Part I: Pipeline Processor

Consider the following program:

```
program pipeline1
        x=10
        y=20
        z=30
        y=z – y
        y=z + y
        z=z-x
        z=z+1
end
```

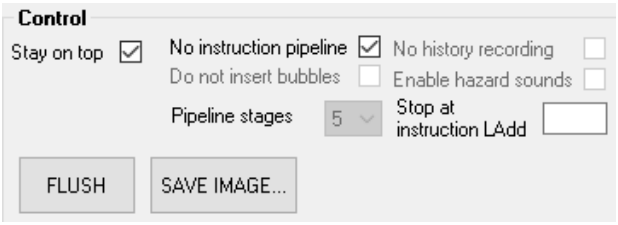
Compile the code and load it in CPU-OS simulator. Perform the following:

Execute the above program using non-pipelined processor and pipelined processor and answer the following questions.

Note: Every time flush the pipeline before running the code

A) Non-pipelined Processor:

To enable non-pipelined processor, check "No instruction pipeline" check box in control panel.



a) How many stages are there in non-pipelined processor? List them.

Solution:

5 stages

- Fetch
- Decode
- Read Operands
- Execute
- Write Result

b) Fill in the following after executing of above program using non-pipelined processor.

	Clocks	Instruction Count	CPI	Speed up Factor
Non-Pipelined Processor	**127**	**23**	**5.52**	**0.91**

c) What are the contents of General-purpose registers after the execution of the program?

Solution:
R00 – 0
R01 – 10
R02 – 40

R03 to R05 - 21
R06 to R31 - 0

B) Pipelined processor:

To use, enable pipelined processor, uncheck "No instruction pipeline" check box in control panel.

a) Fill in the following table with respect to pipelined processor execution of the above program:

Pipelined processor conditions	**Clocks**	**Instruction Count**	**CPI**	**Speed up Factor**	**Data hazard (Yes/No)**	**Contents of registers used by the program**
Check "Do not insert bubbles" check box	**39**	**23**	**1.7**	**2.94**	**No (0)**	R00 – 0 R01 – 0 R02 – 20 R03 - 50 R04 to R05 – 51 R06 to R31 - 0
Uncheck "Do not insert bubbles"	**46**	**23**	**2**	**2.5**	**Yes (17)**	R00 – 0 R01 – 10 R02 – 40 R03 to R05 - 21 R06 to R31 - 0

b) Is there a way to improve the CPI and Speed up factor? If so give the solution.

Solution:

From the above observation, inserting bubbles improving the CPI but decreasing the speed up factor and vice versa.

Some common ways to improve the CPI and speed up factor are:

1) Dynamic scheduling:
2) Multi-threading
3) Increasing pipeline stages
4) Use larger instruction window
5) Implement branch prediction

Part II: Process Scheduling

Consider the following source codes:

```
program My_Pgm
        read(i)
        for n = 1 to 10
                x = i + n
        next
end
```

Compile the above source code and load it in the main memory.

We are now going to use the OS simulator to run this code. To enter the OS simulator:

1) Click on the OS O… button in the current window. The OS window opens.
2) You should see an entry, titled LoopTest, in the PROGRAM LIST view.
3) Now that this program is available to the OS simulator, we can create as many instances, i.e. processes, of it as we like. You do this by clicking on the CREATE NEW PROCESS button.

PART-II_A

- Select the **First-Come-First-Served (FCFS)** option in the SCHEDULER/Policies view
- Time slice should be considered as **seconds.**
- Create four processes P1, P2, P3 and P4 from source code respectively (Use the Priority drop-down list in the PROGRAM LIST / Process View): **3, 2, 4,1**
- Slide the Speed selector half-way down and then hit the START button.
- **Arrival delay** should be considered in **seconds** in the OS simulator

Now, give answer for the following:

a) What is the order in which processes are executed?

P1 → P2 → P3 → P4

Avg. Process Waiting Time = 83.6 sec

b) What is the ***Elapsed time , Average Process Waiting Time*** and ***Average Burst Period*** and of each process? (To see this, Click on VIEWS button available on the left of your OS control, the click VIEW LOG)

Process	*Arrival Time/Delay*	*Elapsed Time (sec)*	*Average Process Waiting Time (sec)*	*Average Burst Period*
P1	0	127.758	0.26	~~44~~
P2	3	235.428	55.58	~~44~~
P3	4	343.610	110.85	~~44~~
P4	6	451.778	165.45	~~44~~

PART-II_B

- Select the **Shortest Job First (SJF)** option in the SCHEDULER/Policies view
- Select the Priority (static) as **Pre-emptive** option in the SCHEDULER/Policies view
- Time slice should be considered as **seconds.**
- Create four processes P1, P2, P3 and P4 from source codes respectively (Use the Priority drop-down list in the PROGRAM LIST / Process View): **3, 2, 4**,**1**
- Slide the Speed selector half-way down and then hit the START button.
- **Arrival delay** should be considered in **seconds** in the OS simulator

Now, give answer for the following:

a) What is the order in which processes are executed?

P1 → P2 → P3 → P4

Avg. Process Waiting Time = 81.76 sec

b) What is the ***Elapsed time , Average Process Waiting Time*** and ***Average Burst Period*** and of each process? (To see this, Click on VIEWS button available on the left of your OS control, the click VIEW LOG)

Process	***Arrival Time/Delay***	***Elapsed Time (sec)***	***Average Process Waiting Time (sec)***	***Average Burst Period***
P1	0	123.995	0.26	44
P2	3	231.657	53.14	44
P3	4	339.845	108.19	44
P4	6	447.773	163.06	44

PART-II_C

- Select the **Round Robin (RR) with 5 seconds as time slice** option in the SCHEDULER/Policies view.
- Select the Priority (static) as **Pre-emptive** option in the SCHEDULER/Policies view
- Time slice should be taken in terms of **seconds** instead of **ticks**
- Create four processes P1, P2, P3 and P4 from source codes respectively (Use the Priority drop-down list in the PROGRAM LIST / Process View): **3, 2, 4**,**1**
- Slide the Speed selector half-way down and then hit the START button.
- **Arrival delay** should be considered in **seconds** in the OS simulator

Now, give answer for the following:

a) What is the order in which processes are executed?

P4 → P2 → P1 → P3

Avg. Process Waiting Time = 41.91 sec

b) What is the ***Elapsed time , Average Process Waiting Time*** and ***Average Burst Period*** and of each process? (To see this, Click on VIEWS button available on the left of your OS control, the click VIEW LOG)

Process	***Arrival Time/Delay***	***Elapsed Time (sec)***	***Average Process Waiting Time (sec)***	***Average Burst Period***
P1	0	156.701	7.37	7
P2	3	104.191	3.47	6
P3	4	198.680	70.08	44
P4	6	59.194	0.09	8

PART-II_D

a) Plot a graph from the results obtained by FCFS, SJF and Round Robin scheduling and explain which algorithm is better among these with proper justification

To compare the three scheduling algorithms, we need to analyse the average process waiting time, which is a critical factor to measure the efficiency of the scheduling algorithm. The process waiting time is the amount of time a process spends waiting in the ready queue.

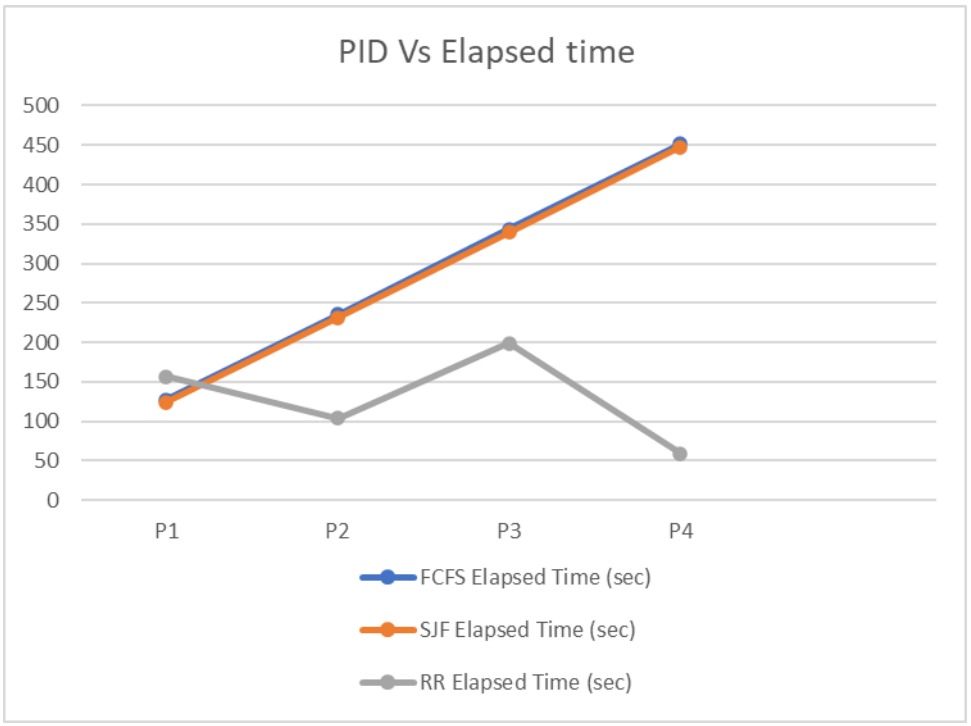
From the results obtained, we can see that the Round Robin algorithm has the lowest average process waiting time of 41.91 seconds, followed by SJF with 81.76 seconds, and FCFS with 83.6 seconds. This result shows that the Round Robin scheduling algorithm is the best among the three.

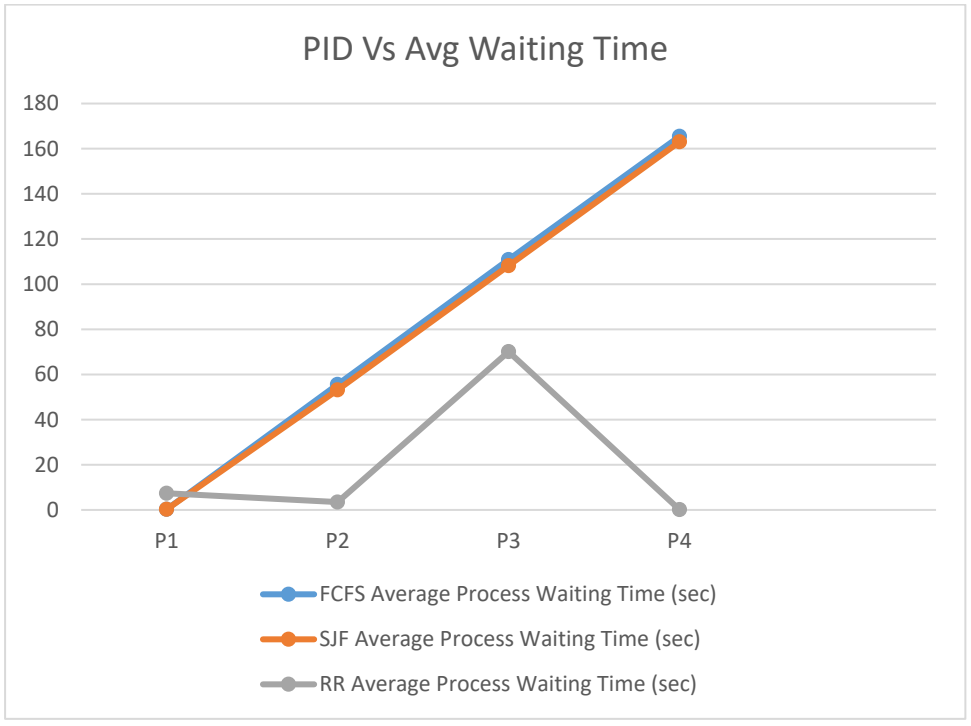
The FCFS algorithm has a higher average process waiting time than SJF and Round Robin, indicating that it is not an efficient algorithm, especially when there are long-running processes. The FCFS algorithm suffers from the convoy effect, where long processes hold up short ones, leading to a higher average process waiting time.

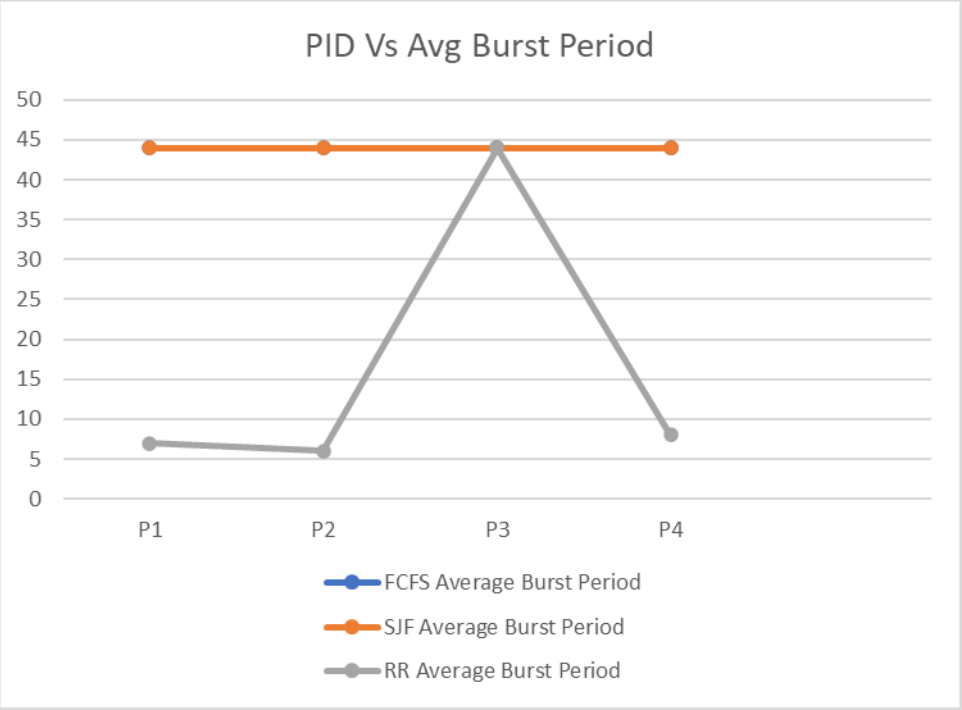
The SJF algorithm has a lower average process waiting time than FCFS, indicating it is a better algorithm. The SJF algorithm selects the process with the shortest burst time, resulting in faster completion of shorter processes. However, SJF may not be the best algorithm in all cases. It may lead to starvation, where a process with a long burst time never gets a chance to run.

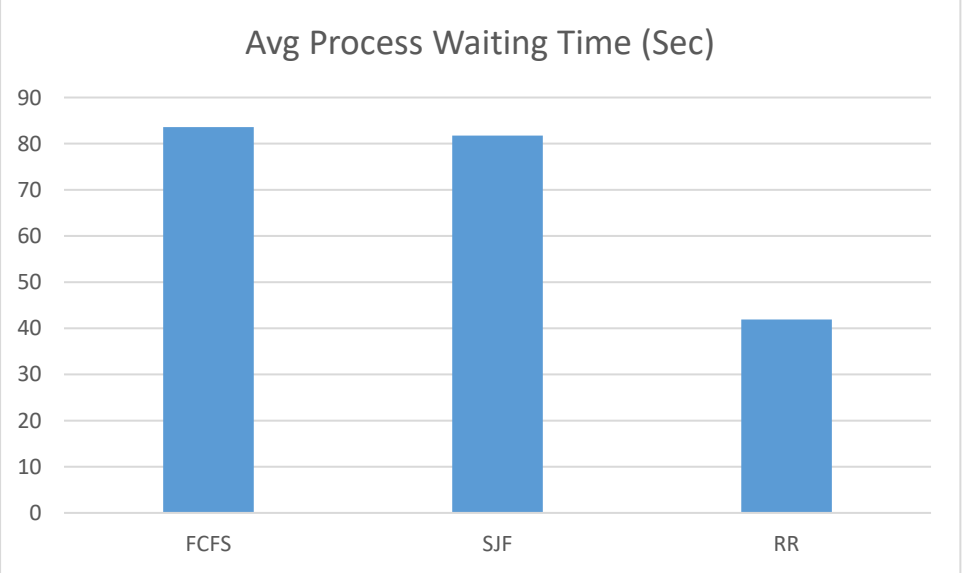
The Round Robin algorithm divides the CPU time into small slices and allocates each process a fixed time slice. If the process has not completed its execution in the allocated time slice, it is pre-empted and sent back to the ready queue. Round Robin provides fair CPU time allocation to all processes, and hence, the average process waiting time is the lowest among the three algorithms. However, Round Robin has its disadvantages. It may lead to a higher context switching overhead if the time slice is too small.

Therefore, based on the results obtained, we can conclude that the Round Robin algorithm is the best algorithm among the three. It provides fair CPU time allocation, a lower average process waiting time, and avoids the convoy effect and starvation.


PID Vs Elapsed time
500
450
400
350
300
250
200
150
100
50
0
P1
P2
P3
P4
FCFS Elapsed Time (sec)
SJF Elapsed Time (sec)
RR Elapsed Time (sec)


PID Vs Avg Waiting Time
180
160
140
120
100
80
60
40
20
0
P1
P2
P3
P4
FCFS Average Process Waiting Time (sec)
SJF Average Process Waiting Time (sec)
RR Average Process Waiting Time (sec)


PID Vs Avg Burst Period
50
45
40
35
30
25
20
15
10
5
0
P1
P2
P3
P4
FCFS Average Burst Period
SJF Average Burst Period
RR Average Burst Period


Avg Process Waiting Time (Sec)
90
80
70
60
50
40
30
20
10
0
FCFS
SJF
RR

Part III: Multi-Threading

Consider the following source code

```
program ThreadTest
        total = 0
        sub thread1 as thread
            for i = 1 to 10
                total = total + i
            next
        end sub
        sub thread2 as thread
            for i = 11 to 20
                call thread1
                total = total * i
            next
        end sub
        sub thread3 as thread
            for i = 21 to 30
                call thread2
                total = total * i
            next
        end sub
        call thread3
        wait
        writeln (“Total =”, total)
end
```

Compile the above source code and load it in the main memory. Create a single process, choose RR scheduling algorithm with time quantum of 5 seconds with no-priority. Run the Process.

Answer the following questions:

a) What is the value of “Total” ?

The value of "Total" would be 0, as the program is designed to print the final value of "total" after all the threads have finished executing. However, the threads are designed to update the value of "total" independently, without waiting for other threads to complete. This can lead to race conditions and unpredictable results.

b) How many processes and how many threads are created?

There is one process created, which spawns three threads.

c) Identify the name of the processes and threads.

P1 and T1, T2, T3

d) What is the PID and PPID of the processes and threads created?

Since there is only one process, it will have a PID (Process ID), here it is P1, but no PPID (Parent Process ID). Each thread will have a TID (Thread ID) and will inherit the PID of the process.

e) Represent the parent and child relationship using tree representation

Here, the process "ThreadTest" is the parent of thread3, which in turn is the parent of thread2, which is the parent of thread1. All the threads share the same address space as the parent process.

ThreadTest(Process)
|
thread3(Thread)
|
thread2(Thread)
|
thread1(Thread)