Overview

In this lab, you will be implementing the following C utilities:

- `time`
- `env`

Notes:

- Do not worry about flags or features that we do not mention.
- Do not print any of your debug information out for your final submission.
- All printing (except env vars) should be handled with `format.h`.
- A common issue is double printouts. If this happens to you, try flushing stdout before you fork/exec. If this solves your issue, ask yourself why.

WARNING!

If you fork bomb on *any* autograder run, you will receive a zero on this assignment.

To prevent you from fork bombing your own VM, we recommend looking into `ulimit`. This will allow you to set a limit for various operations on your system, including how many files you can open concurrently or how many times you can fork. The side effect is that a poorly chosen limit may throttle your system's operations (e.g. setting the fork limit too low may make your terminal unable to execute commands!).

format.c and .h

Since this lab requires your programs to print messages to `stdout` and `stderr`, we have provided you with `format.c` and `format.h`. You should not be printing out to stdout and stderr at all. Instead, you should be using the provided functions. You can find documentation for each function in `format.h`. Please *read* the documentation in `format.h` *multiple* times to determine when each function should be used. This is our way of ensuring that you do not lose points for formatting issues, but it also means that you are responsible for handling any errors mentioned in `format.c` and `format.h`.

It is common for students to fail certain test cases on this assignment with seemingly functional code, it is almost always because of improper usage of `format.h`.

time

In this lab, you will be implementing `time`.

```
time - run a program and report how long it took
```

So if a user enters:

```
./time sleep 2
```

then time will run `sleep` with the argument `2` and print how long it took in seconds:

```
sleep 2 took 2.002345 seconds
```

For more examples, you can play with Linux's builtin `time` command by typing `time YOURCOMMAND` (`time ls -l`, for example) in your terminal. Be sure to add `./` to the beginning (or use the full path to your `time` executable file if you are in another directory), otherwise the builtin `time` will be called.

We've also provided a test executable to run basic tests on your time implementation. Note that although these tests are similar to those that will be run on the autograder they are not identical, so passing locally does not guarantee you will receive full credit. It is still your responsibility to ensure you have functional code.

Note that we only care about wall-clock time, and we recommend using `clock_gettime` with `CLOCK_MONOTONIC`.

Pro tip: 1 second == 1,000,000,000 nanoseconds.

Nota bene:

- You **may not** use the existing `time` program.
- You must use `fork`, `exec`, and `wait` (no other solutions will be accepted).
- If the child process does not terminate successfully (where its exit status is non-zero), you should exit with status 1 *without* printing the time.
- We will only run `time` with one program.
- The commands we will run can take any number of arguments.
- Do your time computations with double-precision floating pointer numbers (`double`) rather that single-precision (`float`).
- We have provided functions in `format.h` that we expect you to use wherever appropriate.

Useful Resources

- Program arguments: argc & argv
- fork, exec, wait
- fork and waitpid

env

In this lab, you will be implementing a special version of `env`.

```
env – run a program in modified environments
```

Usage:

```
./env [key=val1] [key2=val1] ... -- cmd [args] ..
```

Please re-read this section *multiple* times before starting:

- Each variable is in the form of `NAME=v1`, separated by spaces.
- Values may contain references to environment variables in the form `%NAME`, including variables that were set earlier. As a result, variables should be processed from left to right.
- Each reference should be replaced with its value.
- The names of variables (both in `key` and in `value`) only contain letters, numbers, or underscore characters.
- For each environment variable `key`/`value` pair, `env` will assign `value` to `key` in the child environment.
- Each execution must be done with `fork`, `exec,` and `wait`.
- The last variable/value(s) pairing is followed by a `--`.
- *Everything* following the `--` is the command and any arguments that will be executed by env.
- Invalid input should result in the usage being printed. Invalid usage includes:
    - Cannot find `--` in arguments
    - Cannot find `=` in an variable argument
    - Cannot find `cmd` after `--`

This is the canonical example and a practical use case:

```
$ ./env TZ=EST5EDT -- date
Sat Sep  9 19:19:42 EDT 2017
$
```

Alternatively:

```
$ ./env TZ=EST5EDT -- date
Sat 09 Sep 2017 07:19:42 PM EDT
$
```

Example of using references to other variables:

```
$ ./env TEMP=EST5EDT TZ=%TEMP -- date
Sat Sep  9 19:19:42 EDT 2017
$
```

Accordingly:

```
$ ./env TEMP=EST5EDT TZ=%TEMP -- date
Sat 09 Sep 2017 07:19:42 PM EDT
$
```

This has the exact same behavior as before, because `TEMP` is first set to `EST5EDT`, and then when `TZ` is set to `%TEMP`, the value of `EST5EDT` is retrieved and then `TZ` is set to that. Notice that the variables are set sequentially, or else it wouldn't work.

We have provided you with a reference executable `env-reference` for you to test your understanding of `env`'s expected behavior. You can also use it to see if your `env`'s output matches the expected output.

Again like `time`, you can play with Linux's builtin `env` command by typing `env <var-list> <command-name>` (`env MYVAR=CS341 printenv`, for example) in your terminal. Again, remember to add `./` to the beginning (or the full path to your `env` executable file if you are in another directory), otherwise the builtin `env` will be called. **Do not use the built-in env, or you will immediately fail the assignment**

In addition, keep in mind that the builtin `env` uses `$` instead of `%` to denote environment variables. In practice, it can be very useful to change some environment variables when running certain commands.

Extra: Why Env?

For example, you may notice people write `#!/usr/bin/env python` on the first line of their Python script. This line ensures the Python interpreter used is the first one on user's environment `$PATH`. However, users may want to use another version of Python, and it may not be the first one on `$PATH`. Say, your desired location is `/usr/local/bin` for instance.

One way to solve this is by exporting `$PATH` to the correct position in your terminal, however, this may mess up other commands or executable under the same session.

An alternative and better way is to use our `env`, and enter:

```
./env PATH=/usr/local/bin -- ./XXX.py
```

then it runs the script with the desired Python interpreter.

Nota bene

- You **may not** use the existing `env` program. (Our specification is different than the existing `env` program.)
- You **may not** replace `%` with `$` or use `wordexp(3)`.
- You **may not** use `execvpe`, `execve`, or `execle`.
- All changes in environment variables and execution must happen only in the child process.
- You must use `fork`/`exec`/`wait`.
- If a variable doesn't exist, interpret its value as a zero-length string.