Emre Uludağ / 50209 30 March 2019  
Arda Arslan / 31760

**COMP304 Project 1 Report**

**Question 1**

In the first part of the Project, we focused on forking, background processing and running commands with execv function. We forked the processes just below the shouldrun check. We used an auxilary variable for understanding if the process must run background or not. And lastly the main principle for using execv command is concatenating “/bin/” with command given by the user in shelldon and using this argument as the first argument in execv function. We used special functions stated in the Project description in the child process.

Forking the process part:

if (shouldrun)

{

child = fork();

if (child == 0)

{ // child

After checking if a background process is running or not we append “&” symbol to:

if (background == 1)

{

strcat(current\_command, " &");

}

Our general method for running processes is concatenating to pass this argument into execv function.

if (strcmp(args[0], "gcc") == 0)

{

strcpy(path, "/usr/bin/");

}

In the first part also we are quitting the child process if we encounter with the “cd” command and we deal with it in the parent process with chdir() command.

Child process:

else if (strcmp(args[0], "cd") == 0)

{

exit(0);

}

Parent process:

else if (strcmp(args[0], "cd") == 0)

{

chdir(args[1]);

}

**Question 2**

For appending and redirecting check we keep two auxilary variables for control. They are called shouldAppend and shouldRedirect. After checking these variables with if statements, if we must append to the file we use :

freopen(filename, "a+", stdout);

And if we must write to the file from scratch then we use:

freopen(filename, "w+", stdout);

For the history part we keep an array of commands on the top of main method.

For helping the operations with the array we also keep history\_count. We add commands to our array accordingly. After inserting each command to the history array as we print history we use the following function:

if (strcmp(args[0], "history") == 0)

{

if (history\_count == 1)

{

printf("No history data found \n");

break;

}

*int* historyEndingIndex = history\_count - 1;

*int* index = history\_count;

printf("Last commands: \n");

for (*int* i = historyEndingIndex; (i > historyEndingIndex - 10) & (i >= 0); i--) //print at most last 10 elements in the histroy

{

printf("%d %s\n", index, history[i]);

index--;

}

}

We append the commands to the history array as follows:

if (!((args[0][0] == '!') || (args[0][1] == '!')))

{

strcpy(history[history\_count], current\_command);

history\_count++;

}

We also check if history is zero when user demands for history list. Lastly our implemetation for going back in time with “!” sign is as follows:

The part for !n:

if (args[0][1] != '!')

{

*int* length = strlen(args[0]);

*char* subbuff[5];

memcpy(subbuff, &args[0][1], length);

subbuff[length] = '\0';

*int* targetHistory = atoi(subbuff);

strcpy(historyCommand, history[targetHistory - 1]);

isInHistory = 1;

strcat(historyCommand, "\n\0");

// strcpy(history[history\_count], historyCommand);

// Above part is not communicating with the history variable in parent process so we need to use a shared memory section to make it happen.

history\_count++;

*char* path[20];

shouldrun = parseCommand(inputBuffer, args, &background, &shouldRedirect, &shouldAppend, isInHistory, historyCommand);

The part for !!:

strcpy(historyCommand, history[history\_count - 1]);

isInHistory = 1;

strcat(historyCommand, "\n\0");

*char* path[20];

**Question 3**

Codesearch command has 3 usages:

1) Default usage: codesearch “foo”. This command will search for string “foo” inside the files located in the current working directory. However it will not check the files inside the folders that are located in the current working directory.

2) Recursive usage: codesearch -r “foo”. This command will search for string “foo” inside the files located in the current working directory. And it will do this recursively so that all the files located in both the current working directory and the sub-directories under the current work directory will be checked.

3) Targeted usage: codesearch “foo” -f ./include/util.h. This command will search for string “foo” inside the file which is located at the path specified as the fourth argument of the command line.

Each time an occurrence was found, a line like this will be printed in the console:

45: ./foo.c -> void foo(int a, int b); where 45 is the line number, “./foo.c” is the file path and the rest is the line which includes the string that we are searching for.

To implement codesearch, we wrote a generic function:

else if ((strcmp(args[0], "codesearch") == 0) && (args[1] != NULL)) {

if (strcmp(args[1], "-r") == 0) {

*char*\* full\_search\_str = args[2];

*char* search\_str[1024];

if ((full\_search\_str[0] == '\"') && (full\_search\_str[strlen(full\_search\_str)-1] == '\"')) {

*int* length = strlen(full\_search\_str) - 2;

*int* position = 1;

*int* c = 0;

while (c < length) {

search\_str[c] = full\_search\_str[position+c];

c++;

}

search\_str[c] = '\0';

}

else{

strcpy(search\_str, full\_search\_str);

}

codesearch(".", search\_str, true, NULL);

}

else if ((args[2] != NULL) && (strcmp(args[2], "-f") == 0)) {

*char*\* full\_search\_str = args[1];

*char* search\_str[1024];

if ((full\_search\_str[0] == '\"') && (full\_search\_str[strlen(full\_search\_str)-1] == '\"')) {

*int* length = strlen(full\_search\_str) - 2;

*int* position = 1;

*int* c = 0;

while (c < length) {

search\_str[c] = full\_search\_str[position+c];

c++;

}

search\_str[c] = '\0';

}

else{

strcpy(search\_str, full\_search\_str);

}

codesearch(".", search\_str, false, args[3]); //TODO

}

else {

*char*\* full\_search\_str = args[1];

*char* search\_str[1024];

if ((full\_search\_str[0] == '\"') && (full\_search\_str[strlen(full\_search\_str)-1] == '\"')) {

*int* length = strlen(full\_search\_str) - 2;

*int* position = 1;

*int* c = 0;

while (c < length) {

search\_str[c] = full\_search\_str[position+c];

c++;

}

search\_str[c] = '\0';

}

else{

strcpy(search\_str, full\_search\_str);

}

codesearch(".", search\_str, false, NULL);

}

}

In the function provided above, we take a string “name” as the path of the root directory, a string “search\_str” as the string that we are searching for, a bool “is\_recursive” to indicate whether we should check the subdirectories or not and a string “forced\_filename” to use when the codesearch will be done in a certain file. If the “forced\_filename” is not NULL than a targeted search will be performed. If it is NULL and “is\_recursive” bool is False, than a default search will be done. If it is NULL and “is\_recursive” bool is True, than a recursive search will be done.

When setting crontab, we give three mp3 options to the user for listening. We use two files for setting crontab. One of them is a Shell executable file for giving play command with the specified mp3 file. The other one is the crontabfile which we are going to pass execv function with crontab command. crontabFile is the second command.

fpMusic = fopen("/home/user/Desktop/comp304/Assignment2/play.sh", "w");

fprintf(fpMusic, "play /home/user/Desktop/comp304/Assignment2/%s trim 0.0 60",musicFileName);

fclose(fpMusic);

FILE \*fpCrontab;

fpCrontab = fopen("crontabFile", "w");

fprintf(fpCrontab, "%s %s \* \* \* /home/user/Desktop/comp304/Assignment2/play.sh\n",timeArray[1],timeArray[0]);

fclose(fpCrontab);

*char*\* arguments[] = {"crontab","crontabFile",NULL};

execv("/usr/bin/crontab",arguments);

Our special method is countOccurances. It returns how many times you used a specific command since you initiated shelldon

It works as follows: shelldon>countUsages ls

It will print: **Usage of ls is 4 times** for example.

else if (strcmp(args[0], "countUsages") == 0){

*char*\* historyElement = args[1];

*int* count = 0;

for(*size\_t* i = 0; i < history\_count; i++)

{

if(strcmp(history[i],args[1])==0){

count++;

}

}

printf("Usage of %s is %d times\n",args[1],count);

**Question 4**