Lab 2: We're Going Deeper Than Subprocess `1.0.0`

ECE 353: Systems Software
Jonathan Eyolfson
January 30, 2023 **Due: February 13, 2023 @ 11:59 PM ET**

In this lab you'll create a small library called *Subsubprocess* (ssp) that acts as a subreaper in addition to creating and monitoring processes. Your implementation should use the concepts learned during the lectures, along with some new system calls. As a change, you create library called `libssp` this lab, not an executable. You'll be using Git to submit your work and save your progress.

Lab setup. Ensure you're in the repository (`cd ~/ece353-labs`) directory. Make sure you have the latest skeleton code from us by running: `git pull upstream main`.

This will create a merge, which you should be able to do cleanly. If you don't know how to do this read Pro Git. **Be sure to read chapter 3.1 and 3.2 fully.** This is how software developers coordinate and work together in large projects. For this course, you should always merge to maintain proper history between yourself and the provided repository. **You should never rebase in this course, and in general you should never rebase unless you have a good reason to.** It will be important to keep your code up-to-date during this lab as the test cases may change with your help.

You can finally run: `cd ssp` to begin the lab.

Your task. You're going to create a process manager similar to the Python `subprocess` module in some respects. Your version is going to be a C library with the following API:

```
void ssp_init();
int ssp_create(char *const *argv, int fd0, int fd1, int fd2);
void ssp_send_signal(int ssp_id, int signum);
int ssp_get_status(int ssp_id);
void ssp_wait();
void ssp_print();
```

The description of what each function should do is below:

`void ssp_init()`

This will always be called once before a user makes any other call to your library. You should initialize or setup anything you need here.

`int ssp_create(char *const *argv, int fd0, int fd1, int fd2)`

You will create a new process in this function, that new process should eventually call `execvp(argv[0], argv)`. You must set file descriptors 0, 1, and 2 to match the arguments `fd0`, `fd1`, and `fd2` through calls to `dup2`. Afterwards you must then close all other file descriptors except for 0, 1, and 2.

You will not rely on library users to properly manage file descriptors (and in fact they can't since your library calls `fork`). You may find some interesting file descriptors left open by VSCode if you're using the dev container. Your experience with Lab 1 will help you close the other file descriptors, since each process has an `fd` directory. As a hint: within this directory only consider files with `d_type` set to `DT_LNK`.

The `execvp` wrapper will re-use the current `env` variable, and search for the program given by `argv[0]` using the `PATH` environment variable (this variable is a list of directories to search for executables). If `execvp` fails you should immediately exit with `errno` set by the `execvp` call. Your library should record the process ID of the newly created process, the name (you need to copy the `argv[0]` string, the library user may re-use the memory for something else so you can't rely on it), and its status.

You should record the created process and initially set the status to -1 to indicate it's running. You should return a unique `ssp_id` that your library will use to refer to this created process. The IDs should be sequential and start with `0`.

```
void ssp_send_signal(int ssp_id, int signum)
```

You should send a signal `signum` to the process referred to by `ssp_id`. If the process is no longer running, you should not return an error and instead do nothing.

```
int ssp_get_status(int ssp_id)
```

You should return the current status of the process referred to by `ssp_id` without blocking.

```
void ssp_wait()
```

This function should block and only return when all processes created through `ssp_create` terminate. As a sanity check, all processes should have a status between `0` and `255` after this call completes.

```
void ssp_print()
```

This is a non-blocking call that outputs the PID, name, and current status of every process created through `ssp_create`. This should reflect the current state of the processes, so you should query them in this function.

You should start by printing a header, which will be `PID` right-justifeid with a width of 7 characters, a space, then `CMD` left-justified to the width of the longest process name, a space, then `STATUS`. After the header, for each process created by `ssp_create` you should output its pid, name, and current status. Recall that the name is your copied `argv[0]` string.

`status`

The status of each process should match it's exit status if it exits normally. However, if the process terminates through a signal you should set the status to be equal to the signal number plus 128. Recall that a status of -1 means the process is currently active.

Errors. You need to check for and properly handle errors. Some errors are expected and should be handled without additional output or exiting the process. For fatal errors, you should exit with the `errno` of the first fatal error.

Become a Subreaper (20% of the grade). This task may make your implementation more complex, or you may have to scrap your first attempt. It's advised to complete the other parts of the lab first. However, when you're ready you should add a call to `prctl(PR_SET_CHILD_SUBREAPER)` in `ssp_init` to become a subreaper.

A subreaper will adopt all orphan processes created by child processes. In other words, your process will be the new parent when an orphan process gets re-parented (instead of `init`). It'll be your job to call `waitpid` on any adopted process as soon as they terminate.

As part of being a subreaper you should record any time an adopted process terminuates. You should record its pid and status. For its name you should simply call it `"<unknown>"`. Add these unknown processes to be displayed when you call `ssp_print` **after** all the processes managed directly by the library. These processes should be output in the order they terminate.

Building. First, make sure you're in the ssp directory if you're not already. After, run the following commands:

```
meson setup build
meson compile -C build
```

Whenever you make changes, you can run the compile command again. You should only need to run setup once.

Testing. You cannot execute your library directly, however you can run the test programs manually. Please find the files in `tests/*.c`. You should be able to read and understand what they're doing with your library. You'll find the executables in `build/tests/*`.

You may also choose to run the test suite provided with the command:

```
meson test -C build
```

Grading. Run the `./grade.py` script in the directory. This will rebuild your program, run the tests, and give you a grade out of 100 based on your test results. Note that these test cases may not be complete, more may be added before the due date, or there may be hidden test cases. These labs are new, so we may need to change.

Tips. You'll want to read the documentation on some more C functions (some are light syscall wrappers). Some header files you'll need to use are provided for you in the skeleton code. You may include additional parts of the standard library. It's highly recommended to at least use the following functions:

```
open fdopendir readdir closedir dup2 waitpid fork execvp malloc perror exit
```

Submission. Simply push your code using `git push origin main` (or simply `git push`) to submit it. *You need to create your own commits to push, you can use as many as you'd like*. You'll need to use the `git add` and `git commit` commands. You may push as many commits as you want, your latest commit that modifies the lab files counts as your submission. For submission time we will *only* look at the timestamp on our server. We will never use your commit times (or file access times) as proof of submission, only when you push your code to the course Git server.