BLG312E - Computer Operating Systems
Homework 1

Volkan Ilbeyli

March 9, 2013

1 Answer to Question 1

The output is:

```
Parent: My process ID: 4283
Parent: My child's process ID: 4284
   Child: My process ID: 4284
   Child: My parent's process ID: 4283
Parent: Terminating...
```

First, the parent process starts to execute. It prints its own PID using `getpid()` function. As two identical processes start after the `f=fork()` function (except for their PIDs), the parent process gets its child's PID using the `f` variable. The `wait(NULL)` line helps the child process to terminate before its parent terminates. If the parent did not `wait(NULL)`, the child process might have printed its parent's PID as `0` since the parent process might have terminated before the child process prints its parent's PID. The child process starts running after the `wait(NULL)` line is reached and prints its parent's PID using `getppid()` function and its own PID using `getpid()` function. At this process, the value of `f` is 0 since the child process doesnt have a child process. The `sleep(1)` line is not as crucial as the `wait(NULL)` line as it just postpones the output only 1 millisecond. When the child process `exit(0)`s the parent process continues to execute where it was left, at the `wait(NULL)` line. The last line simply states that the parent is terminating.

The difference between the child PID and the parent PID is one, simply stating that a *child* process is created *after* the parent, thus having a PID greater than its parents only by 1.

2 Answer to Question 2

The output is:

```
Parent: My parent's process ID: 3021
Parent: My process ID: 5450
Parent: My child's process ID: 5451
   Child: My process ID: 5451
   Child: My parent's process ID: 5450
Parent: Terminating...
```

The difference between the PIDs of parent(5450) process and the parent of parent(3021) process is larger compared to a simple, 1 child forking situation. Judging the large difference and the parent of parent process' PID always being 3021, I can conclude that the PID with 3021 is the process which is started with the operating system and is responsible for executing my `main()` function.

3 Answer to Question 3

Below are 2 different outputs when the `wait(NULL)` line is omitted:

```
Parent: My parent's process ID: 3021
Parent: My process ID: 5840
Parent: My child's process ID: 5841
   Child: My process ID: 5841
Parent: Terminating...
   Child: My parent's process ID: 5840

Parent: My parent's process ID: 3021
Parent: My process ID: 5838
Parent: My child's process ID: 5839
Parent: Terminating...
   Child: My process ID: 5839
   Child: My parent's process ID: 1
```

The first run seems like a regular run, alike with the ones which have `wait(NULL)` line. But the second run shows that the parent process did not wait for its child to `exit(0)` and terminated before its child terminated. Therefore when child process tried to print its parent's PID, it printed 1 instead of its parents PID, 5838. A PID value of 1 indicates that there is no parent process associated with that process and it is the value of the `init` process, which is the first process started during the boot.

4 Answer to Question 4

Output:

```
Parent: global before assignment = 5
Parent: global = 0
   Child: global before assignment = 5
   Child: global = 10
   Child: Terminating...
Parent: global after child is terminated = 0
Parent: Terminating...
```

The global variable value was `5` before the `fork()` was called. Since the fork *clones* the process, global variable was cloned to the child process with the same value. The distinction here is the newly cloned variable became the *child's global variable* while the parent process maintained *its own global variable*. Thus, assigning different values to the variable did not affect the each other process' global variable.

5 Answer to Question 5

Output:

```
Parent: global address: 25804816
Parent: global before assignment = 5
Parent: global = 0
   Child: global address: 25804816
   Child: global before assignment = 5
   Child: global = 10
   Child: Terminating...
Parent: global after child is terminated = 0
Parent: Terminating...
```

When a process is forked, a location from the main memory is allocated to the child process, **however,** due to virtualization, they look like they are using exactly the same memory address while in fact they don't. The above situation illustrates this phenomena. Even though a global variable is declared, when forked, the child process will have the cloned variables and the pointers will seem to have the same address location due to virtualization. In this situation, changes made to the location pointed by the global pointer in the child process will not affect the memory location allocated by the global pointer in the parent process. In other words, the child and the parent process have their *own* global pointers which seem to have the same address. In physical layer, they don't. That is due to cloning everything to the child when `fork()` is invoked.

6 Answer to Question 6

This question basically compares the threads against processes. Below is the output:

Processes:

```
0: Value= 1
1: Value= 1
Main: Created 3 procs.
2: Value= 1
```

Threads:

```
0: Value= 1
1: Value= 2
main(): Created 3 threads.
2: Value= 3
```

The most important difference between a process and a thread is: A process group get to have the same resources as the parent process, i.e., the variables, memory amount, etc, everything is cloned to the child processes. However for threads: they use the same resources, i.e. they *share* the resources. A change in one variable affects all the threads using that variable. Concluding from the output: All the three processes had their cloned variable incremented by one, meaning that a change in one variable did *not* affect the other while in the threads, a change in on variable affected the others as the variable was incremented three times, all by each thread, which shows us that the resources are *shared* among threads while they are *not shared* among processes.