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

I denne rapport dokumenterer vi vores valg i forhold til implementationen af BOSC shell (bosh).
Vores implementation kan findes i "Source Code/bosh.c". Vi har bygget videre på den implementation, som blev lagt op på bloggen[1]. Vi har dog erstattet "parser.c" med v2 af "parser.c"[2].

Kildekode og testdokumentation kan findes i appendix på side 5. Vores git repository kan findes på https://github.com/esfdk/BOSC . Når vi ser tilbage på processen, føler vi, at der er et par ting, vi kunne have gjort anderledes.

- Gjort det muligt at få en liste af baggrundsprocesser. (fx via en "jobs" kommando)
- Gjort det muligt at hente baggrundsprocesser frem til forgrund.
- Delt implementationen ud i flere dele (istedet for en lang bosh.c fil)

Vi har fået en del hjælp til pipes, redirect af stdin/out og "cd" kommandoen af Ulrik Damm (ulfd@itu.dk). Hans GitHub repository er https://github.com/ulrikdamm/progsh .

Der er en "bug" i parseren, som gør, at den ikke er i stand til at læse argumenter med mellemrum. Fx vil kommandoen 'mkdir "mappe med mellemrum" ' ikke lave en mappe ved navn "mappe med mellemrum", men derimod lave tre mapper med navnene '"mappe', 'med' 'mellemrum"'.

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] https://blog.itu.dk/BOSC-E2012/files/2012/09/oo1.zip
[2] https://blog.itu.dk/BOSC-E2012/files/2012/09/parserc.zip

1.3 Beskrivelse af implementation

1.3.1 Delopgave 1: Køre uafhængigt

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

Vi har ikke lavet nogen decideret test af dette.

1.3.2 Delopgave 2: Display hostname

Vi har valgt at vise både user og hostname i shell'en, da vi havde ønske om, at vores shell skulle ligne Bash shell så meget som muligt.
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 27 - 43 samt 232 i bosh.c.

Tests

Vi tjekker om vores shell viser det samme navn, som terminalen gør.
Forventet resultat: Se billede A.1 på side 6.
Faktisk resultat: Se billede A.1 på side 6.

1.3.3 Delopgave 3: Kør programmer

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 149 i bosh.c.

Tests

Test3.1 Vi kører kommandoen ls i den normale shell, og derefter i vores egen shell til sammmenligning.
Forventet resultat: Se billede A.2 på side 6.
Faktisk resultat: Se billede A.2 på side 6.

Test3.2 Vi kører kommandoen cat i den normale shell, og derefter i vores egen shell til sammenligning.
Forventet resultat: Se billede A.3 på side 6.
Faktisk resultat: Se billede A.3 på side 6.

Test3.3 Vi kører kommandoen wc i den normale shell, og derefter i vores egen shell til sammenligning.
Forventet resultat: Se billede A.4 på side 7.
Faktisk resultat: Se billede A.4 på side 7.

Test3.4 Vi kører kommandoen "42" i den normale shell, og derefter i vores egen shell til sammenligning.
Forventet resultat: Se billede A.5 på side 7.
Faktisk resultat: Se billede A.5 på side 7.

1.3.4 Delopgave 4: Baggrundsprocess

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 180-183 samt 202-214 i bosh.c.
Vi undgår zombieprocesser ved at kalde 'signal(SIGCHLD, SIG_IGN)' (linje 213).

Tests

Vi kører 'sleep 100 &' og derefter kommandoen date for at vise at sleep kører i baggrunden.
Forventet resultat: Forskellen mellem de to tidspunkter fra 'date" kommandoer er mindre end 100 sekunder.
Faktisk resultat: Se billede A.6 på side 7.

1.3.5 Delopgave 5: Redirect af stdin/out

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 123-127 i bosh.c.

Tests

Vi kører kommandoen 'wc -l </etc/passwd >antalkontoer' i den normale shell, og derefter i vores egen shell til sammenligning.
Forventet resultat: Se billede A.7 på side 8.
Faktisk resultat: Se billede A.7 på side 8.

1.3.6 Delopgave 6: Pipes

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. (Linje105 i bosh.c).
Vi kalder så close() på write delen af vores pipe (linje 113). Hvis der er flere commands i shellcmd, lukker vi stdin og dup() read delen af pipe. 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

Test: Vi starter med at køre kommandoen ls i shellen, og derefter prøver vi at køre ls — wc -w
Forventet resultat: Se billede A.8 på side 8.
Faktisk resultat: Se billede A.8 på side 8.

1.3.7 Delopgave 7: Exit

Første iteration af denne delopgave terminerede bare bosh (med exit(0)), men dette ændrede vi, da vi følte det gav bedre mening, hvis vi lod main-funktionen afslutte.
'executeshellcmd' checker om den sidste kommando er "exit". Hvis den er, returner vi 1 til main-funktionen, som så afslutter.
Se linje 58-60 i bosh.c.

Tests

Vi starter shellen og kører exit kommandoen.
Forventet resultat: Shellen afsluttes.
Faktisk resultat: Se billede A.9 på side 8.

1.3.8 Delopgave 8: Ctrl+C

Vores første implementation af Ctrl+C funktionaliteten stoppede alle processer (både forgrunds- og baggrundsprocesser). Dette lavede vi dog om, så det matchede måden, som Bash håndterer Cntrl+C signals. Vi har ikke implementeret en mulighed for at hente baggrundsprocesser frem i forgrunden, hvilket gør, at vi ikke kan afslutte baggrundsprocesser, medmindre de selv terminerer. Sådan funktionalitet bør være højt på dagsordenen, hvis man udvider funktionaliteten af BoSh.
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 220 samt 262-274 i bosh.c.

Tests

Vi kører kommandoen wc uden parameter og trykker ctrl-c for at se om processen afsluttes.
Forventet resultat: wc stoppes efter ctrl-c er blevet tastet.
Faktisk resultat: Se billede A.10 på side 9.

1.3.9 Ekstra funktionalitet 1: Display current working directory

Som nævnt i **1.3.2** har vi haft et ønske om at få vores shell til at ligne Bash, så derfor ville vi gøre det muligt at både se og skifte directory.
Vi bruger en funktion 'getcwd' til at finde current working directory. Dette har vi implementeret i 'getcurrentdir'.
Se linje (47-53) i bosh.c.

Tests

Vi starter shellen for at se om der vises det directory der i øjeblikket arbejdes i.
Forventet resultat: Se billede A.11 på side 9.
Faktisk resultat: Se billede A.11 på side 9.

1.3.10 Ekstra funktionalitet 2: Change directory

Som nævnt i **1.3.9**, så ville vi gøre det muligt at både se og skifte directory, hvilket var grundlaget for at implementere dette.
I vores 'executeshellcmd' tjekker vi om kommandoen er "cd" om den har mere end et argument. I så fald så kalder vi chdir på det første argument til "cd" kommandoen. Hvis det giver en error, printer vi det og returner 0.
Se linje 62-87 i bosh.c.

Tests

Vi går ind i vores shell og prøver derefter at skifte til andet directory end det der arbejdes i.
Forventet resultat: Se billede A.12 på side 9.
Faktisk resultat: Se billede A.12 på side 9.

A Test pictures

A.1 Delopgave 2

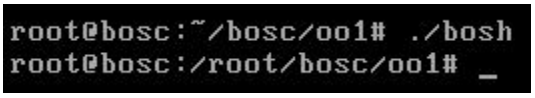


Figure A.1: Test 2

A.2 Delopgave 3

Figure A.2: Test 3.1

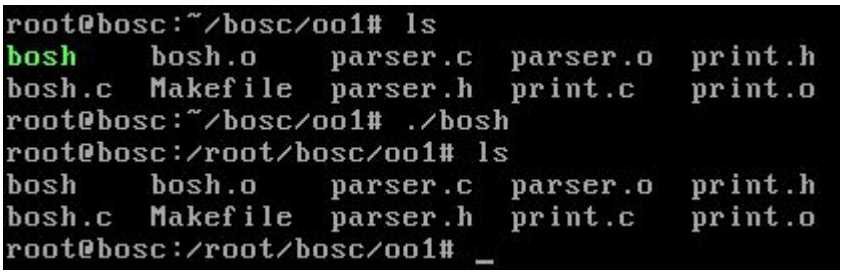


Figure A.3: Test 3.2

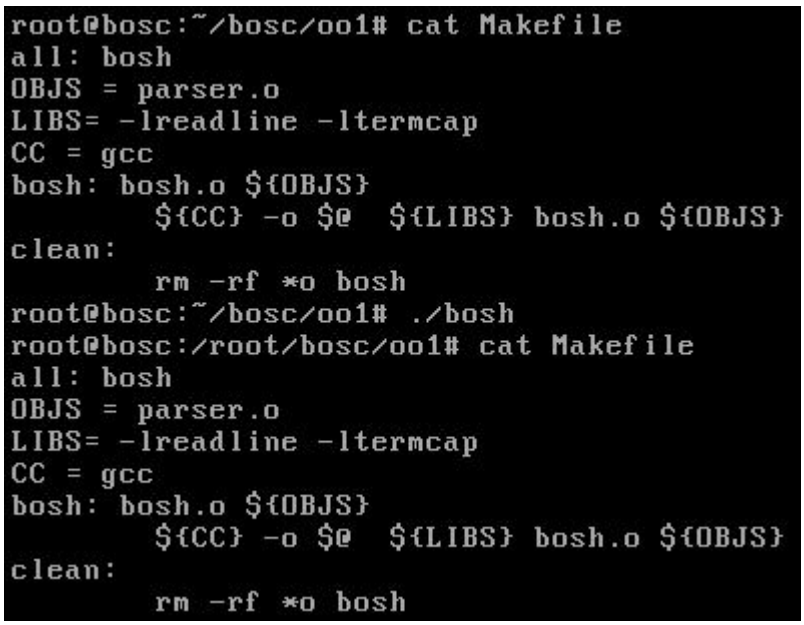


Figure A.4: Test 3.3

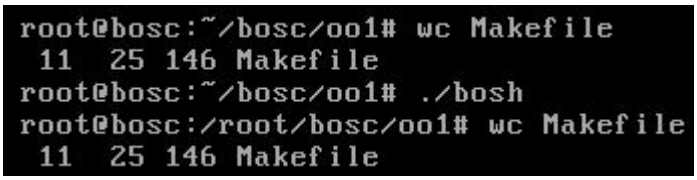
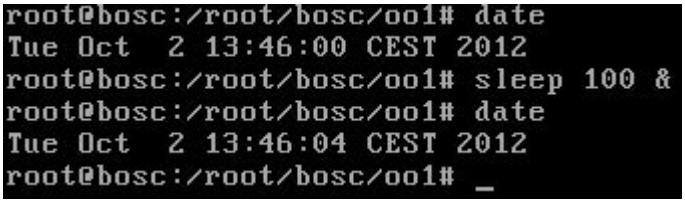


Figure A.5: Test 3.4

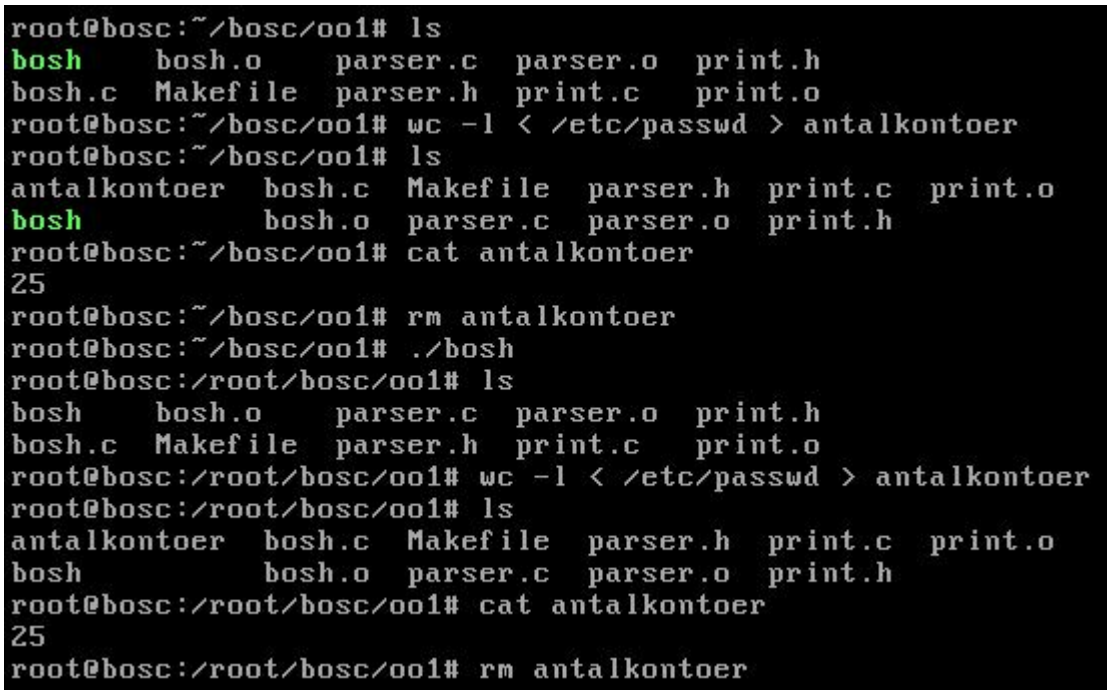
A.3 Delopgave 4

Figure A.6: Test 4



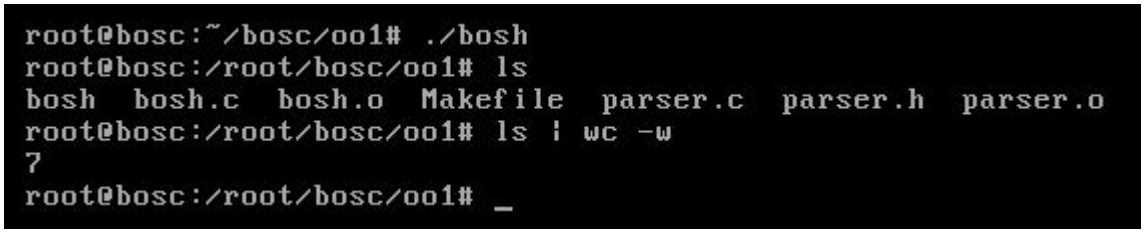
A.4 Delopgave 5

Figure A.7: Test 5



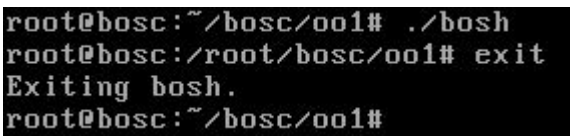
A.5 Delopgave 6

Figure A.8: Test 6



A.6 Delopgave 7

Figure A.9: Test 7



A.7 Delopgave 8

Figure A.10: Test 8

```
root@bosc:~/bosc/oo1# ./bosh
root@bosc:/root/bosc/oo1# wc -l
root@bosc:/root/bosc/oo1# ^C_
```

A.8 Ekstrafunktionalitet 1

Figure A.11: Test 9

```
root@bosc:~/bosc/oo1# ./bosh
root@bosc:/root/bosc/oo1# _
```

A.9 Ekstrafunktionalitet 2

Figure A.12: Test 10

```
root@bosc:~/bosc/oo1# ./bosh
root@bosc:/root/bosc/oo1# cd ..
root@bosc:/root/bosc# cd oo1/
root@bosc:/root/bosc/oo1# cd ..
root@bosc:/root/bosc# cd /root/bosc/oo1/
root@bosc:/root/bosc/oo1# ls
bosh  bosh.c  bosh.o  Makefile  parser.c  parser.h  parser.o
root@bosc:/root/bosc/oo1# _
```

B Makefile

```
1  all: bosh
2  OBJS = parser.o
3  LIBS= -lreadline -ltermcap
4  CC = gcc
5  bosh: bosh.o ${OBJS}
6          ${CC} -o $@  ${LIBS} bosh.o ${OBJS}
7  clean:
8          rm -rf *o bosh
```

C bosh.c

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

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

/* --- symbolic constants --- */
#define HOSTNAMEMAX 100
#define SHELL_MAX_PROCESSES 50

/* Interupt handler */
static void int_handler(int sig);

/* pid's for the running processes */
pid_t running_foreground_processes[SHELL_MAX_PROCESSES];
pid_t running_background_processes[SHELL_MAX_PROCESSES];

/* Gets user and hostnames and inserts them into a string with the format "user@hostname".
char *get_user_and_hostname(char *hostname, size_t size)
{
        FILE *hostnamefile;
        char hname[HOSTNAMEMAX];
        char line[HOSTNAMEMAX];

        hostnamefile = fopen ("/proc/sys/kernel/hostname", "r");

        while (fgets(line, HOSTNAMEMAX, hostnamefile))
        {
                if(sscanf(line, "%s", hname))
                {
                        snprintf(hostname, size, "%s@%s", getenv("USER"), hname);
                }
        }

        return hostname;
}

/* Sets dir to be the current working directory. */
```

```
47 char *getcurrentdir(char *dir, size_t size)
48 {
49         char cur_path_buffer[1024];
50         char *cur_path = getcwd(cur_path_buffer, sizeof(cur_path_buffer));
51         snprintf(dir, size, "%s", cur_path);
52         return dir;
53 }
54
55 /* Executes a shell command. */
56 int executeshellcmd (Shellcmd *shellcmd)
57 {
58         if (!strcmp(*shellcmd->the_cmds->cmd, "exit")){
59                 return 1;
60         }
61
62         char **cd_command = shellcmd->the_cmds->cmd;
63
64         if (strcmp(cd_command[0], "cd") == 0 && cd_command[1] != NULL)
65         {
66                 const char *newdir = cd_command[1];
67                 int error = chdir(newdir);
68
69                 switch (error)
70                 {
71                         case 0: break;
72                         case EACCES:
73                                 printf("%s: access denied.\n", newdir);
74                                 break;
75                         case ENOENT:
76                                 printf("%s: no such file or directory.\n", newdir);
77                                 break;
78                         case ENOTDIR:
79                                 printf("%s: not a directory.\n", newdir);
80                                 break;
81                         default:
82                                 printf("%s: error %i\n", newdir, error);
83                                 break;
84                 }
85
86                 return 0;
87         }
88
89         shell_cmd_with_pipes(shellcmd, 0);
90
91         return 0;
92 }
93
94 /* Main function for executing shell commands. */
95 int shell_cmd_with_pipes(Shellcmd *shellcmd, int write_pipe)
96 {
97         int fd[2];
```

```
98          char **cmd = shellcmd->the_cmds->cmd;
99          shellcmd->the_cmds = shellcmd->the_cmds->next;
100         int proc_pid;
101
102         /* Pipes if the shellcommand has more than one command. */
103         if(shellcmd->the_cmds != NULL)
104         {
105                 pipe(fd);
106         }
107
108         /* Fork process - if parent, just continue. If child, run if.*/
109         if(!(proc_pid = fork()))
110         {
111                 if(shellcmd->the_cmds != NULL)
112                 {
113                         close(fd[1]);
114                 }
115
116                 /* If any commands are left, close stdin and dup pipe.
117                    Else if stdin is redirected, close stdin and dup the
118                    stdin file.                                          */
119                 if(shellcmd->the_cmds)
120                 {
121                         close(fileno(stdin));
122                         dup(fd[0]);
123                 } else if(shellcmd->rd_stdin)
124                 {
125                         close(fileno(stdin));
126                         dup(fileno(fopen(shellcmd->rd_stdin, "r")));
127                 }
128
129                 /* If write_pipe is more than 0, close stdin and dup write_pipe.
130                    Else if stdout is redirected, close stdout and dup the stdout
131                    file.
132                 if(write_pipe > 0)
133                 {
134                         close(fileno(stdout));
135                         dup(write_pipe);
136                 } else if(shellcmd->rd_stdout)
137                 {
138                         close(fileno(stdout));
139                         dup(fileno(fopen(shellcmd->rd_stdout, "w+")));
140                 }
141
142                 /* If stderr is redirected, close stderr and dup redirected stderr.*/
143                 if(shellcmd->rd_stderr)
144                 {
145                         close(fileno(stderr));
146                         dup(fileno(fopen(shellcmd->rd_stderr, "w+")));
147                 }
148
```

```
149                 execvp(cmd[0], cmd);
150                 printf("Could not find command: %s \n", cmd[0]);
151         }
152
153         /* If background process, add proc_pid to list of background
154            processes, else add to foreground processes.                    */
155         if(shellcmd->background)
156         {
157                 add_background_process(proc_pid);
158         }
159         else
160         {
161                 add_foreground_process(proc_pid);
162         }
163
164         /* Close write_pipe if it is more than zero. */
165         if (write_pipe > 0)
166         {
167                 close(write_pipe);
168         }
169
170         /* If more commands are left, close read and run recursively. */
171         if(shellcmd->the_cmds != NULL)
172         {
173                 close(fd[0]);
174                 shell_cmd_with_pipes(shellcmd, fd[1]);
175         }
176
177         int exit_code;
178         /* If shell command is not marked as background, wait for pid. Else
179            do not wait.
180         if(!shellcmd->background)
181         {
182                 waitpid(proc_pid, &exit_code, 0);
183         }
184 }
185
186 /* Adds a process to list of foreground processes. */
187 int add_foreground_process(pid_t process)
188 {
189         int i = 0;
190         pid_t *proc;
191         while(*(proc = &(running_foreground_processes[i])) > 0
192                 && i < SHELL_MAX_PROCESSES) i++;
193
194         /* TODO: Should introduce handling too many processes. */
195
196         *proc = process;
197
198         return 0;
199 }
```

```
200
201 /* Adds a process to list of background processes. */
202 int add_background_process(pid_t process)
203 {
204         int i = 0;
205         pid_t *proc;
206         while(*(proc = &(running_background_processes[i])) > 0
207                 && i < SHELL_MAX_PROCESSES) i++;
208
209         /* TODO: Should introduce handling too many processes. */
210
211         *proc = process;
212
213         return 0;
214 }
215
216 /* --- main loop of the simple shell --- */
217 int main(int argc, char* argv[]) {
218
219         /* Handles Cntrl+c input. */
220         signal(SIGINT, int_handler);
221
222         /* Handles zombie processes. */
223         signal(SIGCHLD, SIG_IGN);
224
225         /* initialize the shell */
226         char *cmdline;
227         char hostname[HOSTNAMEMAX];
228         char currentdir[1024];
229         int terminate = 0;
230         Shellcmd shellcmd;
231
232         if (get_user_and_hostname(hostname, sizeof(hostname)))
233         {
234                 /* parse commands until exit or ctrl-c */
235                 while (!terminate)
236                 {
237                         printf("%s:", hostname);
238                         printf("%s", getcurrentdir(currentdir, sizeof(currentdir)));
239                         if (cmdline = readline("# "))
240                         {
241                                 if(*cmdline)
242                                 {
243                                         add_history(cmdline);
244                                         if (parsecommand(cmdline, &shellcmd))
245                                         {
246                                                 terminate = executeshellcmd(&shellcmd);
247                                         }
248                                 }
249
250                                 free(cmdline);
```

```
251                         }
252                         else terminate = 1;
253                 }
254
255                 printf("Exiting bosh.\n");
256         }
257
258         return EXIT_SUCCESS;
259 }
260
261 /* Interupt handler. Closes all running foreground processes. */
262 void int_handler(int sig)
263 {
264         sig = 0;
265         int i;
266         pid_t *process;
267
268         for (i = 0; *(process = &running_foreground_processes[i]) > 0
269                 && i < SHELL_MAX_PROCESSES; i++)
270         {
271                 kill(*process, SIGINT);
272                 *process = 0;
273         }
274 }
```

D parser.h

```
1  typedef struct _cmd {
2      char **cmd;
3      struct _cmd *next;
4  } Cmd;
5
6  typedef struct _shellcmd {
7      Cmd  *the_cmds;
8      char *rd_stdin;
9      char *rd_stdout;
10     char *rd_stderr;
11     int background;
12 } Shellcmd;
13
14 extern void init( void );
15 extern int parse ( char *, Shellcmd *);
16 extern int nexttoken( char *, char **);
17 extern int acmd( char *, Cmd **);
18 extern int isidentifier( char * );
```

E parser.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <ctype.h>
4  #include "parser.h"
5
6  /* --- symbolic constants --- */
7  #define COMMANDMAX 20
8  #define BUFFERMAX 256
9  #define PBUFFERMAX 50
10 #define PIPE  ('|')
11 #define BG    ('&')
12 #define RIN   ('<')
13 #define RUT   ('>')
14 #define IDCHARS "_-.,/~+"
15
16 /* --- symbolic macros --- */
17 #define ispipe(c) ((c) == PIPE)
18 #define isbg(c)   ((c) == BG)
19 #define isrin(c)  ((c) == RIN)
20 #define isrut(c)  ((c) == RUT)
21 #define isspec(c) (ispipe(c) || isbg(c) || isrin(c) || isrut(c))
22
23 /* --- static memory allocation --- */
24 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)
160 {
161   char *tok;
162   int n, cnt = 0;
163   Cmd *cmd0 = cmds;
164   cmds = cmds->next;
165   cmd0->next = NULL;
166   cmd0->cmd = pp;
167
168   while (1) {
169     n = nexttoken(s, &tok);
170     if (n == 0 || isspec(*tok))
171       {
172         *cmd = cmd0;
173         *pp++ = NULL;
174         return cnt;
175       }
176     else
177       {
178         *pp++ = tok;
179         cnt += n;
180         s += n;
181       }
182   }
183 }
184
185 int isidentifier (char *s)
186 {
187   while (*s)
188     {
189       char *p = strrchr (IDCHARS, *s);
190       if (! isalnum(*s++) && (p == NULL))
191         return 0;
192     }
193   return 1;
194 }
```