used the following data table to execute:

Process	Burst Time	Arrival Time
P1	5	3
P2	2	2
P3	1	5
P4	7	4
P5	6	1

Time Quantum = 4

Gantt chart representation of Round Robin CPU scheduling algorithm is given below:

idle	P5	P1	P2	P3	P4	P5	P1	P4

0 1 5 9 11 12 16 18 19 22

Process	**Start Time**	**End Time**	**Response Time**	**Waiting Time**	**Turnaround Time**
P1	5	19	2	11	16
P2	9	1	7	7	9
P3	11	12	6	6	7
P4	12	22	8	11	18
P5	1	18	0	11	17

Average Response Time: 4.6

Average Waiting Time: 9.2

Average Turnaround Time: 13.4

7. Our Own Scheduling Algorithm:

The function we provided, Own_algo(int ch), appears to implement a scheduling algorithm for processes. Here's a breakdown of what the function does:

Inputs:

- ch: An integer that likely determines the mode of operation. (e.g., ch=8 might be a special mode)

Process Data Structures:

- n: Number of processes
- a[n]: Arrival time of each process
- b[n]: Burst time of each process (CPU execution time)
- ct[n]: Completion time of each process
- tat[n]: Turnaround time of each process (completion time - arrival time)
- wt[n]: Waiting time of each process (turnaround time - burst time)

- rt[n]: Response time of each process (time from submission to start of execution)
- p[n]: Priority of each process (might not be used in this implementation)
- v: A vector to store the order of process execution

Algorithm Steps:

1. **Get the number of processes (n):**
    - If ch is not 8, it prompts the user to enter the number of processes and stores it in n.
    - If ch is 8, it uses a predefined value np for the number of processes.
2. **Initialize process data structures:**
    - Creates arrays a, b, ct, tat, wt, rt, and p of size n to store process data.
3. **Get process data (if ch is not 8):**
    - Prompts the user to enter arrival time and burst time for each process and stores them in a, b respectively.
4. **OR Load process data from predefined variables (if ch is 8):**
    - If ch is 8, it assumes art, bt, and pri (priority) arrays contain arrival time, burst time, and priority of processes and copies them to a, b, and p respectively.
5. **Priority Queue and Variables:**
    - Creates a priority queue pq to store process IDs and a weighted value for scheduling.
    - Initializes variables t (current time), pr (process flags to track if a process has arrived), st (starting time of each process).
    - Initializes variables art (average turnaround time), atat (average waiting time), and awt (average response time) - likely for later calculations.
6. **Scheduling Loop:**
    - Runs a loop until there are no more processes to schedule.
    - Inside the loop:
        - Initializes mn (minimum arrival time) and pp (process found flag) to -1 and 0 respectively.
        - Iterates through all processes:
            - If a process is not yet arrived (pr[i] == 0) and its arrival time is less than or equal to the current time:

        - Sets its process flag (pr[i]) to 1, sets pp to 1, and pushes a pair containing a weighted value (average of arrival and burst time * arrival time) and the process ID to the priority queue pq.
      - Otherwise, if mn is not set yet or the current process's arrival time is less than mn, update mn with the current process's arrival time.
    - Checks if no process has arrived yet (pp == 0 and mn == -1):
      - If true, it means there are no more processes to schedule, and the loop breaks.
    - If the priority queue is empty:
      - Sets the current time t to the minimum arrival time mn.
    - Otherwise, it dequeues the process with the highest weighted value from the priority queue and stores its ID in in and burst time in d.
      - Sets the starting time (st) and completion time (ct) for the process.
      - Updates the current time t by adding the burst time.
7. **Process any remaining processes in the queue:**
   - After the loop exits, it dequeues any remaining processes from the priority queue and calculates their starting and completion times.
8. **Output (if ch is 8):**
   - If ch is 8, it prints a message indicating the algorithm used ("Own-Algorithm").
9. **Call output function (not shown):**
   - Calls a function output_np (not shown in the provided code) to likely output the scheduling results like process IDs, completion.

Pseudocode:

Function Own_algo(ch)

// Input processing

if ch ≠ 8 then

Print "****Chosen Own Algorithm****"

Read n // number of processes

else

n = np // predefined number of processes (assuming np is defined elsewhere)

// Initialize data structures (arrays)

a[1..n] = arrival time of each process

b[1..n] = burst time of each process

ct[1..n] = completion time (to be calculated)

tat[1..n] = turnaround time (to be calculated)

wt[1..n] = waiting time (to be calculated)

rt[1..n] = response time (to be calculated)

p[1..n] = priority (potentially unused in this implementation)

pr[1..n] = process arrived flag (0 - not arrived, 1 - arrived)

st[1..n] = starting time of each process (to be calculated)

v = empty vector (to store process execution order)

t = 0 // current time

art = 0 // average turnaround time (initially 0)

atat = 0 // average waiting time (initially 0)

awt = 0 // average response time (initially 0)

// Get process data (if ch ≠ 8)

if ch ≠ 8 then

for i = 1 to n do

Read a[i] and b[i] // arrival and burst time from user input

else

// Load data from predefined variables (if ch = 8)

for i = 1 to n do

```
a[i] = art[i] // arrival time from predefined array

b[i] = bt[i] // burst time from predefined array

p[i] = pri[i] // priority from predefined array (potentially unused)

// Priority queue for scheduling

pq = empty priority queue (ordered by a function of arrival and burst time)

while True do

mn = -1 // minimum arrival time seen so far

pp = 0 // flag to indicate if a process was found for scheduling

// Check for arrived processes

for i = 1 to n do

if pr[i] == 0 and a[i] <= t then // process not arrived yet and arrived at current time

pr[i] = 1 // set process arrived flag

pp = 1 // set found flag

pq.push({-(a[i] + b[i]) * a[i] / 2, i}) // push process ID with weighted value to queue

else if mn == -1 or a[i] < mn then // update minimum arrival time

mn = a[i]

end for

// No more processes to schedule

if pp == 0 and mn == -1 then

break // exit the loop

end if

// Handle empty queue case (use minimum arrival time as current time)

if pq.isEmpty() then

t = mn
```

```
else
// Dequeue process with highest weighted value from priority queue
top_process = pq.pop()
in = top_process.second // process ID
d = b[in] // burst time
// Update process data
st[in] = t // starting time
v.push_back(in) // add process ID to execution order vector
ct[in] = t + d // completion time
t = t + d // update current time
end if
end while
// Process any remaining processes in the queue
while not pq.isEmpty() do
top_process = pq.pop()
in = top_process.second
d = b[in]
v.push_back(in)
st[in] = t
ct[in] = t + d
t = t + d
end while
// Output (if ch = 8)
if ch == 8 then
```

Print "Algo: Own-Algorithm"

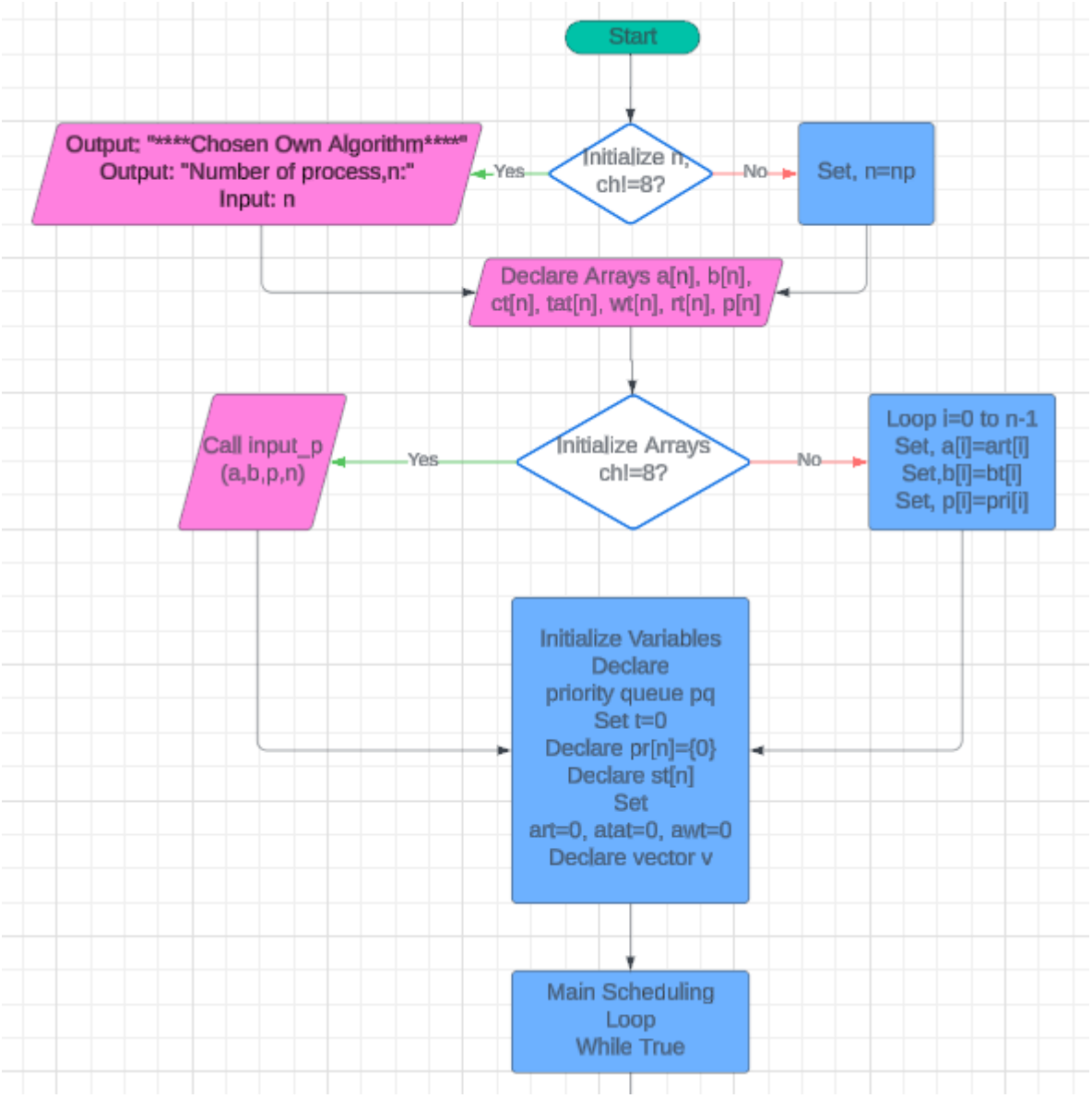
end if

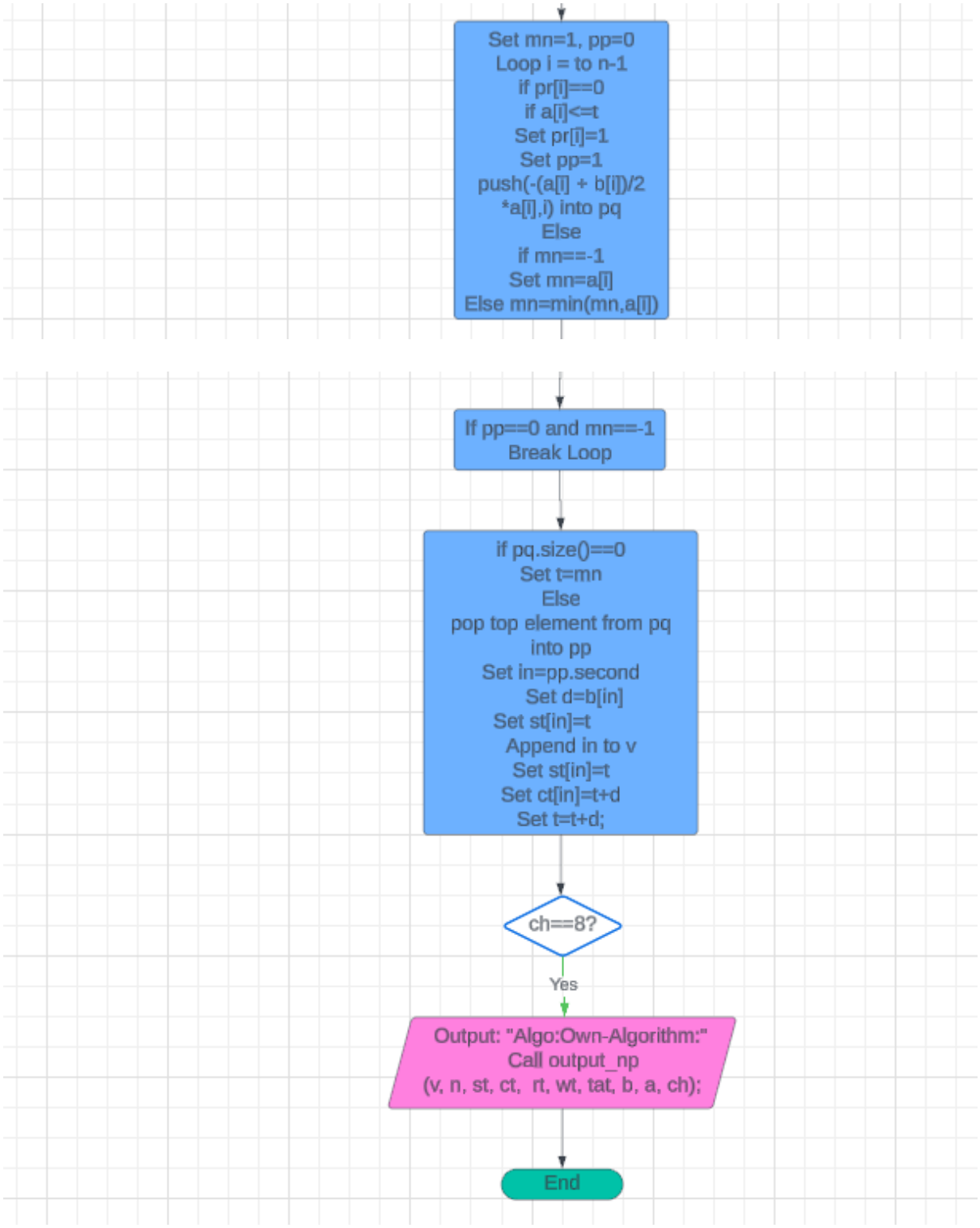
// Call output function (to be implemented elsewhere)

output_np(v, n, st, ct, rt, wt, tat, b, a, ch)

End Function

Flow-Chart:

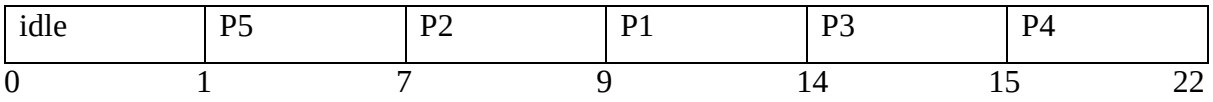



Set mn=1, pp=0
Loop i = to n-1
if pr[i]==0
if a[i]<=t
Set pr[i]=1
Set pp=1
push(-(a[i] + b[i])/2
*a[i],i) into pq
Else
if mn==-1
Set mn=a[i]
Else mn=min(mn,a[i])
If pp==0 and mn==-1
Break Loop
if pq.size()==0
Set t=mn
Else
pop top element from pq
into pp
Set in=pp.second
Set d=b[in]
Set st[in]=t
Append in to v
Set st[in]=t
Set ct[in]=t+d
Set t=t+d;
ch==8?
Yes
Output: "Algo:Own-Algorithm:"
Call output_np
(v, n, st, ct, rt, wt, tat, b, a, ch);
End

Example: We have used the following data table to execute:

Process	Burst Time	Arrival Time	Priority
P1	5	3	3
P2	2	2	1
P3	1	5	2
P4	7	4	1
P5	6	1	3

Gantt chart representation of our own CPU scheduling algorithm is given below:



Process	Start Time	End Time	Response Time	Waiting Time	Turnaround Time
P1	9	14	6	6	11
P2	7	9	5	5	7
P3	14	15	9	9	10
P4	15	22	11	11	18
P5	1	7	0	0	6

Average Response Time: 6.2

Average Waiting Time: 6.2

Average Turnaround Time: 10.4

Comparison:

Algorithms	Average Response Time	Average Waiting Time	Average Turnaround Time
FCFS	7.4	7.4	11.6
Non-Preemptive SJF	5.2	5.2	9.4
Preemptive SJF	3.6	4.2	8.4
Non-Preemptive Priority	7	7	11.2
Preemptive Priority	4	6	10.2
Round Robin	4.6	9.2	13.4
Own Algorithm	6.2	6.2	10.4

Table: Comparison between existing algorithms and our own algorithm

Conclusion:

After completing this assignment, we come to know about various CPU scheduling algorithms and implemented them. We also have compared their average response time, average turnaround time and average waiting time. In priority scheduling if two processes have the same priority then it follows the FCFS algorithm. In our own algorithm we used a formula for priority queue. Though we have faced some difficulties while designing our own algorithm, eventually we have completed our own algorithm. In conclusion, the choice of CPU scheduling algorithm should be based on the specific requirements of the system and the desired trade-offs between fairness, response time, turnaround time, waiting time, throughput and overhead.