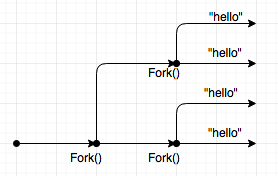
Nathan Bender

CS 283

H6 – Processes

3/5/17

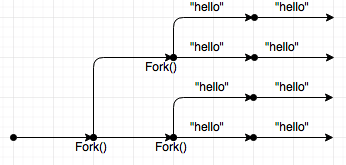
**8.11**



Hello will print four times.

The for loop that encases the Fork() command executes twice. The first time, the fork creates a parent and child process. The second time, the Fork() creates a parent and children process for each of the previous parent and child processes. So the program now has four processes, and each of the processes prints “hello” to the console.

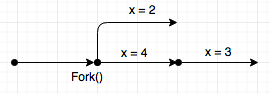
**8.12**



Hello will print 8 times.

Main first calls the doit function. Like the previous example, the doit function forks twice, first creating a parent and child process, and then creating another process for each of the parent and child process. Therefore, there are then four processes running. In the doit function, each of these processes prints “hello” to the screen and returns (total of four times). Then, once back in the main function, there are still four processes, and each of the processes prints hello once (total of four times). So in total, “hello” is printed to the screen eight times.

**8.13**



One possible output is, if we follow the parent process first and then the child process:

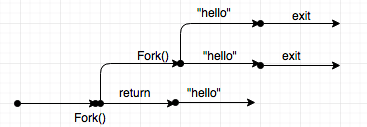
x = 4

x = 3

x = 2

Fork() creates two processes, a parent and a child. Each has their own copy of x, each of which starts at 3. In the case above, the parent is executed first. X has an initial value of 3, so when it triggers the first if statement, due the pid not being zero, x is printed as 4, since ++x (3 + 1). Then, in the next if statement, x is decremented using --x so x is printed as 3 (4 – 1). Finally, once the child is run, only the last print statement is ran, decrementing x (--x). Since the child process has it’s own copy of x, so x = 2 is printed (3-1).

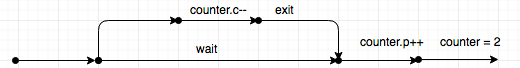
**8.14**



Hello will print 3 times.

The main function first calls the doit function. The doit function first calls fork, creating a parent and child process. Each process then executes the if statement. The parent process, since the pid is not 0, hits the return statement and goes back to the main function. Here, the word “hello” is printed to the screen and the process is done. The child process in the doit function, does have a pid of 0, so it goes inside the if statement. The if statement has another fork, so the child becomes a parent to another child process. There are now 2 processes after the second fork, and each prints “hello”. Each process then calls exit(0) which closes the processes and the function does not return.

**8.16**



The output will be:

Counter = 2

There is a global variable called counter, which is declared as being equal to one. However, even being global, each process has their own copy of the counter variable. On line 6, the main program forks a child and parent process. The child process decrements their copy of counter, and exits. So their value for counter is equal to 0. The parent process, however, waits until the child is finished, and then increments their own counter. Since they have their own copy of the counter variable, counter has the value of 2 (1 + 1). Therefore, “counter = 2” is printed to the screen.

**8.23**

The handler blocks the signals while the parent processes the signal that it was sent by the child process. The 2nd signal that is received becomes the pending signal, and the third signal that is sent is dropped. This is due to the fact that the signal handlers do not queue the signals that they receive. They only have a single bit that states whether or not they have a signal waiting for them. So while the handler is processing the first signal, the second signal is received, and the bit is set. When the third signal is sent, the bit is already set, so nothing happens, and the signal is lost.

Here is a breakdown of what is occurring:

* The main program registers the signal handler with the name “handler” for the signal SIGUSR2
* The program forks, and calls kill using the signal SIGUSR2 five times.
* The first time the child calls kill, the handler is triggered immediately.
* The second time the child calls kill, the handler is already processing a signal, so it sends the bit that a signal is pending.
* The third time the child calls kill, the handler is already processing a signal, and there is already a signal pending, so the signal is dropped. This is possible due to the handler sleeping for 1 second, so the handler is not finished in time to receive the other signals.