

CSC 369 Worksheet 2

August 16, 2020

Source: [link](#)

1 Homework (Simulation)

1. Run `./fork.py -s 10` and see which actions are taken. Can you predict what the process tree looks like at each step? Use the `-c` flag to check your answers. Try some different random seeds (`-s`) or add more actions (`-a`) to get the hang of it.
2. One control the simulator gives you is the fork percentage, controlled by the `-f` flag. The higher it is, the more likely the next action is a fork; the lower it is, the more likely the action is an exit. Run the simulator with a large number of actions (e.g., `-a 100`) and vary the fork percentage from 0.1 to 0.9. What do you think the resulting final process trees will look like as the percentage changes? Check your answer with `-c`.
3. Now, switch the output by using the `-t` flag (e.g., run `./fork.py -t`). Given a set of process trees, can you tell which actions were taken?
4. One interesting thing to note is what happens when a child exits; what happens to its children in the process tree? To study this, let's create a specific example: `./fork.py -A a+b,b+c,c+d,c+e,c-`. This example has process 'a' create 'b', which in turn creates 'c', which then creates 'd' and 'e'. However, then, 'c' exits. What do you think the process tree should look like after the exit? What if you use the `-R` flag? Learn more about what happens to orphaned processes on your own to add more context.
5. One last flag to explore is the `-F` flag, which skips intermediate steps and only asks to fill in the final process tree. Run `./fork.py -F` and see if you can write down the final tree by looking at the series of actions generated. Use different random seeds to try this a few times.
6. Finally, use both `-t` and `-F` together. This shows the final process tree, but then asks you to fill in the actions that took place. By looking at the tree, can you determine the exact actions that took place? In which cases can you tell? In which can't you tell? Try some different random seeds to delve into this question.

2 Homework (Code)

1. Write a program that calls `fork()`. Before calling `fork()`, have the main process access a variable (e.g., `x`) and set its value to something (e.g., 100). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of `x`?
2. Write a program that opens a file (with the `open()` system call) and then calls `fork()` to create a new process. Can both the child and parent access the file descriptor returned by `open()`? What happens when they are writing to the file concurrently, i.e., at the same time?
3. Write another program using `fork()`. The child process should print “hello”; the parent process should print “goodbye”. You should try to ensure that the child process always prints first; can you do this without calling `wait()` in the parent?
4. Write a program that calls `fork()` and then calls some form of `exec()` to run the program `/bin/ls`. See if you can try all of the variants of `exec()`, including (on Linux) `exec1()`, `execle()`, `execvp()`, `execvp()`, and `execvpe()`. Why do you think there are so many variants of the same basic call?
5. Now write a program that uses `wait()` to wait for the child process to finish in the parent. What does `wait()` return? What happens if you use `wait()` in the child?
6. Write a slight modification of the previous program, this time using `waitpid()` instead of `wait()`. When would `waitpid()` be useful?
7. Write a program that creates a child process, and then in the child closes standard output (`STDOUT_FILENO`). What happens if the child calls `printf()` to print some output after closing the descriptor?
8. Write a program that creates two children, and connects the standard output of one to the standard input of the other, using the `pipe()` system call.