Shell LAB: in STY17
Project 2 (11%) and HW 2 (3%)

Gunnlaugur Kristinn Hreidarsson
Reykjavik University
Gunnlaugur15@ru.is

Hjalmar Orn Hannesson
Reykjavik University
Hjalmarh15@ru.is

February 9, 2017

Abstract

The Shell Lab is the second project of this semester in the course Styrikerfi and also the first group project. It involves implementing a small Unix shell and by doing so learning more about process control and signaling. As a programmer this helps us understand how applications interact with the operating system. We found this to be very interesting because even though we've used a shell many times we have never really thought deeply about the inner workings or the implementation of it. Until now.

I. Introduction

Write a short history (time-line) of the Unix shells and mention a few landmark changes. Cite the sources you build your work on (Here it is fine to use regular web pages)

A shell is basically any program that lets you, the user, send commands to the computer. A Unix shell is a shell that provides a Unix-like or *nix command line user interface.

The first such shell was the Thompson shell, developed by Kevin Thompson and introduced in the first Unix version in 1971. This introduced shell concepts such as I/O redirection and piping but was however not a scripting shell.[1]

The PWB Shell was an upgrade of the Thompson shell which came out ca 1975-77 and incorporated more features into the shell. It added shell variables, executable scripts and the use of if/then/else/endif and while statements.[2]

. The Bourne shell, often called with sh, was one of the most widely used an influential of the shells, along with the C shell. The Bourne shell was the shell for Unix version 7 distributed in 1977. It was a complete rewrite of previous shells, written by Stephen Bourne at Bell Labs. It has sparked many extensions that are still used today, such as[3]:

- Korn Shell(1983) - a sort of middle ground between Bourne shell and C shell
- bash(1989) - Bourne Again Shell, the default shell for most Linux and macOS systems.
- Z shell(1990) - zsh, the shell used by our beloved skel.ru.is

The C shell, csh, was developed, around the same time as the Bourne shell, by Bill Joy. It was released initially in 1978. At the time it differed from the other shells because of its interactive features and overall style. It was created to be more like C in style, because most of the Unix system had been written in C. Nowadays, on many systems such as MacOS and Red hat linux, csh is actually tcsh, an improved version of csh.

The Unix shell has come a long way since the Thompson shell, around 40 years ago, and chances are you're using an improved version of one of the old Unix shells.

II. Methods

Like most good assignments, this one looked pretty challenging at first. Writing your own shell is no easy task. So after reading the assignment description we watched the lecture by Freysteinn and got started. After seeing that some of the functions were already implemented we didn't feel as overwhelmed.

We started by implementing the "simple" things in the assignment, like checking for a built in command and executing it and the waitfg function. Our builtincmd function basically parses the command line input and checks if it is one of the built in functions. The waitfg function is just a busy loop which sleeps while our pid argument is the same as the pid of the foreground process. Next we used the parseline function to figure out which variables needed to be in the eval function. We then determine if the user is executing a built in function and if not, we block the signals with sigprocmask and fork. Then we unblock the signals and either try to execute the command with execve or add it as either a foreground or background job.

The SIGINT and SIGTSTP handlers proved pretty straightforward. We basically find the job in the foreground, kill it(using SIGINT or SIGTSTP) and delete it from our jobs list. In the case of SIGTSTP we set the jobs state to ST(stopped). We made sure to use -pid instead of pid in our kill function so as to send the singals to the entire foreground process group.

The functions that caused us the most trouble was the dobgfg function and the SIGCHILD signal handler. The sigchild handler mostly caused us trouble because we neglected it for a while. We thought we'd be able to work on other things and save it for later but we realized later that we needed the handler for other things to work. We then just read the appropriate chapters in the book and used WIFEXITED, WIFSIGNALED and WIFSTOPPED in a while loop with a waitpid that was almost identical to the one in the book. The main difference was that we used the WNOHANG and WUNTRACED options that were recommended in the project description. Our function is reaping the terminated children and deleting the jobs from our job list or changing the state of the job according to the status.

We had some trouble with the dobgfg function, mostly due to error handling. We had trouble finding the right way to use the argv variable to find the things we needed. But once that came we just went through the possible things we could put in with fg/bg and tried them out in tshref to see what output should be printed. We then added error checks for these things. Then we check if the user entered fg or bg and then change the job state of the job and kill it with SIGCONT

We encountered a very stupid error that kept us baffled for at least an hour. Our jid kept coming back as 0 when we tried to print it out. We were going crazy over this until we decided to check the given function pid2jid. We had apparantly accidentally changed it at some point. This teaches us to never just trust that some function works.

III. Results

Our project was pretty straightforward. Hjalmar was sick at home during the week so we were in a bit of a race against the deadline. We really didn't have time for any experiments(save for the trace tests given) so I don't really see the need for this section.

IV. Discussion

We found this project to be pretty interesting, albeit a bit different from the last ones. Attacklab and Bomblab(from Tolvuhogun) involve a bit of an "action" element where you feel like you're sort of in a game, whereas this project was more of a "normal" project if you will. This was however fun in a different way, as it proved to be challenging towards the end.

The main thing we take from this assignment is READ THE BOOK. Especially if your teacher mentions it in the assignment description. Things proved so much easier when we just gave ourselves time to read things over instead of trying to code right away.

V. HomeWork Questions

Following are the questions you need to answer for the HomeWork section. This part is worth 3% of the total grade for the project.

Question 1: The question is about reentrancy in computing.

a) What is *a race condition* in computing?

WRITE YOUR ANSWER FOR 1.a HERE
Race condition is a synchronization error.
It happens when child terminates before the parent is able to run, then addjob and deletejob will be calleed in the wrong order.
We can prevent this to happen if we block SIGCHLD signals before the call to fork and then unblocking them only after we have called addjob. Then we know that the child will be reaped after it is added to the job list.

b) What is a reentrant function?

WRITE YOUR ANSWER FOR 1.b HERE

It has only access to the local variables, no static, global or non-constant data nor return the address to them, must only work only on the data provided to it by the caller.

Its good practise to use them when we are handling signals (for example in "csapp.h" there is good SIO-package(safe in-/output) to use, instead using "printf" it uses "write")

c) How do you prevent signals from being received? Give a few examples, at least one of which blocks multiple signals.

```
#include <signal.h>
int main() {

  /*FIRST.. We can ignore the signal with
  SIG_IGN but if we are dealing with SIGKILL
  or SIGSTOP then it will not
  be ignored(or be caught) */

  signal(SIGINT, SIG_IGN);
  exit(0);

  return 0;
}
```

```
#include <signal.h>
int main() {
  int sigaction(int signum, struct sigaction *act,
                struct sigaction *oldact);
  /*SECOND.. We use
W. Richard Stevens's approach
    and use sigaction there we install
    our  */

  act.sa_handler = handler;
  sigemptyset(&act.sa_mask);//sa_mask specifies
                            //mask of signals
                            //which should be b


  act.sa_flags = SA_RESTART;//restarts syscalls

  if(sigaction(signum, &act, &oldact) < 0){
    unix_error("signal␣error");
  }
  return oldact.sa_handler;
}
```

Question 2: Following is some code that forks child processes and tries to reap them with a handler. The code is however not working as it should and you have been asked the help him out. Find at least 3 flaws?

```
void handler1( int sig ) {
  if ((waitpid(-1, NULL, 0)) < 0)
    sio_error("waitpid␣error");
  Sio_puts("Handler␣reaped␣child\n");
}

int main( int x, int y ) {
  int i, n;
  char buf[MAXBUF];
  if(signal(SIGCHLD, handler1) == SIG_ERR)
    unix_error("signal␣error");
  for (i = 0; i<3; i++) {
    if (Fork() > 0) {
      printf("Child␣process␣%d\n",
        getpid());
      exit(0);
    }
  }
  n = read(STDIN_FILENO, buf, 200)
  sleep(10);
  printf ("Parent␣output,␣%s\n" buf);
  exit();
}
```

WRITE YOUR ANSWER HERE

1. There is no busy loop around sleep function (to check that all children are reaped)
2. Race condition, after we fork we terminate without reaping the childs.
3. Syntax error need ';' in line n = read(STDIN $_FILENO, buf, 200)$
$4. Syntaxerrorneed",\text{"}inlineprintf("Parentoutput,s"buf);$
$5. Youneedtousewhilelooptoreapallthechilds. Ifstatementwillnotdothej$

Question 3:

```
int counter = 0;
void handler(int sig)
{
  counter++;
}

int main() {
  int i;
  signal(SIGCHLD, handler);
  for (i = 0; i < 6; i++) {
    if (Fork() == 0) {
      exit(0);
    }
  }
  /* wait for all children to die */
  while (wait(NULL) != -1);
    printf("counter␣=␣%d\n", counter);
  return 0;
}
```

Does the program output the same value of counter every time we run it? And if so, what is the value?

WRITE YOUR ANSWER HERE
It modifies some part of the global state, changing the number of times it gets called changes the program behavior. We dont know which one is returned first so by doing this the counter might go step ahead and that will pretty much ruin everything.

REMEMBER YOU CAN USE vfill TO FILL STREACHED PAGES AND ALSO TO REMOVE THE INSTRUCTION TEXT LIKE THIS ONE (ALL THE BLUE TEXT WITH IN emph BLOCKS)

References

[1] Wikipedia 2016 Thompson Shell https://en.wikipedia.org/wiki/Thompson_shell

[2] Wikipedia 2016 The PWB Shell https://en.wikipedia.org/wiki/PWB_shell

Wikipedia 2016 Bourne Shell https://en.wikipedia.org/wiki/Unix_shell#Bourne_shell