15213 Lecture 20: Exceptions and Signals

Learning Objectives

- Explain how the shell uses process control functions to launch the executable the user requested.
- Motivate the need for special commands that the shell treats differently.
- Recognize the advantages and limitations of background jobs.
- Attain proficiency in running multiple commands at once from the same shell.

Getting Started

The directions for today's activity are on this sheet, but refer to accompanying programs that you'll need to download. To get set up, run these commands on a shark machine:

1. `$ wget http://www.cs.cmu.edu/~213/activities/ecf-signals.tar`
2. `$ tar xf ecf-signals.tar`
3. `$ cd ecf-signals`

1 The Shell

If you've made it this far in the course, you've certainly used the Unix command line. This interface is provided by a program called the **shell**, which is responsible for repeatedly prompting the user for a command to run, then launching the appropriate program. In this activity, we'll investigate this process, starting with the simple case.

1.1 Commands and Processes

Examine the program `getpid.c`, which simply prints out its PID. When you're ready, build all the programs by typing: `$ make`.

1. Try running the resulting executable: `$ ./getpid`. Thinking back to last class, which process control library functions do you expect the shell called when you typed this command, and in what order?
   The shell calls `fork` followed by `execve` in the child process and `wait` or `waitpid` in the parent process.

Now let's convince the shell to explain what it's doing. We've provided a shared library, `libtrace.so`, that intercepts and logs calls to some C standard library functions. We can ask the runtime dynamic linker to load this library before `libc`, so that its function implementations override the system ones with the same names. We'll do this by defining a special **environment variable**, a type of configuration parameter that a process inherits when it is created and can check using the C library. Start a traced instance of the shell using: `$ LD_PRELOAD=./libtrace.so sh`.

2. Try rerunning `$ ./getpid` under this new shell. Does it make the same sequence of function calls you expected? Draw a process graph, labeling each vertex with a single call.

```
        --> execve --> exit

        |              |

o --> fork --------> waitpid --> ...
```

3. Now look back at the PIDs listed in the trace output. One of them corresponds to the `getpid` process, and the other to the shell. How are the two processes related? How can you tell?
   The `getpid` process is the child of the shell process because the PID of the `getpid` process is returned to the parent (shell) process by the `fork` call (0 is returned to the `getpid` process by the `fork` call).
4. In the same shell, try running: `$ ./notpresent`. How does the shell find out that something has gone wrong? How does it craft the error message it displays?
   The call to `execve` returns -1, so the shell checks `errno` and passes its value to `strerror`, which chooses the error message from the libc error messages.

1.2 Command-line Arguments

Next we'll see how the shell passes command-line arguments to the program in order to (eventually) make them appear in the `argv` array that gets passed to `main()`.

5. We've provided a trivial program, `out`, that simply prints out its command-line arguments. Working in the traced shell, try running it with a sentence of your choice (e.g., `$ ./out Hello, world!`). To which process control library function did the shell pass your sentence? Did the shell pass the arguments to that library function in the positions you'd expected?
   The shell passed my sentence to `execve`. The arguments are in the correct order (the first is the path to the program, the second is the command line arguments array `argv`, and the third is the environment variable array `envp`).

1.3 Path Search

When we ran the programs provided with this activity, we had to specify where to find their executable files by prefixing the filename with a path (the `./`). In contrast, when we run programs that are already installed on the system, we don't have to do this[1].

6. Still running in the traced shell from before, try printing the source code of the `out` program: `$ cat out.c`. Based on the trace output, what is the path to the `cat` program's executable?
   The path to the `cat` executable is `/bin/cat`. You may also see `/usr/bin/cat`.

If no path is provided, the shell looks for a file with the program's name in a configurable set of locations. This search is controlled by the `PATH` environment variable, which contains a `:`-separated list of directories to be searched in order.

7. If any words in the user's command string begin with a `$`, the shell expands them to the contents of the named variables[1]. Use this feature to list the contents of the `PATH` variable: `$ ./out $PATH`. How many directories did the shell have to check before it found `cat`?
   Answers may vary depending on a user's precise `PATH` configuration, but `/bin` is often within the first five directories of a user's `PATH`.

[1] Note that your shell is *not* required to support this feature.

1.4 Another Type of Command

The `out` program is actually redundant: Unix-based operating systems come with an equivalent one called `echo`.

8. From the traced shell, run `$ echo $PATH` and compare against the output from the last question. What do you notice? Can you come up with a possible explanation?
   No child process is `fork`ed to execute `echo`. In fact, no process management library functions are called at all. This is because the `echo` command is implemented directly by the shell.

You just encountered a **built-in command**: its functionality is implemented directly by the shell. Sometimes a single name corresponds to both a built-in command and an executable file:

9. Now try running `$ /bin/echo $PATH` from the same shell. What happens this time?
   A child process is `fork`ed to run `echo`, as before.

As you've discovered, when a name corresponds to both a built-in command and an executable file, the shell's version takes precedence. However, it's still possible to override this decision if you want to run the separate utility.

Whereas some built-in commands (such as `echo`) are implemented within the shell only for performance reasons, others exist because they would be harder to implement as a separate program. To see an example of this, try: `$ exit`.

2 Multitasking

One of the key features of an operating system is support for running multiple programs at once. It's probably no surprise, then, that Unix shells have long provided an interface for running and managing multiple commands at the same time. Let's try it out!

2.1 Background Jobs

Since a shell doesn't have multiple "windows," its multitasking is based on a feature called **background jobs**. If you're still running the tracing shell, exit back to your normal one before answering the first question.

10. The `sleep` program hangs for a number of seconds, then exits. Try invoking with `$ sleep 3`.
    Then try `$ sleep 3 &`.
    How do the two cases differ? Take a guess what the shell did to achieve this new behavior.
    `$ sleep 3` waits for three seconds before returning to the user prompt while `$ sleep 3 &` prints the job id and the PID of the child process and returns to the user prompt immediately. Both cases `fork`ed a child process to execute `sleep`, but in the former, the shell also called `waitpid`. In the latter, the shell does not appear to call `waitpid`.

Now open a traced shell again (`$ LD_PRELOAD=./libtrace.so sh`) and use it to check your answer to the previous question. Continue using this shell as you answer the next questions, until instructed to relaunch it.

2.2 Job Control: Foregrounding

Once created, a background job persists until finished. As long as it still exists, it may be converted into a foreground job like the ones we saw in the last section. To see why this is useful, consider the `prompt.c` program, which sleeps for a few seconds then prompts for user input. Read through its source code, then run it in the background: `$ ./prompt &`

11. Once the program issues its prompt, try typing `fortune` and hitting enter. Where does this input go? Why is this necessary?
    The input (`fortune`) goes to the shell process, which passes it to `execve`. The redirection of output from the background `prompt` program is necessary so that the user can interact with the shell's foreground process while one or more background processes continue to run.

You may have also noticed that the shell printed a line like this:

```
[1]+ Stopped(SIGTTIN) ./prompt
```

This was to inform you as the program became blocked on user input and transitioned from the running state to the stopped one.

12. Try bringing the job into the foreground: `$ fg`. What library call(s)[2] does the shell make in response?
    The shell calls `killpg`, which sends a `SIGCONT` signal to a process, followed by `waitpid`.

13. Where on the system is `fg` located? How could you tell?
    `fg` is a shell builtin, because it does not require a child process to be `fork`ed and causes a specific shell functionality (running `killpg`).

14. Where does subsequent user input get directed?
    Subsequent user input is directed to the `prompt` process, which is now the foreground process.

2.3 Job Control: Backgrounding

Jobs can also be converted in the other direction: from foreground to background. Run the application again, in the foreground this time: `$ ./prompt`. Then immediately press Ctrl-Z, before the prompt appears[3]. (If you're too late, hit Ctrl-Z anyway, relaunch, and try again.)

15. Wait for the program to finish. What do you notice?
    The program never finishes, because it was stopped by the `SIGTSTP` signal sent by the shell (to the foreground process group) after receiving Ctrl-Z.

16. Now run: `$ bg`. What does this command do?
    It resumes execution of a suspended (background) process in the background.

[2] You may not recognize this function yet, but we'll cover it today. The name is somewhat misleading: as used here, it actually does not terminate the process.

[3] You'll notice that the shell doesn't make any process control calls when you do this. This is because this keystroke is handled by the console, which directly notifies all processes that share the foreground job's process group ID. Shells assign each command its own process group using `setpgid()` so such keystrokes won't affect the shell itself.

2.4 Job Control: Listing

After the last section, you should have at least one instance of `prompt` still running in the background. Once you have a lot of background jobs running, it's easy to lose track of them. To help you out, the shell keeps track of each one and assigns it a short **job ID** so that you can refer to it directly.

17. Try running: `$ jobs`. One of the jobs has a `+` next to it, indicating that it is the default target of the next `fg` or `bg`. Do these commands default to the most recently or least recently backgrounded task?
    The `fg` and `bg` commands default to the most recently backgrounded task.

You can operate on a specific job by passing its job ID; for instance, you might type `$ fg %2` to foreground the second-oldest background job. Try doing this until you've cleared out the job list.

Pick up this sheet only if you have extra time at the end of the activity.

2.5 The Zombie Apocalypse *(Advanced)*

Recall that unless a long-running process reaps its terminated children using the `waitpid()`[4] function, these processes persist as zombies!

18. Is there any type of job you worry the shell might not be reaping correctly? *(Hint: Consider the sequence of calls used to launch a command, and how it differs based on the job type.)*
    We may worry that the shell is not correctly reaping background jobs, since we saw earlier that the shell does not call `waitpid` for processes run in the background.

19. Try running this again: `$ sleep 3 &`. Afterward use the `ps` command to check whether the `sleep` process becomes a zombie ("`<defunct>`") after finishing.
    The `ps` command does not show `sleep` as a `<defunct>` process. In fact, it does not show `sleep` at all, meaning it was correctly reaped by the shell.

As it turns out, we've been hiding some `waitpid()` calls from the output so far for brevity. Our call-tracing library decides whether to do this using a special environment variable. Exit the tracing shell you've been using and rerun it as: `$ LD_PRELOAD=./libtrace.so SHOW_WNOHANG= sh`. Then rerun the command from the previous question under the new shell.

20. Rerun the command from the previous question under the new shell. When do the new `waitpid()` calls appear? What event prompts this?
    The new `waitpid()` calls appear right before the user prompt is displayed.

21. Based on your answer to the previous question, what can you conclude about the shell's control flow?
    The shell calls `waitpid` before showing the user prompt to reap terminated background processes.

22. Notice the new option flag that is being called to these additional `waitpid()` calls. Consult the documentation (`$ man waitpid`) to determine the meaning of this flag. How does it change the function's behavior?
    The new `WNOHANG` flag makes `waitpid` return immediately if no child processes has exited.

23. Running a couple more commands, notice that the last call to `waitpid()` consistently returns `-1`, indicating that the process has no children. Why might the shell be making this useless extra call? *(Hint: Consider what the shell program's code structure might look like.)*
    The shell calls `waitpid` in a loop before displaying the user prompt in order to reap all terminated background processes.

[4] or `wait()`