Matthew Gibeault
101323772

im student 2

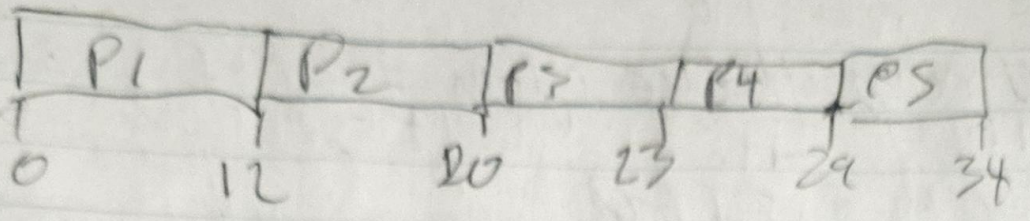
Part 1

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

a)
i) FCFS
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

First come first serve is a very simple non preemptive scheduling algorithm, the order of execution is determined by arrival time, whichever process arrives the earliest will be the earliest one to be executed. Since it is a non preemptive process, once a process begins execution it will continue until completion uninterrupted. Once the process is done running, it will execute the next process in the ready queue. Its implementation is based on a simple data structure which is a FIFO queue, meaning first in first out. Some benefits of this algorithm are that it is very simple and easy to understand, it is also suitable for batch systems where all processes arrive at the same time. Some disadvantages of this algorithm is that if a long process is the first one to arrive it can delay all of the other processes that may require less time to execute. It also has poor response time and is not suitable for time sharing systems.

9)

1. 

2. completion times

P1: 12ms P2: 20ms P3: 23ms P4: 29ms P5: 34ms

3. turn around times

P1: 12-0 = 12ms P2: 20-5 = 15ms

P3: 23-8 = 15ms P4: 29-15 = 14ms

P5: 34-20 = 14ms

4 mean turn around time:

(12+15+15+14+14)/5 = 14ms

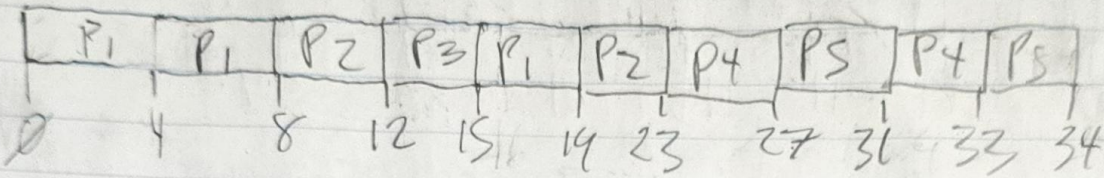
ii) round robin (time slice of 4 ms)
RR explanation:
Round robin runs processes in order of arrival, each process runs for as long as the time slice allows, if the time slice is reached then it moves on to the next process in the queue and moves the previews process to the back of the queue.

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

Assignment 2

a) ii RR 4ms time slice

1.



2. Completion times: P1: 19ms P2: 23ms P3: 15ms
P4: 33ms P5: 34ms

3. Turn around time:
Completion time − arrival time
P1: 19 − 0 = 19ms P2: 23 − 5 = 18ms
P3: 15 − 8 = 7ms P4: 33 − 15 = 18ms P5: 34 − 20 = 14ms

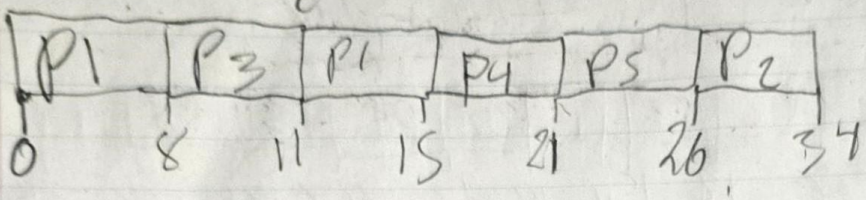
4. mean turnaround time:
(19 + 18 + 7 + 18 + 14) / 5 = 15.2

iii) shortest job first with preemption

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

iii Shortest job. P1 is with preemption

1.



2. Completion times

P1: 15ms P2: 24ms P3: 11ms P4: 21ms P5: 26ms

3. Turnaround times

P1: 15-0 = 15ms P2: 34-5 = 29ms

P3: 11-8 = 3ms P4: 21-15 = 6ms

P5: 26-20 = 6ms

4. Mean turnaround time

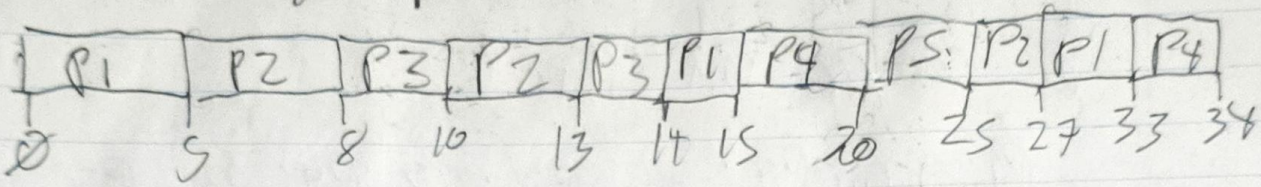
(15+29+3+6+6)/5 = 11.8ms

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

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

iv) Multi level queue with feedback

high-priority queue quan = 2ms
mid " " " = 3ms
low-priority = FIFO



2. Completion times:

P1: 33ms P2: 27ms P3: 14ms P4: 34ms

P5: 25ms

3. Turnaround times:

P1: 33-0 = 33ms P2: 27-5 = 22ms

P3: 14-8 = 6ms P4: 34-15 = 19ms

P5: 25-20 = 5ms

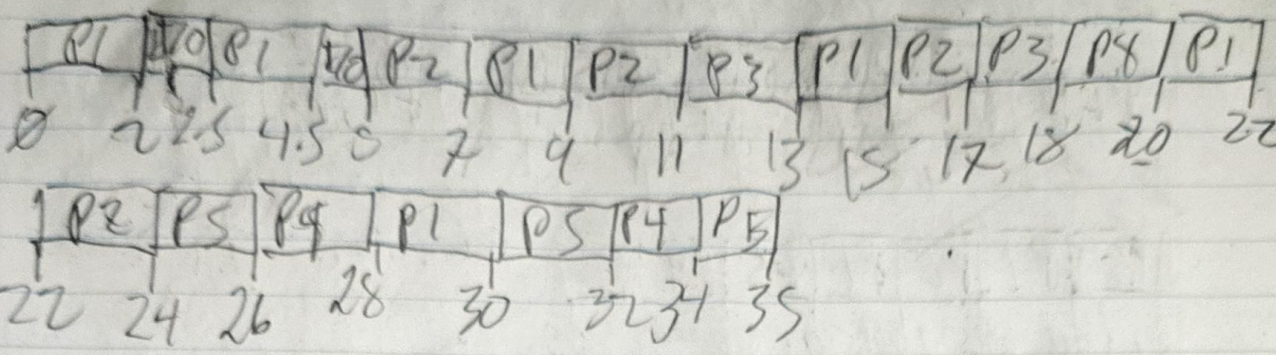
4. Mean turn around time:

(33+22+6+19+5)/5 = 17.0ms

b) 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)

b) i)



2. Completion times

P1: 30ms P2: 24ms P5: 18ms P4: 34ms P5: 35ms

3. turnaround time

P1: 30-0 = 30ms P2: 24-5 = 14ms P3: 18-8 = 10ms

P4: 34 - 15 = 19ms P5: 35-20 = 15ms

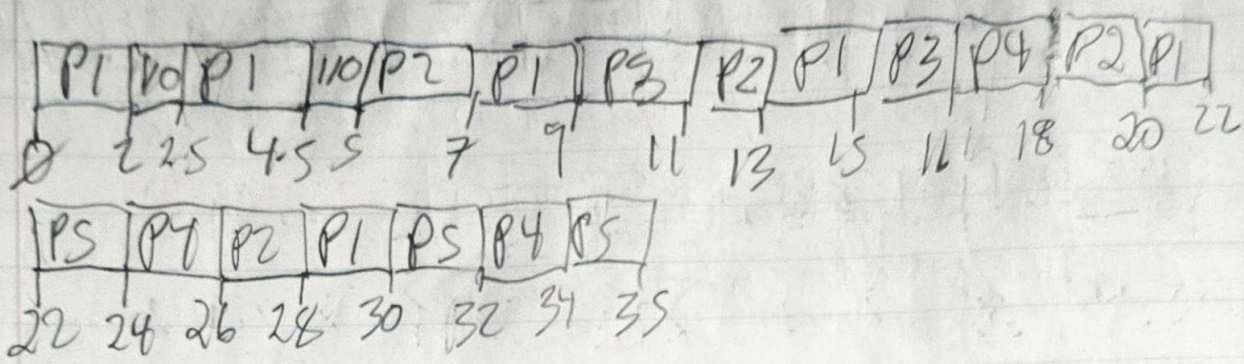
4. avg turn around

(30+14+10+19+15)/5 = 18.6ms

ii)

ii) Round Robin

1. Time slice 4ms



2. Completion times

P1: 30ms P2: 28ms P3: 16ms P4: 34ms P5: 35ms

3. Turn around times

P1: 30-0 = 30ms P2: 28-5 = 23ms P3: 16-8 = 8ms

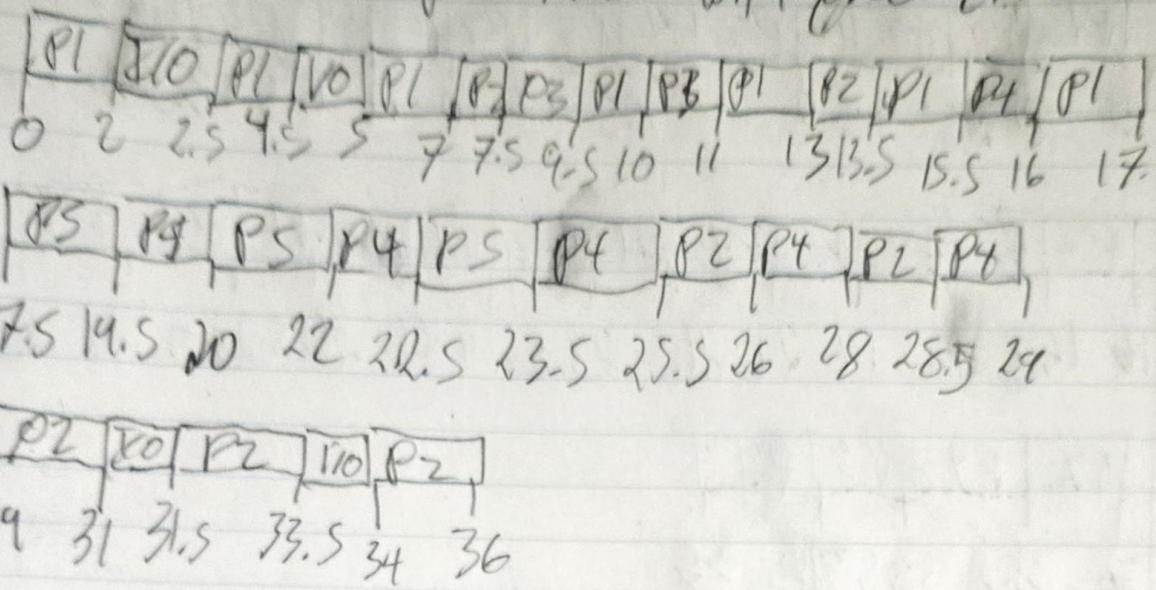
P4: 34-15 = 19ms P5: 35-20 = 15ms

4 Mean turn around time

(30+23+8+19+15)/5 = 19ms

iii)

iii) shortest job first with preemption



2. Completion times

P1: 17.5 ms P2: 36 ms P3: 11 ms P4: 29 ms P5: 23.5 ms

3. turn around times

P1: 17.5 - 0 = 17.5 ms P2: 36 - 5 = 31 ms

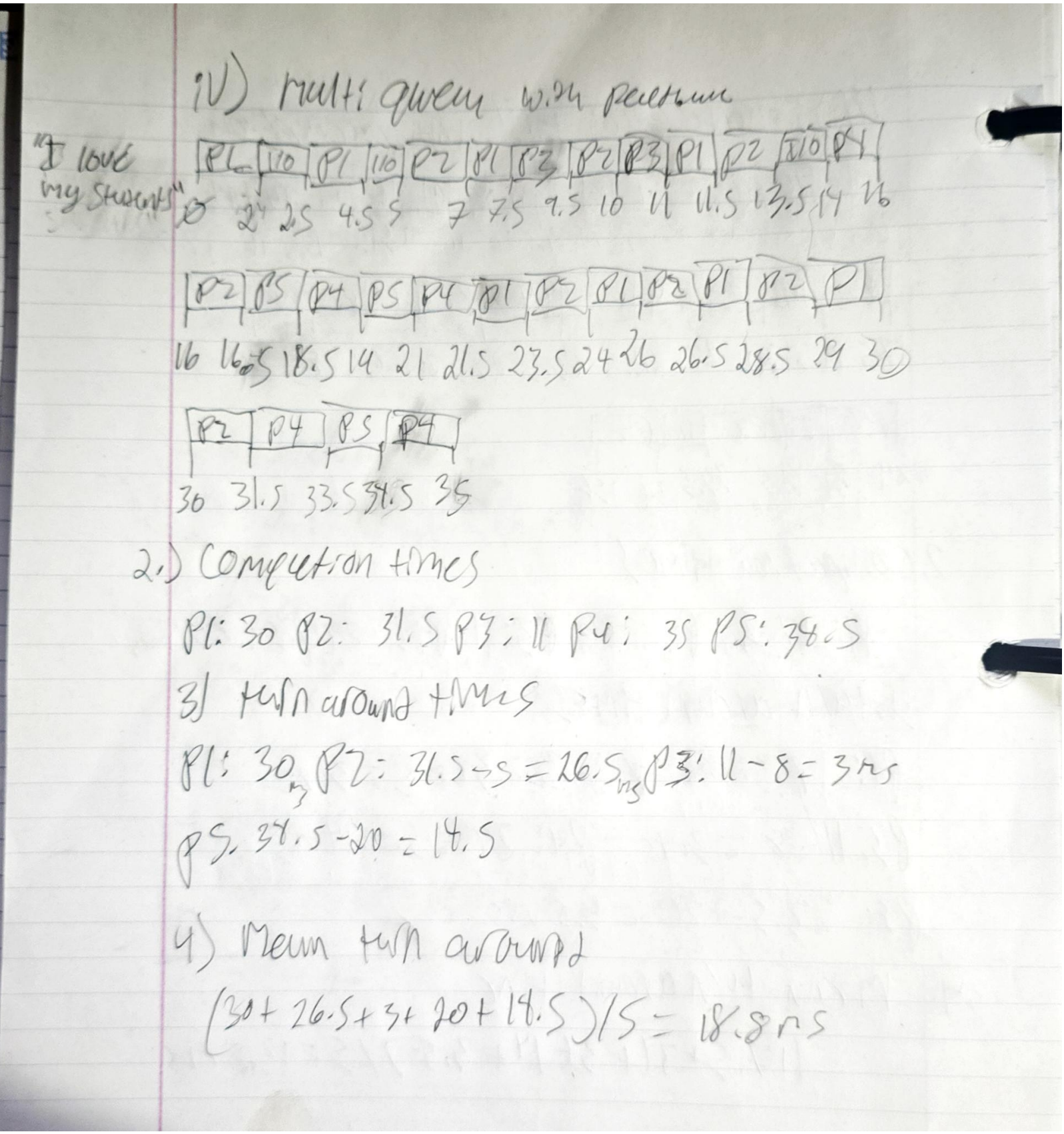
P3: 11 - 8 = 3 ms P4: 29 - 15 = 14 ms

P5: 23.5 - 20 = 3.5 ms

4. Mean turn around time

(17.5 + 31 + 3 + 14 + 3.5) / 5 = 13.8 ms

iv)


iv) multi queue w.rr preemp
"I love my students"
P1 I/O P1 I/O P2 P1 P3 P2 P3 P1 P2 I/O P4
0 2 2.5 4.5 5 7 7.5 9.5 10 11 11.5 13.5 14 16
P2 P5 P4 P5 P4 P1 P2 P1 P2 P1 P2 P1
16 16.5 18.5 14 21 21.5 23.5 24 26 26.5 28.5 29 30
P2 P4 P5 P4
30 31.5 33.5 34.5 35
2.) Completion times
P1: 30 P2: 31.5 P3: 11 P4: 35 P5: 34.5
3) turn around times
P1: 30 P2: 31.5-5 = 26.5 P3: 11-8 = 3 ms
P5: 34.5-20 = 14.5
4) Mean turn around
(30+26.5+3+20+14.5)/5 = 18.8 ms

c) 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

Job	Size	Allocated hole	Remaining Hole size after allocation
J1	140KB	2	200KB
J2	82KB	1	2KB
J3	275KB	8	5KB
J4	65KB	2	135KB
J5	190KB	4	5KB

Remaining free memory: 466KB
Total internal fragmentation: 348KB
Total external fragmentation: 466KB

- (ii) Best Fit

Job	Size	Allocated hole	Remaining Hole size after allocation
J1	140KB	6	20KB
J2	82KB	1	3KB
J3	275KB	8	5KB
J4	65KB	7	10KB
J5	190KB	4	5KB

Remaining free memory: 466KB
Total internal fragmentation: 43KB
Total external fragmentation: 466KB

- (iii) Worst Fit

Job	Size	Allocated hole	Remaining Hole size after allocation
J1	140KB	2	200KB
J2	82KB	8	198KB
J3	275KB	No hole has enough space remaining :(	
J4	65KB	2	135KB
J5	190KB	8	8KB

Remaining free memory: 741KB
Total internal fragmentation: 541KB

Total external fragmentation: 741KB
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.

2) Best fit had the smallest internal fragmentation at 43KB. Best fit is ideal for frequent small allocations as it puts the small jobs into the smallest holes, ensuring that there are still big holes for the big jobs. First fit would have inconsistent performance as it would just depend on what jobs appear first to be allocated, and in which order holes appear. Worst fit could be useful to maximize the usage of large holes in memory, but as shown in this example it can also lead to scenarios where a job that otherwise would have space now doesn't have any memory it can fit into.
Best fit is by far the best for frequent small allocations, whereas first fit is a good middle ground that would be best in a system with mixed workload sizes

I'm really not sure what calculations could be done for 2 that weren't already done in 1

Part 2

Fork system call explanation:
fork() creates a new process by duplicating the calling process. The new process is a child to the original. After fork() is called, both processes continue execution at the instruction after fork(). Fork returns different values depending on where its called, in the parent fork() returns the PID of the child, in the child fork() returns 0. The child copies the parents address space and file descriptors, but has a different PID assigned to it. Changes in the childs memory don't affect the parent unless memory is being shared.

The exec function in Linux is a system call used to replace the current process with a new program. When a process calls exec, it stops running its own code and instead begins executing the code of the specified program, while keeping the same process ID. This means that no new process is created, the existing one is transformed into a different program. The exec family of functions provides different ways to specify the program name and arguments. The exec system call is often used after fork() to allow a parent process to create a child and then have the child run a completely different program.

Output analysis:

Test case	Number of processes created	Analysis
1	2 (parent + child)	Single fork() creates one child
2	3 (1 parent + 2 children)	Contains to separate fork() calls
3	2 (parent + child)	Only one fork() occurs
4	3 (one parent, two children)	A second form occurs after program 1 runs
5	1	Fork was not called

Analysis: the child always runs before the parent resumes, as outlined in the assignment that the child must have higher priority

Memory allocation and loading programs:

Test case	Loaded programs	Memory partitions in use	Analysis
1	Program1: 10MB Program2: 15MB	Partition 4 then partition 3	Larger program causes longer load time
2	Program1: 10MB	Partition 4, then 3, then 2	Test case demonstrated precess replacement
3	Program2 twice:15MB	Partition 4	I/O interrupts occur after execution begins
4	Program1: 10MB	Partition 4 then 3	Multiple syscalls and CPU bursts extend runtime
5	No program loaded	none	Only system calls and CPU bursts occur, no memory reallocation

Across the 5 test cases the simulation consistently reproduced expected IS process management behaviour. Fork() cloned PCB state and immediately scheduled the child. Exec() replaced the running process's memory image and updated PCB and partition entries. Program loading time depended on program size, and system calls followed the required kernel-mode transition steps. Test case 5 provides a contrast since it is a test case that doesn't create and processes or reassigns memory. The tests show the simulator behaves according to the assignment rules and expectations as proven by the outputs.

Github links:
https://github.com/Nhfaris627/SYSC4001_A2_P3

https://github.com/EnderAres183/SYSC4001_A2_P2