Obligatorisk opgave 1

Operativsystemer og C

Bachelor in Software Development,
IT-University of Copenhagen

Jakob Melnyk, jmel@itu.dk
Frederik Lysgaard, frly@itu.dk
Jacob Claudius Grooss, jcgr@itu.dk

October 3rd, 2012

Contents

1 Opgavebesvarelse

1.1 Forord

1.2 Funktionalitet

Krav til funktionalitet:

1. bosh skal køre uafhængigt.
2. Display af hostname.
3. Bruger skal kunne kalde simple kommandoer. Udskriv "Command not found" meddelelse, hvis kommando ikke findes.
4. Kommandoer skal kunne køres som baggrundsprocesser.
5. Det skal være mulighed for at lave redirect af stdin og stdout.
6. Mulighed for at anvende pipes.
7. Funktionen exit skal være indbygget.
8. Cntrl+C skal afslutte det program, som kører, men ikke bosh shell'en.

Ekstra funktionalitet, vi har valgt at lave:

1. Display af "current working directory".
2. Mulighed for at kalde "cd" for at skifte directory.

1.3 Beskrivelse af implementation

1.3.1 Delopgave 1

Vi kalder ikke system() i vores kode. De mest relevante systemkald, vi laver, er pipe(), dup(), fork(), waitpid() og execvp(). Desuden har vi fileno(), close() og fopen() kald.

Tests

1.3.2 Delopgave 2

Vi ændrede 'gethostname' i bosh.c til 'get_user_and_hostname'. Metoden tager en char pointer og sætter den til at pege på en string med formatet "user@hostname". Vi tager 'user' via getenv og vi finder hostname i "/proc/sys/kernel/hostname".
Se linje ?? i bosh.c.

Tests

1.3.3 Delopgave 3

Vi bruger execvp() til at køre programmer i vores shell. execvp() leder efter et program i $PATH. Hvis den finder det, bliver programmet kørt og resten af det program, som execvp blev kørt fra, bliver termineret.
Se linje ?? i bosh.c.

Tests

1.3.4 Delopgave 4

Hvis et program bliver kørt som en baggrundprocess, bliver processen tilføjet til vores array af baggrundsprocessid'er. Desuden kalder vi ikke 'waitpid()' på det processid.
Se linje ?? i bosh.c.
Vi undgår zombieprocesser ved at kalde 'signal(SIGCHLD, SIG_IGN)'.

Tests

1.3.5 Delopgave 5

Hvis der i shellcmd bliver redirected stdin, stdout og/eller stderr, så kalder vi close() på dem, som bliver redirected. Derefter åbner vi filen, som er blevet redirected til. Vi kalder så dup() på den fil.
Se fx linje ?? i bosh.c.

Tests

1.3.6 Delopgave 6

Hvis der er mere end en enkelt command i 'shellcmd', når vi modtager den som argumentet til funktionen 'shell_cmd_with_pipes', piper vi vores fd. (Linje ?? i bosh.c).
Vi kalder så close() på write delen af vores pipe (linje ??) og dup() read. Hvis 'write_pipe' er mere end 0, lukker vi stdout og kalder dup på vores 'write_pipe'.
Hvis vores 'write_pipe' er 0 udenfor vores fork, så kalder vi close() på 'write_pipe'. Hvis der er flere kommandoer tilbage i vores 'shellcmd', kalder vi 'shell_cmd_with_pipes' med fd[1] (write delen af vores pipe) som parameter 'write_pipe'.

Tests

1.3.7 Delopgave 7

'executeshellcmd' checker om den sidste kommando er "exit". Hvis den er, kalder vi 'exit(0)' i shellen.
Se linje ?? i bosh.c.

Tests

1.3.8 Delopgave 8

I begyndelsen af vores main-metode, kalder vi 'signal(SIGINT, int_handler)'. Dette betyder, at 'int_handler' bliver kørt, hver gang bosh programmet modtager en interupt. 'int_handler' er vores interupt handler, som lukker alle forgrundsprocesser.
Se linje ?? i bosh.c.

Tests

1.3.9 Ekstra funktionalitet 1

Tests

1.3.10 Ekstra funktionalitet 2

Tests

A Test pictures

A.1

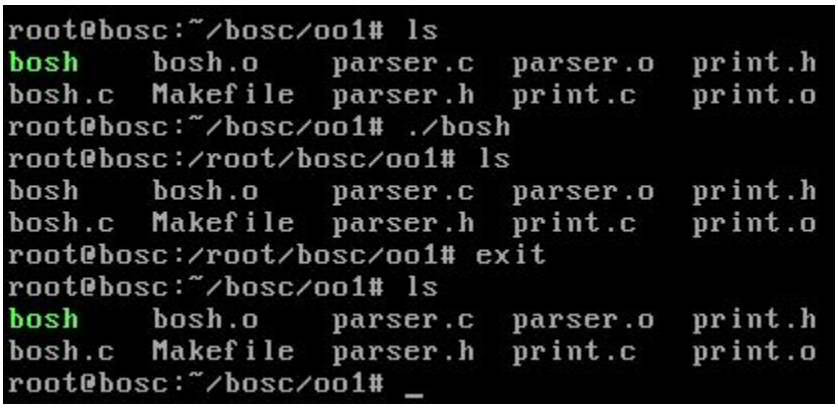
A.2

A.3

A.4

A.5

A.6 Exit test



A.7

B Makefile

```
all: bosh

OBJS = parser.o
LIBS= -lreadline -ltermcap
CC = gcc

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

clean:
	rm -rf *o bosh
```

C bosh.c

D parser.h

```
typedef struct _cmd {
    char **cmd;
    struct _cmd *next;
} Cmd;

typedef struct _shellcmd {
    Cmd  *the_cmds;
    char *rd_stdin;
    char *rd_stdout;
    char *rd_stderr;
    int background;
} Shellcmd;

extern void init( void );
extern int parse ( char *, Shellcmd *);
extern int nexttoken( char *, char **);
extern int acmd( char *, Cmd **);
extern int isidentifier( char * );
```

E parser.c

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "parser.h"

/* --- symbolic constants --- */
#define COMMANDMAX 20
#define BUFFERMAX 256
#define PBUFFERMAX 50
#define PIPE  ('|')
#define BG    ('&')
#define RIN   ('<')
#define RUT   ('>')
#define IDCHARS "-_.,/~+"

/* --- symbolic macros --- */
#define ispipe(c) ((c) == PIPE)
#define isbg(c)   ((c) == BG)
#define isrin(c)  ((c) == RIN)
#define isrut(c)  ((c) == RUT)
#define isspec(c) (ispipe(c) || isbg(c) || isrin(c) || isrut(c))

/* --- static memory allocation --- */
static Cmd  cmdbuf[COMMANDMAX], *cmds;
static char cbuf[BUFFERMAX], *cp;
static char *pbuf[PBUFFERMAX], **pp;

/*
 * parse : A simple commandline parser.
 */

/* --- parse the commandline and build shell commmand structure --- */
int parsecommand(char *cmdline, Shellcmd *shellcmd)
{
        int i, n;
        Cmd *cmd0;

        char *t = cmdline;
        char *tok;

        // Initialize list
        for (i = 0; i < COMMANDMAX-1; i++) cmdbuf[i].next = &cmdbuf[i+1];

        cmdbuf[COMMANDMAX-1].next = NULL;
        cmds = cmdbuf;
        cp = cbuf;
```

```
pp = pbuf;

shellcmd->rd_stdin     = NULL;
shellcmd->rd_stdout    = NULL;
shellcmd->rd_stderr    = NULL;
shellcmd->background = 0; // false
shellcmd->the_cmds       = NULL;

do
{
        if ((n = acmd(t, &cmd0)) <= 0)
        {
                return -1;
        }

        t += n;

        cmd0->next = shellcmd->the_cmds;
        shellcmd->the_cmds = cmd0;

        int newtoken = 1;
        while (newtoken)
        {
                n = nexttoken(t, &tok);
                if (n == 0)
                {
                        return 1;
                }

                t += n;

                switch(*tok)
                {
                        case PIPE:
                                newtoken = 0;
                                break;
                        case BG:
                                n = nexttoken(t, &tok);
                                if (n == 0)
                                {
                                        shellcmd->background = 1;
                                        return 1;
                                }
                                else
                                {
                                        fprintf(stderr, "illegal bakgrounding\n");
                                        return -1;
                                }
                                newtoken = 0;
                                break;
                        case RIN:
```

```
                                        if (shellcmd->rd_stdin != NULL)
                                        {
                                                fprintf(stderr, "duplicate redirection of s
                                                return -1;
                                        }
                                        if ((n = nexttoken(t, &(shellcmd->rd_stdin))) < 0)
                                        {
                                                return -1;
                                        }
                                        if (!isidentifier(shellcmd->rd_stdin))
                                        {
                                                fprintf(stderr, "Illegal filename: \"%s\"\n
                                                return -1;
                                        }
                                        t += n;
                                        break;
                                case RUT:
                                        if (shellcmd->rd_stdout != NULL)
                                        {
                                                fprintf(stderr, "duplicate redirection of s
                                                return -1;
                                        }
                                        if ((n = nexttoken(t, &(shellcmd->rd_stdout))) < 0)
                                        {
                                                return -1;
                                        }
                                        if (!isidentifier(shellcmd->rd_stdout))
                                        {
                                                fprintf(stderr, "Illegal filename: \"%s\"\n
                                                return -1;
                                        }

                                        t += n;
                                        break;
                                default:
                                        return -1;
                        }
                }
        } while (1);
        return 0;
}

int nexttoken( char *s, char **tok)
{
        char *s0 = s;
        char c;

        *tok = cp;
        while (isspace(c = *s++) && c);
        if (c == '\0')
        {
```

```
                return 0;
    }
        if (isspec(c))
    {
                *cp++ = c;
                *cp++ = '\0';
    }
        else
    {
                *cp++ = c;
                do
                {
                        c = *cp++ = *s++;
                } while (!isspace(c) && !isspec(c) && (c != '\0'));
                --s;
                --cp;
                *cp++ = '\0';
        }
        return s - s0;
}

int acmd (char *s, Cmd **cmd)
{
        char *tok;
        int n, cnt = 0;
        Cmd *cmd0 = cmds;
        cmds = cmds->next;
        cmd0->next = NULL;
        cmd0->cmd = pp;

        while (1)
        {
                n = nexttoken(s, &tok);
                if (n == 0 || isspec(*tok))
                {
                        *cmd = cmd0;
                        *pp++ = NULL;
                        return cnt;
                }
                else
                {
                        *pp++ = tok;
                        cnt += n;
                        s += n;
                }
        }
}

int isidentifier (char *s)
{
        while (*s)
```

```
    {
                char *p = strrchr (IDCHARS, *s);
                if (! isalnum(*s++) && (p == NULL))
                {
                        return 0;
                }
    }
        return 1;
}
```