BOSC2013 OO1 - the BOSC shell

Sigurt Bladt Dinesen
sidi@itu.dk

October 1, 2013

Opgaver og løsninger

Værtsnavn

Kommando-prompten skal vise navnet på a den host den kører på

Det sørger funktionen *gethostname* for, som vist i appendix A. Funktionen bruger biblioteksfunktionen *fopen* til at skabe en filstrøm fra */proc/sys/kernel/hostname*, der så bruges i et kald til fgets, sammen med et array af arbitrær længde, og dettes længde. Dette array, givet som parameter til funktionen, bliver da fyldt med den første (og eneste) linje i filen *hostname*.

Enkeltstående kommandoer

En bruger skal kunne indtaste almindelige enkeltstå aende kommandoer, så a som ls, cat og wc. Hvis kommandoen ikke findes skal der udskrives enCommand not found meddelelse.

executecmd (appendix D) bruger biblioteksfunktionen *fork* til at starte en ny process der, via biblioteksfunktionen *execvp*, erstater sig selv med programmet der beskrives af testkstrengen indtastet af brugeren. Magien sker i linje 23 og 29.

Baggrundsprocessor

Kommandoer skal kunne eksekvere som baggrundsprocesser (ved brug af &) sådan at mange programmer kan køres på a samme tid.

I appendix D, linje 40-42, sørger programmet for kun at vente på den forkede process, hvis bg ikke er sat på Shellcmd'en.

Piping

Det skal være muligt at anvende pipes.

Kommandoer behandles rekursivt. Hvert kald opretter et rør, hvis ud-ende overskriver kommandoens stdin. Ind-enden sendes med til den foregående kommando; det næste lag af rekursion, der overskriver sin stdout med den. Det er en simpel løsning, der har den ulempe at rekursionsstakken overflyder hvis rør-kæden bliver for lang (25 kommandoer på mit system). Alternativt kunne man køre en løkke (eller to) over komandolisten.

Redirection

Der skal være indbygget funktionalitet som gør de muligt at lave redirection af stdin og stdout til filer.

To af *executecmd*'s parametre, std_in og std_out, sættes af den kaldende funktion (*executeshellcmd*, ej beskrevet i denne rapport) til de relevante file descriptors. *executecmd* overskriver den sidste kommandos stdout med std_out, og kopierer manuelt std_in til den første komandos stdin. Stdin delen er noget

uelegant, og kan alternativt gøres ved et look-ahead når der forkes. std_in kunne så overskrive den første kommandos stdin ligesom det det er tilfældet for resten af kommandoerne.

SIGINT

Tryk på a Ctrl-C skal afslutte det program, der kører i bosh shellen, men ikke shell'en selv.

Dette gøres ved at ignorere SIGINT i bosh processen (implementeret ved et kald til *signal*i *main*), og registrere en funktion, *siginttrap*, (appendix C) som handler i den forkede process. Det virker dog ikke helt efter hensigten, da siginttrap aldrig printer til konsollen.

A Retrieving the system hostname

```
1 char* gethostname(char *hostname){
2   FILE* f = fopen("/proc/sys/kernel/hostname", "r");
3   char* stat = fgets(hostname, HOSTNAMEMAX, f);
4   fclose(f);
5   hostname[strlen(hostname)-1]=0;
6   return stat;
7 }
```

B Forking and Executing a Command

```
1  int executecmd (Cmd* cmd, int std_in, int std_out, int bg){
2    if(cmd == NULL){
3      if(std_in != 0){
4        puts("copying");
5        void* buf[1024];
6        int size=0;
7        do {
8          size = read(std_in, buf, 1024);
9          write(std_out, buf, size);
10         printf("did %d bytes, errno is: %d\n", size, errno);
11       } while (size > 0);
12       close (std_in);
13     }
14     close (std_out);
15     return 0;
16   }
17
18   int pfds[2];
19   pipe(pfds);
20
21   executecmd(cmd->next, std_in, pfds[1], 0);
22
23   pid_t child = fork();
24   if(child==0){
25     signal(SIGINT, siginttrap);
26     dup2(pfds[0], 0);
27     dup2(std_out, 1);
28     close(pfds[1]);
29     if(execvp(cmd->cmd[0], cmd->cmd)){
30       printf("Error in: %s\n", cmd->cmd[0]);
31       if(errno == 2){
32         printf("%s not found\n", cmd->cmd[0]);
33       }else {
34         printf("errno set to: %d", errno);
35       }
36       exit (errno);
37     }
38   }
39
40   if(std_out == 1 && !bg){
41     waitpid(child, NULL, NULL);
42   }
43   else if (std_out != 1){
44      close(std_out);
45   }
46   return 0;
47 }
```

C A signal handler

```
1 void siginttrap(int signal){
2   printf("\nI gots sigint %d\n", signal);
3   exit (0);
4 }
```

D Code

D.1 bosh.c

```
/*

   bosh.c : BOSC shell

 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <fcntl.h>
#include <string.h>
#include <readline/readline.h>
#include <readline/history.h>
#include <errno.h>
#include "parser.h"
#include "print.h"
#include <signal.h>

/* --- symbolic constants --- */
#define HOSTNAMEMAX 100
#define PROMPT "$-> "

/* --- trap sigint --- */
void siginttrap(int signal){
  printf("\nI gots sigint %d\n", signal);
  exit (0);
}

/* --- use the /proc filesystem to obtain the hostname --- */
char* gethostname(char *hostname){
  FILE* f = fopen("/proc/sys/kernel/hostname", "r");
  char* stat = fgets(hostname, HOSTNAMEMAX, f);
  fclose(f);
  hostname[strlen(hostname)-1]=0;
  return stat;
}

/* --- execute a shell command --- */
//executecmd starts from the last cmd in the cmdline, and moves backwards,
//+hence, the previous command in the cmdline is the next cmd to be processed
//+by executecmd, and vice versa.
//Comments in the function refers to the cmdline when using the terms "next"
//+and previous.
int executecmd (Cmd* cmd, int std_in, int std_out, int bg){
  //if there are no more commands,
```

```
//+close the pipe and return
//if stdin is set, copy it to stdout before closing
//+doing it this way is basically shit, I might look into it later.
if(cmd == NULL){
  if(std_in != 0){
    puts("copying");
    void* buf[1024];
    int size=0;
    do {
      size = read(std_in, buf, 1024);
      write(std_out, buf, size);
      printf("did %d bytes, errno is: %d\n", size, errno);
    } while (size > 0);
    close (std_in);
  }
  close (std_out);
  return 0;
}

//make pipe, to bind the previous cmd's stdout to this one's stdin
int pfds[2];
pipe(pfds);

//setup the previous cmds recursively
//+bg is set to 0, as only the last cmd in the cmdline (first in terms of thi
//+should be affected by backgrounding
executecmd(cmd->next, std_in, pfds[1], 0);

//fork and exec current cmd
pid_t child = fork();
if(child==0){
  //copy the pipe's read part to cmd stdin,
  //+next cmd's
  //+then close the write part of pipe, we don't need it
  dup2(pfds[0], 0);
  dup2(std_out, 1);
  close(pfds[1]);
  signal(SIGINT, siginttrap);
  if(execvp(cmd->cmd[0], cmd->cmd)){
    printf("Error in: %s\n", cmd->cmd[0]);
    if(errno == 2){
      printf("%s not found\n", cmd->cmd[0]);
    }else {
      //there must be a better way of interpreting errnos?
      printf("errno set to: %d", errno);
    }
    exit (errno);
  }
}
```

```
  //if this is the last cmd (indicated by stdout being 1, this should be done
  //+differently) and bg isn't set, wait for the this cmd
  if(std_out == 1 && !bg){
    waitpid(child, NULL, NULL);
  }
  //only close the stdout fd if it isn't the stdout of the program,
  //+otherwise the REPL (yes, I called it REPL) would die
  //+neglecting to close stdout will prevent the reading cmd from terminating,
  //causing the dreaded "ls | cat sort of works, but ls | wc does not" problem
  else if (std_out != 1){
     close(std_out);
  }
  return 0;
}

int executeshellcmd (Shellcmd *shellcmd){
  int std_in = 0;
  if(shellcmd->rd_stdin != NULL){
    std_in = open(shellcmd->rd_stdin, O_RDONLY);
  }

  int std_out = 1;
  if(shellcmd->rd_stdout != NULL){
    std_out = open(shellcmd->rd_stdout, O_WRONLY|O_CREAT);
  }

  executecmd(shellcmd->the_cmds, std_in, std_out, shellcmd->background);
  printshellcmd(shellcmd);
  return 0;
}

/* --- main loop of the simple shell --- */
int main(int argc, char* argv[]) {
  /* initialize the shell */
  char *cmdline;
  char hostname[HOSTNAMEMAX];
  int terminate = 0;
  Shellcmd shellcmd;
  signal(SIGINT, SIG_IGN);

  if (gethostname(hostname)) {

    /* parse commands until exit or ctrl-c */
    while (!terminate) {
      printf("%s", hostname);
      if (cmdline = readline(PROMPT)) {
        if(*cmdline) {
          add_history(cmdline);
          if (!strcmp(cmdline, "exit")){
            terminate=1;
```

```
          } else if (parsecommand(cmdline, &shellcmd)) {
            terminate = executeshellcmd(&shellcmd);
          }
        }
        free(cmdline);
      } else terminate = 1;
    }
    printf("Exiting bosh.\n");
  }

  return EXIT_SUCCESS;
}

// vim: set ts=2 sw=2 et:
```

D.2 Makefile

```
all: bosh

OBJS = parser.o print.o
LIBS= -lreadline -lncurses
CC = gcc

bosh: bosh.o ${OBJS}
	${CC} -ggdb -o $@ bosh.o ${OBJS} ${LIBS}

clean:
	@rm -rf *o

cleanall: clean
	@rm bosh
```