Peter Caprioli
Fabian S

ID2200, Laboration

Implementing a Unix Shell

2015-05-17

Final version, approved by examiner

To students

Please do not use this report or code for cheating, you'll learn a lot from building your own shell and understanding the inner workings of process management in Unix.

Summary

This report addresses a lab exercise in the course ID2200 given at the Royal Institute of Technology. The exercise was to build a small, yet functional shell which can be executed on Unix systems. We have written such a shell in the C programming language.

This report contains information about how the shell was implemented, how the code is structured, how some of the more important functions works and how it handles processes and system calls. The report also contains a short discussion at the end, with full source code included in the appendix.

The reader is expected to have some experience with Unix/Linux, system calls and programming in general.

Introduction

The purpose of this lab assignment was to increase our knowledge about process management within Unix. To do so, we were asked to implement a small shell, similar to Bash or any other available shell. Since Bash and other "real world" shells have a huge set of features, we were only required to implement the core functionality of what a shell should be able to do. We have, for example, not implemented tab completion, history or most of the regular built-in commands like echo.

Both authors of this report are experienced with Linux/Unix, low level programming and C since many years back, even before KTH.

Task description

For full details, please see https://www.kth.se/social/course/ID2200/page/laborations-pm/

The lab specifies several tasks and constraints, mentioned in the above link. In short, the shell should be working, there should be no "special cases" where it would crash or behave in a bad way. Memory management needs to be perfect, no memory leaks are accepted.

The code should compile without any errors on a reference system provided by KTH. The code should also be made available to the examiners via AFS. The report needs to give instructions on how to compile and run the code.

Implementation details

Our code implementation is pretty straight forward. The first thing we do is `register_signal()` in order to catch `SIGINT` and `SIGCHLD`. We then call `setpgid()` in order to create a new process group under which our child processes will live.

We use a main loop which is responsible for reading commands from `STDIN` using our function `readline()`, which in turn uses `fgets()` to read from stdin.

`readline()` takes a `char**` as its only argument which is then populated with one command or argument per element in the input. This makes it easy to call `execv()` later on, since we only need to pass the `char**`. It is also easy to "rewrite" certain words or characters, like `&`. This makes it easy to determine if we should execute in the background or not.

The main loop first examines if the command is a built in command. If it is, the buffer is passed on to the corresponding function which handles the command. If it is not, we pass the buffer to the function `executeCmd()` which then forks, attempts to `exec()` the command and then checks if everything worked out as it should by using `waitpid()`.

`checkEnv()` was the hardest-to-implement function in our code. We wanted to keep our code as clean as possible, and we decided to implement two similar `checkEnv()` functions. One is executed if there are no arguments (and hence `grep` does not need to be executed) and the other one is executed when we have arguments.

The first one uses two pipes and the second one uses three pipes. Piping works the same way in both functions. We have three cases of pipes: first, middle and last. These also works the same way, with the exception that we do not redirect `STDIN` in the first pipe nor `STDOUT` in the last.

`checkEnv()` calls `wait()` two (three with arguments) times and makes sure the processes ended in a sound manner. The last command in the chain is determined by `getenv("PAGER")`. If the environment variable PAGER exists, we attempt to execute it by using `execlp()`. Since `execlp()` replaces the current process, we cannot run any more code after it has been called. This enables us to simply call `execlp()` multiple times. If it returns, we know that the execution failed and our code will resume, executing less or more. Should all three `execlp()` fail, we print an error message on `STDERR` and terminate.

Upon entering the built-in command exit we `free()` all memory which was allocated and call `killpg()` in order to terminate background processes.

Validation

We believe our code fully conforms to the specification requirements.

Build in commands

All commands as per the specifications have been implemented. `checkEnv` was by far the hardest one, but we believe it has been implemented correctly. We have executed the same set of commands in Bash and the output is identical to the one produced by our shell. Each command runs in its own fork, where the output/input is redirected by `dup2()`.

Executing arbitrary commands

Execution of non built in commands are handled by `execvp()`. A compile time macro SIGDET selects whether the shell should detect terminated processes by either signals or by polling. If polling is being used, the parent fork regularly runs `waitpid(t_pid, &status, WNOHANG)` to check if the process has terminated. If signals are being used, we catch the `SIGCHLD` signal and then use `wait()` to determine which process ended.

Memory management

Valgrind was used with the `--leak-check=full` argument. We tested all commands, both with polling and signals, and Valgrind did not find a single memory leak. In addition, our code uses `malloc()` very sparingly, making it easy to follow where we need to `free()` memory.

Memory is `free()`'d once a command is done executing. Memory usage is therefore constant during the lifetime of the shell process. We have inspected our code to make sure memory is `free()`'d every time `exit()` is called, a child is terminated and for any other reason we do not need the memory any more. Our design tries to `free()` as much memory as possible, as soon as it is possible.

ANSI C

We compile our shell with the `-pedantic` and `-ansi` flags in GCC without errors or warnings.

System call return values

We have gone through our code line by line looking for system calls. For every system call we use, we checked the corresponding `man` page to make sure we check for errors. Some system calls, like `getpid()`, never fails and hence we do not check it for errors.

Correctly terminating processes

When exiting our shell, we send `SIGTERM` to our process group. Processes may catch the signal and ignore it, but that's not bad programming on our side. Besides, we do not want to use "stronger" signals like `SIGKILL`, since it might terminate a process in the middle of doing something important.

The shell should behave controlled

We believe it does. We have tried executing lots of background processes. We handle signals like `SIGINT` in a correct way and we use `sighold()` and `sigrelse()` to prevent signals from reaching us while we are executing another process. It behaves much like Bash, with (much) less features. And no Shellshock.

We ran automated tests which we simply pasted into the terminal, causing pretty much all the functionality of our shell to be tested. During these tests, we usually kept one or more background process running while at the same time running Valgrind.

Keeping process lists

We do not keep any process list. We use process groups, `wait()`, `waitpid()`, etc in combination with processes groups.

How to compile

Copy the source code from the AFS folder:

```
$ cp -R /afs/kth.se/home/f/a/??????/OS-Labb .
```

Compile the shell with

```
$ cd OS-Labb && gcc -pedantic -Wall -ansi -O4 -o shell shell.c
```

A binary executable `shell` will be created. Run it by typing

```
$ ./shell
```

Finally, when you wish to exit our shell, type `exit`.

Reflection

This was a fun lab exercise. We both had previous experience dealing with C, system calls and low level programming. This probably made the lab easier for us, but none the less we learnt a lot by actually implementing this sort of program. We do have a lot more practical experience with “exotic” system calls like `pipe()` now.

We spent approximately 4 evenings programming this lab, a total of about 15 hours of actual programming.

The difficulty was reasonable, we both wish the instruction would have been clearer though. For example, the instructions stated that our shell should “behave controlled, i.e. as bash”. What exactly does this mean? Does it mean we have to implement tab completion? What about pipes in commands? What about double ampersands or redirecting `STDERR` to `STDOUT`?

Appendix: Source Code

shell.c

```
#ifndef _XOPEN_SOURCE
#define _XOPEN_SOURCE 500
#endif

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

#define SIGDET 1
#define BUFFSIZE 128
#define PROMPTSTRING "$ "

#define PIPE_READ 0
#define PIPE_WRITE 1

/*
 * buff is an array of pointers, in total <= BUFFSIZE pointers
 * if buff[i] is not null, free it and set it to null (to avoid freeing it
 * again)
 */
void freeBuff(char **buff)
{
    int i;
    for (i=0; i<BUFFSIZE; i++) {
       if (buff[i] != NULL) {
             free(buff[i]);
             buff[i] = NULL;
       }
    }
}

/* return 0 on success, else 1
 * read up to BUFFSIZE bytes from stdin, then split the bytes by space.
```

```
 * Allocate memory for each word and place (the pointers to) each word in
 * buff.
 */
int readline(char **buff)
{
    void *res;
    char tmp[BUFFSIZE];
    int index;
    char* current;
    char* argv;
    int i;

    printf(PROMPTSTRING);

    /* to avoid breaking things there must be no signals when reading with
     * fgets which calls read()
     */
    if (sighold(SIGCHLD)) perror("sighold");
    if (sighold(SIGINT)) perror("sighold");
    res = fgets(tmp, BUFFSIZE, stdin);
    if (sigrelse(SIGINT)) perror("sigrelse");
    if (sigrelse(SIGCHLD)) perror("sigrelse");

    if (res == NULL) {
       /* read eof or an error occurred */
       return 1;
    }

    /* the current position in buff */
    index = 0;

    tmp[strlen(tmp)-1] = '\0'; /* remove the newline */

    while ( (current = strtok( (index==0 ? tmp : NULL), " ")) != NULL) {
       /*printf("Read: %s.\n", current);*/
       argv = malloc( sizeof(char) * strlen(current) + 1);
       if (argv == NULL) {
             /* could not allocate memory, free the memory already
              * allocated and call exit */
             for (i=0; i<BUFFSIZE && buff[i] != argv; i++) {
                   if (buff[i] != NULL) {
                         free(buff[i]);
                   }
             }
             exit(1); /* Cannot allocate memory */
       }
       strcpy(argv, current);
       buff[index] = argv;
```

```
        index++;
    }
    return 0;
}

/* call getenv and return the string if successfull */
char *getHOME()
{
    char *HOME;
    if ((HOME = getenv("HOME")) == NULL) {
        fprintf(stderr, "could not getenv(HOME): %s\n", strerror(errno));
        return NULL;
    }
    return HOME;
}

/* if called with no argument, cd to $HOME
 * else cd to the given posision with chdir */
int cd(char *cdpos)
{
    /* cdpos holds the string to which we want to cd */
    if (cdpos == NULL) {
        return cd(getHOME());
    } else if (chdir(cdpos)) {
        fprintf(stderr, "Could not cd to '%s': %s\n", cdpos, strerror(errno));
        return 1;
    }
    return 0;
}

/* print the result of getcwd() */
void pwd()
{
    char buff[BUFFSIZE*100]; /* just a (too large) buffer */
    if (getcwd(buff, BUFFSIZE*100) == NULL) {
        fprintf(stderr, "Could not get cwd: %s\n", strerror(errno));
    } else {
        printf("%s\n", buff);
    }
}

/* set up a struct sigaction with the parameters
 * sig (the signal)
 * handler (the function to call)
 * flags
 */
void register_signal(int sig, void (*handler)(int sig), int flags)
{
```

```
    struct sigaction params;
    int ret;

    params.sa_handler = handler;
    if (sigemptyset(&(params.sa_mask))) perror("sigemptyset");
    params.sa_flags = flags;
    ret = sigaction(sig, &params, NULL);
    if (ret == -1) perror("sigaction");
}

/* handle SIGINT (and do nothing) */
void SIGINT_handler(int sig)
{
    /*printf("Caught SIGINT\n");*/
}

/* handle SIGCHLD,
 * wait for a child process to change status, then return
 */
void SIGCHLD_handler(int sig)
{
    int status;
    pid_t pid;
    int fd = STDOUT_FILENO;
    char *buff = "exited!\n";
    size_t count = 8;
    /*printf("SIGCHLD_handler waiting...\n");*/
    status = 0;
    pid = wait(&status);
    /*printf("have waited, got pid %d\n", pid);*/

    if (pid == -1) {
       /* an error occurred */
       perror("wait");
    }

    if (WIFEXITED(status)) {
       if (-1 == write(fd, buff, count)) {
             /*
              * error!
              * though there is nothing reasonable to do since we are
              * in the signal handler
              */
       }
       /*printf("exited!\n");*/
    } else if (WIFSIGNALED(status)) {
       /*printf("WIFSIGNALED signal signaled\n");*/
    } else if (WIFSTOPPED(status)) {
```

```
       /*printf("WIFSTOPPED signal signalled stopped\n");*/
    }

    /*printf("in signal handler: pid: %d\n", getpid());*/
}

/* detection with signals, when the child process terminates the signal handler
 * will be called */
#if SIGDET == 1
void executeCmdBg(char **buff)
{
    pid_t pid;

    /*printf("sigdet 1: prepare to fork!\n");*/
    if ((pid = fork()) == 0) {
       execvp(buff[0], buff);
       freeBuff(buff);
       exit(1);
    } else if (pid == -1) {
       perror("fork");
    }

    return;
}
#endif /* sigdet == 1 */

/* detection with polling, spawn a polling process which calls waitpid
 * nonblocking to find out then the child process has exited */
#if SIGDET == 0
void executeCmdBg(char **buff)
{
    pid_t pid;
    int status;

    /* detection with polling */
    /*printf("sigdet 0: prepare to fork!\n");*/
    if (fork() == 0) {
       /* this is the polling process */

       /*printf("forked\n");*/
       if ((pid = fork()) == 0) {
             /* and here is the actual (executing) process */
             execvp(buff[0], buff);
             freeBuff(buff);
             exit(1);
       } else if (pid == -1) {
             perror("fork");
       }
```

```
        /* we don't need this in the polling process */
        freeBuff(buff);

        while (1) {
              /* break if there is a change in the child process or
               * an error */
              if (waitpid(pid, &status, WNOHANG)) break;
        }

        if (WIFEXITED(status)) {
              /*printf("exited!\n");*/
              return;
        } else if (WIFSIGNALED(status)) {
              /*printf("signaled\n");*/
        } else if (WIFSTOPPED(status)) {
              /*printf("signalled stopped\n");*/
        }
        exit(0);
    }
    return;
}
#endif /* sigdet == 0 */

/* fork and execute the command and print the time it took */
void executeCmd(char **buff, int background)
{
    int status;
    pid_t pid;
    struct timeval t1, t2;
    double t1val, t2val;

    if (background) {
        /* execute in background instead */
        executeCmdBg(buff);
        return;
    }

    /* work and start executing */
    if ((pid = fork()) == 0) {
        execvp(buff[0], buff);
        freeBuff(buff);
        exit(1);
    } else if (pid == -1) {
        perror("fork");
    }

    /* get the current time */
```

```
    if (gettimeofday(&t1, NULL) != 0) {
       fprintf(stderr, "Could not get time: %s\n", strerror(errno));
    }

    /* wait for the process to finish */
    waitpid(pid, &status, 0);

    /* and get the time now */
    if (gettimeofday(&t2, NULL) != 0) {
       fprintf(stderr, "Could not get time: %s\n", strerror(errno));
    }

    /* compute the difference and print it */
    t1val = ((double) t1.tv_sec) + ((double) t1.tv_usec)/1000000.0;
    t2val = ((double) t2.tv_sec) + ((double) t2.tv_usec)/1000000.0;
    printf("%s exited, time used: %f s\n", buff[0], (double) t2val - t1val);
}

/* execute: printenv | sort | less,
 * fork for the programs to execute,
 * use dup2 to replace stdout with pipes for printenv and sort and use dup2 to
 * replace stdin with pipes for sort and less */
void checkEnv()
{
    int pipe_1[2], pipe_2[2];
    int status;
    pid_t cp;
    char *pager;

    /*printf("checkEnv\n");*/

    if (0 != pipe(pipe_1)) {
       fprintf(stderr, "Failed to execute pipe()\n");
    }
    if (0 != pipe(pipe_2)) {
       fprintf(stderr, "Failed to execute pipe()\n");
    }

    cp = fork();
    if (cp == 0) { /*In child fork, printenv */
       /* replace stdout with the pipe */
       if (dup2(pipe_1[PIPE_WRITE], STDOUT_FILENO) == -1) {
             fprintf(stderr, "Could not dup2()\n");
       }

       /* and close all pipes which are not used in this process */
       if (close(pipe_1[PIPE_READ]) == -1) {
             fprintf(stderr, "Could not close()\n");
```

```
    }

    if (close(pipe_1[PIPE_WRITE]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_2[PIPE_READ]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_2[PIPE_WRITE]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }
    /*printf("exec printenv\n");*/
    execlp("printenv", "printenv", (char *) 0);
    exit(1);
}
if (-1 == cp) {
    perror("fork");
}

cp = fork();
if (cp == 0) { /*In child fork, sort */
    /*printf("starting sort\n");*/
    /* replace stdin and stdout with the pipe */
    if (dup2(pipe_1[PIPE_READ], STDIN_FILENO) == -1) {
          fprintf(stderr, "Could not dup2()\n");
    }

    if (dup2(pipe_2[PIPE_WRITE], STDOUT_FILENO) == -1) {
          fprintf(stderr, "Could not dup2()\n");
    }

    /* and close all pipes which are not used in this process */
    if (close(pipe_1[PIPE_READ]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_1[PIPE_WRITE]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_2[PIPE_READ]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_2[PIPE_WRITE]) == -1) {
          fprintf(stderr, "Could not close()\n");
```

```
    }
    /*printf("exec sort\n");*/
    execlp("sort", "sort", (char *) 0);
    exit(1);
}
if (-1 == cp) {
    perror("fork");
}

cp = fork();
if (cp == 0) { /*In child fork, PAGER/less/more */
    /*printf("starting pager\n");*/
    pager = getenv("PAGER");

    /* replace stdin with the pipe */
    if (dup2(pipe_2[PIPE_READ], STDIN_FILENO) == -1) {
          fprintf(stderr, "Could not dup2()\n");
    }

    /* and close all pipes which are not used in this process */
    if (close(pipe_1[PIPE_READ]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_1[PIPE_WRITE]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_2[PIPE_READ]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_2[PIPE_WRITE]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }
    /*printf("exec pager\n");*/
    if (pager != NULL) {
          execlp(pager, pager, (char *) 0);
          return;
    }
    execlp("less", "less", (char *) 0);
    execlp("more", "more", (char *) 0);
    fprintf(stderr, "Could not find less/more/PAGER\n");
    exit(1);
}
if (-1 == cp) {
    perror("fork");
}
```

```
    /* close all pipes which are not used in this process (which is none) */
    if (close(pipe_1[PIPE_READ]) == -1) {
       fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_1[PIPE_WRITE]) == -1) {
       fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_2[PIPE_READ]) == -1) {
       fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_2[PIPE_WRITE]) == -1) {
       fprintf(stderr, "Could not close()\n");
    }


#if SIGDET == 1
    if (sighold(SIGCHLD)) perror("sighold");
#endif /* SIGDET == 1 */

    /* wait for all the processes to finish */
    /*printf("wait 1\n");*/
    cp = wait(&status);
    if (-1 == cp) {
       perror("wait");
    }

    /*printf("wait 2\n");*/
    cp = wait(&status);
    if (-1 == cp) {
       perror("wait");
    }

    /*printf("wait 3\n");*/
    cp = wait(&status);
    if (-1 == cp) {
       perror("wait");
    }

#if SIGDET == 1
    if (sigrelse(SIGCHLD)) perror("sigrelse");
#endif /* SIGDET == 1 */

    /*printf("done\n");*/
```

```
}

/* just like checkEnv except that grep with arguments is used between printenv
 * and sort */
void checkEnvArg(char** buff)
{
    int pipe_1[2], pipe_2[2], pipe_3[2];
    int status;
    pid_t cp;
    char *pager;

    /*printf("checkEnv\n");*/

    if (0 != pipe(pipe_1)) {
       fprintf(stderr, "Failed to execute pipe()\n");
    }
    if (0 != pipe(pipe_2)) {
       fprintf(stderr, "Failed to execute pipe()\n");
    }
    if (0 != pipe(pipe_3)) {
       fprintf(stderr, "Failed to execute pipe()\n");
    }

    cp = fork();
    if (cp == 0) { /*In child fork, printenv */
       freeBuff(buff);

       /* replace stdout with pipe */
       if (dup2(pipe_1[PIPE_WRITE], STDOUT_FILENO) == -1) {
             fprintf(stderr, "Could not dup2()\n");
       }

       /* close unused pipes */
       if (close(pipe_1[PIPE_READ]) == -1) {
             fprintf(stderr, "Could not close()\n");
       }

       if (close(pipe_1[PIPE_WRITE]) == -1) {
             fprintf(stderr, "Could not close()\n");
       }

       if (close(pipe_2[PIPE_READ]) == -1) {
             fprintf(stderr, "Could not close()\n");
       }

       if (close(pipe_2[PIPE_WRITE]) == -1) {
             fprintf(stderr, "Could not close()\n");
       }
```

```
   if (close(pipe_3[PIPE_READ]) == -1) {
         fprintf(stderr, "Could not close()\n");
   }

   if (close(pipe_3[PIPE_WRITE]) == -1) {
         fprintf(stderr, "Could not close()\n");
   }
   /*printf("exec ls\n");*/
   execlp("printenv", "printenv", (char *) 0);
   exit(1);
}
if (-1 == cp) {
   perror("fork");
}

cp = fork();
if (cp == 0) { /*In child fork, grep */
   /* replace stdin and stdout with pipes */
   if (dup2(pipe_1[PIPE_READ], STDIN_FILENO) == -1) {
         fprintf(stderr, "Could not dup2()\n");
   }

   if (dup2(pipe_2[PIPE_WRITE], STDOUT_FILENO) == -1) {
         fprintf(stderr, "Could not dup2()\n");
   }

   /* the pipes are no longer used */
   if (close(pipe_1[PIPE_READ]) == -1) {
         fprintf(stderr, "Could not close()\n");
   }

   if (close(pipe_1[PIPE_WRITE]) == -1) {
         fprintf(stderr, "Could not close()\n");
   }

   if (close(pipe_2[PIPE_READ]) == -1) {
         fprintf(stderr, "Could not close()\n");
   }

   if (close(pipe_2[PIPE_WRITE]) == -1) {
         fprintf(stderr, "Could not close()\n");
   }

   if (close(pipe_3[PIPE_READ]) == -1) {
         fprintf(stderr, "Could not close()\n");
   }
```

```
   if (close(pipe_3[PIPE_WRITE]) == -1) {
         fprintf(stderr, "Could not close()\n");
   }
   /*printf("exec grep\n");*/
   buff[0][0] = 'g';
   buff[0][1] = 'r';
   buff[0][2] = 'e';
   buff[0][3] = 'p';
   buff[0][4] = '\0';
   /*printf("exec grep\n");*/
   execvp("grep", buff);
   freeBuff(buff);
   exit(1);
}
if (-1 == cp) {
   perror("fork");
}

cp = fork();
if (cp == 0) { /*In child fork, sort */
   freeBuff(buff);
   /*printf("starting sort\n");*/
   /* replace stdin and stdout with pipes */
   if (dup2(pipe_2[PIPE_READ], STDIN_FILENO) == -1) {
         fprintf(stderr, "Could not dup2()\n");
   } if (dup2(pipe_3[PIPE_WRITE], STDOUT_FILENO) == -1) {
         perror("pipe3 write");
   }

   /* the pipes are no longer used */
   if (close(pipe_1[PIPE_READ]) == -1) {
         fprintf(stderr, "Could not close()\n");
   }

   if (close(pipe_1[PIPE_WRITE]) == -1) {
         fprintf(stderr, "Could not close()\n");
   }

   if (close(pipe_2[PIPE_READ]) == -1) {
         fprintf(stderr, "Could not close()\n");
   }

   if (close(pipe_2[PIPE_WRITE]) == -1) {
         fprintf(stderr, "Could not close()\n");
   }

   if (close(pipe_3[PIPE_READ]) == -1) {
         fprintf(stderr, "Could not close()\n");
```

```
    }

    if (close(pipe_3[PIPE_WRITE]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }
    execlp("sort", "sort", (char *) 0);
    exit(1);
}
if (-1 == cp) {
    perror("fork");
}

cp = fork();
if (cp == 0) { /*In child fork, PAGER/less/more */
    freeBuff(buff);
    /*printf("starting pager\n");*/
    pager = getenv("PAGER");

    /* replace stdin with pipe */
    if (dup2(pipe_3[PIPE_READ], STDIN_FILENO) == -1) {
          fprintf(stderr, "Could not dup2()\n");
    }

    /* close the pipes */
    if (close(pipe_1[PIPE_READ]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_1[PIPE_WRITE]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_2[PIPE_READ]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_2[PIPE_WRITE]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_3[PIPE_READ]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }

    if (close(pipe_3[PIPE_WRITE]) == -1) {
          fprintf(stderr, "Could not close()\n");
    }
    /*printf("exec pager\n");*/
```

```
        if (pager != NULL) {
              execlp(pager, pager, (char *) 0);
        }
        execlp("less", "less", (char *) 0);
        execlp("more", "more", (char *) 0);
        fprintf(stderr, "Could not find less/more/PAGER\n");
        exit(1);
    }
    if (-1 == cp) {
        perror("fork");
    }


    /* the parent process does not use the pipes */
    if (close(pipe_1[PIPE_READ]) == -1) {
        fprintf(stderr, "Could not close()\n");
    }
    if (close(pipe_1[PIPE_WRITE]) == -1) {
        fprintf(stderr, "Could not close()\n");
    }
    if (close(pipe_2[PIPE_READ]) == -1) {
        fprintf(stderr, "Could not close()\n");
    }
    if (close(pipe_2[PIPE_WRITE]) == -1) {
        fprintf(stderr, "Could not close()\n");
    }
    if (close(pipe_3[PIPE_READ]) == -1) {
        fprintf(stderr, "Could not close()\n");
    }
    if (close(pipe_3[PIPE_WRITE]) == -1) {
        fprintf(stderr, "Could not close()\n");
    }

#if SIGDET == 1
    if (sighold(SIGCHLD)) perror("sighold");
#endif /* SIGDET == 1 */

    /* wait for the processes to be finished */

    /*printf("wait 1\n");*/
    cp = wait(&status);
    if (-1 == cp) {
        perror("wait");
    }

    /*printf("wait 2\n");*/
    cp = wait(&status);
    if (-1 == cp) {
```

```
        perror("wait");
    }

    /*printf("wait 3\n");*/
    cp = wait(&status);
    if (-1 == cp) {
        perror("wait");
    }

    /*printf("wait 4\n");*/
    cp = wait(&status);
    if (-1 == cp) {
        perror("wait");
    }

#if SIGDET == 1
    if (sigrelse(SIGCHLD)) perror("sigrelse");
#endif /* SIGDET == 1 */

    /*printf("done\n");*/
}

int main()
{
    /* an array to hold strings */
    char *buff[BUFFSIZE];

    /* our built in commands */
    char *cmd_exit = "exit";
    char *cmd_cd = "cd";
    char *cmd_pwd = "pwd";
    char *cmd_checkEnv = "checkEnv";
    char *cmd_amp = "&";

    int i;

    /* the process group to which our children belong */
    pid_t ourpgrp;

    /* register for signals */
    register_signal(SIGINT, SIGINT_handler, 0);
#if SIGDET == 1
    register_signal(SIGCHLD, SIGCHLD_handler, 0);
#endif /* sigdet == 1 */
    /*printf("done\n");*/

    /* set the process group */
    if (setpgid(0, 0) != 0) {
```

```
   fprintf(stderr, "Could not set process group: %s\n", strerror(errno));
}

/* get the process group (will never fail) */
ourpgrp = getpgrp();

/* clear the memory */
memset(buff, 0, BUFFSIZE*sizeof(char*));

while (1) {
   /* free buff for the previous round */
   freeBuff(buff);

   /* and clear the memory */
   memset(buff, 0, BUFFSIZE*sizeof(char*));

   /* read a line, if EOF or error, quit */
   if (readline(buff)) break;

   /* continue if the empty line */
   if (buff[0] == NULL) continue; /* No command entered */

   /* execute built in commands */
   if (strcmp(buff[0], cmd_exit) == 0) break;
   if (strcmp(buff[0], cmd_pwd) == 0) {
         pwd();
         continue;
   }
   if (strcmp(buff[0], cmd_cd) == 0) {
         cd(buff[1]);
         continue;
   }
   if (strcmp(buff[0], cmd_checkEnv) == 0 && buff[1] != NULL) {
         checkEnvArg(buff);
         continue;
   }
   if (strcmp(buff[0], cmd_checkEnv) == 0) {
         checkEnv();
         continue;
   }

   /* it wasn't built in!
    * locate the first & and execute the command in background */
   for (i=0; i < BUFFSIZE; i++) {
         if (buff[i] != NULL && strcmp(buff[i], cmd_amp) == 0) {
               free(buff[i]);
               buff[i] = NULL;
               executeCmd(buff, 1);
```

```
                break;
            } else if (buff[i] == NULL) {
                /* there was no &, execute in foreground */
                executeCmd(buff, 0);
                break;
            }
        }
    }

    /* free buff */
    freeBuff(buff);

    /*printf("exiting main!\n");*/
    /*printf("in main: pid: %d\n", getpid());*/

    /* send SIGTERM to all our children */
    if (killpg(ourpgrp, SIGTERM)) {
        perror("killpg");
        return 1;
    }

    /* bye bye! */
    return 0;
}
```