Name: Gurumanie Singh Dhiman
Section: A
University ID: 911192340

Lab 2 Report Summary:

I think that this lab was mostly about figuring out and truly understanding how processes are created, managed and killed in Unix based systems. We started the lab by using fork(), ps, and execl, and saw how new processes come into existence, how they're assigned unique process IDs, and how they interact with their parent processes. I also noticed that fork() returns different values depending on whether the code is running in the parent or child. Removing sleep() revealed how parents can exit quicker and then leave their children process orphaned, in which case systemd adopts them. Additionally, I also noticed that the kernel scheduler time-slices CPU execution between processes, sometimes causing output to overlap and also appear truncated at some times.

The later experiments mainly focused on synchronization and termination. Using wait() just to make sure that the parent didn't finish until the child was also done, which gave more predictable output. I also saw how kill can end a child process which is in a visibly infinite loop, and how return values communicate success or failure in a minimal way (like the true and false commands). Finally, working with exec showed me that if the call succeeds, the old process image is replaced entirely, which explains why code after the exec line doesn't run unless exec fails. All in all, this lab gave me a clearer picture of process control, scheduling, and communication, while also showing me some abrupt and weird behavior like truncated outputs and an unexpected adoption by systemd that made the lab less about theory and more about real-world system behavior.

Lab Questions:

3.1:
6pts In the report, include the relevant lines from "ps -l" for your programs, and point out the following items:

- output
- process name
- process state (decode the letter!)
- process ID (PID)
- parent process ID (PPID)

```
bash-5.1$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 460819   5514    5495  0  80   0 - 58356 do_wai pts/0    00:00:00 bash
0 S 460819   7861    5514  0  80   0 -   657 hrtime pts/0    00:00:00 main
0 S 460819   7873    5514  0  80   0 -   657 hrtime pts/0    00:00:00 main
4 R 460819   7954    5514  0  80   0 - 56437 -      pts/0    00:00:00 ps
bash-5.1$
```

Output: Given above

Process Name (both): main

Process State (both): S - Interruptible Sleep (waiting for an event to complete)

Process ID for 1st main (PID): 7861

Process ID for 2nd main (PID): 7873

Parent Process ID (PPID) (both): 5514

2pts Repeat this experiment and observe what changes and doesn't change.

```
bash-5.1$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 460819   8994    8975  0  80   0 - 58099 do_wai pts/0    00:00:00 bash
0 S 460819   9117    8994  0  80   0 -   657 hrtime pts/0    00:00:00 main
0 S 460819   9136    8994  0  80   0 -   657 hrtime pts/0    00:00:00 main
4 R 460819   9279    8994  0  80   0 - 56437 -      pts/0    00:00:00 ps
bash-5.1$
```

Output: Given above

Process Name (both): main

Process State (both): S - Interruptible Sleep (waiting for an event to complete)

Process ID for 1st main (PID): 9117

Process ID for 2nd main (PID): 9136

Parent Process ID (PPID) (both): 8994

The only changes are the PID and PPID. I noticed the PPID only changes if you close your terminal and try with a new one. Using the same terminal would give you the same PPID during both experiments.

2pts Find out the name of the process that started your programs. What is it, and what does it do?

```
bash-5.1$ ./main &
[1] 7673
bash-5.1$ Process ID is: 7673
Parent process ID is: 5697
^C
bash-5.1$ ps
    PID TTY          TIME CMD
   5697 pts/0    00:00:00 bash
   7673 pts/0    00:00:00 main
   7691 pts/0    00:00:00 ps
bash-5.1$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 460819   5697    5678  0  80   0 - 58611 do_wai pts/0    00:00:00 bash
0 S 460819   7673    5697  0  80   0 -   657 hrtime pts/0    00:00:00 main
4 R 460819   7710    5697  0  80   0 - 56437 -      pts/0    00:00:00 ps
bash-5.1$ I am awake.
^C
[1]+  Done                    ./main
bash-5.1$ ps
    PID TTY          TIME CMD
   5697 pts/0    00:00:00 bash
   9150 pts/0    00:00:00 ps
bash-5.1$ ./main &
[1] 9158
bash-5.1$ Process ID is: 9158
Parent process ID is: 5697
^C
bash-5.1$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 460819   5697    5678  0  80   0 - 58611 do_wai pts/0    00:00:00 bash
0 S 460819   9158    5697  0  80   0 -   657 hrtime pts/0    00:00:00 main
4 R 460819   9170    5697  0  80   0 - 56437 -      pts/0    00:00:00 ps
bash-5.1$ I am awake.
^C
[1]+  Done                    ./main
```

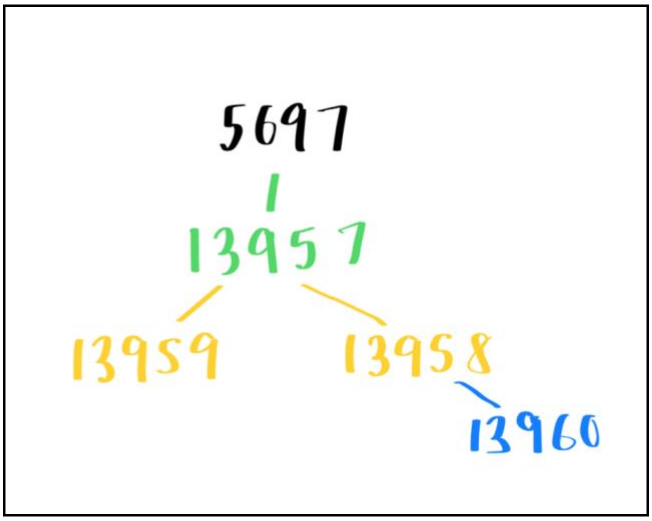
As seen above, based on the two times that main was run, both times the parent process (PPID) was listed as 5697, which as we can see from the output above, is the bash process. Bash is an shcompatible command language interpreter that executes commands read from the standard input or from a file. In our case, bash works as a command language interpreter to execute the "./main &" command that we enter to run our main file in the background.

3.2:

1pt Include the output from the program

```
bash-5.1$ ./main
Process 13957's parent process ID is 5697
Process 13959's parent process ID is 13957
Process 13958's parent process ID is 13957
Process 13960's parent process ID is 13958
bash-5.1$
```

4pts Draw the process tree (label processes with PIDs)



3pts Explain how the tree was built in terms of the program code

The program code calls fork() back to back in lines 1 and 2 right after we start main(). We know that process ID 5697 is bash, which then creates process ID 13957 which is the main(), and then the process ID's 13959 and 13958 are both the fork() calls.

2pts Explain what happens when the sleep statement is removed. You should see processes reporting a parent PID of 1. Redirecting output to a file may interfere with this, and you may need to run the program multiple times to see it happen.

```
bash-5.1$ ./main
Process 13244's parent process ID is 8994
Process 13246's parent process ID is 13244
Process 13245's parent process ID is 3306
bash-5.1$ Process 13247's parent process ID is 13245
```

When the sleep statement is removed, the child process is left orphaned because the parent finishes execution earlier. In our case, 8994 is bash, which creates 13244 which is the main(). The children in my example are 13245 and 13246. However, in our case as I mentioned, the parent process exits much sooner than the child, which leaves the child in an orphaned state. What this means is that our child process is then adopted by 3306 (which is systemd). I am not sure if I am wrong on this because the question says we should see a PID of 1 but I did not. I looked into what PID 3306 reported as, and in my case that was systemd.

3.3:

2pts Include the (completed) program and its output

```
int main() {
	int ret;
	ret = fork();
	if (ret == 0) {
		/* this is the child process */
		printf("The child process ID is %d\n", getpid());
		printf("The child's parent process ID is %d\n", getppid());
	} else {
		/* this is the parent process */
		printf("The parent process ID is %d\n", getpid());
		printf("The parent's parent process ID is %d\n", getppid());
	}
	sleep(2);
	return 0;
}
```

```
bash-5.1$ ./main
The parent process ID is 20915
The parent's parent process ID is 20009
The child process ID is 20916
The child's parent process ID is 20915
```

4pts Speculate why it might be useful to have fork return different values to the parent and child. What advantage does returning the child's PID have?

It is useful to have fork() return different values to the parent and the child because otherwise it is difficult to tell them apart. Because fork() returns the childs process ID to the parent, it is easier for the parent to know and keep track of which process was created under it.
After a fork(), we get two nearly identical processes that are created, so we need a way to tell them apart from one another. Since we know child processes get a 0 and the parent process gets the child's ID, this makes it easy to tell which is which. Furthermore, since the parent has access to the child's process ID, it is able to keep track of it and know which child process is which and does not need to guess.

3.4:
2pts Include small (but relevant) sections of the output

```
Parent: 499998
Parent: 499999
 499456
Child: 499457
Child: 499458
```

```
Parent: 499471
Parent: 499472
Parent:d: 498871
Child: 498872
Child: 498873
```

```
Parent: 498925
Parent: 498926
Parenild: 498286
Child: 498287
Child: 498288
```

4pts Make some observations about time slicing. Can you find any output that appears to have been cut off? Are there any missing parts? What's going on (mention the kernel scheduler)?

Yes, I found multiple lines that have been cut off and even noticed some lines that were mixed with some other lines. The above screenshots are only some of the relevant screenshots and examples of the output. There were many other instances of such cut off lines and also missing parts as seen in the first screenshot above.

What is happening is that the kernal scheduler is switching between the processes for the child and the parent and in doing so it creates a break or a cut off when it switches but this discrepency only happens sometimes. This was discussed in class and I believe it is mainly due to the scheduler allocating a specific amount of time for each process and switching to the next when the time is up then rotating back to the first process when the second process runs out of its allocated time.

3.5:

6pts Explain the major difference between this experiment and experiment 4. Be sure to look up what wait does (*man 2 wait*).

```
Child: 499998
Child: 499999
Child ends
Parent starts
Parent: 0
Parent: 1
```

The main difference between this experiment and experiment 4 is that this one consists of print statements to indicate when a child process starts and ends, when a parent process starts and ends, and the biggest difference that makes an impact in the output is the wait (NULL) statement. The 2 wait man page states that the parent waits until the child is done processing.
This is clearly why the outputs are very different between experiments 4 and 5. Due to the 2 wait command, the processes are allowed to fully complete before the next one begins. This is why the child process goes first, and only once it ends, does the parent process begin.

3.6:

2pts The program appears to have an infinite loop. Why does it stop?

```
bash-5.1$ ./main
Parent sleeping
Child at count 1
Child at count 2
Child at count 3
```

```
Child at count 978
Child at count 979
Child has been killed. Waiting for it...
done.
bash-5.1$
```

The program stops because we have a kill function in the else loop. What this is mainly doing is, it allows the child to run for 10 seconds, after which the kill function from the else takes over and stops the child process. Since the child process is the only one in the forever while loop, and it is terminated forcefully, we are only left with the parent (else) loop, which is not an infinite loop and terminates when it is completed.

4pts From the definitions of *sleep* and *usleep*, what do you expect the child's count to be just before it ends?

The child’s count just before it ends is 979 (from my output). However, running it multiple times we notice that the number fluctuates slightly from 1000 (which is the theoretical value it should reach).

2pts Why doesn't the child reach this count before it terminates?

I think there may be a few reasons for this count not reaching the theoretical amount. I strongly feel that one of them is the fact that all these print statements are not instant and take time to be output to the terminal. Since the print statements need to be sent out to the terminal, the execution of the process running is slowed down ever so slightly.

Another reason that this might be happening could be due to the usleep and sleep functions not be exactly the time that they say they are. If we read into the details for both usleep, it states that they guarantee a minimum of 10,000 microseconds, but this also means that depending on a multitude of reasons, it could take longer.

3.7:

8pts Read the man page for *execl* and relatives (*man 3 exec*). Under what conditions is the *printf* statement executed? Why isn't it always executed?
(consider what would happen if you changed the program to execute something other than */bin/ls*.)

```
bash-5.1$ ./main
main  main.c  main.o  outputlab2-3.4.txt  outputlab2-3.5.txt
bash-5.1$
```

The printf statement is only executed if our call to exec fails. This is because the man page states that if exec fails, it will return -1 and set errno, then continue to the next line.

The printf isnt always executed because if exec is successful, the image of the process is replaced with a new program. This in turn would not allow it to have access or go back to the old code.

If we tried to change the program to execute something else other than /bin/ls, for example /bin/hello and it does not exist, the exec call will fail and our printf statement would be executed. However, if /bin/hello does exist, then our printf statement is not executed and instead, whatever function hello has will be executed.

3.8:

2pts What is the range of values returned from the child process?

```
bash-5.1$ ./main
Child exited with signal 15
bash-5.1$ ./main
Child exited with status 199
bash-5.1$ ./main
Child exited with status 219
bash-5.1$ ./main
Child exited with signal 15
bash-5.1$ ./main
Child exited with signal 15
bash-5.1$ ./main
Child exited with status 252
bash-5.1$ ./main
Child exited with status 100
bash-5.1$ ./main
Child exited with status 233
bash-5.1$ ./main
Child exited with status 11
bash-5.1$ ./main
Child exited with status 239
bash-5.1$ ./main
Child exited with status 45
bash-5.1$ ./main
Child exited with signal 15
bash-5.1$ ./main
Child exited with status 52
bash-5.1$ ./main
Child exited with status 52
bash-5.1$ ./main
Child exited with status 43
bash-5.1$ ./main
Child exited with signal 15
```

```
bash-5.1$ ./main
Child exited with status 129
bash-5.1$ ./main
Child exited with status 129
bash-5.1$ ./main
Child exited with status 245
bash-5.1$ ./main
Child exited with signal 15
bash-5.1$ ./main
Child exited with signal 15
bash-5.1$ ./main
Child exited with signal 15
bash-5.1$ ./main
Child exited with status 93
bash-5.1$ ./main
Child exited with status 93
bash-5.1$ ./main
Child exited with status 133
bash-5.1$ ./main
Child exited with signal 15
bash-5.1$ ./main
Child exited with status 212
bash-5.1$ ./main
Child exited with status 113
bash-5.1$ ./main
Child exited with status 61
bash-5.1$ ./main
Child exited with status 212
bash-5.1$ ./main
Child exited with status 179
bash-5.1$ ./main
Child exited with status 179
```

In my scenario, where I ran the program multiple times, I noticed that the range of values for signal stay constant at 15 and status fluctuates from 11 to 252.

However, after looking into the code and what it is doing a little more, I realized the value for signal remains constant at 15 because signal 15 represents SIGTERM. Basically, if we put in a different signal instead of SIGTERM in the code then it would represent that signal and give its value.

And the values for status could range from 0 to RAND_MAX which is different in different implementations. In our case if I had to guess, it would represent the largest 32 bit signed integer which is 2,147,483,647. However, additional to this information, the reason why none of the values seem to go that high and often stay at around a max of the high 200s is because the value that the parent gets is actually truncated to only show the lowest 8 bits of the child's return value.

2pts What is the range of values sent by child process and captured by the parent process?

The parent only see the values from 0 to 255 returned by the child. This is because the parent process receives the status of the child using WEXITSTATUS(status) which only guarantees to hold the lowest 8 bits of the return value. Which means in reality, the parent only receives what the result from the rand() function modded with 256.

2pts When do you think the return value would be useful? Hint: look at the commands *true* and *false*.

From reading about the commands true or false, they use a simple convention where true exits with a status 0 which basically means success, whereas false exits with a status 1 which means failure. I think return values are especially useful if there is no need to provide a full log or a lot of data on exactly what happened. If we just need to know if a program passed or failed, or some third situation, we can use return values to simplify things.