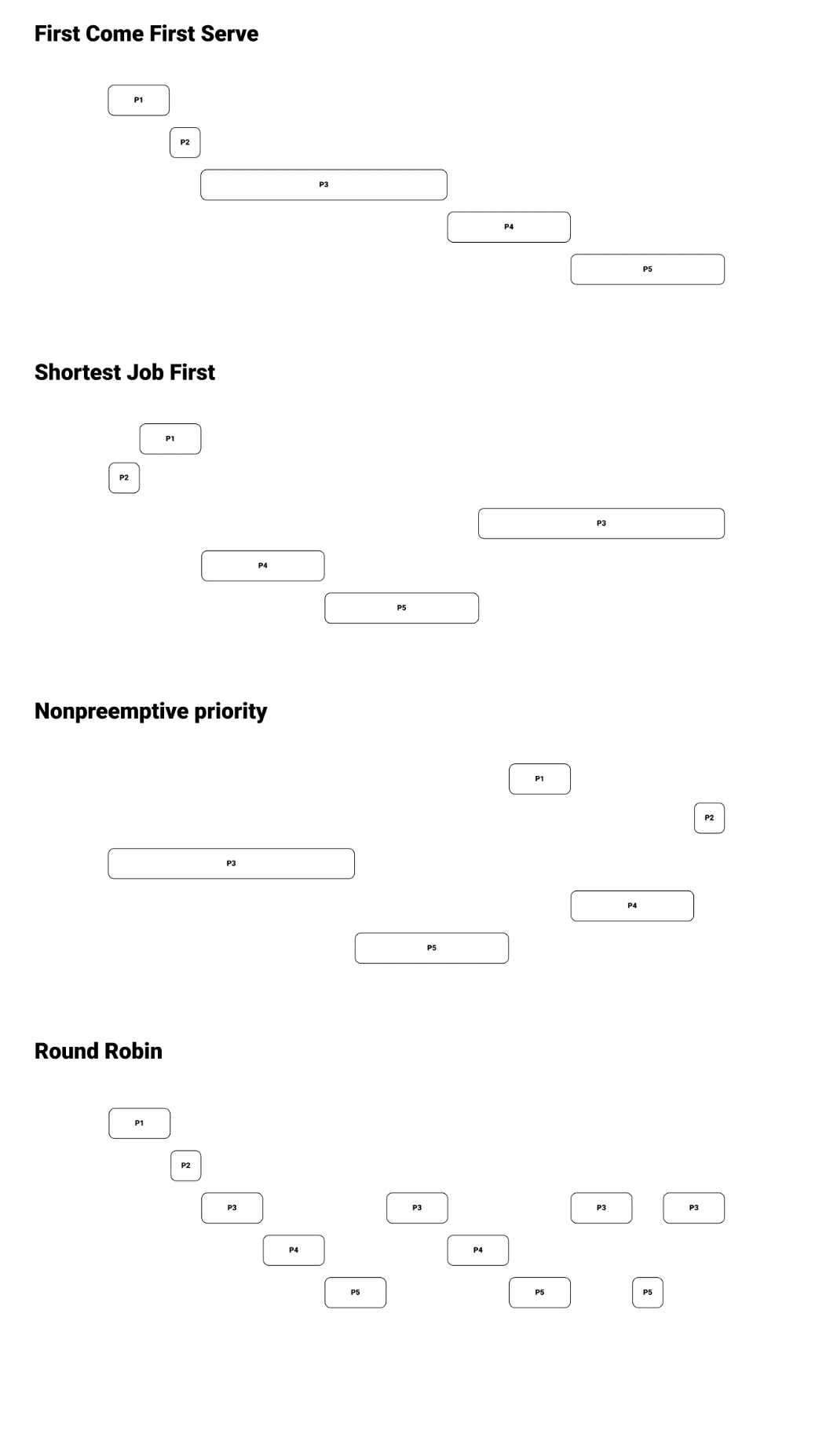
1. a



b. Process vs Algorithm turnaround time.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Process | FCFS | SJF | Non-Preemptive Priority | RR |
| P1 | 2 | 3 | 15 | 2 |
| P2 | 3 | 1 | 20 | 3 |
| P3 | 11 | 20 | 8 | 20 |
| P4 | 15 | 7 | 18 | 13 |
| P5 | 20 | 12 | 13 | 18 |

c. Process vs Wait time

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Process | FCFS | SJF | Non-Preemptive Priority | RR |
| P1 | 0 | 1 | 13 | 0 |
| P2 | 2 | 0 | 19 | 2 |
| P3 | 3 | 12 | 0 | 12 |
| P4 | 11 | 3 | 14 | 9 |
| P5 | 15 | 7 | 8 | 11 |

d.

FCFS: 6.2

**SJF: 4.6**

Non-Preemptive priority: 10.8

RR (quantum = 2): 6.8

SJF has the minimum average waiting time at 4.6 per process.

* 1. The shorter burst I/O bound tasks will use their time and then call there I/O operation. This quantum is favorable to the I/O bound tasks, because they will have shorter waits to start their I/O operation. The longer running CPU-Bound task will suffer from this quantum because it will need to swap in many times before it can complete, this will add more swap overhead. Because there are more I/O bound tasks this is still probably the best option.
  2. This time quantum favors the CPU-bound task and will do less swaps to complete. However, the I/O bound tasks will be stuck waiting for the long CPU-Bound operation to run and will end up doing more waiting. This waiting will far outweigh the switching overhead. This seems to be the worse option.
  3. A. FCFS does not favor short processes
  4. Processes shorter than the quantum because they will get to complete after one pass and will not have to wait for the next time around.
  5. Shorter process can complete after the first RR queue, they will not have to wait for the next pass and will not be bumped down to the longer waiting queues.
  6. Use only if the wait is expected to be very short.
  7. When you have a critical section and only want one thread in at a time.
  8. When you need to keep track of multiple resources.
  9. When needing to sync threads up.
  10. To read/write on data.
  11. Disabling Interrupts means the code that is currently running can not be stopped. Because there is only 1 core it will finish before any other code affects any data, making it mutually exclusive.
  12. An un-healthy process might just hold onto that CPU forever and will force a restart.
  13. If two threads both check the if on line 8 at the same time before they increment the resource counter, if the resource counter is at MAX\_RESOUCES -1 they will both think they can get the resource when only 1 resource is available.
  14. Lock on line 7 release and release right before the return.
  15. No because the check of the value and the increment of that value need to happen atomically. The way the code is setup, those two things happened in different spots.