CSci 4061: Introduction to Operating Systems Fall 2024
Project #1: Basic Autograder

Instructor: Jon Weissman

Due: Sept 25 midnight
Intermediate Due: Oct 2 midnight

1 Objective

In this project you learn about the use of *system calls for* ***processes***. In particular, you will gain experience with `fork`, `execl`, `wait`, `waitpid`, `return`, `sleep`, `kill`, and `exit`, and basic I/O operations such as `fopen`, `fgets` (although the purpose of this project is not I/O per-se). Additionally, you will learn how to use */proc* to extract information about processes in your program, and the system.

Processes have several **benefits**: (1) they can run arbitrary compiled code unknown until execution time, (2) isolation, and (3) they can provide performance benefit on a multicore/multiprocess system. In this project you will explore and observe all of these different benefits.

Some source code, object code, and test cases will be provided as needed in a downloadable tar file from Canvas.

2 Problem Statement

In this project, you will write a rudimentary *autograder*. The autograder runs and checks a set of "submitted" executables or "solutions" (presumably submitted by a class of students). Each executable has a unique name (e.g., `a_sol`, `b_sol`, `c_sol`, etc.) and is located in the `solutions` directory. In real-life these executable names would likely be the name of the student or student group. Each "solution" is programmed to accept a single integer parameter and return an associated "answer". The autograder is going to run every submitted student "solution" for a given set of testing input parameters, each one an integer, (e.g. the parameters could be 1, then 2, then 3) that are provided on the autograder command-line. In this case, you would run "a_sol" with parameter 1, "b_sol" with parameter 1, and so on, then "a_sol" with parameter 2, "b_sol" with parameter 2, and so on.

There are 3 possible events that occur when a submitted program is run with a parameter:

1) It runs successfully and returns the correct answer

2) It runs unsuccessfully returning a wrong answer

3) It crashes

You will generate a final tally for each submitted executable one per-line of the form (where 1, 3, 7, … are example input parameters representing different outcomes):

a_sol: 1 (crash), 3 (crash), 7 (crash), 9 (correct), …

b_sol: 1 (crash), 3 (correct), 7 (crash), 9 (incorrect), …

…

Did you get the right answer? We will provide you with the expected outputs for particular submissions.txt for validation. We will test you on these as well as some others chosen by us.

3 Going deeper

Your autograder will find the list of executables, one per line, in a file called "`submissions.txt`". **We have provided the function *write_filepath_to_submissions* that generates `submissions.txt` for you** (Note: This function does not sort the executables in alphabetical order. If you wish to implement your own function to achieve alphabetical sorting, you are encouraged to do so). You will then use `fopen/fgets` to acquire the names and read each line. Remember that `fgets` puts a '\n' at the end of the string which you will have to strip out. Use the executable names exactly as they appear in the call to `exec*`. (after '\n' removal).

Make sure the "executable solutions" have the proper permissions: chmod -R +x

A) Your auotograder will run (i.e. `fork/exec`) on your set of differing executables using their names.

Notice how this achieves Benefit (1) - your autograder can determine at runtime what executables to launch (i.e. no hardcoding of names).

B) You will run the submitted executables in batches of size B. Where the batch runs in parallel and finishes before the next batch is submitted.

How would you know B?

`cat /proc/cpuinfo | grep processor | wc -l` gives the number of cores on that machine. Try various values of B below and above this value (but not too big e.g. <10), and notice any performance changes. Your final batch may be < B, depending on how many submitted solutions you are executing.

To make grading easier for us, please form the batches by processing the contents of submissions.txt in sequential order and write the output lines in the order they appear in submissions.txt. Write the output to a file named autograder.out (use fopen and fputs). For testing purposes, we have provided the necessary input parameters. You can compare the autograder.out file with the provided expected.out to verify that your implementation produces the expected results for the executables listed in "submissions.txt". Here is a sketch of the suggested code structure (Note: this is pseudocode, and you will need to implement it in C with the appropriate syntax):

```
# For each parameter, run all executables in batch size chunks
for p in parameters:
    # Run through all executables in batch size chunks
    done_executables = 0
    while done_executables < total_executables:

        # Fork and exec batch_size executables
        for b in batch:
            ...

        # Wait for the executables to finish
        finished = 0
        while finished < batch_size:
            for b in batch:
                # Wait for the executables to finish
                # and determine the cases
                ...

        done_executables += batch_size
```

If the batch size > 1, the "solutions" should run in **parallel** on the different cores (hopefully)! Make sure that your autograder does not serialize execution of each solution.

Can you notice how this achieves Benefit (3) - that your autograder runs faster as the batch size is increased up to a limit?

C) Each provided submission executable will `exit/return` (<answer>) to return their answer or crash. You will find `pid_t waitpid (-1, &status, WNOHANG)` to be very handy coupled with the macro `WIFEXITED (status)` to do this. Read up on `waitpid` and its arguments, the return values, and the macro `WIFEXITED`. When will the executables finish? Since this is unknown you may have to go around the wait loop multiple times. To make this more efficient, you make want to insert a `sleep (n)`, n ~ 2-5 seconds.

If an answer=0 is returned, the autograder will mark it as correct. If an answer=1 is returned, the autograder will mark it as incorrect. Thus, events 1 and 2 are straightforward to determine.

D) You can similarly handle event 3 very easily: Read up or try out the behavior of `waitpid` on a child that crashes.

Notice how this achieves Benefit (2) isolation– a crashed executable has no effect on your autograder.

4 Intermediate Submission

In one week you will submit a version of your autograder that can "grade" one fixed size batch of submitted executables (e.g. 2 is fine) as correct or incorrect only with a single input parameters (1 is fine). **For this upload autograder.c**.

5 Implementation Notes

<u>Check</u>: make sure that you do not leave any stray processes before you log out. Run `ps -u <userid>` or better still `top -u <userid>` to see what processes are still running. Then run `kill -9 <pid>` at the shell to dispose of them.

<u>Warning:</u> if you are writing more than 100s of lines of code, something is wrong ☹

6 Deliverables

There will be 2 submissions, one intermediate submission due 1 week after the release of the project and a final submission due 2 weeks after the release.

Intermediate Submission :

Complete autograder.c

Both intermediate and final submissions should contain the following.

One student from each group should upload to Canvas, a zip file containing the following (autograder.c (with any supported source files as needed), a Makefile and a README that includes the following details):

- How to compile the program
- Any assumptions outside this document
- Team id, team member names and x500's
- Contribution by each member of the team for *final submission only*

7 Rubric: Subject to change

- 10% README including answers to questions posed in the writeup.
- 15% Intermediate submission [Including README, autograder.c code for correct/incorrect cases].
- 10% Documentation within code, coding, and style: indentations, readability of code, use of defined constants rather than numbers. The code should be well commented (need not explain every line). You might want to focus on the "why" part, rather than the "how", when you add comments.
- 65% Test cases: correctness, error handling, meeting the specifications.

  You must error check ALL *system* calls in your autograder.
- A test folder of executables, input parameters to test, a "solution" and the templates will be provided.
- We will use the GCC version installed on the CSELabs machines to compile your code. Make sure your code compiles and run on CSELabs.
- **Please make sure that your program works on the CSELabs machines** e.g., KH 4-250 (csel-kh4250-xx.cselabs.umn.edu). You will be graded on one of these machines.

8 Miscellaneous

- We will provide an initial package of code, but you will be doing most of the coding.
- To run your program, type **autograder<B><p_1>< p_2> …**

  Where B is the batch size, and each p_i is a test input parameter to pass to the submitted "solutions" – it is an integer
- Do not use the system call "`system`".
- Said before: KILL all of your stray processes during debugging as needed.
- Any provided binaries are meant for the CSELAB Linux environment. No other binaries will be distributed.
- ChatGPT or other significant "other" code reuse prohibited. The purpose of this course is to learn

by doing, and not meeting some deadline. If you are unsure about any located online code, contact us.

- On the other hand, locating code snippets that show how system calls can be used is fine.

9 Suggested Workplan

1. Untar the package and understand what we have given you
2. Start coding
3. Verify you can read in all of the executables in submissions.txt and print out their names
4. Verify you can launch these executables from your program with an integer parameter(s)
5. Verify you can obtain an answer from a completed executable
6. Verify you can detect all answer types: do the easier ones first
7. Produce the scoring for each submitted executabl