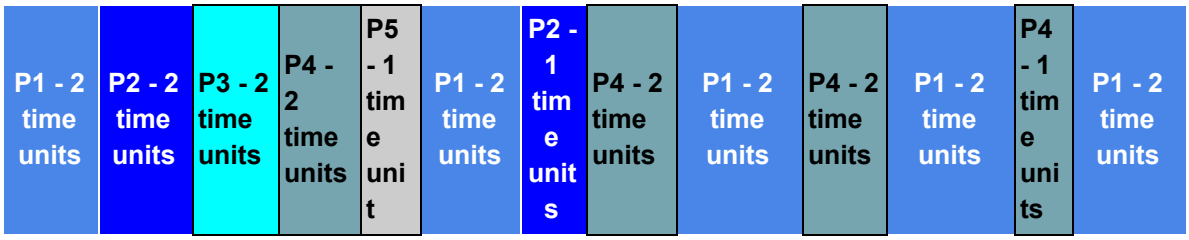
Theoretical Part:

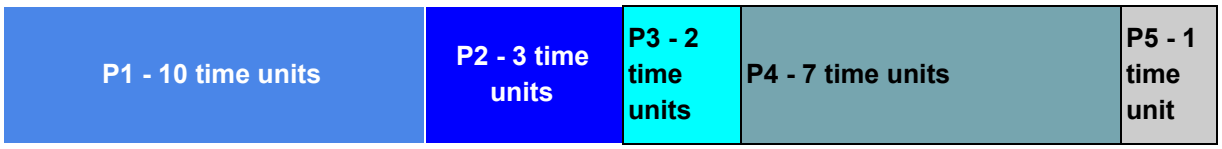
Part I:

Round Robin Gantt table (with quantum size 2):



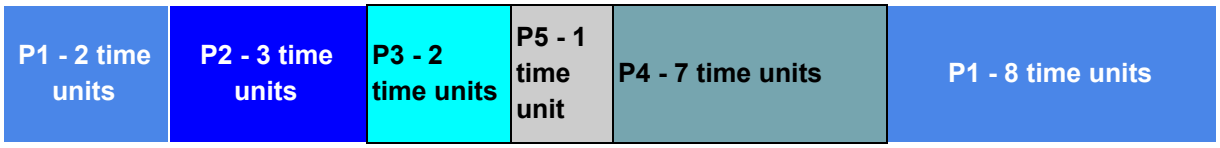
Average waiting time is: (12 + 6 + 2 + 12 + 6) / 5 = 7.6
Turnaround time is: (22 + 9 + 4 + 19 + 7) / 5 = 12.2

FCFS Gantt table:



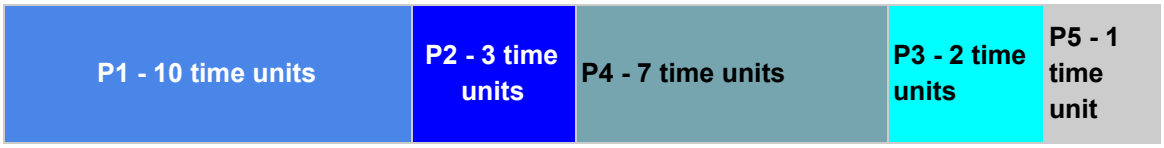
Average waiting time is: (0+ 8 + 9 + 11 + 15) / 5 = 8.6
Turnaround time is: (10 + 11 + 11 + 18 + 16) / 5 = 13.2

SRTF Gant table::



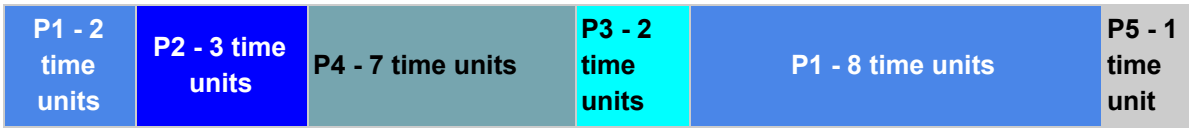
Average waiting time is: ((15-2) + (2-2) + (5 - 4) + (8 - 4) + (7 - 7)) / 5 = 3.6
Turnaround time is: (23 + 3 + 3 + 11 +1) / 5 = 8.2

Priority:



Average waiting time is: (0 + 8 + 9 + 16 + 15) / 5 = 9.6
Turnaround time is: (10 + 11 + 16 + 18 + 16) / 5 = 14.2

Priority and Preemption:



Average waiting time is: (12 + 0 + 8 + 1 + 15) / 5 = 7.2
Turnaround time is: (22 + 3 + 10 + 8 + 16) / 5 = 11.8

2. The files' blocks that we save in the exercise are saved in the heap segment of the process, which is saved in the RAM. Time access to the RAM is faster in a couple of orders of magnitude to disk as we have seen. Thus the answer is yes.

3. RAM access is "easier" in the sense that it is done almost completely by the CPU itself. Whereas Disk access is more complex and thus involves a file system implementation which is done by the OS for the most part. As we have seen Disk access is slower in a couple order of magnitudes and so we can be less tolerant to redundant disk accesses compared with RAM. And so, we deduce that disk access management is a more challenging task.

4. For example we will have 4 blocks- b1, b2, b3, b4 and in the cache we can hold 2 blocks:
· LRU better than LFU for this pattern: b1, b1, b1, b2, b4, b3
The LRU will hold at the beginning b1 and b2, then b2 and b4, and finally b4 and b3.
The LFU will hold at the beginning b1 and b2, then b1 and b4, and finally b1 and b3.
· LFU better than LRU for this pattern: b1, b1, b1,b2, b3, b4, b1
The LRU will hold at the beginning b1 and b2, then b2 and b3, then b3 and b4, and finally b4 and b1.
The LFU will hold at the beginning b1 and b2, then b1 and b3, and finally b1 and b4.

· Both algorithms fails for this pattern: b1, b2, b3,b4, b1, b2, b3,b4
The LRU will hold at the beginning b1 and b2, then b2 and b3, then b3 and b4, then b4 and b1 and so on...
The LFU will hold at the beginning b1 and b2, then b2 and b3, then b3 and b4, then b4 and b1 and so on...

5. FBR attempts to solve instances of which one block was accessed multiple times during first read which per se does not represent the “real” value of accessing the file. In reality we have performed one read of the file but behind the scenes the multiple accesses causes the counter to be inflated artificially. In this case LFU’s drawback would manifest itself in which it would maintain this block in the cache, whereas the better option could be possibly to delete it since we have used this block only once. Algorithms like LRU would attempt to delete it from the cache if in fact this block was not in use recently. FBR algorithm attempts to combine each of the mentioned before algorithms in order to take advantage of the pros of each one. The problem mentioned above is solved with FBR by partitioning the cache and apply different behaviour for each part of the cache. In particular, blocks in the “new” partition won’t get their counters incremented during this stage. Incrementing will start taking place only after block would “age” enough and move to a different partition. This aging process is used to indicate that this block was not used in the recent time.

Part II:

1. 7 disk accesses overall. Initially we retrieve the file inode which requires 3 - 2 accesses for the directory and one for the inode itself. The second step would be to retrieve the data itself via the pointers. The inode’s ten direct pointers would not be enough to retrieve all the data we need (A block size is 2KB times 10 pointers is 20KB), hence we invoke one indirect pointer so that’s an additional 2 disk accesses. Finally we save the new data and update the inode itself and so we add 2 more accesses.

2. Since we supply seek command offset 0 with the fd of the file (We assume that the intention here is that 0 is the offset argument, plus we did not find the seek in linux manual command only lseek which requires additional arguments). This per se should not cause any interrupts since we only set the reading offset but do not perform disk access. The read command access the disk directly (with the supplied flags) and this generates a hardware interrupt as we have seen.

3. a. Ignoring context switch overhead, Round-Robin with a quanta size of one. This would ensure that the short-jobs will not left hanging in the queue for long periods relatively. Thus average wait time for the short jobs would be optimal in this case.
   b. If we take the overhead into account the solution suggested is the former section Would probably not be optimal since Round-Robin algorithm has many job preemptions. To minimize this we could switch our solution to FCFS algorithm.