ANSWERS:

Part 1:

(1a) round robin:

The gantt chart:

- 0-2p1
- 2-4p2
- 4-6p1
- 6-7p2
- 7-9p3
- 9-11p4
- 11-13p1
- 13-14p5
- 14-16p4
- 16-18p1
- 18-20p4
- 20-22p1
- 22-23p4

Turnaround time - 11.6 without cs

Average wait time - 7 without cs

b) First come first serve:

the gantt chart:

- 0-10p1
- 10-13p2
- 13-15p3

- 15-22p4
- 22-23p1

turnaround time - 13.2 without cs

average wait time - 12 without cs

c) SRTF:

the gantt chart:

- 0-2p1
- 2-5p2
- 5-7p3
- 7-8p5
- 8-15p4
- 15-23p1

turnaround time - 8.2 without cs

average wait time - 3.6 without cs

d) priority scheduling

the gantt chart:

- 0-10p1
- 10-13p2
- 13-20p4
- 20-22p3
- 22-23p5

turnaround time - 14.2 without cs

average wait time - 13 without cs

e) priority scheduling with preemption:

the gantt chart:

- 0-2p1
- 2-5p2
- 5-12p4
- 12-14p3
- 14-22p1
- 22-23p5

turnaround time - 11.8 without cs

average wait time - 7.2 without cs

2)

Yes, it always provide a faster access time than fetching from the disk because the heap is located at the main memory.

The main memory is DRAM and since accessing DRAM is much faster than accessing the disk (which must be accessed mechanically) we get that this way is better.

3)

Swapping pages algorithm is managed by the MMU/CPU, in hardware level, and since that it is very difficult to run sophisticated algorithms that demands sophisticated data structures so it has to use simple clock-oriented algorithms.

In contrast, managing disk buffer is os's responsibility so we have more possibilities in sense of memory and sophisticated data structures.

4)

An example when LRU is better than LFU:

Let the size of the cache be 4.

Initially the user reads the blocks 0,1,2 for a 1000 times.

Then the user reads the block 3,4 for infinity.

An example when LFU is better than LRU:

Let the size of the cache be 3.

Reading order of the blocks is - 1,2,2,3,3,3,4,1,2

An example when LFU and LRU are both not good:

Let the size of the cache be 3.

Reading order of the blocks is - 1,2,3,4,5,6,1,2,3

5)

The reason for that behavior is to "factoring out locality", that means that if we have a block/file that is very necessary for short period of time, and for any time else we don't mean to access it again. So with that kind of file, certain block will have very high number of reference - so it get stuck in the cache for a long time, although no one will access it again.

Part 2:

1)

As we saw in class, every access to file demand access to the I-node and directory instructor of every directory in the path. With that in mind, the following disk access needed:

1. Access to the i-node and instructor of the root directory.("\")
2. Access to the i-node and instructor of os directory.
3. Access to the i-node of "README" file.

Up until now we get 5 access to the disk, now we need to access the block with the single indirect (6) (since we can't access directly after the 20,480 byte), find the pointer that points

to 40000byte, read block's data (7), write (8) the changes we want to do in that block and at last update (9) the inode of this file.

Therefore, we get 9 access to the disk.

2)

First, we observe that seek and read are system calls, so we get 2 trap software interrupts.

Second, read function has to access the disk, so when the disk will finish his job it will send memory hardware interrupt to notify the os that the job has done.

Overall we get 3 interrupts.

3)

a)

We will use two level feedback queue, where every new jobs that came to the system gets into the first queue, that will run Round-Robin algorithm with quantum size=1. Each work that haven't finished her job (has to be type 2 job, with running time of 10 seconds) gets into second queue that will run SJF with priority and preemption for jobs that just came in to this queue.

This preemption and priority means that if a new job enters to the first queue, or moves from first to second queue, we stop the job that running right now and runs this new job.

b)

No, the preemption will cause many more context switch than first come first serve algorithm, and more context switch will cause more overhead time.

First come first serve will minimize overhead time because in that algorithm there are no context switch at all, hence the CPU always busy only with running process.