IT University of Copenhagen

Bachelor: Operating Systems and C, Handin for Group A

The BOSC Shell

Anders Edelbo Lillie
aedl@itu.dk

Andreas Bjørn Hassing Nielsen
abhn@itu.dk

Markus Thomsen
matho@itu.dk

September 27, 2016

Contents

1 Introduction and background

The purpose of this project is to create a simple command line Shell for Linux, in the C programming language. In this Shell (henceforth called BOSH), binary applications in common *bin* directories located on the system, must be able to be executed (*/bin/*, */usr/local/bin/* and */usr/bin/* for instance).

Furthermore, adhering to the project description[1], BOSH must have the following functionality:

- redirecting standard in and standard out
  - redirection of standard in: `program < filename`, feeds the contents of the file at `filename` as input to `program`.
  - redirection of standard out: `program > filename`, writes the printed output of the `program` into the file at `filename`.
- piping of programs
  - `program1 | program2`, feeds the final output of `program1` into `program2` as input.
- running programs in the background
  - `program &`, runs `program` as a process in the background, thus letting the user keep using the shell without being blocked by the execution of `program`.
- interrupting processes running in BOSH, without killing BOSH itself
- attempting to run a command that does not exist must result in BOSH printing a message that the command was not found.

2 Problem analysis

As described in section 1 a set of specific functionality was required of BOSH. In this section it is described how this functionality was analyzed, in order for it to be implemented.

As a template for the program was given, the overall structure and control flow was already designed, which meant the "only" focus was on the required functionality. At first the core functions were identified, so they could be implemented, before the rest of the tasks were assigned to the different members of the group. This would give a basis and common understanding of the program, and gave a skeleton for adding further functionality. Showing the host name in the commando prompt, as well as the execution of single commands were implemented through the use of pair programming (including the whole team). The former was simple, as it only required knowing a way to fetch the host name from the system, and print it on the screen. The latter being a key function in BOSH, had to be implemented carefully. The first idea was to scan through the directories `/bin/` and `/usr/bin/` and search for the entered command, as these directories contain the standard system programs on Linux. If the command was found, a check for whether it could be executed or not was performed, and finally the program would be executed through `execvp`. This was troublesome, and a lookup in the manual page for `execvp`, revealed that the function itself searched the directories listed in the environment variable `$PATH`. From there on, it was much simpler to implement the functionality, by using only `execvp` in a child process, and waiting for it to terminate in the parent process.

As BOSH became capable of executing commands, the rest of the functions were *just* extensions. The remaining tasks were distributed among the team members. Using git and GitHub, it was easy to do version control as well as track work progress. The tasks were created as *issues* on GitHub, and when a task was

[1] Although written in Danish, attached in Appendix, section 7.1

completed, a team member could assign themselves to another task.

Throughout the process some common issues arose. It had not been clear to us that the `parser` was actually fully implemented, meaning that it completely builds the `Shellcmd` struct, allowing functions to exploit this. Therefore, some time was wasted on modifying this, e.g. finding out when a redirection was issued with the token '<', and then assigning the file to the `Shellcmd rd_stdin` struct member, although this function was already taken care of. It then became clear that the functionality was only missing implementation in `bosc.c`.

While the original `while`-loop for running the commands in a `Shellcmd` was easy to understand, we had a hard time extending upon that design. When we implemented the pipe functionality, it was clear that we needed a different execution structure. The execution of commands was redesigned and reimplemented, replacing the `while`-loop with a recursive `execute` function. This allowed us to create a recursive fork-and-execute model, that effectively ended up in yielding less and more concise code.

Originally background processes were thought to be as easy to implement as simply invoking the `setsid(2)` function in a childprocess, if the command struct indicated it was a background process. However, after trying to get it to work, we realized that BOSH was still waiting for the process. We decided to solve the problem by forking once more into a new process from the child process, and kill the process linking it to BOSH, effectively severing the background process from its grand-parent.

3 Examples

- **Redirection of stdin and stdout**
  Read the system hostname and print it to a textfile "hostname":

  ```
  cat < /etc/hostname > hostname
  ```

  Of course, there can only be one redirection of `stdin` and one redirection of `stdout` per command, as it does not make sense to have input/output redirected to multiple locations.

- **Execute command as background process**
  To execute a command as a background process, simply append the '&' as you'd normally do in `bash`.

  ```
  datahub:# sleep 100 &
  datahub:# Background process ID: 19507

  datahub:# ps -x
    PID TTY      STAT   TIME COMMAND
  19411 ?        S      0:00 sshd: anon@pts/0
  19412 pts/0    Ss     0:00 -bash
  19505 pts/0    S+     0:00 ./bosh
  19507 ?        Ss     0:00 sleep 100
  19508 pts/0    R+     0:00 ps -x
  ```

  As can be seen, the pid of the new background process will be printed, and when we print an overview of the processes, we can see `sleep 100` running in the background, with no attached TTY[2].

- **Chaining commands with the pipe operator**
  The '|' operator can be used to pipe out from one command into another, as you'd normally do in bash.

[2] TeleTYpewriter, the terminal text input/output environment. Usually the parent process of any non-background processes called through it.

```
datahub:# ls | wc -c
117
datahub:#
```

In the example, the output of listing files in the current working directory is listed (`ls`), and the results of that command is piped into `wc -c`, which counts the total amount of characters in the file names.

- **Interrupting processes**
  Processes can be interrupted by pressing `Ctrl-C`, at any time. A child process running under BOSH will be interrupted, and if no process is running, BOSH itself will exit.

```
datahub:# sleep 99999999
^Cdatahub:#
```

- **Exiting and interrupting bosh**
  You can exit BOSH at any time, by using the builtin `exit` command or `Ctrl-C`.

4 Technical description of the program

In this section, the specified functionality is described with a technical description, both covering how they work, and how they are implemented in the source code.

4.1 Show hostname for commando prompt

Whenever BOSH is waiting for a command to be executed, the system host name must be visible to the left in the commando prompt. A normal bash shell would appear like the following, which displays the user "john", and the hostname "desktopPC":

```
john@desktopPC:~/
```

The specific signs following the host name is purely design choice, and was by default in the template ":# ". Determining the given host name is one of the first functions executed in the `main` entry point of BOSH, which makes sense, as it should be printed to the shell right away.

The function `gethostname2`[3], opens the file "/proc/sys/kernel/hostname", reads the first line with `fgets(3)`, closes the file stream, and removes trailing newlines (if any). The hostname is then assigned to a `char HOSTNAME[]`, and printed as the first thing in the command prompt. It will look like the example above, except it will not display which user is using the shell. Different signs following the hostname, as well as color highlighted letters (which most shells implement) are not present either. Example below:

```
desktopPC:#
```

[3]The number `2` is added to the function name, as `gethostname(2)` is a known system function from `unistd.h`

4.2 Executing commands

BOSH executes commands through spawned child processes. When a command is run, a child process is created through forking, and the new child process executes the command. In the meantime, BOSH waits for the child process to finish, unless the command is set to run in the background.

The child process executes the given command, by passing the command name to the `execvp(3)` function, along with an array of null-terminated arguments. When the child process is done, it exits. If the command was not set to run in the background, BOSH will at this point continue and await the next command to be entered.

4.2.1 Builtin commands

One builtin command exists: `exit`, which gracefully exits BOSH. The command is implemented via string comparison (`strcmp(3)`) on the command entered by the user, and is checked before forking.

4.2.2 Background commands

Using the standard `bash` token '&' for executing a process in the background, will flag the command to be executed independently of the BOSH parent process.

BOSH will parse any command ending with a '&' character, as a command to be run in the background, and fork into a new process, which will be severed from the parent process, making it the head of a new process group.

This is accomplished by forking from the first child process, invoking the `setsid(2)` function in the new child process, and killing the old child process with a `exit(EXIT_SUCCESS)` call, severing the link between the grand-parent process, and the new child process.

As BOSH no longer has any child processes, it can stop waiting for them to terminate, and continue executing commands.

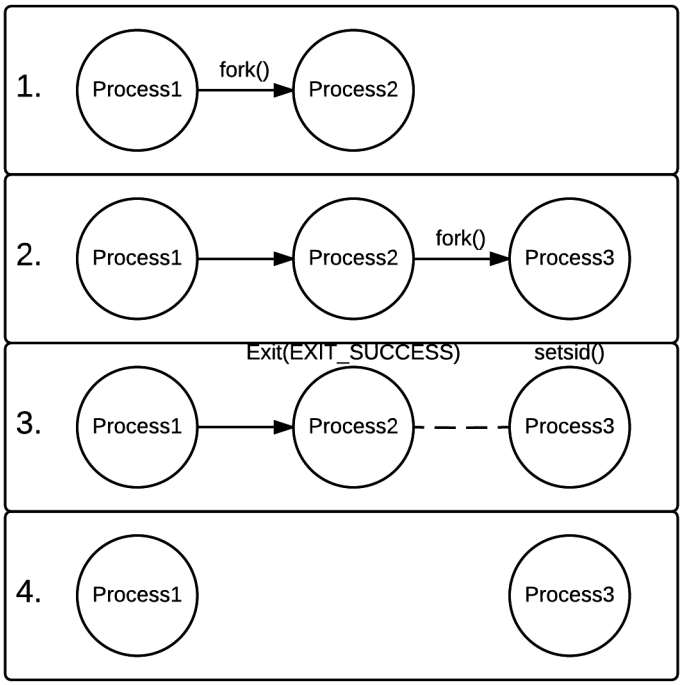

Figure 1: Background process execution, effectively detaching from the original parent process.

1. The main process forks as usual, when executing a command.
2. The child process checks that it's a background process, in which case it forks again.
3. The child process exits with success, separating it's link to the main process, and new child process, while the new child process calls the `setsid(2)` function, to make it the head of its own process group.
4. BOSH is now free to run a new command, while the new child process executes in the background.

4.2.3 Piping commands

When piping commands together, the parser in BOSH generates a `Shellcmd` with a list containing at least 2 commands (depending on the amount of commands that are piped together). Take note that the `Shellcmd` commands array is ordered in reverse, as this is important.

BOSH creates a pipe for the processes to communicate through, then forks the process, before recursively executing the next command, creating yet another pipe and fork, and so on. The first command that is executed, is the last one written in BOSH, as it will be the one expecting input from the next, and so on (this is why they are reverse ordered).

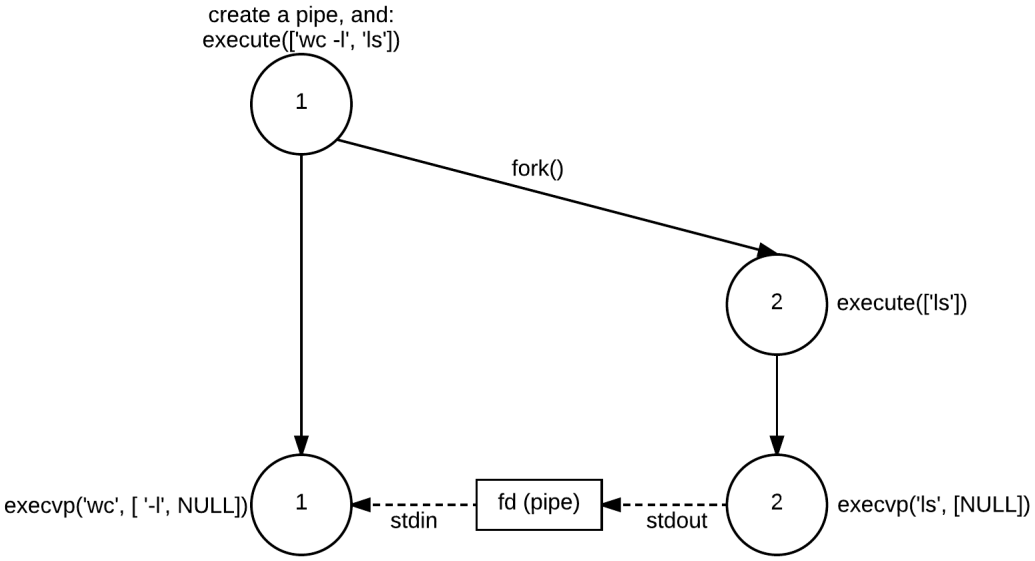

Figure 2: Example of running `ls | wc -l`, listing the current directory, and counting the number of lines in the output.

4.2.4 Redirection of stdin and stdout

Using the standard bash tokens '<' and '>' for redirecting `stdin` and `stdout` in BOSH, redirects any input and output from the terminal to the given file. Below is an example:

```
wc -l < /etc/passwd > newfile
```

BOSH will parse this command line and assign "/etc/passwd" and "newfile" in the `Shellcmd rd_stdin` and `rd_stdout` struct members respectively, as the identifiers '<' (in) and '>' (out) are parsed. This information is then accessible from `bosc.c`, where the redirection functionality is implemented in the `executeshellcmd` function. Before mentioned struct members are checked whether they contain anything or a reference to `NULL`. If the former is the case for either of them, `redirect.c` takes care of redirecting `stdin` and `stdout` to the filenames referred to from the `Shellcmd` struct. If an error occurs in the redirection, the process is exited, `stdin` and `stdout` are redirected to the terminal, and BOSH is ready for new commands. If no errors occur, the command is executed, as it "normally" would, but with custom `stdin`/`stdout`, and when terminated, the terminal is restored as `stdin`/`stdout` as described before.

4.3 Control-C interrupt

To interrupt BOSH, the `Ctrl-C` key binding can be pressed. This will interrupt and kill the BOSH process, returning control to the original instantiator of BOSH. A user, however, may want to exit a program that is

currently running through BOSH, without actually closing BOSH itself.

This functionality is obtained by propagating the `SIGINT`[4] signal BOSH receives, while currently running a process, onward to the child process that runs the actual program. When the child process is finished and exits, BOSH reverts its handling of SIGINT to default, such that the user, once again, can interrupt BOSH with `Ctrl-C`.

In technical terms, the `signal(2)` function is used to 'catch' the `SIGINT` signal, and change the default behavior to the one defined in `void propagate_signal(int sig);`. The function `propagate_signal` receives the ID of the signal that was interrupted (in the case of BOSH, always SIGINT), and sends the signal onward to the child process ID set by BOSH, via the `kill(2)` function. The function `kill` has a misleading name, what it actually does is to send some signal to a specified process ID, and in the case of BOSH the process running BOSH sends `SIGINT` to the process ID of the child executing the current command.

5 Test

BOSH functionality relies heavily on input/output, which makes it difficult to test the program in a systematic way, as well as the internal components. It is the assumption that the template, on which the program is implemented, worked as it should, thereby leaving the extra functionalities added throughout this project to be tested. The examples illustrated in section 3, as well as several more have been used to test BOSH every time a new function was added. These test cases were inspired by the requirements found in the project description (Appendix 7.1).

6 Conclusion

We set out to implement a minimal Linux shell, and have succeeded in covering all the required functionality. Our implementation can do the following:

- Show the name of the host machine.
- Run simple system commands, and respond with an error message if the call doesn't exist.
- Run commands as background processes using the '&' operator.
- Redirect `stdin` and `stdout` to files, using the '<' and '>' operator.
- Take the output of one command and use it as input for another command, using the '|' operator.
- Exit the shell using a built-in `exit` command.
- Interrupt processes by pressing the `Ctrl-C` key combination.

We're very satisfied with the result, and we don't feel like we've made any compromises or quick-fixes in terms of functionality. We had no major obstacles, but it was sometimes a bit confusing reading and understanding the given template of the shell implementation, and that may have slowed down the process of writing an otherwise simple function.

6.1 Possible improvements and extensions

One possible improvement to the program, would have been to implement a directory change command, like `cd` in `bash`. We could have used the `getcwd(3)` call to get the working directory, in which the bosh shell is started, and used the `chdir(2)` to change the current working directory of the shell.

Another improvement could've been to use the strategy design pattern for built-in commands, as that would make it a lot easier to implement new commands for the shell. The current hardcoded approach works fine for a single command, and maybe even two, but falls short if you need more.

[4] Interrupt Signal, commonly sent to the currently running program via a terminal, by typing `Ctrl-C` on the keyboard

7 Appendix

7.1 Project description

Intentionally blank space, as a seperate .pdf file is included in the report, and begins on the next page

IT-Universitetet i København 6. september 2016

Operativsystemer og C

Obligatorisk opgave 1

Rapporten - som **pdf-fil** - samt kildekode skal pakkes og navngives „BOSC-opg1-dit-navn“ og uploades til LearnIT - en per gruppe - senest onsdag den 28. september, klokken 12:00.

Baggrund

En shell er et program som tilbyder en brugergrænseflade til operativsystemet. Brugeren kan taste kommandoer ind og få dem udført på den ønskede måde.

Et eksempel kunne være en bruger, der ønsker at se filerne i en folder. Brugeren intaster `ls` i shell'en, hvorefter shell'en parser og processerer strengen „ls“ og begynder at søge efter den udførbare fil ls i filsystemet. Efter noget tid finder den filen `/usr/bin/ls`, der skal eksekveres. Til det formål starter shell'en en ny proces som sættes til at eksekvere `/usr/bin/ls`. Herefter venter shell'en på at programmet bliver færdigt. Hvis der er fejl meddeles denne, ellers forsættes med at læse den næste kommando som brugeren indtaster.

I denne opgave skal du lave din egen shell, som vi kalder `bosh` (BOSC shell), med funktionalitet svarende til en begrænset version af `bash` på Linux.

Specifikation af `bosc`

Hvad skal `bosh` - som minimum - kunne:

- `bosh` skal kunne virke uafhængigt. Du må ikke bruge andre eksisterende shells, f.eks. er det ikke tilladt at anvende et systemkald `system()` til at starte `bash`.
- Kommando-prompt'en skal vise navnet på den host den kører på.
- En bruger skal kunne indtaste almindelige enkeltstående kommandoer, så som `ls`, `cat` og `wc`. Hvis kommandoen ikke findes i operativ systemet skal der udskrives en „Command not found“ meddelelse.
- Kommandoer skal kunne eksekvere som baggrundsprocesser (ved brug af `&`) sådan at mange programmer kan køres på samme tid.
- Der skal være indbygget funktionalitet som gør de muligt at lave redirection af stdin og stdout til filer. F.eks skal kommandoen

```
wc -l < /etc/passwd > antalkontoer
```

lave en fil „antalkontoer“, der indeholder antallet af brugerkontoer.

- Det skal være muligt at anvende pipes. F.eks. skal

  ```
  ls | wc -w
  ```

  udskrive antallet af filer.

- Funktionen `exit` skal være indbygget til at afslutte shell'en.
- Tryk på Ctrl-C skal afslutte det program, der kører i `bosh` shell'en, men ikke shell'en selv.

Du er velkommen til at tilføje mere funktionalitet efter eget ønske.

Hints

På kursusbloggen er uploadet `oo1.zip`, som indeholder en ufuldstændig version af `bosh`, der kan benyttes som udgangspunkt. Hertil hører også en simpel `Makefile`, som du selv kan modificere efter behov.

Den ufuldstændige version af `bosh` giver mulighed for at indtaste kommandoer, som parses til en shellcmd `struct` med følgende definition:

```
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;
```

Programmet benytter sig af GNU readline biblioteket, hvilket betyder at man kan indtaste de samme tegn og kommandoer som i `bash`. Desuden er der implementeret en „history“ funktion som gør det muligt at browse igennem tidligere kommandoer.

Første skridt efter at du har downloadet `oo1.zip` (og evt. `apt-get install unzip`):

```
# unzip oo1.zip
# cd oo1
# make
# ./bosh
```

Bemærk, at kommandoerne parses i modsat rækkefølge af den de er indtastet i (med god grund). Du kan kigge i `print.c` og i funktionen `printshellcmd()` for at se hvordan `struct`'en tilgås.

Fremgangsmåde

Det er muligt at implementere funktioner for alle specifikationerne i den rækkefølge de er skrevet overfor. Implementér dem én af gangen. Test funktionaliteten og når det ser ud til at fungere som ønsker, så gå videre til næste punkt.

I forbindelse med at skrive C funktionerne er det nødvendigt at studere diverse systemkald. For nogle af dem, f.eks. `exec`, eksisterer der forskellige varianter. Du er nødt til at finde den der bedst passer til dit behov - nogle gange er der adskillige der er brugbare.

Systemkald som du helt sikkert skal bruge: `fork, exec, wait, pipe, dup`.

Rapport

Rapporten skal dels være dokumentation af jeres arbejde med projektet og give indblik i jeres tankegang. Samtidigt er rapporten et produkt i sig selv og ment som en træning i at skrive rapporter inden i skal til at lave bachelor rapporten senere i jeres uddannelse. Rapporten skal derfor være et selvstændigt dokument (forside, indledning, beskrivelse af hvordan hver enkelt funktionalitet er implementeret, testet, samt en konklusion). Alt relevant kildekode (+Makefile) skal inkluderes i et appendix samt zippes med afleveringen.

7.2 Code

7.2.1 Makefile

```
all: bosh

OBJS=bosh.o parser.o redirect.o
LIBS=-lreadline -ltermcap
CC=gcc
CCFLAGS=-Wall -Werror

bosh: $(OBJS)
        $(CC) $(CCFLAGS) -o $@ $(OBJS) $(LIBS)

bosh.o: bosh.c
        $(CC) $(CCFLAGS) -c bosh.c parser.h

parser.o: parser.h parser.c
        $(CC) $(CFLAGS) -c parser.h parser.c

redirect.o: redirect.h redirect.c
        $(CC) $(CFLAGS) -c redirect.c

.PHONY: clean

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

debug: CCFLAGS += -DDEBUG -g
debug: clean bosh
```

7.2.2 bosh.c

```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 "parser.h"

#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include "redirect.h"

/* --- symbolic constants --- */
#define HOSTNAMEMAX 100
static int SAVED_STDIN;
static int SAVED_STDOUT;

static pid_t pid_to_interrupt = 0;

void propagate_signal(int sig) {
    // kill seems aggressive, but all it does is
    // to send a signal (2nd argument) to another
    // process
    kill(pid_to_interrupt, sig);
}

/* --- use the /proc filesystem to obtain the hostname --- */
char *gethostname2(char *hostname)
{
    FILE *hostnamefile;
    hostnamefile = fopen("/proc/sys/kernel/hostname", "r");
    fgets(hostname, HOSTNAMEMAX, hostnamefile);
    fclose(hostnamefile);

    // remove trailing newline from hostname
    sscanf(hostname, "%[^\n]", hostname);

    return hostname;
```

```
}

/*Restore std IO to terminal*/
int restore_std_IO() {
    dup2(SAVED_STDIN, STDIN_FILENO);
    close(SAVED_STDIN);
    dup2(SAVED_STDOUT, STDOUT_FILENO);
    close(SAVED_STDOUT);
    return 0;
}

/*Execute command with terminal IO*/
void execute(Cmd *cmdlist) {
    char **cmd = cmdlist->cmd;
    cmdlist = cmdlist->next;

    pid_t pid = -1;
    int fd[2];

    // any more commands remaining?
    if (cmdlist) {
        // if so, setup the pipe
        if (pipe(fd) < 0) {
            fprintf(stderr, "Failed to setup pipe");
            exit(1);
        }
        pid = fork();
    }

    if (pid == 0) {
        // pipe sender
        close(STDOUT_FILENO);
        close(fd[0]);
        dup(fd[1]);

        execute(cmdlist);
    } else {
        if (cmdlist) {
            // pipe receiver
            close(STDIN_FILENO);
            close(fd[1]);
            dup(fd[0]);
        }

        execvp(*cmd, cmd);
        switch (errno) {
            case ENOENT:
                fprintf(stderr, "command not found\n");
```

```
                exit(ENOENT);
                break;
            case EACCES:
                fprintf(stderr, "permission to run %s denied\n", *cmd);
                exit(EACCES);
                break;
        }

        exit(0);
    }
}

/* --- execute a shell command --- */
int executeshellcmd (Shellcmd *shellcmd)
{
    Cmd *cmdlist = shellcmd->the_cmds;

    char **cmd = cmdlist->cmd;

    // if builtin command `exit` is run, kill the shell
    if (!strcmp(*cmd, "exit"))
        return 1;

    pid_t pid = fork();
    if (pid == 0) {
        if (shellcmd->rd_stdin) {
            /*Redirect stdin*/
            if(redirect_stdincmd(shellcmd->rd_stdin) == -1) {
                fprintf(stderr, "Unable to redirect stdin to %s\n", shellcmd->rd_stdin);
                exit(0);
            }
        }
        if (shellcmd->rd_stdout) {
            /*Redirect stdout*/
            if(redirect_stdoutcmd(shellcmd->rd_stdout) == -1) {
                fprintf(stderr, "Unable to redirect stdout to %s\n", shellcmd->rd_stdout);
                exit(0);
            }
        }
        if (shellcmd->background) {
            // divorce from parent process by forking child again,
            // and killing the middle-child
            int child;
            child = fork();
            if (child < 0)
                exit(EXIT_FAILURE);
            if (child > 0)
                exit(EXIT_SUCCESS);
```

```
            // set new child as process group leader, unless something bad happens
            if (setsid() < 0)
                exit(EXIT_FAILURE);
            printf("Background process ID: %d\n", getpid());
        }

        execute(cmdlist);
    } else {
        // interrupt ctrl+c, and propagate it to command process
        pid_to_interrupt = pid;
        signal(SIGINT, propagate_signal);

        waitpid(pid, NULL, 0);
        // revert the interrupt signal so bosh can exit
        signal(SIGINT, SIG_DFL);
    }

    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;
    SAVED_STDIN = dup(STDIN_FILENO); //Save current stdin
    SAVED_STDOUT = dup(STDOUT_FILENO);
    Shellcmd shellcmd;

    if (gethostname2(hostname)) {

        /* parse commands until exit or ctrl-c */
        while (!terminate) {
            printf("%s", hostname);
            if ((cmdline = readline(":# "))) {
                if(*cmdline) {
                    add_history(cmdline);
                    if (parsecommand(cmdline, &shellcmd)) {
                        terminate = executeshellcmd(&shellcmd);
                        restore_std_IO();
                    }
                }
                free(cmdline);
            } else terminate = 1;
        }
        printf("Exiting bosh.\n");
```

```
    }

    return EXIT_SUCCESS;
}
```

7.2.3 redirect.h

```
/*
   Redirection of standard in/out

 */

#ifndef _REDIRECT_H
#define _REDIRECT_H

int redirect_stdincmd(char *);
int redirect_stdoutcmd(char *);

#endif
```

7.2.4 redirect.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 "redirect.h"

// S_IRUSR = user can read
// S_IWUSR = user can write (overwrite and delete)
// S_IRGRP = group can read
// S_IROTH = other users can read
#define FILE_RIGHTS (S_IRUSR | S_IWUSR) | S_IRGRP | S_IROTH

int redirect_stdincmd(char *infilename)
{
    /* Create a handle (file descriptor) to a file*/
    int fid = open(infilename, O_RDONLY); //Handle (file descriptor) to a file
    close(STDIN_FILENO); //close the standard input
    int new_fid = dup(fid); //Assign "fid" as new standard input
    close(fid); // The file descriptor is duplicated close the opened file
    return new_fid; //Return -1 if error
}

int redirect_stdoutcmd(char *outfilename)
{
    int fid = open(outfilename, O_CREAT|O_WRONLY|O_TRUNC, FILE_RIGHTS);
    close(STDOUT_FILENO);
    int new_fid = dup(fid);
    close(fid);
```

```
    return new_fid;
}
```

7.2.5 parser.h

We did not modify `parser.c` in any way, so we have not included it in our appendix. We did however alter `parser.h` slightly, to avoid compiler warnings on a implicit declaration of `parsecommand`.

```
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;

int parsecommand(char *cmdline, Shellcmd *shellcmd);

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