Lab 3

6 February 2023

This week's lab is about practicing three things in C: file I/O with `FILE*`, fork/wait, and signal handling.

Black box spec

Your program will take as its command line argument a partial filename, to which the program will append `.in` and `.out` to generate the actual filenames used for the file-based I/O.[1] (You may print an error message, but should not crash, if the provided partial filename is longer than 20 characters.) On startup, it will print a message "`Welcome`" to the screen, then simultaneously interact with the user, and with the provided files.

For both the user interaction and the file interaction, the behaviour should be that it repeatedly reads in a single positive integer, pauses to think for one second, and then prints the number twice as large as that. If the input is zero, the interaction ends (a "normal" end). If the input is negative, the interaction prints an error message and ends (an "error" end).

If both interactions end normally, the original process prints a message to that effect and exits. If either interaction ends with an error while the other is still running, the still-running interaction immediately prints that the other interaction had an error and then exits itself; and then the original (controller, interactive) process prints a message that says *which* interaction ended with an error, and exits. If the error happens after the other interaction is done, the original process simply prints its error message and exits.

Examples

Contents of `test1.in`:

```
3
42
```

[1] So, if the argument were `foo`, the input would come from `foo.in` and the output would go to `foo.out`.

```
7
0
```

Interaction, including initial command prompt and both input and output:

```
shannon -> ./a.out test1
Welcome
8
16
1000
2000
17
34
2401
4802
0
Both interactions normal
```

Contents of `test1.out` after execution:

```
6
84
14
```

Example 2

Contents of `errex2.in`:

```
8
13
-5
2
0
```

Interaction, including initial command prompt and both input and output:

```
shannon -> ./a.out errex2
Welcome
123
```

```
246
2
Error in other interaction
File interaction ended with error
```

(Note that the exact result/length of the interaction depends slightly on how fast/slow the user types.) Contents of `errex2.out` after execution:

```
16
26
Error
```

Example 3

Contents of `third.in`:

```
1
2
3
4
5
0
```

Interaction, including initial command prompt and both input and output:

```
shannon -> ./a.out third
Welcome
25
50
-1123
Error
User interaction ended with error
```

Contents of `third.out` after execution:

```
2
4
Error in other interaction
```

(Note that the exact contents of the file depend slightly on how fast/slow the user types.)

Internal and other requirements

The two interactions, with the file and with the user, should each be their own process separate from the original process (for a total of three processes).

In case it's not clear, the one-second pause is a purely artificial requirement to make this basically testable (otherwise the file-based interaction would go much too fast). This can be done with the `sleep` function.

The indication from a child process as to whether it is ending normally or with an error should be through its exit condition: 0 for normal, 1 for an error. The original process will check for this condition to decide how to respond to it.

The signal from the parent process to the other child after one child errors should be a SIGTERM which is handled appropriately.

The actual work of read-pause-print should be done by a function that takes `FILE*` parameters, so that it can work with actual files or with standard input and output.

Use functions appropriately and observe principles of good design. Avoid globals unless that's the only way to transfer a value—in this lab you will actually use one global variable (acting as a boolean) to communicate from the signal handler. The handler will set the variable, and your running interaction will check it to see if it needs to gracefully terminate.

As before, your handin should include a readme documenting how to compile, run, and test your program. You may use a makefile; you should not rely on `compile`. (If you wish to use unci, i.e. `.u` files, to test any functions you write, you can use `uncic` directly to compile the `.u` files into `.o` files. See me for details if you need help.)

Programs that do not compile will get few or zero points.

Use test cases to show me what works.

Duedate and handin

The lab is due Monday at 4pm.

Hand it in using the handin script:

```
handin cmsc242 lab3 dir_of_stuff/
```

Rubric (tentative)

RUBRIC

General and design

3	compile, run, test instructions
2	fopen, fclose
1	good design, use of functions, variables, etc
1½	checks return values in cases where something could go wrong
1	checks at least one of fork, fscanf, fopen
½	...all of them
½	prints messages as specified

CL arguments and string handling

1	parameters of main
1	uses parameter of main
1	builds correct filenames per spec
1	... with enough space allocated for largest valid filenames
1	... and protecting against buffer overflow

Function for I/O and math

1	puts at least some I/O in separate function(s)
1	uses function as specified (one function for both interactions)
1	... with parameters as specified
2*	fscanf, fprintf *(original handout said 1)*
1	read a number, print its double if positive
1	pause for one second
1	read/prints in non-infinite loop
1	... that breaks on negative and/or zero input, and not on positives
½	... ... error and exit loop on negative
½	... ... exit loop without error on zero

Basic process management

1	calls fork
1	... and stores or uses return value
1	clearly distinguishes child vs parent
1	two children per spec
1	calls wait
1	... and successfully gets status
1	calls wait for both forked children
1	... along all execution paths
1	child processes exit/return normally if normal
1	child processes exit/return nonzero if error
1	Parent distinguishes error from normal exit (WIFEXITED, WEXITSTATUS)
1	...and correctly ids which child errored based on exit status, sometimes
½	... ... and prints normal/user error/file error, correctly in all cases

Signal sending and handling

1	On error, send SIGTERM to other child
1	On receipt of SIGTERM, react in any custom way
1	... updates variable (global or equiv) to notify process to wrap up
1	... print error message
½	... closes files if any