Shell Lab Roadmap

Hey all, here's a roadmap for shell lab that my TAs made when I was taking the class! Good luck!

Shell lab has a lot of moving parts and require synthesizing information about a lot of different system calls and concepts. But don’t worry, as it is possible to tackle those things on after the other.

This roadmap outlines a way of approaching the lab piece wise and to not get overwhelmed. (It is not the only solution, but it is a good strategy)

Important note : tsh_helper is overzealous when it comes to signal blocking. This is meant to help you prevent race condition in the later stage of the lab, but initially it can get in the way, and possible warn for something that is not an issue when you have no signal handlers at all.

This roadmap is a suggestion. Some alternative ordering are possible (see at the end)

Getting started

Trace 00: This one should work out of the box

Trace 01: Start by implementing the is this a builtin command logic, handle correctly the one builtin command and stub out the other ones + add some stub for real jobs (that prints the command line ?)

Trace 02: First time creating a child process. Don’t worry about the job list, reaping it or waiting for it right now. Get the new child and child

loads and run the correct binary logic running. (These a 2 syscalls)

Trace 03: Now that you can create process, you can start thinking about how to reap them when they are done running. You can use the naïve obvious method to wait for the foreground job and reap it, for now, but remember that you will need to change this in order to get simultaneous background and foreground job to work properly. (including reaping them in a timely fashion).

Trace 04: Check that your run a new program logic handles argument correctly. Now would be a good time to figure out how to block signals in the signal mask and how to restore it, and add the proper job list management call. Make sure to block signals just around the critical section where global state is being changed and may not be consistent.

Trace 05: Background job creation : Change the logic at the end of eval to only wait for foreground job.

The shell job management core

Trace 06: Now is time for the trickiest part of shell lab. You need to completely change the way you are waiting for foreground jobs, and add the logic to reap all the childs in a timely fashion, but without getting stuck anywhere. There are 3 simultaneous tricky parts and it is quite possible you will break some of the earlier traces doing so.

- Making sure you detect that at least a child is not longer running, and reap all the child that can be currently reaped. You will need to read the manual page carefully. You are likely going to use a signal handler so review signal safety.
- Make sure your job list manipulation is safe and race free. Review

what global state exist and is being changed (this may actually include kernel state changed by syscalls and state in your program), and ensure no signal can observe the state when it is not consistent, or change it when it is not expected to change. (Technically you only need to block the signal for which you have a handler, but blocking 3 of them will make the job list happier and the warning disappear).
- Make sure eval returns as soon as there is no longer a running foreground job, and no sooner. You will probably have to rely on the job list as the ground truth, as updated by the first thing, and making sure it is not messed up by races

Trace 07: Add the job builtin. Hopefully your job list is correctly updated.

Trace 08: Some tests to make sure the logic you built in trace 06 was robust. If not, it is possible that you may fail to reap a child and delete it from the job list

Let's go a little bit out of order on the traces here.

Handling job termination conditions

Trace 12: Sometime job crash / fail / do not terminate properly, so we need to do something about it when reaping them. Read the man page for the sys call used to reap and (paying attention to signal safety) print out the appropriate message when such thing happen.

Trace 13: Stopping isn't the same as dying you may need something special to make sure you get told about those too, similar to trace 09

Trace 11, trace 14: Other different child process crashes

Handling other signals your shell receives

Trace 09, trace 10: Sometime you get a signal and you are supposed to forward it to the foreground job. Time to add two new signal handlers.

Trace 15, trace 16: Make sure you don't forward signals to innocent bystanders

Trace 17, trace 18: Make sure you do send the signals to everyone involved. This is actually achieved thanks to a small call at child process set-up (that groups all the relevant processes of the job), make sure you did not forget it. Then kill can send signals to all those processes at once.

Trace 19 to 21: Race condition and other edge cases trying to break your shell.

Handling the various other builtins

Trace 22, 23: A new builtin, bg (you should be aware that someone else could tell your child to continue, and you should probably still print about it, if that could inform the separation of concerns between telling a child to continue and printing stuff about t continuing)

Trace 24: Yet another builtin

Trace 25: Some stress test of those builtins

Trace 26: Race conditions are back

Trace 27: Probably similar to 17 and 18.

I/O redirection and error handling

Trace 28: This is where I/O comes really into play. Remember that you can do stuff in your child process before loading and running the new program. Review the Unix sys calls for I/O and what file descriptors are. Remember that you need to have one close call per open call at program execution, in each process.

Trace 29: Same thing

Trace 30: The job command is done differently, look carefully at the interface of the API you use, no need to fiddle with your shell stdout.

Trace 31: tests that all of this is robust

Trace 32: Cause a bunch of errors in various syscalls. It is not exhaustive, but make sure to review all of your error checking, and ideally this should just be a matter of changing the action on those error to match the message and behaviour from the shell. (Be careful with error in the child process before loading the new program)

Looks like you are done. Time to get the style right (error checking is already done properly everywhere hopefully ;-)

Alternative orderings :

- It is possible to implement some amount of I/O redirection without the signal handling stuff. This mean trace 28-32 can go right after trace 08.
- There is some potential of doing things a little bit out of order in the signal and built-in stuff, but you have to look carefully at the traces to see what they require, the order I suggest is probably the

more robust. Trace 22-26 require trace up to 8 and 11-14, but not the signal forwarding logic. Trace 27 does require everything before it.