<u>Question 11</u>

My answer: **C**

Right answer: **A**

The code was
```
int main() {
                    fprintf(stderr, "abc\n");
                    fprintf(stdout, "234\n");
            }
```
and so I assumed that because both file descriptors 1 and 2 went to the screen that we would see
```
abc
234
```
as output. The command was **.a/out | grep 2** and I assumed that looked for any line with "2" in it, which would be the 234 line alone, so I picked **C-234** as my answer. But now I realize that grep only looks for stuff in lines of the standard output and that all lines of the standard output are printed when grep is used, so the correct answer is
```
A-abc
  234
```

What grep does is
```
     ___stderr___________
./a [___stdout___grep____] put these together -> output
```
but I incorrectly assumed grep did
```
     ___stderr___
./a [___stdout___] put these together -> grep -> output
```

Question 21

My answer: **D**

Right answer: **B**

The code was
```
void doit() {
        char *args[] = { "/bin/echo", "hello", NULL };
        char *newenviron[] = { NULL };
        if (fork() == 0) {
                execve(args[0], args, newenviron);
                printf("hello\n");
        }
}
int main() {
        doit();
        printf("hello\n");
}
```
and I assumed that what happened was
```
call main___call doit___fork()___parent___return from doit___print hello___exit
                          |
                          child___execve print hello___return from doit___print hello___exit
                                       | |
                             ._________. L_________________.
                             |                             |
                             echo hello___return from execve
```
which would be 4 instances of print hello, so I chose **D-4** but what actually happens is
```
call main___call doit___fork()___parent___return from doit___print hello___exit
                          |
                          child___execve
                                       |
                             ._________.
                             |
                             echo hello___exit execve if there are no errors
```
which would be 2 instances of print hello, so the correct answer is **B-2** instead.

Question 30

My answer: **A**

Right answer: **D**

At t=0, everything gets set up and a child is forked. When 0<t<10, both are sleeping. At t=10, someone gives the foreground job, which is the parent, the ***^C*** thing which sends it a **SIGINT** and because the child is in the same process group as the parent, the child gets a **SIGINT** also so since the **SIGINT** handler is set up before the fork, both the parent and child go into it. Parent attempts to sleep for 10 seconds while the child attempts to sleep for 20 seconds; when 10<t<20, both are sleeping. At t=20, the parent wakes up and gives a **SIGTERM** to the child, and since this is different signal, the handler of the child is interrupted. Before the fork **SIGINT** handler is set up to also handle **SIGTERM** so child enters this handler and attempts to sleep for 20 seconds; at this same time, parent attempts to sleep for 10 seconds. So when 20<t<30, both are sleeping. At t=30, the parent wakes up and tries to reap the child without the **WNOHANG** and child is still sleeping, so parent blocks. When 30<t<40, child is still sleeping and parent is still blocking waiting for the child to be done. At t=40, the child calls exit and is done, and the parent can be done waiting, so it has now fully reaped child and the wait call returns the pid of child, so parent attempts to sleep for 20 seconds. So if 40<t<50, such as if t=45, child does not exist and parent is sleeping, so **D-parent is running/sleeping and child no longer exists** is correct. I correctly knew at t=0, stuff was set up and the child was forked. I correctly knew that when 0<t<10, both were sleeping. I correctly knew that at t=10, someone gave the foreground job, which was the parent, the ***^C*** thing which sent it a **SIGINT** and since child was in the same process group as parent, child gets a **SIGINT** also so because **SIGINT** handler was set up before the fork, both the parent and child go into it. I correctly knew the parent attempts to sleep for 10 seconds while the child attempts to sleep for 20 seconds, so when 10<t<20, both were sleeping. I correctly knew that at t=20, the parent wakes up and gives a **SIGTERM** to the child, and since this was a different signal, the handler of the child was interrupted. Before the fork the **SIGINT** handler was set up to also handle **SIGTERM** so the child enters this handler and attempts to sleep for 20 seconds, and at this same time, the parent attempts to sleep for 10 seconds. I correctly knew that when 20<t<30, both were sleeping. At t=30, I knew the parent wakes up and tried to reap the child without **WNOHANG** and child was still sleeping, so the parent blocked. I also knew that when 30<t<40, child was still sleeping and parent was still blocking waiting for child to be done. But I failed to notice the exit call after this so I incorrectly assumed that at t=40, child exited the signal handler it was doing and went back to the old one where there were 10 seconds of sleep left. So I assumed when 40<t<50, child was sleeping and parent was blocking, waiting for child, so I put **A-parent and child are both running/sleeping**

Question 32

My answer: **D**

Right answer: **C**

With ***setpgid(getpid(), getpid());*** between lines 29 and 30 as well as line 9 being changed from ***if (waitpid(pid, &status, 0) > 0) {*** to the line ***if (waitpid(pid, &status, WNOHANG) > 0) {*** we now have different process groups for the parent and child after the fork. This fork happens at t=0 when everything gets set up, which I correctly knew. So I correctly knew that when 0<t<10, both the parent and the child were sleeping. Then at t=10, I correctly knew that someone gave the foreground job, which was the parent, the ***^C*** thing which sent it a **SIGINT** and since child was in different process group from parent, child does not receive it and was still sleeping. So I correctly knew that only the parent would go into the **SIGINT** handler and would attempt to sleep for 10 seconds. So I correctly knew when 10<t<20, the parent was sleeping in the handler and the child was sleeping in main, and I correctly knew at t=20 that the parent would send a **SIGTERM** to the child and since the **SIGINT** handler was also set up to handle **SIGTERM** the child would go there. So I correctly knew the child would then attempt to sleep for 20 seconds, while the parent would attempt to sleep for 10 seconds. So I correctly knew that when 20<t<30, both parent and child were sleeping. Then I correctly knew at t=30 that the parent would attempt to reap the child if done but that the child would not be done, so the wait call would return 0 because of **WNOHANG** and that that would get the parent to send a **SIGKILL** to the child which would have the child terminate immediately, but I incorrectly assumed that because there was a wait call, the parent would automatically reap the child whenever it was exited or terminated, but in reality, **WNOHANG** prevents any reaping if there are no children done or terminated, meaning the child would not be reaped. I correctly knew, however, that at t=30, the parent would attempt to sleep for 10 seconds and that the **SIGKILL** would terminate the child. Therefore I correctly knew that when 30<t<40, such as when t=35, the parent would be sleeping, but I incorrectly assumed the child would no longer exist, when really, the child would be terminated but exist, making it a zombie, so the correct answer is **C-parent is running/sleeping and child is terminated/zombie** but my incorrect assumptions made me mark **D-parent is running/sleeping and child no longer exists**

Question 39

My answer: **D**

Right answer: **B, D, F**

I correctly assumed that section 4 needed the line ***close(pipefd[PIPE WRITE]);*** because the second child was only reading. I incorrectly assumed that section 6 did not need the line ***close(pipefd[PIPE WRITE]);*** as I incorrectly assumed that the parent having this file descriptor open in the parent would cause no problems, although it actually does cause problems, so section 6 needs this line. I incorrectly assumed that section 2 did not need the line ***close(pipefd[PIPE WRITE]);*** because I was thinking of how the first child has to write to the file the descriptor pipefd[PIPE_WRITE] describes, but in reality, the first child is simply writing to its file descriptor 1, and so it needs the line ***close(pipefd[PIPE WRITE]);*** and I correctly assumed that section 1 didn't need the line ***close(pipefd[PIPE WRITE]);*** because the pipe would have just barely have been created, meaning that this would close that file for both children when they were forked. I correctly assumed that section 3 did not need the line ***close(pipefd[PIPE WRITE]);*** because this was just a section to have a line about exiting with errors, and I correctly assumed that section 5 also did not need it either, since this was also just a section about adding a line to exit with errors. So my assumptions caused me to answer only **D-section 4** while the 3 correct answers were **B-section 2** and **D-section 4** and **F-section 6**