Bryan Arnold

CSE 4300

2/8/2018

Homework 1

1.1) When context-switching between processes occurs, the first thing to happen is an interrupt is detected by the OS. When this happens, the OS saves the PC and registers of the current process, as well as the user stack pointer. Now, control is handed over to the kernel clock interrupt handler. The kernel then saves all the registers and other machine states of the current process into the process' PCB. The OS then determines which process is next by the scheduler and retrieves the PCB of that next process. The new process' PCB is retrieved, and the registers are restored for the process. Now this process is in the state before it was interrupted, and user control is granted back.

1.2) The context-switching between two threads is simpler than that of processes. Instead of having to have a data structure such as a PCB, the only thing that needs to be swapped between the threads is CPU registers. The OS gets an interrupt, saves the CPU registers of the current thread, and then transfers to kernel control. The kernel takes the new thread's CPU registers, restores them, and then gives control back to the OS and user.

2.1) One example of programming where multithreading is more optimal, is matrix operations. Each thread can compute certain areas of the matrices, instead of sequentially going through all the operations in one thread. I had to implement Conway's Game of Life with threads, and it was much quicker than one thread when enough threads were added. Another example is some sort of web server. It would service individual requests from users on separate threads, instead of having each request taken care of by one thread.

2.2) A good example of a single thread being more optimal would be operations on a list of numbers. This is because using multiple threads would effectively do nothing, as each thread must wait for the before finishing the operation, and the operations taken are constant speed. Lastly, another example would be memory allocation. Overhead for creating multiple threads would be much longer in terms of performance. It is much better just to use one thread for this.

3) CPU efficiency is the sum of CPU being useful divided by the total CPU time.

a) A process is running for T time, with a switch occurring at the block. So, the CPU efficiency is T/(T+S).

b) Since the quantum is lower than the time before a block occurs, each time the CPU runs for T, T/Q process switches are required. So, the overhead is ST/Q. Using the same principle as the previous questions, the efficiency would be T/(T+(ST/Q)). This simplifies to Q/(Q+S).

c) If the quantum required is the same as the overhead time, just use the previous problem solution for this one as well, Q/(Q+Q) = 50%.

d) If the quantum is 0, then the efficiency for which the quantum is used one would also be 0. So, as Q gets closer and closer to 0, so does CPU efficiency.

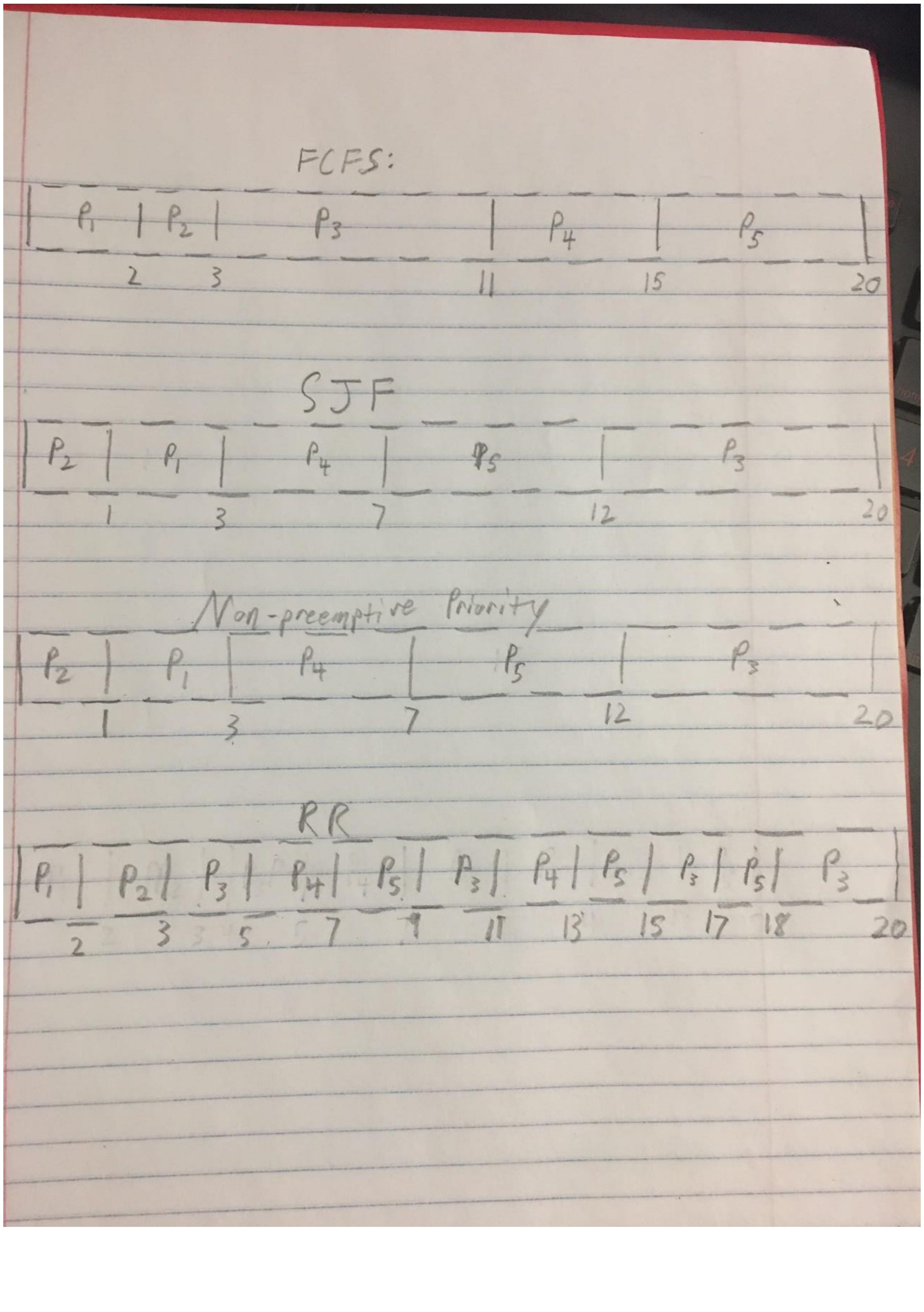
4)

a) When kernel threads < processors, not all the processors would be utilized. The scheduler maps kernel threads to processors and not user-level threads to processors, so by not having threads mapped to each processor, some processors would just be idle.

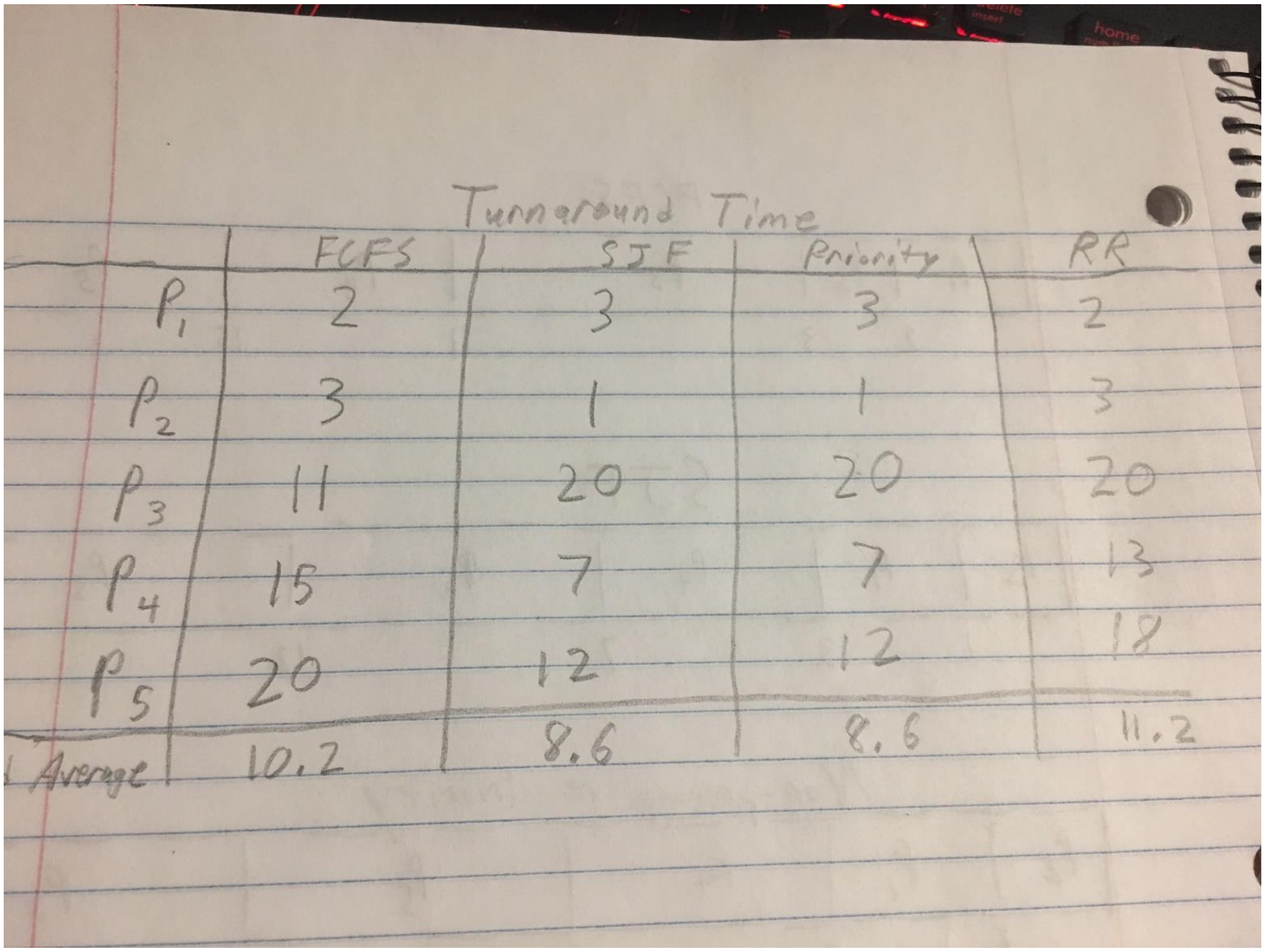
b) When kernel threads = processors, then the processors would be used optimal to some extent. They would be used simultaneously, but when a kernel thread blocks inside the kernel, the processor it is mapped to would become idle and remain idle.

c) When kernel threads > processors, when a kernel thread is being blocked, it can be swapped for another kernel thread that isn’t blocked and is ready to execute. This would fully utilize each processor, and none would be idle.

5) a)


FCFS:
P1 | P2 | P3 | P4 | P5
2 3 11 15 20
SJF
P2 | P1 | P4 | P5 | P3
1 3 7 12 20
Non-preemptive Priority
P2 | P1 | P4 | P5 | P3
1 3 7 12 20
RR
P1 | P2 | P3 | P4 | P5 | P3 | P4 | P5 | P3 | P5 | P3
2 3 5 7 9 11 13 15 17 18 20

b)

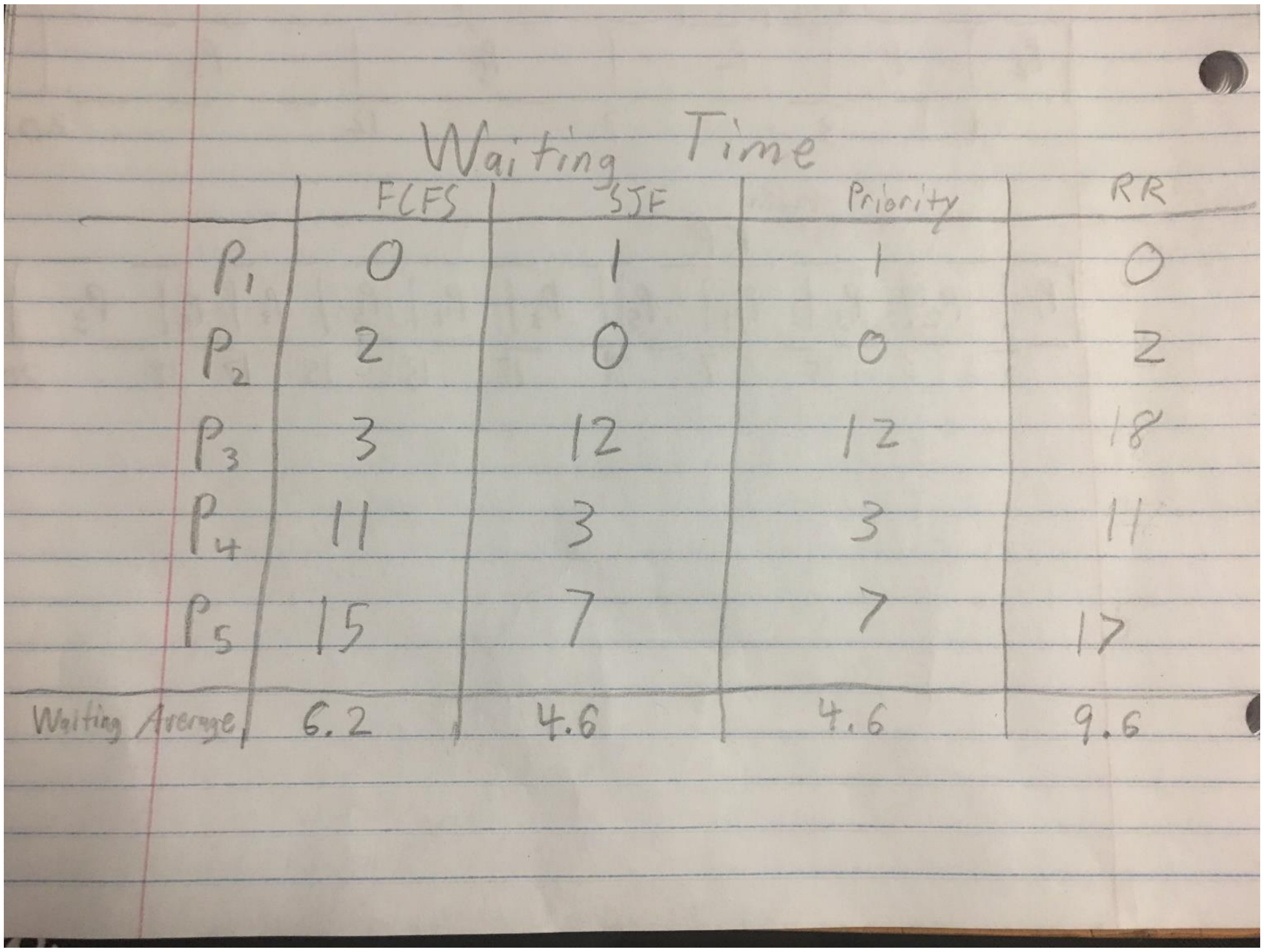


Turnaround Time

	FCFS	SJF	Priority	RR
P_1	2	3	3	2
P_2	3	1	1	3
P_3	11	20	20	20
P_4	15	7	7	13
P_5	20	12	12	18
Average	10.2	8.6	8.6	11.2

Average turnaround times: FCFS = 10.2, SJF = 8.6, Non-preemptive Priority = 8.6, RR = 11.2

c)



Waiting Time

	FCFS	SJF	Priority	RR
P_1	0	1	1	0
P_2	2	0	0	2
P_3	3	12	12	18
P_4	11	3	3	11
P_5	15	7	7	17
Waiting Average	6.2	4.6	4.6	9.6

Average waiting times: FCFS = 6.2, SJF = 4.6, Non-preemptive Priority = 4.6, RR = 9.6

d) Since the non-preemptive priority and SJF algorithms ended up being scheduled the same with the given jobs, both share the minimal average waiting time of 4.6. Usually though, SJF would be the shorter waiting time in most scenarios.