

# CSC 369 Worksheet 2 Solution

August 17, 2020

## 1 Homework (Simulation)

1. I need to create process trees at each step when the command `./fork.py -s 10` is run.

1) Action: a forks b



2) Action: a forks c



3) Action: c EXITS



4) Action: a forks d



5) Action: a forks e



### Notes

- **fork()**
  - Is used to create a new process
  - **Creator** → parent process
  - **Newly Created** → child process
  - Child process is nearly identical to parent process
- **exec()**
  - Allows a child to break free from its similarity to its parent and execute an entirely new program.
- **wait()**
  - Is used to let parent code delay its execution until the child finishes executing.
  - Makes the output deterministic

2. I need to write what the resulting final process trees will look like as the fork-percentage changes. Here I ran command (`./fork.py -s 10 -a 10 -f 0.1` and `./fork.py -s 10 -a 10 -f 0.9`)

Notes

- `./fork.py -s 10 -a 10 -f 0.1`



- `./fork.py -s 10 -a 10 -f 0.9`



Based on the diagram above, I can deduce that the lower the fork percentage, the more likely that `exit()` is executed by the childmost process, and the final tree will either have a single node or none.

On the other hand, the higher the fork-percentage is, the more likely that `fork()` is executed by the childmost process, and the final tree will have nodes that are deeply nested.

3. I need to fill out blank entries created by the command `(./fork.py -t)`



4. I need to write what happens when a child exits; what happens to its children in the process tree.

When a child exists, all of its children will also exit.

I am not sure what happens when `-R` flag is used.

Correct Solution

I need to write what happens when a child exits; what happens to its children in the process tree.

When a child exists, its parentmost child, along with its children, will be attached to the parentmost node



When `-R` flag is used (i.e. `./fork.py -A a+b,b+c,c+d,c+e,c- -R`) and a child exists, its parentmost child, along with its children will be attached to the parent node of the child that exits



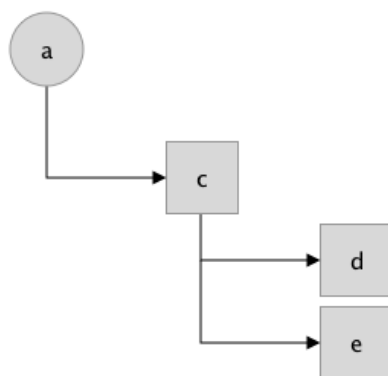
- I need to write down the final tree by looking at the series of actions generated (here, the command `./fork.py -F` is used).

```
Process Tree:
a

Action: a forks b
Action: b EXITS
Action: a forks c
Action: c forks d
Action: c forks e

Final Process Tree?
```

Answer:



6. First, I need to fill the actions that took place given the final process tree.



Given the final diagram, the missing actions are:

1. Action: a forks b
2. Action: b forks c
3. Action: a forks d
4. Action: a forks e
5. Action: a forks f

Second, I need to write whether I can determine the exact actions that took place, and write where can I tell and cannot tell.

No. I cannot tell exact actions that took place. I can tell what happened upto the latest visible node in the diagram (e.g a, b, c, d, e, f in above diagram), but I cannot tell actions that took place afterwards (e.g. Action: f forks g, Action: a forks h, Action: h EXITS, and Action: g EXITS).

## 2 Homework (Code)

1. Let  $x = 1000$ .

First, I need to write the value of the variable  $x$  in the child process.

The value of  $x$  in child process is the same as the parent (source code is provided in `question_7_part_1.c`).

```
hello, I am parent 9112 (pid: 9111)
-----hello, I am child (pid: 9112)
value of x is: 1000-----
```

Second, I need to write what happens to variable  $x$  when both child and the parent change the value of  $x$  (source code is provided in `question_7_part_2.c`).

When the value of  $x$  is changed in both child and parent, each possess their own values as if it's their own.

```
hello, I am parent 10035 (pid: 10034)
value of x is: 30
-----hello, I am child (pid: 10035)
value of x is: 20
-----
```

### Notes

- C file can be compiled via command `gcc -o OUTPUT_FILE_NAME SOURCE_FILE_NAME.c`
2. Yes. Both the child and parent can access the file descriptor returned by `open()`.

When they are writing to file concurrently, parent's `write()` is considered before children.

```
This is a parent
This is a child
```



3. Yes. Child can be called first by sufficiently delaying the execution of parent's print function. (Please refer to `question_9.c`).
4. First, I need to write a program that calls `fork()` and then calls some form of `exec()` to run the program `/bin/ls`.

Please refer to file `question_10.c` for solution.

Second, I need to write why there are so many variants of the same basic call?

According to Wikipedia ([link](#)), the variations of `exec()` are to satisfy requirements of various programming languages (e.g Python, BASH) and operating systems (e.g Linux, Windows).

## Notes

- **execl**

- Doesn't use PATH (a shortcut command). Requires full path of the executable file.

```
1  #include <unistd.h>
2
3  int main(void) {
4      char *binaryPath = "/bin/ls";
5      char *arg1 = "-lh";
6      char *arg2 = "/home";
7
8      execl(binaryPath, binaryPath, arg1, arg2, NULL);
9
10     return 0;
11 }
```

- **execlp**

- Uses PATH
- Is structurally similar to `execl()`

```
1  #include <unistd.h>
2
3  int main(void) {
4      char *programName = "ls";
5      char *arg1 = "-lh";
6      char *arg2 = "/home";
7
8      execlp(programName, programName, arg1, arg2, NULL);
9
10     return 0;
11 }
```

- **execle**

- Works like `execl()`
- Can provide your own environment variables. (e.g. `$PORT`)

```
1  #include <unistd.h>
2
3  int main(void) {
4      char *binaryPath = "/bin/bash";
5      char *arg1 = "-c";
6      char *arg2 = "echo \"Visit $HOSTNAME:$PORT from your browser.\"";
7      char *const env[] = {"HOSTNAME=www.linuxhint.com", "PORT=8080",
8                          NULL};
9
10     execle(binaryPath, binaryPath, arg1, arg2, NULL, env);
11
12     return 0;
13 }
```

- **execv**

- **Syntax:** `int execv(const char *path, char *const argv[]);`
- Passes all arguments in array `argv`

```
1  #include <unistd.h>
2
3  int main(void) {
4
5      int ret;
6      char *cmd[] = { "ls", "-l", NULL};
7      ret = execv ("/bin/ls", cmd);
8      return 0;
9  }
```

- **execvp**

- Works the same as `execv()`, but the `PATH` environment variable is used.

```
1  #include <unistd.h>
2
3  int main(void) {
4
5      int ret;
6      char *cmd[] = { "ls", "-l", NULL};
7      ret = execvp ("ls", cmd);
8      return 0;
9  }
```

- **execve**

- Works similarly to `execle()`
- Can provide own environment variables along with `execv()`.

```
1  #include <unistd.h>
2
3  int main(void) {
```

```
4
5     int ret;
6     char *cmd[] = { "ls", "-l", NULL};
7     char *const env[] = {"HOME=/usr/moegu", "LOGNAME=moegu",
8     NULL};
9     ret = execve ("/bin/ls", cmd, env);
10    return 0;
11 }
```

## References

- 1) Linuxhunt, Exec System Call in C, [link](#)
  - 2) SysTutorials, execl (3p) - Linux Man Pages, [link](#)
5. First, I need to write a program that uses `wait()` for the child process to finish the parent.

The solution is attached in file `question_11_part_1.c`.

Second, I need to write the returned value of `wait()`.

The value of `wait()` returns an integer. If `wait() = -1`, it means current process is the childmost process. If `wait() > -1`, it is the PID of recently terminated child-process.

(Please refer to `question_11_part_2.c`)

Third, I need to write what happens when `wait()` is used in child.

When `wait()` is used in child, -1 is returned.

(Please refer to `question_11_part_2.c`)

6. First, I need to write a slight modification of the previous program, replacing `wait()` with `waitpid()`.

The solution is attached in file `question_12.c`.

Second, I need to write when `waitpid()` would be useful.

Given the definition of `waitpid()`, `waitpid()` is useful when

- 1) Waiting for a specific child process
- 2) Waiting for child processes with specific group pid

## Notes

- **waitpid**

- **Syntax:** `pid_t waitpid(pid_t pid, int *stat_loc, int options);`

The value of `pid` can be

- \* **< -1:** wait for any child process whose process group ID is equal to the absolute value of `pid`
    - \* **-1:** wait for any child process
    - \* **0:** wait for any child process whose process group ID is equal to that of the calling process.
    - \* **> 0:** wait for the child whose process ID is equal to the value of `PID`
  - **stat\_loc** can have value `NULL` or one of its status options
  - **options** can have value `0` or one of its options