Practical 2

Author Name

December 23rd, 2022

Table of Contents

Introduction

This practice proposes a series of activities with the aim that the student can apply on a Unix system some of the concepts introduced in the last modules of the subject. The student will have to carry out a series of experiments and answer the questions posed. You will also need to write a short program in C language.

Question 1

Write a C program that displays the current time every N seconds. To display the current time, you must execute the date command date. To wait N seconds, you cannot use signals or routines such as sleep, you must do so using a child process that executes the sleep command N.

- First, we set N equal to 2 seconds.
- Now make N a parameter that must be specified on the command line.

Answer 1

The code can be found in the files `main_a.c` and `main_b.c` in the folder for this question.

To compile the code, you can run a command like the following in the appropriate directory:

```
gcc -Wall -Wextra -Werror main_a.c
```

Note: The compilation flags `-Wall -Wextra -Werror` are optional, used to optimize the code and avoid having unused variables.

To run the code, you can simply run the following command:

```
./a.out
```

As an added bonus, the code I created passes the so-called ***`norminette`****, a linter for* `.c` *and* `.h` *files I use to keep code clean.*

Question 2

This exercise highlights the differences between the `_exit` system call and the exit library routine. Although we usually refer to "the `exit` system call", `exit` is a library routine that ends up invoking the `_exit` system call.

a)

Refer to the manual page (man order) for `exit` and `_exit`. Attach the first few lines of the result. How to know who is called and who is called by a library routine?

Answer 2 a) Here are the first lines of the output of the command `man exit`:

```
EXIT(3)                    Library Functions Manual                    EXIT(3)

NAME
       exit - cause normal process termination
```

And here are the first lines of the output of the command `man _exit`:

```
_EXIT(2)                      System Calls Manual                      _EXIT(2)

NAME
       _exit, _Exit - terminate the calling process
```

As we can see, `_exit` is the actual system call to terminate a process whereas `exit` is merely a command utility for shells. The `exit` command is typically implemented as a built-in command on most common shells (bash, zsh, fish, etc).

b)

From reading the manual pages, what two main functionalities does the `exit` routine add to the `_exit` system call?

Answer 2 b) `_exit` simlpy terminates the calling process and closes any open file descriptors. The `exit` routine has two extra features:

1. All open streams are flushed and closed, and temporary files from `tmpfile` are removed.
2. Every function called with `atexit` and `on_exit` is called in reverse order of registration.

c)

exitae.c is a program that uses one of these new features. Compile it, run it, and attach the result of several executions. Explain the behaviour of this program.

Answer 2 c)

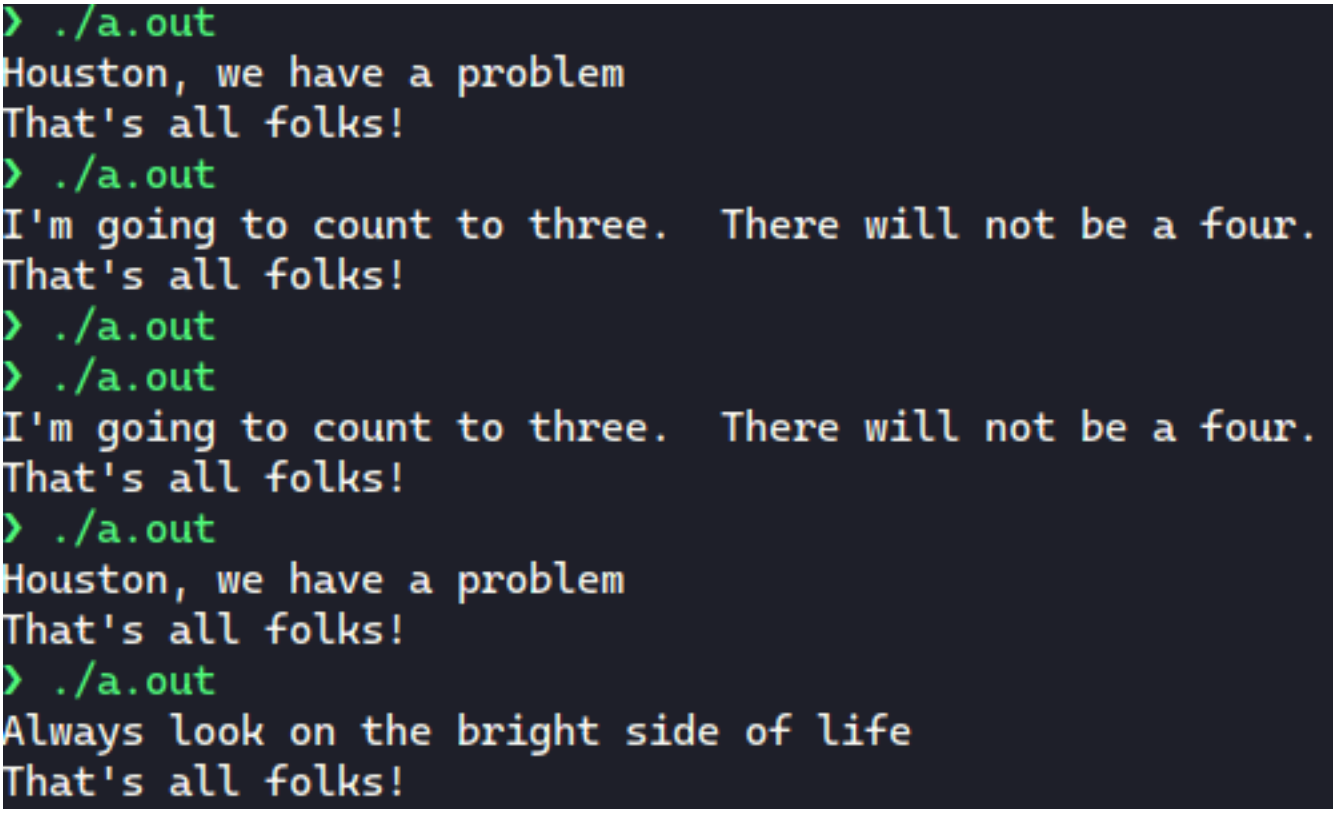

Figure 1: Sample Executions

From the behavior we can clearly see how `exit` and `_exit` differ. Every case in the `switch` prints a unique sentence and executes the `exit` routine, and thanks to `atexit`, a unique sentence will be printed afterwards. The random number mod `4` might not be `0`, `1` or `2` so it won't fall into any of the switch cases, running the `_exit` system call instead. In this last case we see that the `atexit` function is never called, thus explaining the peculiarity of the `exit` routine.

d)

exitwe.c, exitw_.c, exitpe.c and exitp_.c are four programs that write a message to standard output and `exit`. They differ in how they write the message (`write` vs `printf`) and how they terminate (`exit` vs `_exit`). Compile and run them. Attach the result of their execution. Justify why we observe different behaviours?

Hint: notice that the message to be written does not end with a line break. Find out what consequences this may have when you do a printf

Answer 2 d)

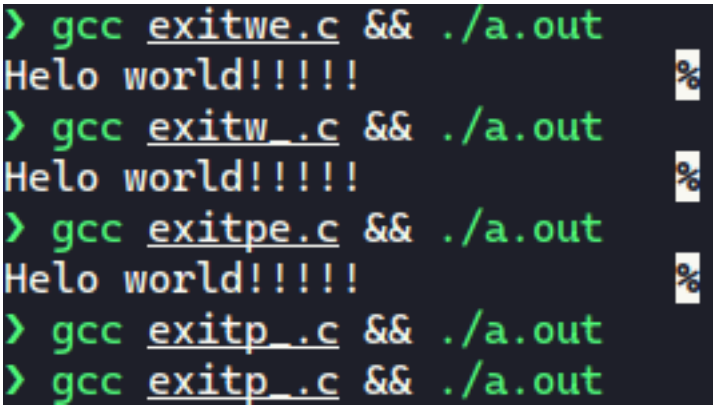

Figure 2: Sample Executions

In the first three cases (`exitwe`, `exitw_` and `exitpe`) the output is the expected one. However, it seems like the combination of using `printf` instead of `write` with `_exit` instead of `exit` causes the last example `exitp_` to not show any output. Reading the manuals for `_exit` we see the following:

`Causes normal program termination to occur without completely cleaning the resources.`

This means that the problem exits, without properly flushing `stdout` before terminating. Now `write` also "writes" to `stdout` in `exitw_` and uses `_exit`, but the difference is that `printf` has buffer management in place whereas `write` does not, and because `stdout` is buffered per lines (which means that every line is flushed from memory when a newline `\n` is found) and the message has no newline, the message cannot be displayed with `printf` and `_exit`.

e)

exitp2e.c and exitp2_.c are two programs analogous to those in the previous section, but which write a message ending in a line break. If you run them without redirecting the standard output and redirecting it to a file, you will get the following result.

Figure 3: Sample Outputs

How the file out2 is zero bytes in size (ie it's empty, no has any message been written there)?

Hint: See the man page for the setbuf library routine

Answer 2 f) Similarly to last time, we are now trying to flush the contents of `stdout` into a file. However, the `_exit` routine does not flush contents while the `exit` routine does. `write` is unbuffered, so content will be flushed immediately to the file, whereas `printf` is line-buffered.

Question 3

This exercise deals with the family of exec routines (execl, execlp, execle, execv, execve, execvp execvpe), routines that allow a process to load a new executable and start its execution.

a)

Of all these routines, which one is really system call, and which are library routines that end up invoking the system call? (Proceed in the same way as exercise 2a)

Answer 3 a) From the manals:

```
EXEC(3)                    Library Functions Manual                    EXEC(3)

NAME
       execl, execlp, execle, execv, execvp, execvpe - execute a file
DESCRIPTION
       The  exec()  family of functions replaces the current process image with a
       new process image.  The functions described in this manual page  are  lay-
       ered  on top of execve(2).  (See the manual page for execve(2) for further
       details about the replacement of the current process image.)
```

```
EXECVE(2)                    System Calls Manual                    EXECVE(2)

NAME
       execve - execute program
DESCRIPTION
       execve()  executes  the  program referred to by pathname.  This causes the
       program that is currently being run by the calling process to be  replaced
       with  a  new program, with newly initialized stack, heap, and (initialized
       and uninitialized) data segments.
```

As we can read, `execve` is the main system call, whereas the others are simply routines that use this call in it and add certain extras on top.

b)

Write two programs that execute the command "/bin/ls -l /". The first program should use execl and the second execv. Attach a screenshot of how they work.

Answer 3 b) *Note: check the code in the folder for this question*

Since both `execl` and `execv` are built on top of `execve`, they do essentially the same: replace the parent process to execute a new program on the process. Because it substitutes the other running process, it is generally required to only call these functions inside children, that is why my code has these function calls inside child processes (from `fork`). The structure of these commands is very well thought out. Here is a general overview of what each part of the command name reflects in the functionality of the function:

- `exec`: root name of command, executes a command.
- `v`: passes the command's argument list as an array (null-terminated).
- `l`: passes the command's argument list as subsequent arguments (last argument is null).
- `p`: the first argument of the function can be the name of the executable instead of its full path. These functions will internally look for the command in the argument using the `PATH` environment variable.
- `e`: Environment (`envp`) is passed to the command to be executed.

With these things explained, it is clear what the difference between `execl` and `execv` are: both take the full path of an executable as the first argument, but `execl` receives an undefined number of extra arguments representing the `argv` for the new command, while `execv` receives an array as `argv`. Neither of them get an extra argument in the end for the `envp`.

```
❯ gcc -Wall -Wextra -Werror execl.c && ./a.out
total 20
lrwxrwxrwx   1 root   root      7 Oct 18 23:01 bin -> usr/bin
drwxr-xr-x   1 root   root      0 Oct 18 23:01 boot
drwxr-xr-x  21 nobody nobody 4340 Dec 23 15:54 dev
drwxr-xr-x   1 root   root   2028 Dec 23 19:19 etc
drwxr-xr-x   1 root   root     10 Dec 23 11:59 home
lrwxrwxrwx   1 root   root      7 Oct 18 23:01 lib -> usr/lib
lrwxrwxrwx   1 root   root      7 Oct 18 23:01 lib64 -> usr/lib
drwxr-xr-x   2 nobody nobody   40 Dec 23 11:41 media
drwxr-xr-x   1 nobody nobody    0 Nov 13 11:19 mnt
drwxr-xr-x   1 root   root     30 Dec 23 12:48 opt
dr-xr-xr-x 616 nobody nobody    0 Dec 23 12:15 proc
drwxr-x---   1 root   root     20 Dec 23 16:44 root
drwxr-xr-x   1 root   root    366 Dec 23 11:59 run
lrwxrwxrwx   1 root   root      7 Oct 18 23:01 sbin -> usr/bin
drwxr-xr-x   1 nobody nobody    0 Nov 13 11:19 srv
dr-xr-xr-x  13 nobody nobody    0 Dec 23 11:41 sys
drwxrwxrwt  21 nobody nobody  540 Dec 23 19:21 tmp
drwxr-xr-x   1 root   root     62 Dec 23 16:45 usr
drwxr-xr-x   1 root   root     50 Dec 23 11:59 var
❯ gcc -Wall -Wextra -Werror execv.c && ./a.out
total 20
lrwxrwxrwx   1 root   root      7 Oct 18 23:01 bin -> usr/bin
drwxr-xr-x   1 root   root      0 Oct 18 23:01 boot
drwxr-xr-x  21 nobody nobody 4340 Dec 23 15:54 dev
drwxr-xr-x   1 root   root   2028 Dec 23 19:19 etc
drwxr-xr-x   1 root   root     10 Dec 23 11:59 home
lrwxrwxrwx   1 root   root      7 Oct 18 23:01 lib -> usr/lib
lrwxrwxrwx   1 root   root      7 Oct 18 23:01 lib64 -> usr/lib
drwxr-xr-x   2 nobody nobody   40 Dec 23 11:41 media
drwxr-xr-x   1 nobody nobody    0 Nov 13 11:19 mnt
drwxr-xr-x   1 root   root     30 Dec 23 12:48 opt
dr-xr-xr-x 616 nobody nobody    0 Dec 23 12:15 proc
drwxr-x---   1 root   root     20 Dec 23 16:44 root
drwxr-xr-x   1 root   root    366 Dec 23 11:59 run
lrwxrwxrwx   1 root   root      7 Oct 18 23:01 sbin -> usr/bin
drwxr-xr-x   1 nobody nobody    0 Nov 13 11:19 srv
dr-xr-xr-x  13 nobody nobody    0 Dec 23 11:41 sys
drwxrwxrwt  21 nobody nobody  540 Dec 23 19:21 tmp
drwxr-xr-x   1 root   root     62 Dec 23 16:45 usr
drwxr-xr-x   1 root   root     50 Dec 23 11:59 var
```

Figure 4: Sample Outputs

c)

The execw.c and execp.c programs differ only in that one uses the write system call while the other uses the printf routine. If you run them, you will see that the behaviour is different. What is it due to?

```
enricm@pcmartorell-i-ribas:~/.../dev$ ./execw
Current weekday is: Tuesday
enricm@pcmartorell-i-ribas:~/.../dev$ ./execp
Tuesday
enricm@pcmartorell-i-ribas:~/.../dev$
```

Figure 5: Sample Output

Answer 3 c) Just like with the case in Question 2d, `printf` has buffer management (it is line-buffered), and the string to be printed before the date is not printed because it has no newline `\n`. The family of `exec` functions all replace the process, so if the buffer is not flushed before it will never be printed on the screen.

Summary

This second practical activity was nice. Even though I had some background knowledge on processes, there were still various details I completely ignored.