Project 1

Due: 3 March 2021

20210226-0100

This project aims to give you more practice in C and tie together several of the systems programming concepts we've seen so far (notably `fork`, direct I/O, and inter-process communication). Unlike the week-long labs, a bit more of the design is up to you; there are a few intermediate deadlines to help you plan that out.

The overall task is to build a test harness for small programs that operate entirely via standard input and output (such as those you wrote for the first two labs). One kind of testing is to provide pairs of files with input and expected output; another is to run the program and manually enter stuff to see if it does what one would expect. Your program will do both, simultaneously, and report on the results.

Objectives

In the course of this project, the successful student will:

- process buffers of string data, in C, without crashing or overflowing;
- manage multiple simultaneous processes within a program using different forms of inter-process communication; and
- use man pages and other documentation to learn how to use a new aspect of the system (directory listings).

Expected behaviour

Your program should take, as its command-line arguments, the name of another program (the tested program) and the name of a directory with test files in it. It will begin auto-running the tested program with those tests (details below), and while those are running, it also runs the tested program with direct user input/output, for the user to personally evaluate whether the behaviour seems right (while they wait for the automatic tests to run).

The automatic side of the program should take every file in the test file directory that ends with "`.in`" and run the tested program with the contents of that file as standard input. If there is a matching "`.expect`" file, the standard output of the tested program should be compared with the contents of that file to see if it is an exact match; otherwise that particular input file is only being run to determine if it successfully completes with a successful exit code (as opposed to other exit codes, or being terminated by a signal).

If the auto tests finish while the user is still playing with it, the program should interrupt with a message indicating that the automatic tests have finished, but should let the user continue interacting with the tested program until they naturally finish (and this message should be the only interruption); if the user wraps up and the automatic tests are still running, the program should print a message to that effect but otherwise wait for the auto tests to finish.

Once both sides of the program are done, it should print a summary indicating how the manual testing completed (successful exit code, error exit code, or termination by signal) and a summary of the automatic testing (tests passed vs failed, plus how they completed). As in other testing systems, tests that pass can just be counted but a test that fails in some way should be indicated by name (so that the user can go investigate it, and presumably fix it).

Prep work

Your initial prep work mostly involves getting the fork-and-exec aspect of the program working on the "manual testing" side of things. It should accept both command line parameters (and verify they're there) but ignore the directory for now; it will fork a side process and run the tested program in that side process (using one of the exec functions, probably `execl`), and the main process will wait for that to finish and print a message when it does.

Hand in the prep work (as `proj1`) by 4pm on Friday the 12th.

Design work

Once you've got the shell of the program running (or perhaps while you are working on that), you can start thinking about the larger system and communication design. In particular, you should write/draw:

1. a diagram showing what each process in the program is doing at different points in time, and when/what messages they send to each other (you can use my vertical multi-process diagrams as a model)
2. a protocol plan for exactly what information gets sent between processes—both the medium (i.e. which type of IPC) and the message (i.e. what do the byte-by-byte messages look like?)
3. a list of what files you'd need to write or acquire to test this project, and what properties each file would have (which code path/aspect of the spec is the file testing?)
4. a list of systems functions you'll need to read up on in order to accomplish all of this

Write your design work on paper (or do it on your laptop, but paper's probably easier for this) and bring it to class; this work is due **at the start of class** on Monday the 15th. If you're really stuck on something, do your best, make a note of it, and move on; we'll be discussing this extensively in class.

Final version

A full-credit final version will be a complete, non-buggy, working implementation of the test harness TOGETHER WITH convincing proof that it is correct. The program should be able to run with arbitrary input without crashing (a graceful exit with an error message is not a crash), even if the user gives bad input or the tested program is itself buggy and crashes; it will run the tested program both for manual testing and for automatic testing, and print a summary of the results when all the testing is finished (whether through orderly exit or signaled termination). The "proof" will take the form of a clean and complete set of test cases, including both input and expected results.

Note that I am not able to spend a ton of time with your program (and in fact may not read it at all, and definitely won't do your debugging for you), so your documentation will need to tell me anything I need to know to run and test your program. There need to be clear instructions on how to run it in general as well as how to run each/all of the tests and quickly verify that they ran correctly (and which rubric items each one corresponds to).

Having complete and correct documentation is an easy 10 points, but if your documentation omits important info or tells me the wrong thing, you'll get less than full credit there.

After prep work (10 points), design work (10 points), and documentation (10 points), there remain 70 points in the rubric, which will be awarded according to the table below. Under each score, I show (for your convenience) the total cumulative points if you get that item plus *all* the previous points, and the letter grade this corresponds to.

NOTE: if your code doesn't compile, or immediately crashes when it's run, you will get zero of these points. Don't let this happen to you!

Score	Description
Process management	
10 (40/D−)	Forks at least one child process, execs the specified executable in a child process, and parent waits for child to terminate before printing some final message (i.e. the prep work)
5 (45/D)	Forks two child processes from main program, one for user-directed stuff and one for auto-run stuff; lets each do their thing and finish up, handling whatever else it does correctly regardless of which order the children terminate
5 (50/D+)	Reports termination conditions (normal/error/signal) of any process that exec'd the tested program, either immediately† or as part of the final summary output
Directory processing	
5 (55/C−)	Iterates through the entire provided directory, either printing the filenames† or using them (the `.in` files at least) as input for the tested program
5 (60/C)	Identifies the `.in` files in the directory
5 (65/C+)	Pairs each `.in` file with its matching `.expect` file or determines that there isn't one
5 (70/B−)	Successfully opens for reading every `.in` file in the directory (and, all the `.expect` files, if they've been found), perhaps printing the first few characters of each† or, of course, actually using them with the tested program

Score	Description
Input/output redirection	
5 (75/B)	With at least one `.in` file, run the tested program redirecting standard input to use that file instead *
5 (80/B+)	With at least one `.in`/`.expect` pair, run the tested program and compare its actual output with the expected output; report whether the test passed or failed, either immediately† or as part of the final summary output *
5 (85/A−)	...without dumping any output to an extraneous temporary file *
5 (90/A)	Does at least one automated test with a `.in` file, tests all the `.in` and `.expect` files it knows about, and, whatever automated testing is done, doesn't put any extra stuff on the screen that could interfere with the user-facing testing process or is otherwise extraneous (just the final report at the end). (If you want to add a command-line option that "turns on extra output" that's fine, as long as it doesn't print by default.) *
Results IPC	
5 (95/A)	Sets up and opens appropriate form of IPC to communicate results from auto-testing child process back to original main parent process
5 (100/A+)	Successfully communicates results of auto-testing back to main parent process and prints it there as final output of the program

* If you are completely stuck on the directory management stuff, you can give the `.in` file as a parameter instead (and an additional `.expect` parameter as appropriate) to get the I/O points. But then you're not going to get the directory processing points.

† Several phrases in the rubric are marked with this symbol to indicate that this is a way to demonstrate that specific point, especially while you're still building and debugging the program, but it might interfere with later points so remember to upgrade/revisit it as you finish later stuff.

Handing in

For both the prep work and the final version, hand it in as `proj1` using the handin script. The final version is due at 4pm on Monday, 3 March.