

CS314: Lab Assignment 2

B Siddharth Prabhu

200010003@iitdh.ac.in

15 January 2023

1 Part I - Hello World Process Chain

Task: Print the string “Hello World” on screen. Each character must be printed by a different process. The process that prints the i^{th} letter must have been spawned by the process that printed the $(i - 1)^{th}$ letter. Each process must print its designated character, as well as its own process ID, then sleep for a random number of seconds (from 1 to 4 seconds), and then, do anything else it must do to achieve the given task.

1.1 What is the minimum lines of C code with which you can achieve the above?

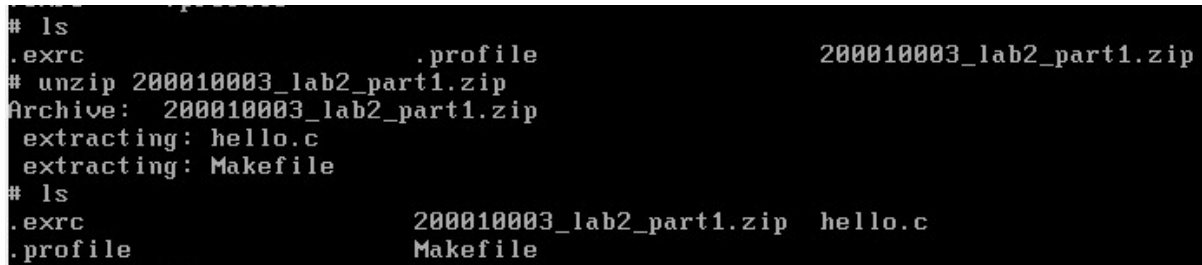
In C, the term “line of code” is technically meaningless. We can put the entire program (except for the #include directives) on one line, with statements separated by semicolons. So, in theory, the minimum lines of code would be $4 + 1 = 5$ lines.

Following good coding practices, the lines of code for the program comes to be around 20-25. In my program, there are 24 lines of code (including ONE empty line between the #include directives and the main function).

1.2 Testing and Running the Code

First, `scp` is used to transfer the zip file for Part 1 into the Minix3 Virtual Machine. Then, `unzip` is used to extract the contents of the zip file. I have done this in the /root directory. Finally, the code was tested using Clang compiler, using `gmake hello`.

1.3 Screenshots



```
# ls
.exrc                                .profile                                200010003_lab2_part1.zip
# unzip 200010003_lab2_part1.zip
Archive:  200010003_lab2_part1.zip
  extracting: hello.c
  extracting: Makefile
# ls
.exrc                                200010003_lab2_part1.zip  hello.c
.profile                            Makefile
```

Figure 1: Transfer of the zip file

```
# ls
.exrc                200010003_lab2_part1.zip  hello.c
.profile             Makefile
# gmake hello

letter|current_pid|parent_pid|
:H      :376      :375      :
:e      :377      :376      :
:l      :378      :377      :
:l      :379      :378      :
:o      :380      :379      :
:      :381      :380      :
:W      :382      :381      :
:o      :383      :382      :
:r      :384      :383      :
:l      :385      :384      :
:d      :386      :385      :
#
```

Figure 2: Testing the code

1.4 Some Remarks

- The `fork()` system call is invoked to create child process for the current (parent) process. After this, both processes will execute the next instruction following the system call. A child process uses the same PC (program counter), registers, and open files which are used by the parent process.
- The `wait()` system call blocks the calling process until one of its child processes exits, or a signal is received. After child process terminates, parent continues its execution from the next instruction following the system call. We have used this so that the parent process waits for the descendant processes to complete execution before closing. This is done to avoid the child processes becoming orphans, hence being adopted by the init (superparent) process.

2 Part II – Sequential Execution with Same PID

Task: Write a collection of programs twice, half, square such that they execute sequentially with the same process-id, and each program should also print its PID. The user should be able to invoke any combination of these programs, to achieve the required functionality.

2.1 Testing and Running the Code

- There are three programs, viz., `twice.c`, `half.c`, `square.c` in the submitted zip file. Just like in the previous part, these are transferred to the VM using `scp`, and then unzipped, and built. Then, command line arguments are used to execute the intended programs.
- The control first goes to the program whose executable appears first in the list of arguments entered. The last element of the list of arguments is the numerical input on which various operations are done.
- When the program executes, it creates a new command line argument without the first argument (that called it), and with an updated numerical input. This carries on until there are no executables left in the list of arguments.
- `execvp()` system call is used to transfer control between the programs without creating a new process.

2.2 Screenshot

```
# ls
.exrc      .profile
# ifconfig
/dev/ip: address 10.196.7.161 netmask 255.255.0.0 mtu 1500
# ls
.exrc      .profile      200010003_lab2_part2.zip
# unzip 200010003_lab2_part2.zip
Archive: 200010003_lab2_part2.zip
  extracting: half.c
  extracting: Makefile
  extracting: square.c
  extracting: twice.c
# gmake
clang half.c -o half
clang square.c -o square
clang twice.c -o twice
# ./twice ./square ./half ./twice ./half 10
Twice: Current process id: 375, Current result: 20
Square: Current process id: 375, Current result: 400
Half: Current process id: 375, Current result: 200
Twice: Current process id: 375, Current result: 400
Half: Current process id: 375, Current result: 200
#
```

Figure 3: Unzipping and Testing the code

2.3 Some Remarks

- The command line input `./twice ./square ./half ./twice ./half 10` calculates the value of `half(twice(half(square(twice(10)))))` and prints 200 as result. It also prints the process ids of each program as it executes.
- After the program does its job of modifying the numerical input and reporting it, a new command line input is formulated by ‘shifting’ the arguments to the left by one, and changing the numerical input argument.
- Then, `execvp()` system call moves the control to the next program in line, within the same process.

3 Part III – MINIX3 Process Messages

Task: Modify the Minix3 source code such that:

- A message “Minix: PID <pid> created” is printed, whenever a process is created. (Let us follow the convention throughout this course that anything printed by the Operating System code will be prepended by the string `Minix: .`)
- A message “Minix: PID <pid> exited” is printed, whenever a process ends.

Comment on the order in which processes are created and processes exit and justify whether it is as expected.

3.1 Testing and Running

The file `forkexit.c` located at `/usr/src/minix/servers/pm` is modified at 2 places, by inserting the desired print statements in the `do_fork()` and `do_exit()` functions.

3.2 Screenshots

```
# ls
Minix: PID 210 created
.exrc      .profile
Minix: PID 210 exited
#
```

Figure 4: ls process created and exited

```
# ls
Minix: PID 210 created
bin      boot_monitor  home      proc      service    usr
boot     dev            lib       root      sys        var
boot.cfg etc           mnt       sbin      tmp
Minix: PID 210 exited
# ifconfig
Minix: PID 211 created
/dev/ip: address 10.196.8.11 netmask 255.255.0.0 mtu 1500
Minix: PID 211 exited
#
```

Figure 5: More processes created and exited

3.3 Some Remarks

- The files related to Process Management are in `/usr/src/minix/servers/pm`, and we observe that the forking and exiting of processes occurs in `forkexit.c`.
- Unless the parent terminates or becomes a zombie, the child process ends before the parent process does. Otherwise, the init (superparent) will adopt the orphan process.
- A bash script `run.sh` is present in the submitted zip file, which copies the modified source files to the correct directories, and builds the system.