Instructions:

1. **Please write the answer for each question ONLY in the space provided below the question.**

2. **No electronic gadgets are allowed.**

[1] 1. Name three ways in which the processor can transition form user mode to kernel mode? Can the user execute arbitrary code after transition?

Solution: ① If a ~~user~~ program invokes a trap instruction via a systemcall, the processor switches to kernel mode → Mode switch

0.75

② If an illegal operation is performed, then processor transitions to kernel mode. → (privileged or, say, malicious)

③ If we want to switch from one user process to another, a context switch is necessary; one of its parts is a (user→kernel mode switch).

(After transition, the processor executes code based on the kernel stack and the interrupt descriptor table. User would have had to request code ~~execution before transition, or after~~ return-from-trap)

[2] 2. The following code is provided to you:

```
int main(int argc, char *argv[]) {
printf("a");
fork();
printf("b");
return 0;
}
```

a) Assuming `fork()` succeeds and `printf()` prints its outputs immediately (no buffering occurs), what are possible outputs of this program?

b) Assuming fork() might fail (by returning an error code and not creating a new process) and printf() prints its outputs immediately (no buffering occurs), what are possible outputs of the same program as above?

Solution: a) With successful fork, output (fully) is: abb

The two 'b's are each printed by parent and child, in an order that depends upon the system.

2

b) If fork() fails, no child process is created, so only one 'b' is printed, resulting in a (complete) output of: ab

2 3. Consider a parent process P that has forked a child process C in the program below.

```
int a = 5;
int fd = open(...) //opening a file
int ret = fork();
if(ret >0) {  // parent block
close(fd);
a = 6;
...
}
else if(ret==0) {    //child block
printf("a=%d\n", a);
read(fd, something);
}
```

After the new process is forked, suppose that the parent process is scheduled first, before the child process. Once the parent resumes after fork, it closes the file descriptor and changes the value of a variable as shown above. Assume that the child process is scheduled for the first time only after the parent completes these two changes.
(a) What is the value of the variable a as printed in the child process, when it is scheduled next? Explain.
(b) Will the attempt to read from the file descriptor succeed in the child? Explain.

Solution: (a) when fork() is called, child gets a copy of all variables, PC, etc. So, the value of 'a' printed in child process is $\boxed{a=5}$. (followed by newline). changes in local variables by the parent after the fork() don't affect the child.

2

(b) We are given that parent is scheduled first. So, the file descriptor is closed in the parent's content. For the child, the 'fd' ~~posix~~ ~~va~~ hasn't been closed. So, the attempt to read from fd will succeed.
↳ As long as file is opened in read mode, and file is valid (path, etc.)

1 4. Five jobs are waiting to be run. Their expected run times are 9, 6, 3, 5, and 'X'. In what order should they be run to minimize average response time (of course your answer will depend on X). Explain the different possible cases based on the value of 'X'.

Solution: For minimum average response time, Round Robin scheduling would be ideal. Since Round Robin deals with time slices, ~~let us assume~~ (which we can assume to be 1 here), the average response time is $\left(\frac{0+1+2+3+4}{5}\right)$ as long as X≥1. This is 2 time units.
~~time~~

What if we were not allowed to do time slicing? We ~~were~~ could take a look at how SJF or STCF works. For SJF, we would try to run the jobs in increasing order of runtimes.
If X>9, then avg. response time = $\frac{0+3+8+14+23}{5} = \frac{48}{5} = \boxed{9.6 \text{ time units}}$
For STCF, the same thing would occur since we assume that all jobs arrived at t=0

Could show more cases if ~~space~~ it was more feasible here.

2 5. Consider the following preemptive-priority-scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the processor (in the ready queue, but not running), its priority changes at a rate A [i.e P(t) = P0 +A * (t - t0) where t0 is the time at which the process joins the ready queue] ; when it is running, its priority changes at a rate B. All processes are given a priority of zero when they enter the ready queue. The parameters A and B can be set to give many different scheduling algorithms.

(A) What is the algorithm that results from B > A > 0 ? Explain how

(B) What is the algorithm that results from A < B < 0 ? Explain how

Solution: (A) If B>A>0, priority increases faster in running state than in ready state. This tends towards first come first serve (FCFS)

↳ Since running processes get more-and-more priority.

↳ In the ready queue, earlier processes accumulate more priority.

−1

(B) If A<B<0, priority decreases faster for running processes, than those of the ready queue. This tends towards Round Robin since running processes are more likely to get "kicked out", and resumed later on.

2 6. The Multi-level Feedback Queue (MLFQ) is a fancy scheduler that does lots of things. Are the below given things TRUE/FALSE about the MLFQ approach? Give reasons for your answer. Please note, merely stating TRUE or FALSE without justification/ reason will not fetch any marks.
a) MLFQ learns things about running jobs
b) MLFQ starves long running job
c) MLFQ uses different length time slices for jobs
d) MLFQ forgets what it has learned about running jobs sometimes

Solution: a) TRUE, since MLFQ decreases priority level of jobs that run until the end of their time slice, and doesn't do the same for ones that relinquish CPU (I/O bound). Hence, MLFQ learns about running jobs.
↳ (ie their CPU-intensive/ I/O bound nature)

−1.25

b) FALSE, subject to the fact that, on a periodic/regular basis, all jobs are sent to topmost queue → [This is Rule (5) in the textbook.]

c) FALSE, time slices stay the same, and after each time slice, a decision is made based on the MLFQ, of which job to run.
↳ [Note that time slice differs from CPU burst]

d) TRUE, this "forgetting" follows from (6 b), where the priorities are periodically reset. This prevents long jobs from starvation.

~~Same~~