SYSC 4001 Assignment #2

Maxim Creanga 101298069
Braedy Gold-Conlin

Part I – Concepts [1.5 marks]

Answer the following questions. Justify your answers. Show all your work.

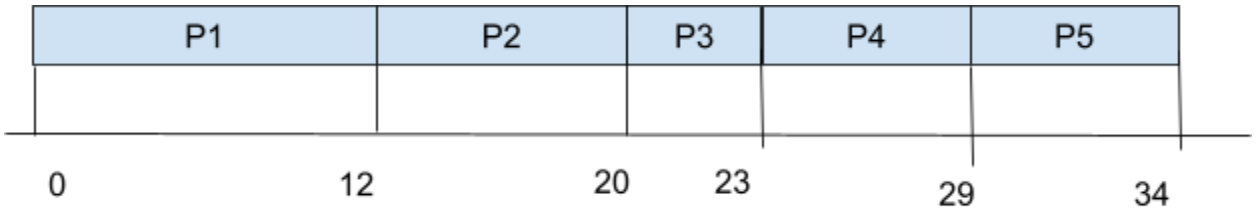
a) [0.5 mark] Consider the following set of processes. Each process has a single CPU burst and does not perform any I/O.

Process	Arrival Time (ms)	Execution Time (ms)
P1	0	12
P2	5	8
P3	8	3
P4	15	6
P5	20	5

With the help of Gantt charts, draw the execution timeline for the following scheduling algorithms:
(i) FCFS (First Come First Serve). [Student 1: submit a separate document explaining the FCFS algorithm]

For each scheduling algorithm:

1. Draw the Gantt chart showing the execution timeline
2. Calculate the completion time for each process
3. Calculate the turnaround time for each process
4. Calculate the mean turnaround time of all the processes

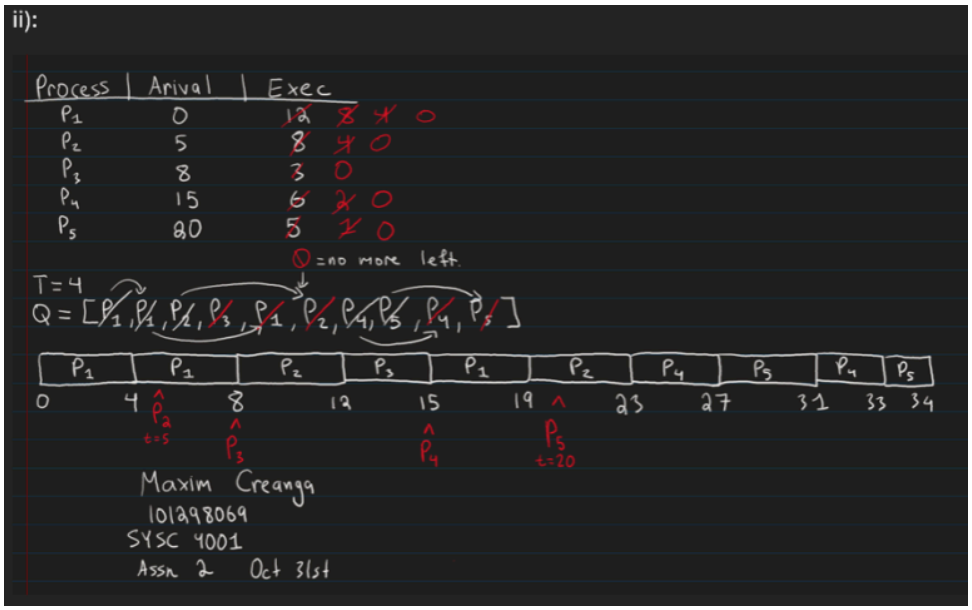


Completion time for P1 = 12
Completion time for P2 = 20
Completion time for P3 = 23
Completion time for P4 = 29
Completion time for P5 = 34

Turnaround time for P1: 12
Turnaround time for P2: 8
Turnaround for P3: 3
Turnaround time for P4: 5
Turnaround time for P5: 5

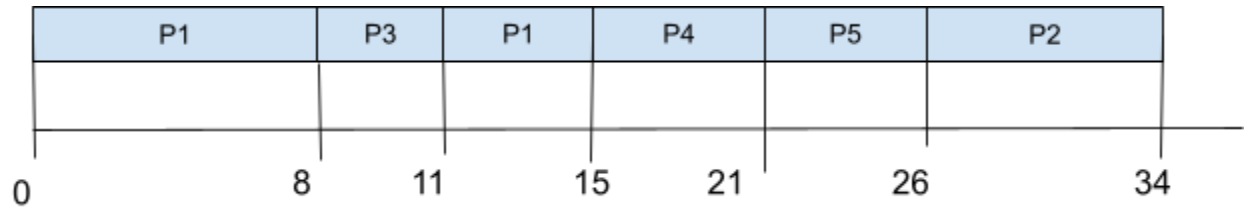
Mean turnaround time = 6.6

(ii) Round Robin (time slice of 4 ms). [Student 2: submit a separate document explaining the RR algorithm]



(taken from individual submission - maxim creanga) The round-robin scheduling algorithm is an algorithm which aims to equally give each
process some time to execute, which is T = 4ms in the example. This eliminates the problem of having very large task take up a super long amount of time, if they came first, like in FCFS, as smaller processes arriving slightly later would be delayed by a tremendous amount of time, which is not fair to those processes. However, by giving each process in the queue a time-slice before kicking them out, the average waiting time and the turnaround time for these processes are going to be quite high, as they are all going to end towards the end of the CPU burst. However, this algorithm aims to provide fairness, and eliminates the convoy effect in FCFS scheduling, as well as starvation of larger processes in algorithms which favor smaller processes, while kicking out said bigger processes, like in SJF or its preemptive version.

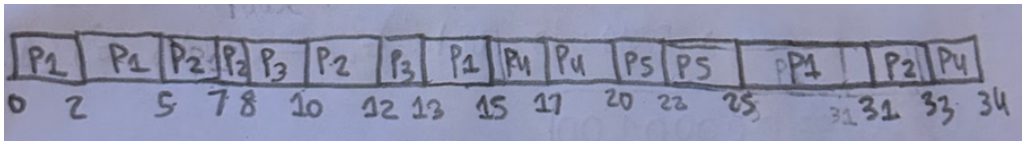
(iii) Shortest Job First with preemption



Turnaround time for P1: 15
Turnaround time for P2: 29
Turnaround time for P3:3
Turnaround time for P4: 6
Turn around time for P5: 6
Mean turn around time = 11.8

Completion times can be seen above, they don't need to be calculated as they are shown in the diagram.

(iv) Multiple queues with feedback (high-priority queue: quantum = 2; mid-priority queue: quantum = 3; low-priority queue: FIFO)



Do turnaround time later (this is more complicated and I have no idea if it is right)

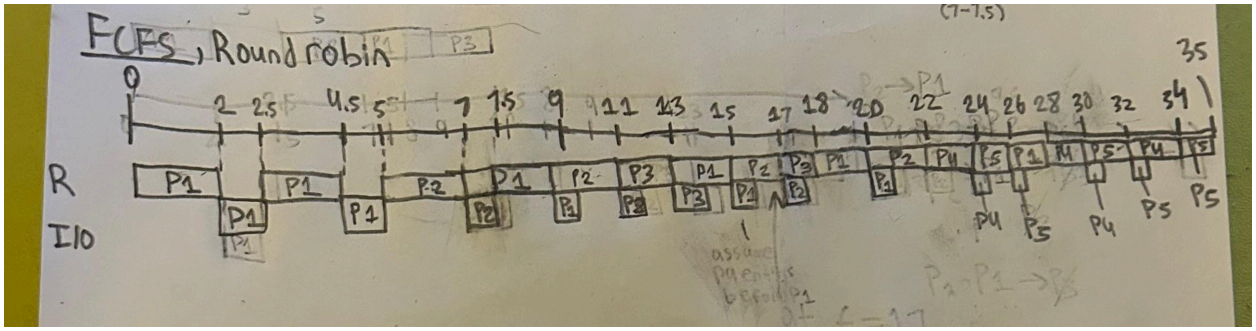
Turnaround time for P1: 31
Turnaround time for P2: 28
Turnaround time for P3: 5
Turnaround time for P4: 19
Turn around time for P5: 5

Mean turn around time = 17.6ms

Completion times can be seen above, they don't need to be calculated as they are shown in the diagram.

b) [0.5 mark] Now assume that each process in part a) requests to do an I/O every 2 ms, and the duration of each of these I/O is 0.5 ms. Create new Gantt diagrams considering the I/O operations and repeat all the parts done in part a) using this new input trace.

i , ii) Both are the same due to the fact that the I/O basically acts as the timer interrupt, except it is instead the I/O interrupt.



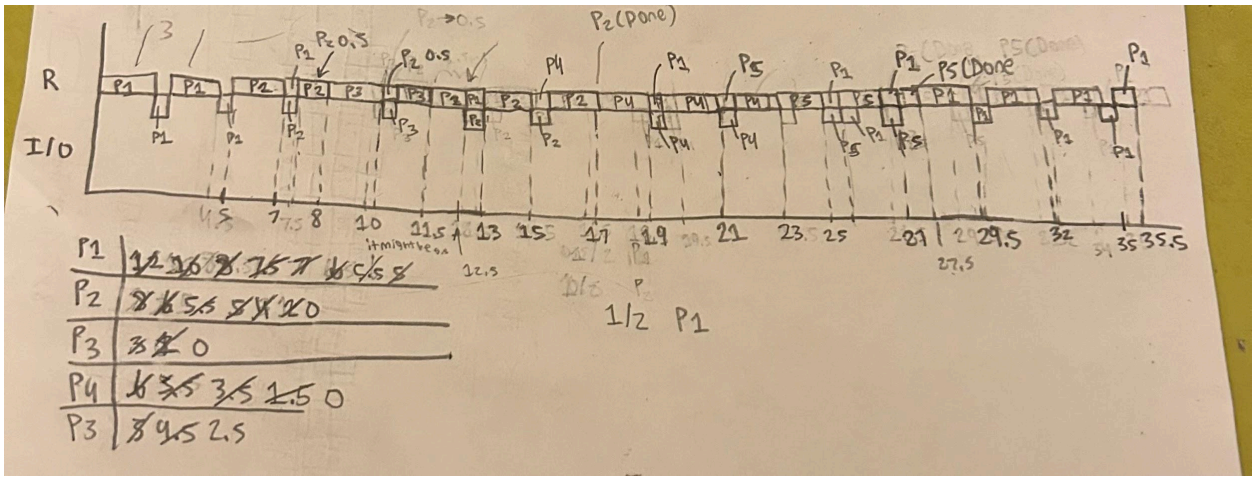
Turnaround time for P1: 20
Turnaround time for P2: 17
Turnaround time for P3: 10
Turnaround time for P4: 19
Turn around time for P5: 15

Mean turn around time = 16.2ms

Completion time for P1 = 28
Completion time for P2 = 22
Completion time for P3 = 18
Completion time for P4 = 34
Completion time for P5 = 35

(this is for both RR and FCFS)

iii)



Turnaround time for P1: 35.5
Turnaround time for P2: 12
Turnaround time for P3: 3.5
Turnaround time for P4: 8s
Turn around time for P5: 27.5

Completion time:
Completion time for P1 = 35.5
Completion time for P2 = 17
Completion time for P3 = 11.5
Completion time for P4 = 23
Completion time for P5 = 29.5

c) [0.5 mark] Consider a multiprogrammed system that uses multiple partitions (of variable size) for memory management. A linked list of holes (the "free" list) is maintained by the operating system to keep track of the available memory in the system.

At the start of the exercise, the free list consists of holes with the following sizes (in KB):

Position	Hole Size	Status
1	85 KB	Free
2	340 KB	Free
3	28 KB	Free
4	195 KB	Free
5	55 KB	Free
6	160 KB	Free
7	75 KB	Free
8	280 KB	Free

The free list is ordered in the sequence given above: the first hole is 85 KB, followed by 340 KB, and so on.

When the system is in this state, the following jobs arrive; each of them has different memory requirements, and they arrive in the following order:

Job No.	Arrival Time	Memory Requirement
J1	t_1	140 KB
J2	t_2	82 KB
J3	t_3	275 KB
J4	t_4	65 KB
J5	t_5	190 KB

[with $t_1 < t_2 < t_3 < t_4 < t_5$]

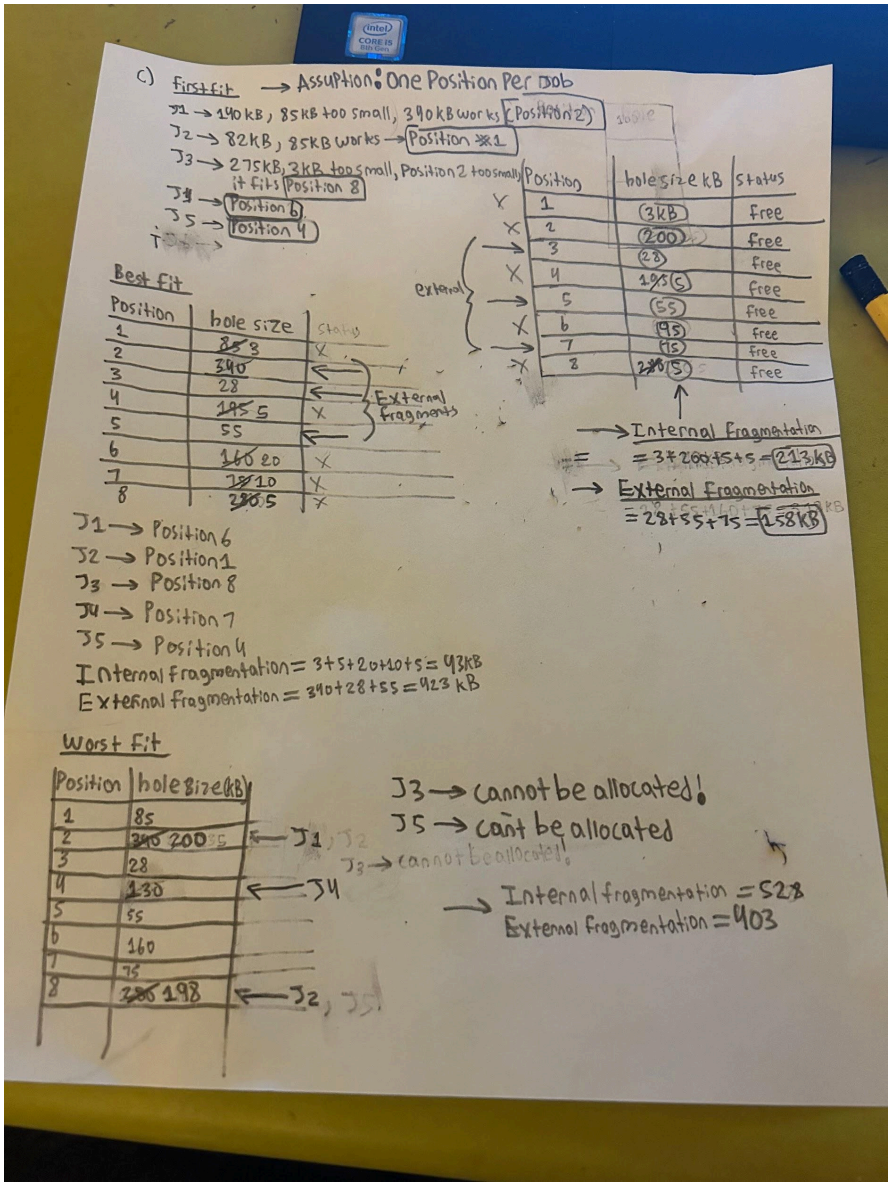
1) Determine which free partition will be allocated to each job for the following algorithms:

- (i) First Fit
- (ii) Best Fit
- (iii) Worst Fit

For each algorithm, show the allocation table (which job gets which partition), the remaining free memory after all allocations, the total internal fragmentation and the total external fragmentation.

Note: This question did not specifically mention whether or not each position can take multiple processes.
→ If we allocate one position for each process (there will be internal fragmentation).
→ If we allow for multiple processes in the position, then there will be no external fragmentation

Assuming one job per hole we get:



Assuming multiple jobs per hole we get:

c) first fit → Assuption: One Position Per Job

J1 → 140 kB, 85 kB too small, 340 kB works (Position 2)

J2 → 82 kB, 85 kB works → Position #1

J3 → 275 kB, 3 kB too small, Position 2 too small, it fits Position 8

J4 → Position 2

J5 → Position 4

Position	hole size kB	status
1	3 kB	free
2	~~340~~ ~~205~~ 135	free
3	28	free
4	~~195~~ 5	free
5	55	free
6	160	free
7	75	free
8	~~280~~ 5	free

J1, J4 →

→ Internal Fragmentation = = 0

→ External Fragmentation = 466

Best fit

Position	hole size	status
1	~~85~~ 3	
2	~~340~~	
3	28	
4	~~195~~ 5	
5	55	
6	~~160~~ 20	
7	~~75~~ 10	
8	~~280~~ 5	

J1 → Position 6

J2 → Position 1

J3 → Position 8

J4 → Position 7

J5 → Position 4

Internal fragmentation = 0

External fragmentation = 466

Worst fit

Position	hole size (kB)	
1	85	
2	~~340~~ ~~205~~ 135	← J1, J4
3	28	
4	~~195~~	← J4
5	55	
6	160	
7	75	
8	~~280~~ ~~188~~ 8	← J3, J5

J3 → J3 can be allocated

J5 → cannot be allocated

→ External fragmentation = 791

2) [0.2 marks] Based on your calculations, analyze and compare the three algorithms according to memory utilization efficiency and fragmentation. Based on your analysis, justify which algorithm would be most appropriate for a system with frequent small allocations, and a system with mixed workload sizes

Show all calculations step by step; also draw the memory state after each allocation. Calculate fragmentation metrics for comparison and provide detailed justification for your analysis

First fit: This is quick, as it doesn't have to iterate through RAM and calculate hole sizes and figure out which one is more optimal. However, it has very average efficiency in terms of Internal fragmentation and external fragmentation. Because it just literally just finds the first hole that is big enough, it causes a lot of internal fragmentation as it can take a hole size that is too big for it leaving internal fragmentation.

Best fit: It would be slower, as it has to iterate through the user area of RAM to find out the smallest hole that can hold the job. It yields a very low internal fragmentation, and it leaves a good external fragmentation (which allows for future processes to have large enough continuous holes to hold their process).

Worst fit: This is pretty useless unless you have a massive job you need to allocate that will need lots of storage for dynamic allocation for example. Unless you had a very large process, and you wanted to give it the hole with the most space, then it is useful, but besides that, it is a waste of time and space.

For a system with frequent but small allocations, best fit would most likely work the best (if you the system is fast enough), as it would be able to compact everything for the most part, leaving small amounts of internal fragments. First fit on the other hand would leave lots of unusable holes between processes (lots of internal fragmentation), and worst fit would be slow, and generally **bad for small allocations** because it uses big holes unnecessarily.

For a system with mixed workload sizes, First Fit would work best because it quickly finds the first hole large enough for each job, efficiently utilizes available memory without wasting the largest holes on small jobs, and tends to leave reasonably sized free spaces for both small and large jobs, balancing speed and fragmentation.

Note: I bolded stuff so it is easier for you to skim through and mark.

Part II – Concurrent Processes in Unix [1 mark]

https://github.com/BraedyCoding/SYSC4001_A2_P2

Assumption: I made the assumption that process2 decrements it at a slower rate than process increments it, so it still goes up to 500.
I also made the assumption that when it said "create two new processes", that meant that we were creating 2 child processes, so we have 3 processes total.

Part 3

GitHub for Part 3: https://github.com/Volley24/Assignment_2

In the third part of this assignment, the task was to create a small API simulator, building up on the interruption system created in assignment 1. The API simulator contains various components, such as code memory partitions, the PCB, the wait queue, and the trace file parser. Just like in assignment 1, an initial text file, containing the trace file of the main program, is present as an input file. However, two major additions were added: FORK and EXEC, where FORK allows the forking of the currently running process (and as such, the PCB and wait queue), and EXEC replaces the currently running process with another program, stored on disk, or somewhere external. As such, files such as programN.txt denote programs to be loaded by EXEC, and can contain things from CPU calls and SYSCALL's, to nested FORKs. The simulator generates two output files: the execution output, and the system status, which contains the PCB wait queue and the currently running PCB, after FORK / EXEC PCB updates.

The main function of the simulator is called simulate_trace, and it is being called recursively in FORK and EXEC. For FORK, this is to simulate the operation of forking a parent and creating a child process, without having a dedicated scheduler in place. Special instructions, such as IF_CHILD, and IF_PARENT, which like their name suggests, only executes the next instructions if they are a child or a parent, respectively. As such, whenever the parent calls simulate_trace, the trace sent to said function only contains the ones the CHILD needs to execute, excluding any PARENT traces. When that function terminates, the PCB for the child is deallocated, and the PARENT runs. Of course, before recursively calling simulate_trace, memory is allocated for the code, and a new PCB is cloned from the parent, aborting if there is no memory partition available. EXEC is similar, except the new process replaces the currently running one, and the PCB is modified to reflect this.

The wait queue portion of the simulator is simulated (as the name suggests), as this simulator does not contain a scheduler. As such, it is mimicked; whenever FORK is called, the PARENT

process is added to the queue, and whenever EXEC is called, the parent is removed from the queue.

There are 6 memory partitions in the simulator, ranging from size 40 to size 2 (simulated in MB, defined in external_files.txt as programN.txt, sizeInMB), as seen in ***[Appendix 1].*** The algorithm used to allocate processes to memory partitions follows the best fit algorithm, as it will try to fit a program's code with the best fitting memory partition. This is subject to internal fragmentation, as each partition can only hold one program, and there may be wasted space. PCBs are stored in the simulate_trace arguments, as well as the PCB wait queue, and contain the memory partition number, which points to the process in memory.

This report contains various examples using various inputs, and explanations on how the results obtained are logical, and how the components interact to produce these results.

Input files: https://github.com/Volley24/Assignment_2/tree/master/input_files/

Output files: https://github.com/Volley24/Assignment_2/tree/master/input_files/

ex_1

The first example (), focuses on having two very simple programs: one which does CPU work, and another which executes a SYSCALL. Looking at the output, we can see the first switch to kernel mode is part of the FORK operation, which does a mimicked SYSCALL (pos 2), before then cloning the PCB, calling the schedule, and executing the CHILD's code. In this case, it calls EXEC, program1, which does another system call (pos 3) to simulate the EXEC, and then executes the program1's content, which is a CPU burst of duration 100ms. After that, since program1 is finished, the PARENT goes to execute, which contains EXEC, program2. This actually does two system calls, one for the EXEC, which is SYSCALL 3, and another for SYSCALL, 4 which mimics an I/O device call. Analyzing the system status, we can see clearly that two programs are present after FORK, with the same name, which is logical, as FORK duplicates the currently running process. After reaching the EXEC, program1, we clearly see that the CHILD process (the one with PID 1) has replaced the program name with program1, size 4 and a different memory partition, which indicates a successfully EXEC call. The second EXEC calls shows only one program, which was initially PID = 0 'init' (i.e: the PARENT from the FORK), as after the CHILD program finished running everything, it's memory partition was freed from the simulator, resulting in only the PARENT remaining.

ex_2

The second example touches upon recursive FORKs, with only a singular program. Since both the INIT program and any subsequent programN programs are parsed with the same simulator, the externally EXEC'd programs can themselves FORK or EXEC. Similarly to example 1, a FORK statement is executed, which creates a CHILD process and a PARENT process waiting on that CHILD. This can be seen from the system status as two INIT's, one with PID = 1 and another with PID = 0, just like in ex_1. In the CHILD program however, after executing EXEC and going into program1 (thus making the PCB have program1 + init, like in ex_1) another

FORK is executed (without any code logic for the IF_CHILD and IF_PARENT, so only the last EXEC program2, 33 gets called, and only by the PARENT). This results is there being two ‘program1’s, where the one with PID = 2 is the CHILD from the CHILD which forked, and the one with PID = 1 is the original CHILD from the PARENT. Logically, we can see that when EXEC program2 is ran, the original ‘program1’ entry from the PCB with PID = 2 is replaced with program2.

ex_3

The third example deals with the handling of both SYSCALLs and END_IO events, where program1 will simulate doing some CPU activity, then doing a SYSCALL, more CPU activity, and END_IO, presumably to mimic a printer. This example doesn’t introduce any new discussion points, as the init simply calls a FORK, where only the PARENT calls EXEC program1, with some CPU, 10 work for only the PARENT after the ENDIF.

ex_4

The fourth example deals with an edge-condition in the EXEC logic: attempting to allocate a program with a size greater than 40 into the memory partition. As seen by the trace file, whenever the init program attempts to call EXEC bigprogram with a size greater than 40 (the maximum memory partition size), a message saying allocation failed is displayed. This has the consequence of not actually doing anything, and as a consequence, the PCB is NOT swapped. This means that the code AFTER the EXEC, which would have been replaced by the program swap, did not occur, which allowed the final CPU, 100 burst to execute.

Ex_5

The fifth example deals with an interesting potential oversight in the EXEC code. We see that init contains seven calls to program1, with an EXEC, and one might expect program1 to run seven times. However, due to the way this is implemented, EXEC explicitly REPLACES the current process with another one, as well as the current PCB, which would effectively render all of the subsequent EXEC’s as useless, only calling the CPU burst after the initial EXEC. This could be fixed in our simulator by forking every time we wanted to call program1. In this case, if all partitions are occupied, further EXECs will fail to allocate memory, and the simulator will display an error and all other instructions. (this would have been a very big simulation due to the amount of FORK’s needed to exhaust all of the resources, but it is still a good discussion point.)

To conclude, this API simulator successfully models process creation, execution, and memory management, with logical output logs and good handling of edge cases.

Appendices

Appendix 1

```
27  memory_partition_t memory[] = {
28      memory_partition_t(1, 40, "empty"),
29      memory_partition_t(2, 25, "empty"),
30      memory_partition_t(3, 15, "empty"),
31      memory_partition_t(4, 10, "empty"),
32      memory_partition_t(5, 8, "empty"),
33      memory_partition_t(6, 2, "empty")
```