On my honor, I have not given, nor received, nor witnessed any unauthorized assistance on this work.

Print name and sign: ______________________________

Question:	1	2	3	4	5	Total
Points:	4	4	6	8	8	30
Score:						

1. (4 points) You write a Unix shell, but instead of calling `fork` and then `exec` to create a new process, you make a subtle mistake: you first call `exec` and then `fork`. How does this change the functioning of your shell (if it does). Explain your answer.

Solution: Doesn't work at all. It's important to understand that `exec` **replaces** the currently running process with the argument supplied to `exec`. A lot of people were under the impression that `exec` creates another process (a "child"). It does not! The currently running program is **replaced** by the arguments to `exec`. Thus, if a user called `exec("ls" -l -r)` from the main/original process, that process is now running the `ls` code. So the `fork()` call would never execute at all because `exec` doesn't return to the original code.

This was very obviously a confusing point to most people. I was generous with partial credit and gave credit for well-reasoned explanations. I generally only deducted points for flat-out incorrect statements or responses that were poorly worded or confusing to the point I couldn't follow your logic/thought process.

2. (4 points) Assume we have three jobs which arrive one after the other at time 0 in the following order:

- Job A which needs 10 seconds of CPU time
- Job B which needs 15 seconds of CPU time
- Job C which needs 10 seconds of CPU time

Assuming a SJF policy, at what time does B finish?

Solution: B finishes last. A runs first (because it arrived first), then C (because it's shorter than B) and then B. A takes 10, C takes 10, and then B takes 15, so B finishes at 35 seconds.

3. When a system call occurs, the hardware will redirect execution to special trap handler code in the OS. The OS must set up a trap table to inform the hardware of the location of all interrupt-handling routines.

   (a) (3 points) Explain when and how the trap table is initialized and who is responsible for the initialization.

   Solution: OSTEP Ch. 6, bottom of page 4 and top of page 5.

   Kernel is responsible for setting up the trap table at boot time. This table stores the locations of code for privileged operations. So if a user process requests a specific (privileged) operation, the code to run is NOT supplied by the process (this would be a huge security issue). Instead, the code can be looked up in the trap table, and the starting address of the code to be executed will be stored and can then be executed.

   Common mistake: confusing trap table with trap handler. Several people descibed the trap handler process (part b) instead of the trap table.

   (b) (3 points) When a user process executes a system call, it uses a a special trap machine language instruction (`syscall` on the x86-64 architecture, `trap` on the ARM8). What does this instruction do?

   Solution: OSTEP, Ch. 6, pgs 3-5 (chart on pg. 5 particularly relevant)

   The user process requests a `syscall` with a specific number to identify which syscall it needs. It does this via a `trap` instruction. The hardware elevates the mode from user mode to kernel mode, and the OS will execute the correct trap handler (via the lookup procedure described in part a). Eventually, when the handler has finished, the OS issues an instruction, `return-from-trap`. This causes the hardware to return the mode to user mode, and the user process can continue via the LDE protocol.

   A common error was not being clear about how the user process, OS, and HW interacted to handle the different parts of the process.

4. Below is a drawing showing a **round robin** scheduler with a 2 second time-slice. Three jobs: A, B, and C arrive in order, at time 0. Each of these jobs run for 6 time units. The scheduler itself takes 1 second to change jobs. This leads to a diagram:

```
SAA SBB SCC SAA SBB SCC SAA SBB SCC
012 345   8   1   1   1   2   2   2
              1   4   7   0   3   6
```

(a) (4 points) Calculate the average **response time** for this round robin implementation for jobs A, B, and C.

Solution: See Chapter 7.6: $time_{response} = time_{firstrun} - time_{arrival}$

$R_A = 1, R_B = 4, R_C = 7$

$R_{avg} = \frac{1+4+7}{3} = \frac{12}{3} = 4$

Common mistake: instead of using the time of arrival, several people used the finish time: $T_{response} = T_{firstrun} - T_{finish}$ which is incorrect. Response time is exactly that: the time the process spends waiting until it gets its first time cycle from the CPU.

(b) (4 points) Calculate the average **turnaround** time for this round robin implementation for jobs A, B, and C.

Solution: See Chpater 7.2: $T_{turnaround} = T_{completion} - T_{arrival}$

$T_A = 21, R_B = 24, R_C = 27$

$R_{avg} = \frac{21+24+27}{3} = \frac{72}{3} = 24$

Common mistake: Not realizing that that process A (for example) runs for 21 seconds! 0-1 is 1 second, 0-2 is 2 seconds. So A ends after 19-20 for a total of 21 seconds. Propagating this error lead many people to calculate an average response time of 23 instead of 24. (If this is confusing to you, think of it like counting elements in an array which starts at index 0. If you count the elements between indexes 0 and 20, you would have 21 elements, not 20.)

5. Scheduling algorithms can be classified along two axes: preemptive versus non-preemptive and size-based versus non-size-based.

   (a) (4 points) Explain the the terms *preemptive* and *size-based* in the context of scheduling algorithms.

   Solution:
   See OSTEP, Ch. 7, pgs 5 (preemptive) - scheduler can interrupt a job before it's done; not a requirement that job runs to completion

   Size based: scheduler takes into account the size (more formally, the service requirement) of a job (or portion of a job) when determining what to do next.

   Common mistake: stating that preemptive algos had something to do with size or priority. They may not. But a preemptive algo is one which can stop one job from running and start another one (i.e., force a context switch).

   (b) (4 points) Fill in the table below, placing the five scheduling algorithms you learned about in this sprint (FIFO, SJF, STCF, RR, and MLFQ) into the appropriate quadrant.

   Solution:

	size-based	non-size-based
preemptive	STCF (Ch 7, p7)	RR (Ch 7, p7-8) MLFQ (Ch 8)
non-preemptive	SJF (Ch 7, p4)	FIFO (Ch 7, p3)

   Common mistake: Putting MLFQ into the preemptive, size-based quadrant. MLFQ is not explicitly sized based. It is PRIORITY based. For example: a job starts in the highest priority queue regardeless of its size. The EFFECTS of the MLFQ policies mean that short jobs are handled quickly, but the algo. itself isn't explicitly sized based.