Theoretical Part:

Part 1:

Question 1:

Round Robin With Quantum = 2 Gantt Chart:

0-2	P1
2-4	P2
4-6	P3
6-8	P4
8-9	P5
9-11	P1
11-12	P2
12-14	P4
14-16	P1
16-18	P4
18-20	P1
20-21	P4
21-23	P1

First Come First Serve Gantt Chart:

0-10	P1
10-13	P2
13-15	P3
15-22	P4
22-23	P5

Shortest Remaining Time First Gantt Chart:

0-2	P1
2-5	P2
5-7	P3
7-8	P5
8-15	P4
15-23	P1

Priority Scheduling With Preemption Gantt Chart:

0-2	P1
2-5	P2
5-12	P4
12-14	P3
14-22	P1
22-23	P5

Priority Scheduling WITHOUT preemption Gantt Chart:

0-10	**P1**
10-13	**P2**
13-20	**P4**
20-22	**P3**
22-23	**P5**

Question 2:

No. First, each time we need to read, we check if the cache holds a block of data that we need, this surely means that we always search the cache first, which means, in a case of a cache miss – where the block isn't found in the cache, we need to read anyways from the disk, but now, not only the runtime of reading from disk is applied, but also the overhead of checking whether the block we need is in cache, therefore, no, not always this method gives us the best runtime possible, but usually it does.

Question 3:

The CPU holds responsible for accessing memory with hardware implementation, and the OS is the one responsible for accessing files. This means the efficient implementation between different computers depends on different hardware. So Implementing a good efficient algorithm which actually matches the hardware or the OS, is harder.

Question 4:

LRU – Least Recently Used – meaning we need a working pattern where we need to keep the blocks of data that were least recently used. For example, if we used a big database recently – Perhaps for inserting something – but not as frequently as other stuff, meaning we add only after some checks in some end of a function – we may assume we need this big database afterwards in order to check where to insert the new item, or perhaps deleting an item.

LFU – Least Frequently Used- meaning we need a working pattern where we need to keep the blocks of data that were least frequently used. For example, some sort of file that we write data to – like, a file that represents a log, we use this many times in our program, however not necessary the least recently, but we do use it plenty of times therefore we want to make sure we keep it on cache.

Question 5:

It tries to solve the problem of LFU – a file that is being referenced a lot of times in a short period of time, however, isn't referenced at all after that short period of time. Meaning it has a high reference count but isn't actually participating in anything right now, this is a classic problem of LFU which FBR comes to solve.

<u>Part 2 – Questions From Exams:</u>

<u>Question 1:</u>

There are 9:

1. **Access to root "/"**
2. **Access to root guide**
3. **Access to OS**
4. **Access to OS folder**
5. **Access to "readme.txt" i-node**
6. **Access to single direct of "readme.txt"**
7. **Access to the right block**
8. **Write to it**
9. **Access and update the i-node of "readme.txt"**

<u>Question 2:</u>

Seek and read are system calls of type trap – a software interrupt.

Also, once the OS finishes handling read system call, she will have to use the disk, which means there is a hardware interrupt of disk.

<u>Question 3 a:</u>

We should use Round Robin Scheduler with quantum = 1. This way, jobs that finish within one quantum will end, and the other will keep running till they finish without knowing the needed runtime for every job.

<u>Question 3 b:</u>

Round Robin Scheduler uses preemptions after a quantum of 1, which makes all the jobs of runtime 1 ends on the first pass, we use more context switching than other schedulers and this makes the overhead as small as possible.