* Student ID: 18125086

Student name: Nguyen Huu Khang

* Student ID: 18125141

Student name: Dang Phuong Nam

Project OS:

I. Project requirements analysis:

What to do:

*** Design a C program to serve as a shell interface (accept & execute commands)**

*** Features:**

- Creating the child process and executing the command in the child
- Providing a history feature
- Adding support of input and output redirection
- Allowing the parent and child processes to communicate via a pipe

*** Addition:**

- Shell builtins commands: pwd, cd, ls, sort, history, exit
- Not exit on Ctrl C or Ctrl Z (only print out)
- Catch SIGCHLD signal when child processes exit.

II. Pipeline:

1. Preprocessing.
2. Split input command into args
3. Check io redirection
4. Run child process
5. Adding communication between parent and child processes.

III. Preprocessing:

Core:

```
init_shell(cmd);
set_shell_state(args, args2, &iFlag, &oFlag, &pFlag, &bgFlag,
                &doneBuiltins);
```

Init the shell and reset all the variables to default.
We use args to store input params and multiple flags to detect mode.

Other problem:

1. Prevent Ctrl C - Ctrl Z:

Idea: Use signal handle

```
signal( SIGINT, SIG_IGN );
signal( SIGTSTP, SIG_IGN );
```

```
(base) khang@khang-Vostro-3500:~/Downloads/ProjectOS-Shell_Interface (1)$ ./main
osh>^C^C^C^Z^Z
```

There is a bug for this prevention:
Ctrl C can go into execvp and create a new process, preventing exit commands.

```
osh>^C^C^C^C^C^C
osh>exit
osh>
```

Sol: Convert args[0][0] to int and make sure it must larger than 50

```
int c = args[0][0];
if (c<50){
    continue;
}
```

```
(base) khang@khang-Vostro-3500:~/Downloads/ProjectOS-Shell_Interface (1)$ ./main
osh>^C
osh>exit
```

2. Save and load history:

Idea:

- + Save history into history.txt
- + For each input command, compare it with the previous command. If they are different then save into file
- + For "!!" case: copy the previous command to the current command.

```
(base) khang@khang-Vostro-3500:~/Downloads/ProjectOS-Shell_Interface (1)$ ./main
osh>!!
ls > out.txt
preprocessing.h  output1.txt  main  abc.txt  io_redirection.h  history.txt  file_sorting.h  out.txt  out2.txt  main.c  shell_builtins.h  .git  output.txt
 a.txt  main1
```

IV. Split input command into args:

Idea: Split input by " " in args. There are some mode we have to check:

+ Input redirection
+ Output redirection
+ Pipe
+ Background running

Compare and check for each type of flag

```
void process_command(char *cmd, char *args[], int *argsCount, int *iFlag,
                     int *oFlag, int *pFlag, int *bgFlag) {
    strtok(cmd, "\n");
    char *token = strtok(cmd, " ");
    int i = 0;
    args[i] = token;
    // printf("Tokens[%d]: %s \n",i, token);
    while (token != NULL) {
        if (!strcmp(token, "<"))
            *iFlag = i + 1;
        else if (!strcmp(token, ">"))
            *oFlag = i + 1;
        else if (strcmp(token, "|") == 0)
            *pFlag = i;
        else if (strcmp(token, "&") == 0) {
            *bgFlag = i;
            // printf("bgFlagChecked");
        }
        args[i++] = token;
        token = strtok(NULL, " ");
        // printf("Tokens[%d]: %s \n",i, token);
    }
```

V. Shell builtins processing:

Idea: We tried to use switch cases and some other structures to be able to scale more builtins commands but they're not working. So we stick with simple if else.

```
void process_shell_builtins_cmd(char *args[], int *running, int iFlag,
                                int *doneBuiltins) {
    *doneBuiltins = 1;
    if (strcmp(args[0], "history") == 0) {
        getHistory();
    } else if (strcmp(args[0], "pwd") == 0)
        getWorkingDirectory();
    else if (strcmp(args[0], "ls") == 0)
        listSubDirectory();
    else if (strcmp(args[0], "cd") == 0) {
        if (args[1] != NULL)
            if (chdir(args[1]) != 0) perror(args[1]);
    }

    else if (strcmp(args[0], "sort") == 0) {
        if (args[1] != NULL && !iFlag) {
            sortFile(args[1]);
        } else {
            *doneBuiltins = 0;
        }
    } else
        *doneBuiltins = 0;
}
```

For sort command, we use simple interchange sort algorithm:

```
void sort(char **array, int filelinecount)
{
    int i, j;
    char t[1000];

    for(i=1;i<filelinecount;i++)
    {
        for(j=1;j<filelinecount;j++)
        {
            if(strcmp(array[j-1], array[j]) > 0)
            {
                strcpy(t, array[j-1]);
                t[1000] = 0;
                strcpy(array[j-1], array[j]);
                strcpy(array[j], t);
            }
        }
    }
}
```

VI. Execute other commands with execvp and parent-child connection:

There are 4 flags in the program:

```
int iFlag = 0, // if exists "<" in args[]
int oFlag = 0, // if exists ">" in args[]
int pFlag = -1,// if exists "|" in args[] ret the pos in args[]
int bgFlag = 0,// if exists "&" in args[]
```

Flow of process:

1) **if there are no flag, the program will run the**

```
process_shell_builtins_cmd(args, &running, iFlag, &doneBuiltins);
```

2) **if there is one of three flags: bgFlag, iFlag, oFlag -> the exec function will be called.**

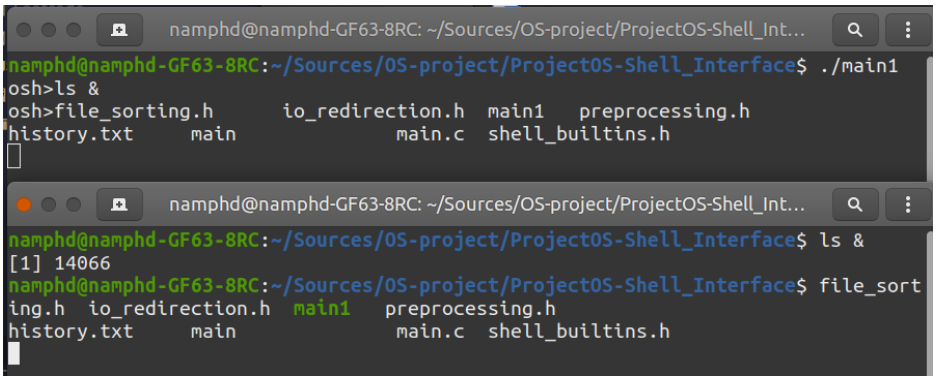
```
void exec(char *args[], int n, int bgFlag, int iFlag, int oFlag) {
   pid_t pid;
   if (bgFlag > 0) args[--n] = NULL; // delete "&" in args[]

   pid = fork();
   switch (pid) {
       case -1: {  // if can not fork
           perror("forkError");
           printf("\nCan't fork!!!");
           return;
       }
       case 0: {
           if (iFlag || oFlag) { // if there exist 1 of 2 I/O flag:
               // printf("I/O process: ");
               if (iFlag > 0) {
                   args[iFlag - 1] = NULL;
                   set_i_mode(args, iFlag);
               }
               if (oFlag > 0) {
                   args[oFlag - 1] = NULL;
                   set_o_mode(args, oFlag);
               }
           }
           if (execvp(args[0], args) < 0) {
               perror("execvpError");
               printf("\nCouldn't execute in command...");
           }
           exit(0);
       }
       default:
           if (bgFlag == 0) { // if there is no bgFlag, the
                              // par and child will run parallel
```

```
                wait(NULL);
            }
            return;
    }
}
```

set_o_mode is setting for output mode.

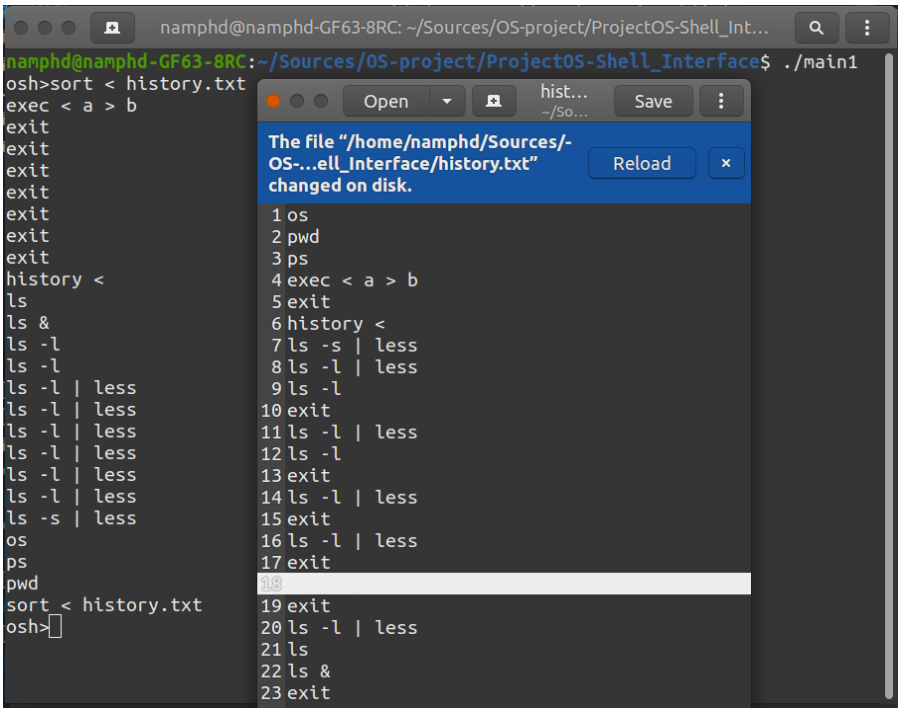
Compare when run: **ls & in my Osh and Terminal:**



Testing Redirection:

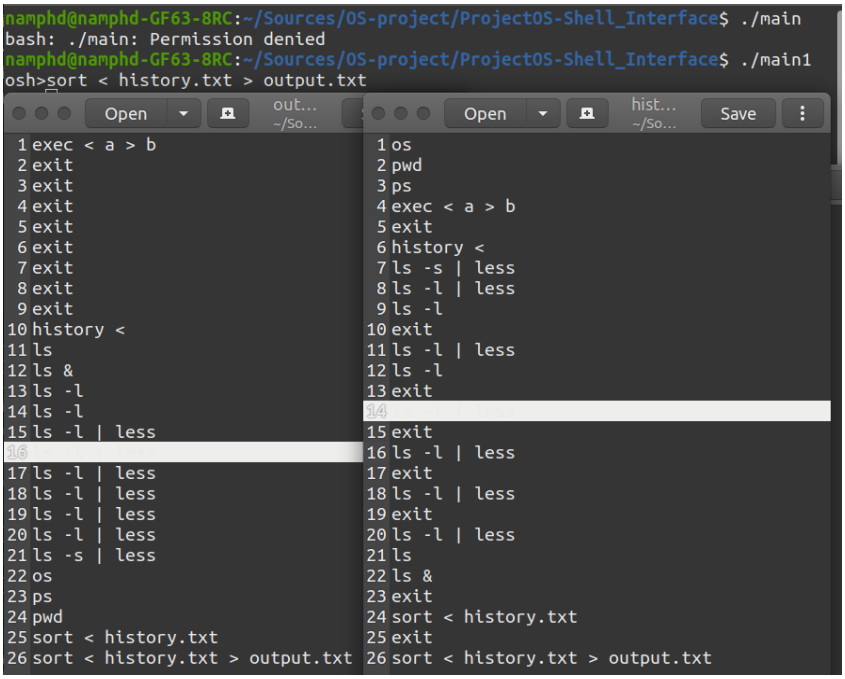
When we execute the cmd: **sort < history.txt:**

Result: the file is sorted and printed to the Osh>.



When we execute the **sort < history.txt > output.txt**

Result: the history.txt is transferred into the sort then print to the another file output.txt



3) In case there is a Pipe:

The **args[]** is splitted into "**cmd1**", "|", "**cmd2**". Because there are only **2 cmds** in this program, we don't need to run a **for-loop** to execute each cmd. Instead, we use pid folk **2 times** and create a **pipe** to create a communication channel among them.

```
void execPipe(char *args[], char *args2[], int bgFlag, int iFlag, int
oFlag) {
   pid_t pid;

   int pipefd[2];  // pipefd[0] refers to the read end of the pipe.
                   // pipefd[1] refers to the write end of the pipe.

   if (pipe(pipefd) == -1) {
       perror("pipeError");
       printf("\nCan't create pipe...");
       return;
   }
   pid = fork();
   switch (pid) {
       case -1: {  // if can't fork
           perror("forkError");
           printf("\nCan't folk...");
           return;
       }
       case 0: {                              // this is child-process
           dup2(pipefd[1], STDOUT_FILENO);    // write into pipefd[1]
           close(pipefd[0]);                  // close unused pipefd[0]
           close(pipefd[1]);                  // close used pipefd[0]
           if (execvp(args[0], args) < 0) { // "ls -l" cmd1
```

```
            perror("execError");
            printf("\nCan't folk...");
        }
        exit(1);
    }

    default: {  // this is parent-process
        pid = fork();
        switch (pid) {
            case -1: {
                perror("forkError");
                printf("\nCan't execute cmd1...");
                return;
            }
            case 0: {
                dup2(pipefd[0], STDIN_FILENO);  // read into
                                                        pipefd[0]
                close(pipefd[1]);               // close unused
                                                        pipefd[1]
                close(pipefd[0]);               // close used
                                                        pipefd[0]

                if (execvp(args2[0], args2) < 0) { // "less" cmd2
                    perror("execError");
                    printf("\nCannot execute cmd2...");
                }
                exit(1);
            }
            default: {
                int status;
                close(pipefd[0]);
                close(pipefd[1]);
                if (bgFlag == 0) waitpid(pid, &status, 0);
                return;
                // status is used to check if there is an Error.
                 // if (WIFSIGNALED(status)) {
                //     printf("Error\n");
                // } else if (WEXITSTATUS(status)) {
                //     printf("Exited Normally\n");
                // }


            }
        }
        break;
```

```
        }
    }
}
```

*Compare with run in **Terminal** and **Osh**:*

```
namphd@namphd-GF63-8RC:~/Sources/OS-project/ProjectOS-Shell_Interface$ ./main1
osh>ls -l | less
osh>
osh>exit
namphd@namphd-GF63-8RC:~/Sources/OS-project/ProjectOS-Shell_Interface$ ls -l | less
```

This is from Osh:

```
total 76
-rw-rw-r-- 1 namphd namphd  1069 Thg 5  14 21:48 file_sorting.h
-rw-rw-r-- 1 namphd namphd    53 Thg 5  15 21:20 history.txt
-rw-rw-r-- 1 namphd namphd   814 Thg 5  15 20:49 io_redirection.h
-rw-rw-r-- 1 namphd namphd 17488 Thg 5  14 21:48 main
-rwxrwxr-x 1 namphd namphd 22848 Thg 5  15 17:58 main1
-rw-rw-r-- 1 namphd namphd  2344 Thg 5  15 19:41 main.c
-rw-rw---- 1 namphd namphd   235 Thg 5  15 20:40 output.txt
-rw-rw-r-- 1 namphd namphd  1864 Thg 5  15 12:09 preprocessing.h
-rw-rw-r-- 1 namphd namphd  5112 Thg 5  15 21:20 shell_builtins.h
(END)
```

This is from Terminal:

```
total 76
-rw-rw-r-- 1 namphd namphd  1069 Thg 5  14 21:48 file_sorting.h
-rw-rw-r-- 1 namphd namphd    59 Thg 5  15 21:24 history.txt
-rw-rw-r-- 1 namphd namphd   814 Thg 5  15 20:49 io_redirection.h
-rw-rw-r-- 1 namphd namphd 17488 Thg 5  14 21:48 main
-rwxrwxr-x 1 namphd namphd 22848 Thg 5  15 17:58 main1
-rw-rw-r-- 1 namphd namphd  2344 Thg 5  15 19:41 main.c
-rw-rw---- 1 namphd namphd   235 Thg 5  15 20:40 output.txt
-rw-rw-r-- 1 namphd namphd  1864 Thg 5  15 12:09 preprocessing.h
-rw-rw-r-- 1 namphd namphd  5112 Thg 5  15 21:20 shell_builtins.h
(END)
```

VII. IO redirection:

We tried to put IO in the main loop in order to use IO for both shell builtins command and others.

```
set_io_mode(args, iFlag ,oFlag, &stdmode); // becaus
process_shell_builtins_cmd(args, &running, iFlag, &
if (!doneBuiltins){
    if (pFlag==-1){
        exec_no_pipe(args, numOfArgs, bgFlag, iFlag)
    }

}
reset_io_mode(args, iFlag, oFlag, &stdmode);
```

The output redirection works fine but the input redirection causes the bugs of not returning to normal input after done.
So we decided to process the shell builts in without IO redirection and use it only in exec other commands.

```
if (iFlag || oFlag) {
    // printf("I/O process: ");
    if (iFlag > 0) {
        args[iFlag - 1] = NULL;
        set_i_mode(args, iFlag);
    }
    if (oFlag > 0) {
        args[oFlag - 1] = NULL;
        set_o_mode(args, oFlag);
    }
```

VIII. Buggs solving:

1. Non executable command:

```
osh>a
00osh>a
00osh>a
00osh>exit
20987osh>exit
20986osh>exit
20985osh>exit
(base) khang@khang-Vostro-3500:~/Code/reproduce_project$
```

Because every command (not builtins) will get in execvp and create a new process and loop forever and prevent exit commands (like the Ctrl C, Ctrl Z above).
We tried to solve by:

- WNOHANG flag -> failed
- Create timeout options

```c
sigset_t sigmask;
sigemptyset(&sigmask);
sigaddset(&sigmask, SIGCHLD);
sigprocmask(SIG_BLOCK, &sigmask, NULL);
if (pid == 0) {    // Child never returns
    for( ; ; );
}
if (sigtimedwait(&sigmask, NULL, &((struct timespec){2, 0})) < 0) {
if (errno == EAGAIN) {
        printf("%s is not recognized\n", args[0]);
        kill(pid, SIGINT);
        return;
}

waitpid(pid,0,0);
```

And solved:

```
(base) khang@khang-Vostro-3500:~/Downloads/ProjectOS-Shell_Interface (1)$ ./main
osh>a
a is not recognized
osh>pwd
/home/khang/Downloads/ProjectOS-Shell_Interface (1)
osh>
```