(+15) Suppose we have a round-robin scheduler in the OS kernel of a preemptively time sliced system, and three programs in its ready queue: P1, P2, and P3. P1 takes 10 seconds to complete, P2 takes 8.5 seconds, and P3 takes 3 seconds. Suppose the time slice is 2 seconds, and the overhead for OS context switching is 0.2 seconds. If P1 starts executing first at time 0, when do each of the programs finish executing? Draw a timeline of execution. What is the percentage overhead due to context switching? Assume that a process that finishes early transfers control back to the OS scheduler.

P1(10) > P2(8.5) > P3(3)>P1(8)>P2(6.5)>P3(1)>P1(6)>P2(4.5)>P1(4)>P2(2.5)>P1(2)>P2(0.5)>P1(0)

	0	2.2	4.4	6.6	8.8	11	12.2	14.4	16.6	18.8	21	23.2	23.7
P1	10			8			6		4		2	DONE	
P2		8.5			6.5			4.5		2.5		0.5	DONE
P3			3			1	DONE						

P1 finishes at 23.2s, P2 at 23.7s and P3 at 12.2s.
There are 11 context switches (assuming the first time slice is executed without a context switch) that amounts to 2.2 seconds of overhead. The total execution time is 23.7 seconds, so 2.2/23.7= 9.28% of overhead for context switch.

(+15) Repeat above for a batch mode multiprogrammed OS. Which system, preemptively multitasked or batch mode multiprogrammed finished execution of all three programs the fastest? Under what conditions might the reverse be true?

	0	2	4	6	8	10	10.2	12.2	14.2	16.2	18.2	18.7	18.9	20.9	21.9
P1	🟦	🟦	🟦	🟦	🟦	🟦									
P2							🟨	🟨	🟨	🟨	🟨	🟨			
P3													🟧	🟧	🟧

The batch mode OS outperformed the multitasked system in this task easily, finishing 1.8 seconds faster. The cause of this can clearly be seen when comparing the context switching overhead (2.2 s vs. 0.4s). Also, we're assuming that P1 and P2 aren't IO bound and idling in this case.

The reverse is true when we have programs that misbehave. Pre-emptively multitasked OS's force badly behaved programs to yield to other tasks while batch mode could get stuck endlessly.

(+15) Explain the differences between the four different kinds of exceptions found in an OS exception table. What is a software interrupt, and how does that differ from a hardware interrupt?

The 4 exceptions can be divided into 2 groups, unintentional and intentional interrupts. We have an unintentional abort interruption (which you can't recover from) and an unintentional interrupt (fault) caused by software failure (which can be recovered from). We also have an intentional hardware interrupt from IO devices interfacing with the system and and an intentional software interruption (trap) where we do a system call. Software interrupts are caused by failures in software or intentional interrupts whilst hardware interrupts are caused by failures in hardware or intentional IO interrupts.

(+10) What role does the jump table play in executing a system call?

The jump table or trap table handles the software interrupts (traps) and, through the handler, executes the required system call**.**

(+15) Explain in what way overlapping I/O with CPU processing is advantageous. Explain two ways that I/O can be overlapped with CPU execution and how they are each an improvement over not overlapping I/O with the CPU.

It frees up the CPU while the I/O device is processing a read/write. I/O devices block often and are much slower than the CPU, so blocking the CPU based on them would be disastrous. We can do this in two ways:

Direct I/O with polling: Whilst we're running continuous loops (busy waiting) to poll the IO devices for information, we can still make use of the CPU in parallel as opposed to getting blocked.

Interrupt driven I/O: Similarly, the CPU is free for utilization in parallel. The CPU sets up an interrupt handler and the I/O device interrupts on data transfer completion, so we don't have to wait on the I/O devices.

(+30) Describe each step of a write() operation to disk, from the application first calling a write() through the OS processing the write() to the final return from the write() call upon completion of the disk operation. Assume interrupt driven I/O operation. You may draw and label a figure if you'd like. Your answer should include components such as the device controller, interrupt handler, device handler, device driver, and any other OS components you deem appropriate to add.

Write() is often called to display an output. The process is as follows:

1. CPU calls the driver for write
2. CPU checks if the device is available
3. Create interrupt with interrupt handler
4. When device is ready, interrupt CPU
5. Call device handler, move data to required device
6. Interrupt CPU
7. CPU calls write command
8. Device writes, success
9. CPU success, done.