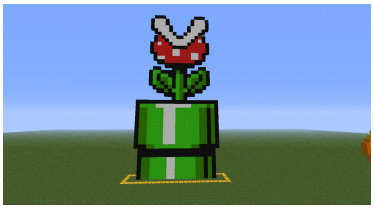
IT University of Copenhagen

IT-Universitet i København

Operativsystemer og C

Obligatorisk aflevering 1

Tróndur Høgnason (thgn@itu.dk)
Frederik Madsen (mfrm@itu.dk)
Holger Borum (hstb@itu.dk)



Cece n'est pas une pipe.

October 2, 2015

Contents

1 Indledning

Denne rapport er skrevet som en del af afleveringen "Obligatorsk Opgave 1" i faget "Operativsystemer og C" på ITU. I afleveringen skulle der implementeres et simpelt shellprogram, som skulle lade brugerne skrive kommandoer, som eksekverer programmer. Programmer skulle kunne eksekveres som enten baggrunds- eller forgrundsprocesser. Et program skulle kunne tage input fra, og give output til, specificerede filer. Derudover skulle programmet kunne tage flere kommandoer, hvor kommando n sendte sit output til kommando n+1, også kendt som piping. Vi fik udleveret kode som kunne stå for at parse strenge fra brugeren. Udfordringen har således hovedsageligt været at implementere piping og omdirigering af programmets in- og output. Det er implementeringen af disse dele, samt nogle mindre funktioner, som vi vil beskrive i denne rapport.

2 Implementering

2.1 Hostname

For at udskrive hostnavnet på maskinen i shell'en, har vi åbnet filen "/proc/sys/kernel/Hostname" med fopen(). Derefter læser vi første linje i filen ind i en buffer med fgets(), og scanner linjen ind i vores hostname variabel med sscan(). Til sidst lukker vi filen med fclose(). Der er ikke nogen fejlhåndtering, hvis filen ikke indeholder noget, eller hvis den ikke eksisterer. Begrundelsen for, at vi ikke fejlhåndterer her, er, at vi initialiserer vores hostname variabel med strengen "DEFAULT" og derefter overskriver denne variabel. Hvis filen ikke eksisterer, bliver hostname ikke overskrevet, og shell'en vil altså udskrive "DEFAULT".

2.1.1 Valg af exec funktion

Vi valgte først at afgrænse de mulige eksekveringsfunktioner til de tre, der understøttede automatisk opslag efter eksekverbare filer i de filkataloger, der er specifiseret i PATH miljøvariabelen. Disse er execlp(), execvp() og execvpe(). Dette gør det let at køre kommandoer som ls og wc.

Funktionen execvpe() blev udelukket, fordi den understøtter ekstra funktionalitet, som vi ikke har brug for i form af muligheden for at specifisere miljøet for kommandoen, der eksekveres.

Forskellen på execlp() og execvp() er kun formateringen af argumenterne, og her foretrak vi execvp(), fordi vi har brugt den før. Alle funktionerne benytter sig alligevel i sidste ende af execve().

2.2 Baggrundskørsel af kommandoer

Vi har implementeret to forskellige måder at køre kommandoer på. Det er implementeret i de to metoder foregroundcmd() og backgroundcmd(), der begge er at finde i forback.c. Foregroundcmd forker hovedprocessen, og får forælderprocessen til at vente på at barneprocessen terminerer, dette er hvad vi kalder en forgrundsproces. Backgroundcmd forker hovedprocessen. Hovedprocessen venter på barneprocessen, mens barneprocessen igen forker sig selv og ekskvere kommandoen, uden at vente på den. Der forkes to gange for at undgå, at baggrundsprocessen ender som en såkaldt zombieproces, efter den terminerer.

2.3 Exit kommando

Afslutning af shell'en sker, når brugeren skriver "exit" i kommandolinjen. I filen bosh.c findes metoden executeshellcmd(), hvor der foregår et check af kommandoerne. Hvis den første kommando er "exit", så returner executeshellcmd() tallet 1 til den kaldende metode, main metoden. Her bliver terminate-variablen sat til 1 og while loopet, der holder shell'en kørende, ved at chekke om terminate IKKE er 1, vil slutte.

2.4 ctrl+c

Ctrl+c sender et SIGINT-signal til shell'en. Metoden som håndterer dette signal, vælges med ved at kalde signal(). InterruptRun() håndterer således alle SIGINT-signaler i stedet for default håndteringen. InterruptRun() udskriver blot "caught ctrl+c". Grunden, til at vi ikke sender signalet videre til eventuelle børneprocesser, er, at disse processer ligeledes vil modtage et SIGINT-signal sendt fra bash-terminalen. Altså er der ikke behov at vi aktivt terminerer dem. Det skal dog også nævnes, at dette vil terminere eventuelle baggrundsprocesser. Det har ikke været muligt for os, at undgå at baggrundsprocesser terminerer ved ctrl+c, på trods af at dette er opførslen i bash-terminalen.

2.5 Piping og redirection

Redirection og piping er blevet implementeret i samme kode, da det stort set er samme funktionalitet. Der oprettes en pipe, hver gang en kommando (k1) skal sende sit output til en anden kommando (k2). Pipens skrive-ende gives som k1's input fil og læse-enden gives til k2's læse-ende. Dette kan ses på figur 1. Input- og output-redirection kan således ses som en pipe-ende, som gives til henholdvis den første eller sidste kommando. Det sker ved at filerne åbnes i hovedprocessen, og deres file-descriptors kan herefter bruges på samme måde som pipe-ender. Løbende lukker hovedprocessen for filer og pipe-ender, som fremover ikke skal bruges af kommandoer.

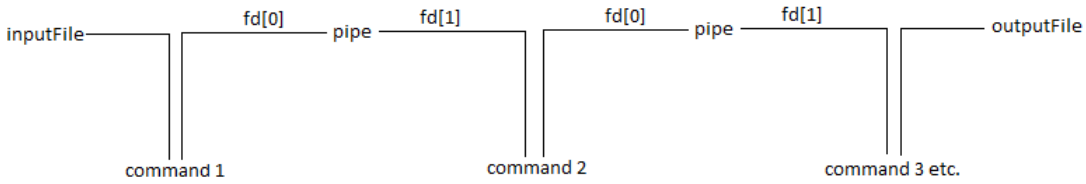


Figure 1: En simpel figur over hvordan der pipes mellem de forskellige kommandoer og in- og output filerne.

Vi valgte at udføre kommandoerne i den rækkefølge som resultaterne skulle bruges. Med andre ord, hvis kommando k1's output skal pipes til kommando k2's input, så skal kommando k1 startes før kommando k2. Hovedårsagen til dette valg var, at det gjorde det nemt at køre en kommando-række enten som forgrunds- eller baggrundsprocesser, da kommandoerne bare kan udføres, uden at hovedprocessen venter på en forgrundsproces, der venter på input. For at kunne have samtidig kørsel af kommandoer, der undervejs generere output, der

pipes, så kan kun den sidste kommando køres som forgrundsproces. Det har medført, at hvis den sidste kommando terminerer før de forudgående baggrundsprocesser, kan det ske, at baggrundsprocesser efterlades kørende. Vi er nået frem til, at det er årsagen til, at kommando-listen oprindeligt vendte som den gjorde - med den sidste kommando først. Disse baggrundsprocesser er dog ikke synlige for brugeren, men det er muligivis stadigvæk problematisk, hvis en af de efterladte baggrundsprocessor ender i et deadlock, da det er usynligt for brugeren, hvilket ikke ville forventes med forgrundsprocesser.

3 Test

Vi har primært testet vores shell ved manuelle kørsler af kommandoer. Vi har brugt programmet htop[1], til at tjekke at børneprocesser lukker ned som forventet.

For at overbevise os selv at filer ikke blev efterladt åbne ved piping, har vi håndkørt lukningen af dem. Her er en tabel, der viser, hvilke processer der har hvilke filer åbne.

File descriptors	main	ls	wc	wc
in	Closed	Open	-	-
1	Closed	Closed	Open	-
2	Closed	Open	-	-
3	Closed	-	Closed	Open
4	Closed	-	Open	-
out	Closed	-	-	Open

Table 1: Den resulterende tabel over hvilke processer der har hvilke pipes efter at have håndkørt vores shell med "ls |wc |wc" .

Tegnforklaring:
Open: Processen har en åben file descriptor med dette ID.
Closed: Processen har fået en åben file descriptor med dette ID, men lukket det.
-: Processen har aldrig haft en file descriptor med dette ID.

[1] http://hisham.hm/htop/

4 Konklusion

Vi har implementeret og løst den stillede opgave og manuelt testet, at programmet virker. Programmet kan altså ekskvere kommandorer som baggrunds- eller forgrundsprocessor, omdirigere in- og output til kommandoer samt eksekvere flere kommandoer, hvor én kommandos output pipes til den næstes input. Det mest ubesvarede spørgsmål for os er, hvorfor at der i opgavebeskrivelsen står, at der er en god grund til, at kommandoer parses i omvendt rækkefølge. Vi har forsøgt at give et bud på hvorfor, men da vi ikke har kunne finde nogen klar grund, valgte vi at vende listen om, da det virkede meget nemmere for os.

Appendix

bosh

```c
/*
  bosh.c : BOSC shell
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <readline/readline.h>
#include <readline/history.h>
#include <sys/types.h>

#include <sys/stat.h>
#include <fcntl.h>

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

#include "parser.h"
#include "print.h"
#include "forback.h"

/* --- symbolic constants --- */
#define HOSTNAMEMAX 100

/* --- use the /proc filesystem to obtain the hostname --- */
void gethostname(char* hostname)
{
  char* filename = "/proc/sys/kernel/hostname";
  if( access( filename, F_OK ) != -1 ) {
    FILE *file;

    char line[HOSTNAMEMAX];
    file = fopen(filename,"r");

    fgets(line,HOSTNAMEMAX,file);
    //if unable to scan, then hostname is already set, so no if(scan(..)) necessary
    sscanf(line,"%s",hostname);

    fclose(file);
  }
}

/* --- execute a shell command --- */
int executeshellcmd (Shellcmd *shellcmd)
{
  struct _cmd *the_cmds = shellcmd->the_cmds;
  char* in      = shellcmd->rd_stdin;
  char* out     = shellcmd->rd_stdout;

  //Reversing the list for easier execution
```

```
54    struct _cmd *next = NULL;
55    struct _cmd *temp;
56    int run = 1;
57
58    if(strcmp("exit", (const char*) *shellcmd->the_cmds->cmd) == 0)
59    {
60      return 1;
61    }
62
63    while(run){
64      if(the_cmds->next == NULL){
65        run = 0;
66      }
67      else{
68        temp = the_cmds->next;
69      }
70
71      the_cmds->next = next;
72
73      next = the_cmds;
74      if(run){
75        the_cmds = temp;
76      }
77    }
78
79    // End of reversing list
80
81    int inId, outId, closeId;
82    outId = -1;
83    inId  = -1;
84
85    if(in){
86      inId = open(in, O_RDONLY);
87    }
88
89    int fid[2] = {inId, outId};
90    while(the_cmds != NULL){
91
92      char** cmd = the_cmds->cmd;
93      if(the_cmds->next != NULL){
94        if(pipe(fid) < 0){
95          exit(1); //Not able to create pipe
96        }
97        outId = fid[1];
98      }
99      else{
100       if(out){
101         outId = open(out, O_WRONLY | O_CREAT, 0666);
102       }else{
103         outId = -1;
104       }
105     }
106
107     closeId = fid[0];
108
109     if(shellcmd->background || the_cmds->next != NULL){
110       backgroundcmd(*cmd, cmd, inId, outId, closeId);
```

```
111     }
112     else{
113       foregroundcmd(*cmd, cmd, inId, outId, closeId);
114     }
115     if(fid[1] != -1){
116       close(fid[1]);
117     }
118     if(inId != -1){
119       close(inId);
120     }
121
122     the_cmds = the_cmds->next;
123     inId = fid[0];
124   }
125
126   if(outId == -1){
127     close(outId);
128   }
129
130   return 0;
131 }
132
133 void interruptRun(int dummy){
134   printf("%s\n", "caught ctrl-c - use exit command or crtl+d to exit");
135 }
136
137
138 /* --- main loop of the simple shell --- */
139 int main(int argc, char* argv[]) {
140
141   /* initialize the shell */
142   char *cmdline;
143   char hostname[HOSTNAMEMAX] = "Default";
144   int terminate = 0;
145   Shellcmd shellcmd;
146   signal(SIGINT, interruptRun);
147
148
149   gethostname(hostname);
150   /* parse commands until exit or ctrl-d */
151   while (!terminate) {
152
153     printf("%s", hostname);
154     if (cmdline = readline(":# ")) {
155       if(*cmdline) {
156         add_history(cmdline);
157
158         if (parsecommand(cmdline, &shellcmd)) {
159           terminate = executeshellcmd(&shellcmd);
160         }
161       }
162
163       free(cmdline);
164     }
165     else{
166       terminate = 1;
167     }
```

```
  }
  printf("Exiting bosh.\n");

  return EXIT_SUCCESS;
}
```

forback

```
/*

   Opgave 1

   forback.h

 */

#ifndef _FORBACK_H
#define _FORBACK_H
int foregroundcmd(char*, char**, int, int, int);
int backgroundcmd(char*, char**, int, int, int);

#endif
```

```
/*

   Opgave 1

   forback.c

 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <string.h>

#include "redirect.h"

/*Helper to avoid dublication*/
int redirectAndExec(char *filename, char *argv[], int in, int out, int closeId){
  if(in != -1){
    redirect_stdincmd(in);
  }
  if(out != -1){
    redirect_stdoutcmd(out);
  }
  if(closeId != -1){
    close(closeId);
  }

  if(execvp(filename,argv) == -1){
    printf("Command not found\n");
    exit(1);
  }
  exit(1);
```

```
}


/* start the program specified by filename with the arguments in argv
   in a new process and wait for termination */
int foregroundcmd(char *filename, char *argv[], int in, int out, int closeId)
{
  if(strcmp("exit", filename) == 0)
  {
    return -1;
  }
  pid_t pid = fork();

  if(pid == 0){
    redirectAndExec(filename, argv, in, out, closeId);
  }else{
    int returnStatus;
      waitpid(pid, &returnStatus, 0);
  }

  return 0;
}

/* start the program specified by filename with the arguments in argv
   in a new process */
int backgroundcmd(char *filename, char *argv[], int in, int out, int closeId)
{
  pid_t pid = fork();
  if(pid == 0){ //Avoiding zombie processes
    pid_t pid1 = fork();
    if(pid1 == 0){
      redirectAndExec(filename, argv, in, out, closeId);
    }
    exit(0);
  }else{
    int returnStatus;
      waitpid(pid, &returnStatus, 0);
  }

  return 0;
}
```

parser

```
/*
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;
```

```
14      int  background;
15  } Shellcmd;
16
17  extern void init( void );
18  extern int parse ( char *, Shellcmd *);
19  extern int nexttoken( char *, char **);
20  extern int acmd( char *, Cmd **);
21  extern int isidentifier( char * );
```

```
1   /*
2   parser.c
3   */
4   #include <stdio.h>
5   #include <string.h>
6   #include <ctype.h>
7   #include "parser.h"
8
9   /* --- symbolic constants --- */
10  #define COMMANDMAX 20
11  #define BUFFERMAX 256
12  #define PBUFFERMAX 50
13  #define PIPE   ('|')
14  #define BG     ('&')
15  #define RIN    ('<')
16  #define RUT    ('>')
17  #define IDCHARS "_-.,/~+"
18
19  /* --- symbolic macros --- */
20  #define ispipe(c) ((c) == PIPE)
21  #define isbg(c)   ((c) == BG)
22  #define isrin(c)  ((c) == RIN)
23  #define isrut(c)  ((c) == RUT)
24  #define isspec(c) (ispipe(c) || isbg(c) || isrin(c) || isrut(c))
25
26  /* --- static memory allocation --- */
27  static Cmd  cmdbuf[COMMANDMAX], *cmds;
28  static char cbuf[BUFFERMAX], *cp;
29  static char *pbuf[PBUFFERMAX], **pp;
30
31  /*
32   * parse : A simple commandline parser.
33   */
34
35  /* --- parse the commandline and build shell commmand structure --- */
36  int parsecommand(char *cmdline, Shellcmd *shellcmd)
37  {
38    int i, n;
39    Cmd *cmd0;
40
41    char *t = cmdline;
42    char *tok;
43
44    // Initialize list
45    for (i = 0; i < COMMANDMAX-1; i++) cmdbuf[i].next = &cmdbuf[i+1];
46
47    cmdbuf[COMMANDMAX-1].next = NULL;
48    cmds = cmdbuf;
49    cp = cbuf;
```

```
50  pp = pbuf;
51
52  shellcmd->rd_stdin    = NULL;
53  shellcmd->rd_stdout   = NULL;
54  shellcmd->rd_stderr   = NULL;
55  shellcmd->background = 0; // false
56  shellcmd->the_cmds       = NULL;
57
58  do {
59    if ((n = acmd(t, &cmd0)) <= 0)
60      return -1;
61    t += n;
62
63    cmd0->next = shellcmd->the_cmds;
64    shellcmd->the_cmds = cmd0;
65
66    int newtoken = 1;
67    while (newtoken) {
68      n = nexttoken(t, &tok);
69      if (n == 0)
70  {
71    return 1;
72  }
73      t += n;
74
75      switch(*tok) {
76      case PIPE:
77  newtoken = 0;
78  break;
79      case BG:
80  n = nexttoken(t, &tok);
81  if (n == 0)
82    {
83      shellcmd->background = 1;
84      return 1;
85    }
86  else
87    {
88      fprintf(stderr, "illegal bakgrounding\n");
89      return -1;
90    }
91  newtoken = 0;
92  break;
93      case RIN:
94  if (shellcmd->rd_stdin != NULL)
95    {
96      fprintf(stderr, "duplicate redirection of stdin\n");
97      return -1;
98    }
99  if ((n = nexttoken(t, &(shellcmd->rd_stdin))) < 0)
100   return -1;
101 if (!isidentifier(shellcmd->rd_stdin))
102   {
103     fprintf(stderr, "Illegal filename: \"%s\"\n", shellcmd->rd_stdin);
104     return -1;
105   }
106 t += n;
```

```
107   break;
108       case RUT:
109   if (shellcmd->rd_stdout != NULL)
110     {
111       fprintf(stderr, "duplicate redirection of stdout\n");
112       return -1;
113     }
114   if ((n = nexttoken(t, &(shellcmd->rd_stdout))) < 0)
115     return -1;
116   if (!isidentifier(shellcmd->rd_stdout))
117     {
118       fprintf(stderr, "Illegal filename: \"%s\"\n", shellcmd->rd_stdout);
119       return -1;
120     }
121   t += n;
122   break;
123       default:
124   return -1;
125       }
126     }
127   } while (1);
128   return 0;
129 }
130
131 int nexttoken( char *s, char **tok)
132 {
133   char *s0 = s;
134   char c;
135
136   *tok = cp;
137   while (isspace(c = *s++) && c);
138   if (c == '\0')
139     {
140       *cp++ = '\0';
141       return 0;
142     }
143   if (isspec(c))
144     {
145       *cp++ = c;
146       *cp++ = '\0';
147     }
148   else
149     {
150       *cp++ = c;
151       do
152   {
153     c = *cp++ = *s++;
154   } while (!isspace(c) && !isspec(c) && (c != '\0'));
155       --s;
156       --cp;
157       *cp++ = '\0';
158     }
159   return s - s0;
160 }
161
162 int acmd (char *s, Cmd **cmd)
163 {
```

```
164   char *tok;
165   int n, cnt = 0;
166   Cmd *cmd0 = cmds;
167   cmds = cmds->next;
168   cmd0->next = NULL;
169   cmd0->cmd = pp;
170
171   while (1) {
172     n = nexttoken(s, &tok);
173
174     if (n == 0 || isspec(*tok))
175       {
176         *cmd = cmd0;
177         *pp++ = NULL;
178         return cnt;
179       }
180     else
181       {
182   *pp++ = tok;
183   cnt += n;
184   s += n;
185       }
186   }
187 }
188
189 int isidentifier (char *s)
190 {
191   while (*s)
192     {
193       char *p = strrchr (IDCHARS,  *s);
194       if (! isalnum(*s++) && (p == NULL))
195   return 0;
196     }
197   return 1;
198 }
```

redirect

```
1  /*
2     Opgave 2
3     redirect.c
4  */
5
6  #include <unistd.h> //for STDIN_FILENO
7  #include <stdio.h>
8  #include <sys/types.h>
9  #include <sys/stat.h>
10 #include <fcntl.h>
11
12 #include <unistd.h>
13 #include <sys/wait.h>
14 #include <stdlib.h>
15
16 /* start the program specified by filename with the arguments in argv
17    in a new process that has its stdin redirected to infilename and
18    wait for termination */
19 int redirect_stdincmd(int infileid)
```

```
{

  /* replace stdin of the child process with fid */
  close(STDIN_FILENO);

  //dup, duplicates to lowest id, which should be STDIN_FILENO. dup2 seems more secure.
  dup2(infileid, STDIN_FILENO);

  close(infileid);

  return 0;
}

/* start the program specified by filename with the arguments in argv
   in a new process that has its stdout redirected to outfilename and
   wait for termination */
int redirect_stdoutcmd(int outfileid)
{
  /* manipulate the file descriptor of the child process */
  //int fid = open(outfilename, O_WRONLY | O_CREAT, 0666);

  /* replace stdin of the child process with fid */
  close(STDOUT_FILENO);

  //dup, duplicates to lowest id, which should be STDIN_FILENO. dup2 seems more secure.
  dup2(outfileid, STDOUT_FILENO);

  close(outfileid);

  return 0;
}
```

Makefile

```
all: bosh

OBJS = parser.o print.o bosh.o forback.o redirect.o
LIBS= -lreadline -ltermcap
CC = gcc

bosh: ${OBJS}
  ${CC} -o bin/bosh ${OBJS} ${LIBS}

clean:
  rm -rf *.o bosh
```