IT University of Copenhagen

Operativsystemer og C

BOSC

Obligatorisk Opgave 1

Author:
Omar Khan (omsh@itu.dk)
Mads Ljungberg (malj@itu.dk)

02-10-2015

Contents

1 Formål

Formålet med denne opgave er at lære og bruge koncepterne omkring processer i operativsystemer, redirection af input og output, og pipe i et shell program skrevet i C. Forud for opgaven er vi blevet tildelt oo1.zip, der indeholder kildekoden som vi skal arbejde ud fra.

2 Baggrund

Ifølge opgavebeskrivelsen skal programmet kunne opfylde følgende specifikationer:

- Uafhængighed: Programmet skal kunne virke uden brug af andre shells f.eks. ved at benytte et systemkald system() til at starte en bash.
- Når programmet kører skal det vise navnet på den host der er logget ind.
- Udskrive en "Command not found" meddelelse når man skriver en kommando der ikke findes i systemet.
- Kommandoer skal kunne køres i baggrunden, ved brug af '&'. Dvs. at man skal kunne fortsætte med at benytte bosh.
- Man skal kunne lave redirection af stdin og stdout ved at benytte, '<' og '>'.
- Det skal være muligt at anvende pipes. Dvs. at det skal være muligt at tage flere kommandoer med '|' som separator. Den venstre specificeret kommando skal benyttes som input til den højre specificeret kommando.
- Man skal kunne lukke shellen ved at bruge kommandoen 'exit'.
- Ctrl+C skal ikke lukke shellen, men afslutte det kørende program i bosh.

De udleverede filer indeholder tre C filer:

- bosh.c
- parser.c
- print.c

For at løse opgaverne specificeret er der ikke behov for at ændre i parser.c og print.c. I filen parser.h er der specificeret de to struct som bliver benyttet i programmet, Cmd og Shellcmd.

Cmd har en hægtet liste datastruktur, der består af en kommando streng og peger på den næste kommando i rækken. Når der er flere kommandoer skal der anvendes pipe.

Shellcmd indeholder felter relateret til de andre specificeret opgaver, som hvis brugeren har specificeret at programmet skal køres i baggrunden, eller specificeret en form af redirect.

3 Implementation

3.1 Hostname

I bosh.c er der en metode signatur, `gethostname()`, til at hente brugernavnet. For at finde lokationen af hostname, benyttes det virtuelle filsystem `/proc`. Den fulde sti til hostname filen er `proc/sys/kernel/hostname`. Vi har lavet den antagelse, at vi kun skal hente den første linje i filen for at få hostname. Dette har vi gjort ved at benytte en `FILE` og metoden `fopen()`.

```
1  FILE *fp;
2      char *line = NULL;
3      size_t len = 0;
4      ssize_t read;
5
6      fp = fopen(hostfile, "r");
7      if((read = getline(&line, &len, fp)) != -1)) /* get the hostname
           from line 1 */
8      {
9          strtok(line, "\n"); /* remove newline token */
10         *hostname = line;
11     }
```

3.2 Baggrundsprocesser & Redirection

Implementering af baggrundsprocesser og redirection er gjort i metoden `executecmds()` i filen execmds.c. Metoden tager imod en kommando, `Cmd`, et filnavn til `stdin`, et filnavn til `stdout`, og en binær boolean for at vide om programmet skal køre i baggrunden. Den observante læser har bemærket, at metodens parametre er svarende til `Shellcmd`'s felter.

Metoden bliver kaldt fra bosh.c's `executeshellcmd()`, ~~e executecmds(cmds, shellcmd->rd_stdin, shellcmd->rd_stdout~~

Metoden skaber en ny process hvori den tjekker om parametrene er instantieret.

```
1  ...
2      if(pid == 0)
3      {
4          if(infilename != NULL) { ... }
5
6          if(outfilename != NULL) { ... }
7          ...
8      }
9      else if(bg != 1)
10     {
11         waitpid(pid, &status, 0);
```

```
12      }
13  ...
```

Til redirection er der gjort brug af metoderne `open()`, `close()` og `dup()`. Hvis programmet skal køres i baggrunden ventes der ikke på processen.

3.3 Pipe

I metoden `executecmds()` tjekkes `Cmd` struct'en om der er flere kommandoer og hvis dette er tilfældet kaldes metoden `pipecmd()` i filen pipe.c. Metoden tager en kommando som argument. Metoden er rekursiv og derfor startes der med at tjekke om kommandoen er `NULL`. Hvis det ikke er tilfældet oprettes en `pipe()` og der startes en ny proces.

```
1  ...
2  if(cmds != NULL)
3  {
4      pipe(pfd); /* Create the pipe */
5
6     if((pid = fork()) == 0) /* Child */
7      {
8          ...
9          execvp(*cmd, cmd);
10     }
11     else  /* Parent */
12     {
13         ...
14         pipecmd(nextcmds);
15     }
16 }
17 ...
```

I den nye proces bliver pipe's `stdin` udskiftet med processens `stdin` ved brug af `dup2()`. Tilbage i den gamle proces udskiftes `stdout` på samme vis.

3.4 CTRL+C

Når en bruger trykker Ctrl+C sendes der et signal som kan fanges af det program der bliver kørt. Helt specifikt er signalet for Ctrl+C en `SIGINT`. Når et signal fanges med metoden `signal()`, skal man i dens andet argument angive en metode der skal kaldes og i dette tilfælde er det sat til en tom metode, `sig_handler()`. Dette giver den ønskede funktionalitet.

```
1      #include
2      <signal.h>
```

```
3       ...
4       void  sig_handler(int  signo) { }
5       int  main(int  argc, char  *argv[])
6       {
7           ...
8           signal(SIGINT, sig_handler);
9           ...
```

Funktioner som "exit" kommandoen og "Command not found" beskeden er også implementeret i bosh.c.

4 Testing

Vi har manuelt testet programmet, hvilket betyder, at der er stor sandsynlighed for at alle test cases ikke er blevet dækket.

Eksemplerne fra opgavebeskrivelsen er blevet afprøvet og fungerer efter hesigten. Under afprøvelser af diverse programmer fandt vi problemer med hensyn til indtastning af en forkert kommando. Programmet printer efter intentionen "Command not found", men af en årsag vi ikke er kommet frem til kan man ikke bare lukke bosh ved at bruge "exit". Man skal nemlig udføre "exit" for antallet af gange man har skrevet en forkert kommando. Antagelsen er, at der startes en process der ikke bliver stoppet korrekt.

5 Konklusion

Ud fra specifikationerne til opgaven kan der konkluderes, at bosh kan køre programmer i baggrunden, redirect til filer, kører flere programmer ved brug af pipe og er uafhængig.

6 Appendix A - Sourcecode

bosh.c

```
1  /*
2
3     bosh.c : BOSC shell
4
5   */
6
7  #include <stdio.h>
8  #include <string.h>
9  #include <stdlib.h>
10 #include <ctype.h>
11 #include <string.h>
12 #include <readline/readline.h>
13 #include <readline/history.h>
14
15 #include <signal.h>
16
17 #include "parser.h"
18 #include "print.h"
19
20
21 /* --- symbolic constants --- */
22 #define HOSTNAMEMAX 100
23
24 /* --- use the /proc filesystem to obtain the hostname --- */
25 char *gethostname(char **hostname)
26 {
27   char hostfile[] = "/proc/sys/kernel/hostname"; /* hostname file */
28
29   FILE *fp;
30   char *line = NULL;
31   size_t len = 0;
32   ssize_t read;
33
34   fp = fopen(hostfile,"r");
35   if((read = getline(&line, &len, fp)) != -1) /* get the hostname from
         line 1 */
36   {
37     strtok(line,"\n"); /* remove newline token */
38     *hostname = line;
39   }
40   return *hostname;
41 }
```

```
42
43 /* --- execute a shell command --- */
44 int executeshellcmd(Shellcmd *shellcmd)
45 {
46   printshellcmd(shellcmd);
47
48   Cmd *cmds;
49   char **cmd0;
50
51   cmds = shellcmd->the_cmds;
52   cmd0 = cmds->cmd;
53
54   if(strcmp(*cmd0,"exit") == 0)
55   {
56     return 1;
57   }
58
59   int i = executecmds(cmds, shellcmd->rd_stdin, shellcmd->rd_stdout,
       shellcmd->background);
60
61   if(i == -1)
62   {
63   printf("Command not found\n");
64   }
65
66   return 0;
67 }
68
69 void sig_handler(int signo) { }
70
71 /* --- main loop of the simple shell --- */
72 int main(int argc, char* argv[]) {
73
74   signal(SIGINT, sig_handler);
75
76   /* initialize the shell */
77   char *cmdline;
78   char *hostname[HOSTNAMEMAX]; /* changed to a pointer */
79   int terminate = 0;
80   Shellcmd shellcmd;
81
82   if (gethostname(hostname)) {
83     /* parse commands until exit or ctrl-c */
84     while (!terminate) {
85       printf("%s:$", *hostname);
86       if (cmdline = readline(" ")) {
```

```
87   if(*cmdline) {
88     add_history(cmdline);
89     if (parsecommand(cmdline, &shellcmd)) {
90       terminate = executeshellcmd(&shellcmd);
91     }
92   }
93   free(cmdline);
94       } else terminate = 1;
95     }
96     free(*hostname);
97     printf("Exiting bosh.\n");
98   }
99
100  return EXIT_SUCCESS;
101 }
```

execmds.h

```
1 int executecmds(Cmd *, char *, char *, int *);
```

execmds.c

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5  #include <stdlib.h>
6
7  #include <sys/types.h>
8  #include <sys/stat.h>
9  #include <fcntl.h>
10
11 #include "parser.h"
12
13 /* executes commands */
14 int executecmds(Cmd *cmds, char *infilename, char *outfilename, int bg)
15 {
16   char **cmd0 = cmds->cmd;
17   Cmd *nextcmds = cmds->next;
18
19   pid_t pid = fork();
20   int status;
21
22   int exe;
23
24   if(pid == 0) /* Child */
25   {
26     if(infilename != NULL)
27     {
28       int fid = open(infilename, O_RDONLY,0);
29
30       close(0);
31
32       dup(fid);
33
34       close(fid);
35     }
36
37     if(outfilename != NULL) /* Check for  */
38     {
39       int fid = creat(outfilename,S_IRWXU|S_IRWXG|S_IRWXO);
40
41       close(1);
```

```
42
43      dup(fid);
44
45      close(fid);
46    }
47
48    if(nextcmds != NULL) /* Check if there are more commands */
49    {
50      exe = pipecmd(cmds);
51    }
52    else
53    {
54      exe = execvp(*cmd0, cmd0);
55    }
56  }
57  else if(bg != 1) /* Parent */
58  {
59    waitpid(pid, &status, 0);
60  }
61
62  if(exe) /* Command not found */
63  {
64  return exe;
65  }
66
67  return 0;
68 }
```

pipe.h

```
/*
   Opgave 3

   pipe.h

 */

#ifndef _PIPE_H
#define _PIPE_H

int pipecmd(char *filename, char *argv[], char *filename2, char
    *argv2[]);

#endif
```

pipe.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>

#include "parser.h"

/* execute commands with pipe */
int pipecmd(Cmd *cmds)
{
  int pfd[2];
  pid_t pid;
  int status;

  char **cmd = cmds->cmd;
  Cmd *nextcmds = cmds->next; /* Set next */

  int exe;

  if(cmds != NULL)
  {
    pipe(pfd); /* Create the pipe */

    if((pid = fork()) == 0) /* Child */
```

```
29      {
30        dup2(pfd[0],0); /* Replace stdin */
31
32        close(pfd[1]);
33
34        exe = execvp(*cmd, cmd);
35      }
36      else /* Parent */
37      {
38        dup2(pfd[1],1); /* Replace stdout */
39
40        close(pfd[0]);
41
42        pipecmd(nextcmds);
43      }
44    }
45
46    wait(&status);
47
48    close(pfd[0]);
49    close(pfd[1]);
50
51    if(exe) /* Command not found */
52    {
53    return exe;
54    }
55
56    return 0;
57  }
```

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 * );
```

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;
```

```
25 static char cbuf[BUFFERMAX], *cp;
26 static char *pbuf[PBUFFERMAX], **pp;
27
28 /*
29  * parse : A simple commandline parser.
30  */
31
32 /* --- parse the commandline and build shell commmand structure --- */
33 int parsecommand(char *cmdline, Shellcmd *shellcmd)
34 {
35   int i, n;
36   Cmd *cmd0;
37
38   char *t = cmdline;
39   char *tok;
40
41   // Initialize list
42   for (i = 0; i < COMMANDMAX-1; i++) cmdbuf[i].next = &cmdbuf[i+1];
43
44   cmdbuf[COMMANDMAX-1].next = NULL;
45   cmds = cmdbuf;
46   cp = cbuf;
47   pp = pbuf;
48
49   shellcmd->rd_stdin    = NULL;
50   shellcmd->rd_stdout   = NULL;
51   shellcmd->rd_stderr   = NULL;
52   shellcmd->background = 0; // false
53   shellcmd->the_cmds       = NULL;
54
55   do {
56     if ((n = acmd(t, &cmd0)) <= 0)
57       return -1;
58     t += n;
59
60     cmd0->next = shellcmd->the_cmds;
61     shellcmd->the_cmds = cmd0;
62
63     int newtoken = 1;
64     while (newtoken) {
65       n = nexttoken(t, &tok);
66       if (n == 0)
67   {
68     return 1;
69   }
70       t += n;
```

```
71
72      switch(*tok) {
73      case PIPE:
74  newtoken = 0;
75  break;
76      case BG:
77  n = nexttoken(t, &tok);
78  if (n == 0)
79    {
80      shellcmd->background = 1;
81      return 1;
82    }
83  else
84    {
85      fprintf(stderr, "illegal bakgrounding\n");
86      return -1;
87    }
88  newtoken = 0;
89  break;
90      case RIN:
91  if (shellcmd->rd_stdin != NULL)
92    {
93      fprintf(stderr, "duplicate redirection of stdin\n");
94      return -1;
95    }
96  if ((n = nexttoken(t, &(shellcmd->rd_stdin))) < 0)
97    return -1;
98  if (!isidentifier(shellcmd->rd_stdin))
99    {
100     fprintf(stderr, "Illegal filename: \"%s\"\n",
           shellcmd->rd_stdin);
101     return -1;
102   }
103 t += n;
104 break;
105     case RUT:
106 if (shellcmd->rd_stdout != NULL)
107   {
108     fprintf(stderr, "duplicate redirection of stdout\n");
109     return -1;
110   }
111 if ((n = nexttoken(t, &(shellcmd->rd_stdout))) < 0)
112   return -1;
113 if (!isidentifier(shellcmd->rd_stdout))
114   {
```

```
115       fprintf(stderr, "Illegal filename: \"%s\"\n",
             shellcmd->rd_stdout);
116       return -1;
117     }
118   t += n;
119   break;
120       default:
121   return -1;
122       }
123     }
124   } while (1);
125   return 0;
126 }
127
128 int nexttoken( char *s, char **tok)
129 {
130   char *s0 = s;
131   char c;
132
133   *tok = cp;
134   while (isspace(c = *s++) && c);
135   if (c == '\0')
136     {
137       *cp++ = '\0';
138       return 0;
139     }
140   if (isspec(c))
141     {
142       *cp++ = c;
143       *cp++ = '\0';
144     }
145   else
146     {
147       *cp++ = c;
148       do
149   {
150     c = *cp++ = *s++;
151   } while (!isspace(c) && !isspec(c) && (c != '\0'));
152       --s;
153       --cp;
154       *cp++ = '\0';
155     }
156   return s - s0;
157 }
158
159 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;
}
```

print.h

```
void printshellcmd(Shellcmd *shellcmd);
void printcmdlist(Cmd *p);
```

print.c

```
#include <stdio.h>
#include "parser.h"
#include "print.h"

/* --- print a shell command --- */
void printshellcmd(Shellcmd *shellcmd)
{
  printf("Shellcommand:\n");
  printf("   stdin : %s\n", shellcmd->rd_stdin  ? shellcmd->rd_stdin
      : "<none>" );
  printf("   stdout: %s\n", shellcmd->rd_stdout ? shellcmd->rd_stdout
      : "<none>" );
  printf("   bg    : %s\n", shellcmd->background ? "yes" : "no");
  Cmd *cmdlist = shellcmd->the_cmds;
  while (cmdlist != NULL) {
    char **cmd = cmdlist->cmd;
    cmdlist = cmdlist->next;

    char **printcmd = cmd;
    printf("    [");
    while (*printcmd != NULL) {
      printf("%s ", *printcmd++); // print the cmd and arguments
    }
    printf("]\n");

  }
}
```

Makefile

```
all: bosh

OBJS = parser.o print.o execmds.o bosh.o pipe.o
LIBS= -lreadline -ltermcap
CC = gcc

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

clean:
  rm -rf *o bosh
```