EECS 678 Project I
Group Members:
Wei Fei (2538810)
Jiahui Wang (2586742)
03/07/15

• Introduction

In this project, we mainly focus on the UNIX system call. The group should implement quash using UNIX system calls. This program could run executable with or without arguments. It can run some single commands and run some command in pipeline. This program also gives some error messages when the directories or commands are not found. For example, we may use “access()” function to search the directory in the environment. This program also allows foreground and background execution, we will use character “&” to distinguish it. We also implement “job” feature, which will display the processes currently running. We can use the kill command to kill certain process. Moreover, we could also set HOME and PATH. Before setting them, we could use “echo” to check the current home directory and path, then use “set” to change the home and path. The program also can handle with “> >> <”, which use to implement I/O redirection.

For implementation, the program mainly have two parts. The first one runs some normal commands, and the second part runs commands with pipelines. In the pipeline part, it can implement all the commands that run in normal part, such as “cd”, “job”, “echo”, “set”...and run foreground and back ground and so on (check the code in lines: 423810). We also do the bonus points, which support multiple pipes in one command, and kill command deliver signals to background processes.

• Implementation and Testing

1. Run executables without arguments (10)

Code:
Normal: line 233-328
Pipe: line 593-696

For implementation, we set the first parameter of the function “***execve(cmd, &p[j], envp)***” as “NULL” since we need to run executables without arguments.

Fore example, when we test the program, we may type “cd” directly, the path will change to the home directory as usual:

```
/home/wfei26/2015Spring/EECS678/Project/quash$cd
/home/wfei26$
```

We can see the directory has already been changed.

2. Run executables with arguments (10)

Code:
Normal: line 233-328
Pipe: line 593-696

The basic characters of running executables with arguments are similar to running witout arguments. It is only depend on the second parameter of “execve()” function.
For example, taward the function: ***execve(cmd, &p[j], envp)***
The first parameter is the name command, the second parameter is a string array. If we do not want to run with arguments, we just need to set the first parameter as “NULL”. We have already done these implementation in “strtok_r()” previously.

For example, when we test the program, we may type the command “cd” or “ls” plus an argument likee “cd quash”, “cd ..” or “ls -l”:

```
[wfei26@1005b-19 quash]$ ./quash
/home/wfei26/2015Spring/EECS678/Project/quash/$ls -l
total 92
-rwxr-xr-x 1 wfei26 undergraduate   115 Mar  5 01:15 Makefile
-rwxr-xr-x 1 wfei26 undergraduate  1081 Mar  5 01:15 README
-rw------- 1 wfei26 undergraduate   496 Mar  7 19:06 a.txt
-rwxr-xr-x 1 wfei26 undergraduate   307 Mar  4 12:45 aaa.out
-rwxr-xr-x 1 wfei26 undergraduate 23383 Mar  7 18:53 quash
-rwxr-xr-x 1 wfei26 undergraduate 21186 Mar  6 02:58 quash.c
-rw-r--r-- 1 wfei26 undergraduate 22528 Mar  7 18:53 quash.o
-rwxr-xr-x 1 wfei26 undergraduate    27 Mar  5 01:17 text_in.sh
/home/wfei26/2015Spring/EECS678/Project/quash/$cd ..
/home/wfei26/2015Spring/EECS678/Project$cd quash
/home/wfei26/2015Spring/EECS678/Project/quash$
```

We see the program runs correctly.

3. set for HOME and PATH work properly (5)

Code:
Normal: 196-208
Pipe: 508-520

When we implement, the most important part is the function setenv(), when we get the home directory or path. These function has two parts, the one is running command with pipeline, the one is without pipe.
For example:

```
if (strcmp(p[j], "set") == 0)
{
        if (strncmp(p[j+1], "PATH", 4) == 0 ) {
                sprintf(environ, "%s", p[j+1]);
                setenv("PATH", environ, 1);
                return ;
        } else if (strncmp(p[j+1], "HOME", 4) == 0 ) {
                sprintf(home, "%s", p[j+1]);
                setenv("HOME", home, 1);
                return ;
        }
        return;
}
```

When we test the program, we may use the key command “echo” or “set” to run this part.
For example, we can see the home path when we catch the path of home directory:

```
/home/wfei26/2015Spring/EECS678/Project/quash/$echo $HOME
/home/wfei26
```

Or the path of current environmental variable:

```
/home/wfei26/2015Spring/EECS678/Project/quash/$echo $PATH
/usr/lib64/qt-3.3/bin:/usr/lib64/ccache:/usr/local/bin:/usr/bin:/bin:/usr/games:/usr/local/sbin:/usr/sbin:/usr/lib64/alliance/bin:/opt/android-sdk/tools:/opt/android-sdk/platform-tools:/opt/android-sdk/ndk:/opt/cadence/IC616/bin:/opt/cadence/ASSURA41/bin:/opt/cadence/MMSIM131/bin:/opt/cadence/PVE121/bin:/usr/libexec/sdcc:/home/wfei26/bin:/usr/lib64/alliance/bin:/opt/android-sdk/tools:/opt/android-sdk/platform-tools:/opt/android-sdk/ndk:/opt/cadence/IC616/bin:/opt/cadence/ASSURA41/bin:/opt/cadence/MMSIM131/bin:/opt/cadence/PVE121/bin:/usr/libexec/sdcc
```

If we reset the PATH, we may see the changes:

```
/home/wfei26/2015Spring/EECS678/Project/quash/$set PATH=/usr/bin:/bin
/home/wfei26/2015Spring/EECS678/Project/quash/$echo $PATH
PATH=/usr/bin:/bin
```

4. exit and quit work properly (5)

Code:

Normal: 173-178
Pipe: 485-490

For implementation, when we read exit or quit command, use the strncmp() function to compare, and we will print “bye” and use the function exit();
Here are sample codes:

```
/* handle with 'exit' */
if ( 0 == strcmp(cmd_exit, p[j]) || 0 == strcmp("quit", p[j]))
{
        printf("bye\n");
        exit(1);
}
```

Testing for this part is so easy, we can just type “exit” to quit the whole program after we have everything done.

```
/home/wfei26/2015Spring/EECS678/Project/quash/$exit
bye
[wfei26@1005b-19 quash]$
```

The program works correctly.

5. cd (with and without arguments) works properly (5)

Code:
Normal: 156-172
Pipe: 468-484

For implementation, when read the “cd” command, we store it in variable p[j], if the p[j+1] is NULL, the the program will go to the home. Otherwise, go to the directory the command given. We use the function chdir().
For example, here are some sample codes for this part:

```
/* handle with 'cd' */
if ( 0 == strcmp(cmd_cd, p[j]) ) {
        if (p[j+1] == NULL) {
                chdir(home);
                getcwd(base_dir, MAX_LEN);
                return;
        } else {
        if (0!=access(p[j+1], 0)) {
                printf("Error! No such file or direction\n");
                return;
        }
        if (0 == opendir(p[j+1])) {
                printf("Error! %s is not a directory.\n", p[j+1]);
                return ;
        }
        chdir(p[j + 1]);
        getcwd(base_dir, MAX_LEN);
        }
return;
}
```

When we test hte program, we may just type some basic command for testing this part.
For example:

```
[wfei26@1005b-19 quash]$ ./quash
/home/wfei26/2015Spring/EECS678/Project/quash/$ls -l >> a.txt
/home/wfei26/2015Spring/EECS678/Project/quash/$cd ..
/home/wfei26/2015Spring/EECS678/Project$ls
EECS678_Project1Report_WeiFei_JiahuiWang.odt quash
/home/wfei26/2015Spring/EECS678/Project$cd quash
/home/wfei26/2015Spring/EECS678/Project/quash$cd aaa.out
Error! aaa.out is not a directory.
/home/wfei26/2015Spring/EECS678/Project/quash$cd
/home/wfei26$
```

6. PATH works properly. Give error messages when the executable is not found (10)

Code:
Normal: 283-298
Pipe: 655-667

For the implementation of this part, we use the function “access()” to search whether the parth of environmental variables exist this command. Give error message if it does not exist.
For example:

```
for (k = 0; k < pathnum; k++)
{
        sprintf(cmd, "%s/%s", envp[k], p[j]);
        if (access(cmd, 0)==0)
        {
                flag = 1;
                break;
        }
}
if (flag == 0)
{
        printf("ERROR: command '%s' not found\n", p[j]);
        exit(1);
}
```

When we test the program, if we type “cd + the file that is not a direction” or “cd + the file that does not exist”, the program will show error message. We also have other error messgeas in different conditions.
For example:

```
/home/wfei26/2015Spring/EECS678/Project/quash/$cd quash
Error! quash is not a directory.
/home/wfei26/2015Spring/EECS678/Project/quash/$cd aaa.out
Error! aaa.out is not a directory.
/home/wfei26/2015Spring/EECS678/Project/quash/$cd adadasas
Error! No such file or direction
```

7. Child processes inherit the environment (5)

Code:
Normal: line 325
Pipe: line 693

We use the function “execve()” to let child processes inherit the envionment.
For example, we have the function with three parameters:

```
execve(cmd, &p[j], envp);/* run the child process */
```

Meanwhile, the third parameter represents the envionmental variable that passed to child process.

8. Allow background/foreground execution (&) (5)

Code:
Normal: line 386-411
Pipe: line 730-811

When the variable “background == 0”, the program run in foreground, otherwise, run in background. When the program runs, the default environment is foreground. The key function is “waitpid()”, if we do not use “waitpid()”, the processes will run in background.
For example, in the codes of part of running foreground, we have:

```
if ( 0 == background )
{
        waitpid(pid, NULL, 0);
        /* recover to stdin/stdout */
        if (fileIO==1) {
                dup2(standardOut, STDOUT_FILENO);
        } else if (fileIO==2) {
                dup2(standardOut, STDOUT_FILENO);
        } else if (fileIO==3) {
                dup2(standardIn, STDIN_FILENO);
        ..................................
}
```

When we test the program, only if we add “&” after command, the process will run in background. For example, on the one hand, if we type “sleep 5”, it runs in foreground, we need to wait for this process five second. On the other hand, if we type “sleep 5 &”, it runs in background, we can continue do other things in foreground.

9. Printing/reporting of background processes, (including the jobs command) (10)

Code:
Normal: line 182-187
Pipe: line 497-502

The most significant function is “showjobs()”, when the program is running, it will record all child process which are running at the same time. If we type “jobs”, the program will execute “showjobs()”

function and output all recorded child process.
For example,we implement “showjobs()” function as loops:

```
for (j = 0; j < pid_topptr; j++ )
{
        /* the status can be found here */
        sprintf(path, "/proc/%d/status", jobs_pid[j]);
        if ( NULL != (fin = fopen(path, "r")) )
        {
                fscanf(fin, "%s%s%s%s%s", str1, str2, str3, str4, status);
                for (i = 0; i < strlen(status) - 2; i++)
                {
                        status[i] = status[i + 1];
                }
                status[i] = 0;
                /* output the jobs info */
                printf("Process %10s with PID %5d :%10s\n", str2, jobs_pid[j], status);
        }
}
```

And in this condition, we run “showjobs()” function:

```
/* handle with 'jobs' */
if ( 0 == strcmp(cmd_jobs, p[j]) )
{
        showjobs();
        return ;
}
```

When we test the program, if we want to see what the current background processes we have, we may type “jobs” to look them.
For example, here are some segments of sample output on Terminal:

```
/home/wfei26/2015Spring/EECS678/Project/quash/$sleep 30 &
[27639] running in background
/home/wfei26/2015Spring/EECS678/Project/quash/$sleep 35 &
[27642] running in background
/home/wfei26/2015Spring/EECS678/Project/quash/$jobs
Process      quash with PID 27581 :   running
Process      sleep with PID 27639 :  sleeping
Process      sleep with PID 27642 :  sleeping
```

10. Allow file redirection (> and <) (5)

Code:
Normal: line 213-275
Pipe: line 528-590
The key functions of implementation are “open(), dup(), dup2(), close()”.
For example,

We can use “<” or “>” to do the file redirection.
For example, in the condition of “<”:

```
if (p[ri+1] == NULL)
```

```
{
        printf("Not enough input arguments\n");
        return ;
}
if (access(p[ri+1], 0)!=0) {
        printf("ERROR: no such file to input (%s)", p[ri+1]);
        return ;
}
fileIO = 3;
fileDescriptor = open(p[ri+1], O_RDONLY, 0600);
standardIn = dup(STDIN_FILENO);
dup2(fileDescriptor, STDIN_FILENO);
close(fileDescriptor);
p[ri] = NULL;
```

When we test our program, if we type “ls -l > aaa.out”, the information of all file in current direction will be written in the file “aaa.out”:

```
total 84
-rwxr-xr-x 1 wfei26 undergraduate   115 Mar  5 01:15 Makefile
-rwxr-xr-x 1 wfei26 undergraduate  1081 Mar  5 01:15 README
-rwxr-xr-x 1 wfei26 undergraduate     0 Mar  4 12:45 aaa.out
-rwxr-xr-x 1 wfei26 undergraduate 23383 Mar  5 22:28 quash
-rwxr-xr-x 1 wfei26 undergraduate 21049 Mar  5 01:17 quash.c
-rw-r--r-- 1 wfei26 undergraduate 22400 Mar  5 22:28 quash.o
-rwxr-xr-x 1 wfei26 undergraduate    27 Mar  5 01:17 text_in.sh
```

On the contrary, if I want to read information in the file “aaa.out”, we may use another symbol “<”: like “wc < aaa.out”, the ourput is:

```
/home/wfei26/2015Spring/EECS678/Project/quash/$wc < aaa.out
  8  65 437
```

11. Allow (1) pipe (|) (10)

Code:
line 420-837 (especially for line 586-718)

In order to implement conviniently, we separate the whole program into two parts, which are normal part and pipeline part. We implement similar functions in pipeline part at the second half of our codes. When we test our codes, we may use pipe to do a lot of functions. For example, we can use pipe to do word counting.
For example:

```
/home/wfei26/2015Spring/EECS678/Project/quash/$ls -l | wc
      8      65     437
```

12. Supports reading commands from prompt and from file (10)

I think it means I need to do bactching processing. The most significant thing is that we need to accept standard inputs.
For example, here is a sample outputs for supporting reading command

First, I create a shell file named “text_in.sh” with four commands:

```
ls
ls -l
ls -l | wc
exit
```

If I type command “” run to run the program with reading commands from prompt/file, the output is:

```
[wfei26@1005d-5 quash]$ ./quash < text_in.sh
Makefile  README  aaa.out  quash  quash.c  quash.o  text_in.sh
total 88
-rwxr-xr-x 1 wfei26 undergraduate   115 Mar  5 01:15 Makefile
-rwxr-xr-x 1 wfei26 undergraduate  1081 Mar  5 01:15 README
-rwxr-xr-x 1 wfei26 undergraduate   307 Mar  4 12:45 aaa.out
-rwxr-xr-x 1 wfei26 undergraduate 23383 Mar  5 22:28 quash
-rwxr-xr-x 1 wfei26 undergraduate 21049 Mar  5 01:17 quash.c
-rw-r--r-- 1 wfei26 undergraduate 22400 Mar  5 22:28 quash.o
-rwxr-xr-x 1 wfei26 undergraduate    27 Mar  5 01:17 text_in.sh
      8      65     437
/home/wfei26/2015Spring/EECS678/Project/quash/$
[wfei26@1005d-5 quash]$
```

Obviously, the result shows us the outputs of all command lines in shell file.

14. Bonus points (you can get bonus points only if you have everything else working (or very close to working))

a. Support multiple pipes in one command. (10)

This part is similar to the single pipe, we may add more parameters in functions, and the key functions are still “pipe(), dup(), dup2()”.

When we test our program, we may use multiple pipes to achieve some functions.

For example:

```
/home/wfei26/2015Spring/EECS678/Project/quash/$cat quash.c | grep int | wc
    109     419    3666
```

b. kill command delivers signals to background processes. The kill command has the format: killSIGNUMJOBID, where SIGNUM is an integer specifying the signal number, and JOBID is an integer that specifies the job that should receive the signal. (5)

When we have some process we do not need in background, we can kill them. We just need to type “kill PID”, the process will be killed.

For example, if I create two process with sleeping 30 and 40 second, I can type “jobs” to show their PIDs, and then kill the one I want to kill:

```
/home/wfei26/2015Spring/EECS678/Project/quash/$sleep 30 &
[27590] running in background
/home/wfei26/2015Spring/EECS678/Project/quash/$sleep 40 &
[27593] running in background
/home/wfei26/2015Spring/EECS678/Project/quash/$jobs
Process      quash with PID 27581 :   running
Process      sleep with PID 27590 :  sleeping
Process     sleep with PID 27593 :  sleeping
/home/wfei26/2015Spring/EECS678/Project/quash/$kill 27593

[27593] PID finished
/home/wfei26/2015Spring/EECS678/Project/quash/$jobs
Process     quash with PID 27581 :   running
Process     sleep with PID 27590 :  sleeping
```

Error Analysis

1. The problem of redirection. When we were testing our codes, the results sometimes showed us a condition that the final outputs did not redirect to the standard outputs.
In order to solve it, we tried to save the standard outputs, and then recover it after the current process.
For example, line 531-547 can show our changing:

```
fileIO = 1;
fileDescriptor = open(p[ri+1], O_CREAT | O_TRUNC | O_WRONLY, 0600);
// We replace de standard output with the appropriate file
standardOut = dup(STDOUT_FILENO);
dup2(fileDescriptor, STDOUT_FILENO); close(fileDescriptor);
p[ri] = NULL;
```

2. After I fix all bugs, when we type “make”, it still shows me some warnings:

```
quash.c: In function ‘process_cmds’:
quash.c:165:20: warning: comparison between pointer and integer [enabled by default]
 if (NULL == opendir(p[j+1])) {
    ^
quash.c: In function ‘process_cmdsp’:
quash.c:477:20: warning: comparison between pointer and integer [enabled by default]
 if (NULL == opendir(p[j+1])) {
     ^
quash.c: At top level:
quash.c:39:7: warning: array ‘env_arr’ assumed to have one element [enabled by default]
 char *env_arr[];
         ^
```

We think it is the problem of comparison. So I try to replace “NULL” by “0”. But it still has the third warning. After we changing a declaration, the last warnings were removed.
We changed “char *env_arr” to “char *env_arr[MAX_LEN]” since we need to initialize the length of the array.

3. Previously, we did not know how to output information after the child process ended. So we google it and learned this point. We add handler in signal function, the problem has been solved.
For example, we did it between line 65-75:

```
void sig_chld(int signo)
{
        pid_t   pid;
        int     stat;

        while((pid = waitpid(-1, &stat, WNOHANG)) > 0) {
        printf("\n[%d] PID finished\n", pid);
   }
   return;
}
```

• Conclusion

This program is quite a shell

Usage:
```
	Build:
		make
	Run:
		./quash
	Clean:
		make clean
```

Functions implemented
1. Run executables without arguments (10)
2. Run executables with arguments (10)
3. set for HOME and PATH work properly (5)
4. exit and quit work properly (5)
5. cd (with and without arguments) works properly (5)
6. PATH works properly. Give error messages when the executable is not found (10)
7. Child processes inherit the environment (5)
8. Allow background/foreground execution (&) (5)
9. Printing/reporting of background processes, (including the jobs command) (10)
10. Allow file redirection (> and <) (5)
11. Allow (1) pipe (|) (10)
12. Supports reading commands from prompt and from file (10)

Bonus:
a. Support multiple pipes in one command. (10)
b. kill command delivers signals to background processes. The kill command has the format: killSIGNUMJOBID, where SIGNUM is an integer specifying the signal number, and JOBID is an integer that specifies the job that should receive the signal. (5)

Test Cases Succeed:
1.
```
	ls
	ls -l
```
2.
```
	echo $PATH
	echo $HOME
	set PATH=/usr/bin:/bin
	echo $PATH
```
3.
```
   cd
	cd ..
	cd quash
	cd adadasas
	cd aaa.out
	quit/exit
```

4.

```
aaaaaa   //the program will give error message with the command does not exist
sleep 10 &
sleep 15 &
jobs
kill 993
        (the 993 is the PID of command 'sleep 15 &')
jobs                //bonus points: kill com
```

5.

```
ls -l | wc
cat quash.c | grep int | wc      //bonus points: mutiple pipes
ls -l > aaa.out
wc < aaa.out
```

6.

```
./quash < text_in.sh     //supports reading commands from prompt and from file
```