

Introduction

The objective of this work is to show us the various ways processes can be issued and handled in the terminal. In order to attain this objective, we design a C program to serve as a shell that accepts user commands and then executes each command in separate processes. This work also stretches on the use of some key functions such as `fork()`, `exec()`, `wait()`, `dup2()` and `pipe()`. After many lines of code and some problem solving along the way, we resulted with a program that runs similarly to a Unix shell. Some challenges that we face were mainly due to the fact that our child process was not returning properly as well as understanding how to manipulate the argument arrays.

Methodology

We accomplished this project by using a divide and conquer method. Since building the shell was an enormous task by itself, with the help of the given directions, we started small with a simpler program and built on it with each new feature. First, we wrote a C program that just takes user input and yields the expected results. To accomplish that step we took user inputs and tokenized them in order to extract the arguments. Then, a child process was created and it executes the command entered by the user using the `execvp` function. This means that those arguments were passed to the `execvp()` function to get the desired results (Figure 1). The method of tokenizing was utilized for all features in the project where user input was considered. Also, there were other smaller features like the exit command.

Next, we created a history feature that allows the user to run the command that he/she previously ran. In order to complete this task, we saved the previous command given by the user into a separate variable. The command was tested to see if it contained “!!” and if so, the saved, previous command was executed using the `execvp()` function (Figure 2). This feature also takes care of the case where there are no commands in the history.

After that, we created a feature that allows the user to redirect input and output. To do that, we parsed the user command to verify if there was a “>”(output) or a

"<"(input). For example, if the user entered an ">" in the command, there would be a process to write standard output to the said file. A big part of this feature was the dup2() function, which took in two integers and allowed for the duplication of file descriptors, including standard output and standard input. Therefore, based on the result issued from our parsing process, we opened a file and used it as an input to read or as an output to write using the execvp() function to execute that command (Figure 3).

Last but not least, we implemented communication via piping. To do so, we had to pass the output issued from a command as an input to another command. To accomplish this task, we first verified if the command entered contained a pipe (|). Upon successful verification, a pipe was created via the pipe() function. Much data manipulation was done to get the arguments on either sides of the pipe into their own separate argument arrays so they would not be confused. After that we executed the command located before "|" and wrote it's result to the pipe. Closing the pipe was a very important step. Next, we used the output in the pipe as input for the command located after "|" (Figure 4).

Results

```

osh>ps -al
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 T   1000  2713  2568  0  80   0 -  1088 signal pts/17    00:00:00 a.out
1 T   1000  2714  2713  0  80   0 -  1088 signal pts/17    00:00:00 a.out
0 T   1000  2746  2568  0  80   0 -  1088 signal pts/17    00:00:00 a.out
0 T   1000  2804  2568  0  80   0 -  1088 signal pts/17    00:00:00 a.out
0 S   1000  3554  3439  0  80   0 -  1088 wait  pts/2     00:00:00 a.out
0 R   1000  3555  3554  0  80   0 -  7228 -      pts/2     00:00:00 ps
osh>ls -l
total 32
-rwxrwxr-x 1 osc osc 13776 Feb 20 18:34 a.out
-rw-rw-r-- 1 osc osc   37 Feb 17 14:26 in.txt
-rw-rw-r-- 1 osc osc  7521 Feb 20 18:34 main.c
-rw-rw-r-- 1 osc osc   44 Feb 20 09:42 out.txt
osh>

```

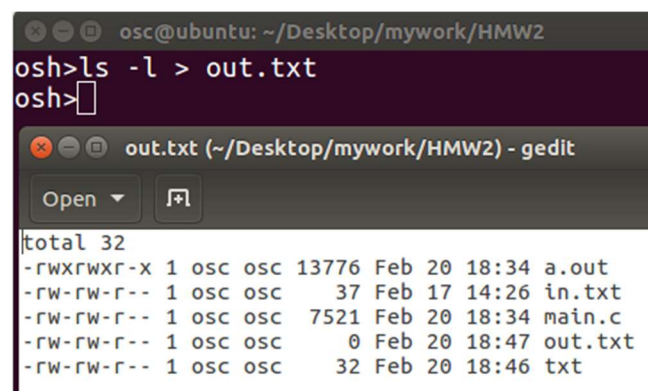
The above figure illustrates the result issued from executing a regular command in a child process.

```

osh>!!
No commands in history.
osh>ls -l
total 32
-rwxrwxr-x 1 osc osc 13776 Feb 20 18:34 a.out
-rw-rw-r-- 1 osc osc   37 Feb 17 14:26 in.txt
-rw-rw-r-- 1 osc osc  7521 Feb 20 18:34 main.c
-rw-rw-r-- 1 osc osc   44 Feb 20 09:42 out.txt
osh>!!
total 32
-rwxrwxr-x 1 osc osc 13776 Feb 20 18:34 a.out
-rw-rw-r-- 1 osc osc   37 Feb 17 14:26 in.txt
-rw-rw-r-- 1 osc osc  7521 Feb 20 18:34 main.c
-rw-rw-r-- 1 osc osc   44 Feb 20 09:42 out.txt

```

The above figure illustrates the result issued from using the history feature. Notice that when there is no previous command entered by the user, “No commands in history.” is echoed to the terminal.



```

osh>ls -l > out.txt
osh>

```

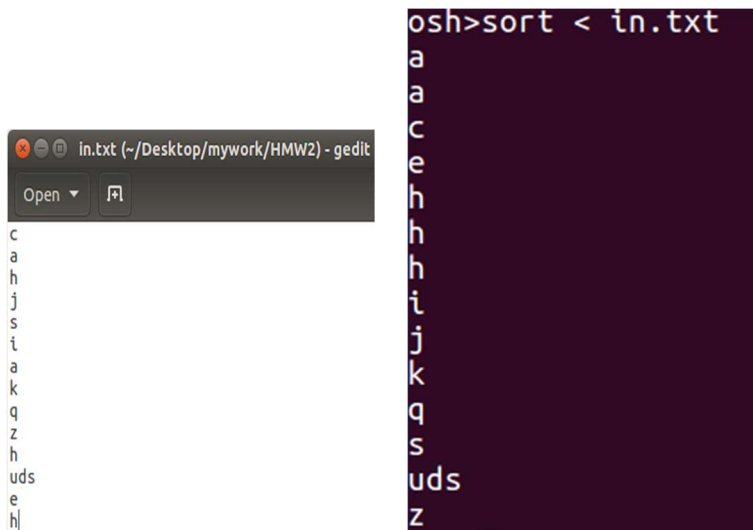
The terminal window shows the command `osh>ls -l > out.txt` being executed. Below it, a file editor window titled `out.txt (~/Desktop/mywork/HMW2) - gedit` is open, displaying the output of the command:

```

total 32
-rwxrwxr-x 1 osc osc 13776 Feb 20 18:34 a.out
-rw-rw-r-- 1 osc osc   37 Feb 17 14:26 in.txt
-rw-rw-r-- 1 osc osc  7521 Feb 20 18:34 main.c
-rw-rw-r-- 1 osc osc   44 Feb 20 18:47 out.txt
-rw-rw-r-- 1 osc osc   32 Feb 20 18:46 txt

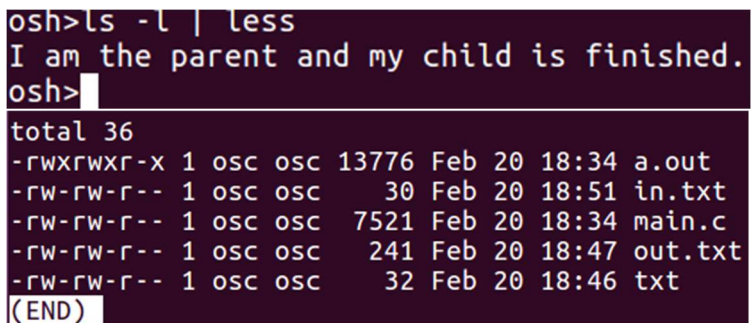
```

The above figure illustrates the result issued from redirecting output.



```
osh>sort < in.txt
a
a
c
e
h
h
h
i
j
k
q
s
uds
z
```

The first figure shows the input file fed to the command that supports input redirecting and the second figure shows the output issued from using redirecting input.



```
osh>ls -l | less
I am the parent and my child is finished.
osh>
total 36
-rwxrwxr-x 1 osc osc 13776 Feb 20 18:34 a.out
-rw-rw-r-- 1 osc osc 30 Feb 20 18:51 in.txt
-rw-rw-r-- 1 osc osc 7521 Feb 20 18:34 main.c
-rw-rw-r-- 1 osc osc 241 Feb 20 18:47 out.txt
-rw-rw-r-- 1 osc osc 32 Feb 20 18:46 txt
(END)
```

The above figure illustrates the result issued from using communication via a Pipe.

An analysis of the results includes the fact that there was more difficulty with the last piping part. The printf statements for debugging worked well for actually seeing what was going on with the code. The features work in terms of the various parts - executing a command, providing a history feature, redirecting input and output and communication via a pipe.

Summary

Overall, this was a decent project in which students learned to create child processes using the `fork()` function. Also, students learned how to use the `exec()` function to have a process execute a command line command. Another important lesson was the `dup2()` function as it was very useful in redirection. Finally, the `pipe()` function was useful for communication via a pipe. If more time was given, improvements would include making code more modular.

Appendix

```
// REGULAR COMMAND IN SHELL

// Get the token which is the first characters before the space or newline
// ie. space or newline is the delimiter in the tokenizer
token = strtok(command, " \n");

// While the token is not null, set the arguments to the token
// and then get another token and finally increment the counter
while(token != NULL)
{
    args[i] = token;
    token = strtok(NULL, " \n");
    i++;
}
// Assign last argument to NULL
args[i] = NULL;

//After reading user input, the steps are:
//(1) fork a child process using fork()
pid = fork();

// If the pid is less than 0, then the fork failed
if(pid < 0)
{
    printf("Fork failed\n");
    return 1;
} else if(pid == 0) {
    // CHILD EXECUTES
    //(2) the child process will invoke execvp()

    if((execvp(args[0], args)) < 0)
    {
        printf("The child did not execute correctly\n");
        return 1;
    }
} else {
    // PARENT EXECUTES
    //(3) if the last arg is &, parent will not wait()

    close(mypipe[1]);
    close(mypipe[0]);
    if(!ampPresent) {
        wait(NULL);
    }
}
```

Figure 1: Executing Command in Child Process.

```

// History command - PART 2
else if(command[0] == '!' && command[1] == '!')
{
    // Strcmp returns 0 if it is the same
    if(!strcmp(theActualPrevCommand, " "))
    {
        printf("No commands in history.\n");
    } else {
        // Tokenize according to space or newline
        token2 = strtok(theActualPrevCommand, " \n");

        // While there is still something left in the string,
        // keep tokenizing
        while(token2 != NULL)
        {
            args[i] = token2;
            token2 = strtok(NULL, " \n");
            i++;
        }
        // Assign last argument to NULL
        args[i] = token2;

        // Create child process
        pid = fork();
        if(pid < 0)
        {
            printf("Fork failed\n");
            return 1;
        } else if(pid == 0)
        {
            // CHILD EXECUTES
            if((execvp(args[0], args)) < 0)
            {
                printf("The child did not execute correctly\n");
                return 1;
            }
        } else {
            // PARENT EXECUTES
            if(!ampPresent) {
                wait(NULL);
            }
        }
    }
}

```

Figure 2: Creating History Feature

```

// Redirection - PART 3
else if(strchr(command, '<') || strchr(command, '>'))
{
    token3 = strtok(command, " \n");

    // While the token is not null, set the arguments to the token
    // and then get another token and finally increment the counter
    while(token3 != NULL)
    {
        args[i] = token3;
        token3 = strtok(NULL, " \n");
        if((strchr(args[i], '<') || strchr(args[i], '>')) || nullifyCopyArgs)
        {
            nullifyCopyArgs = true;
            copyArgs[i] = NULL;
        }else {
            copyArgs[i] = args[i];
        }
        i++;
    }
    // Assign last argument to NULL
    args[i] = token3;
    copyArgs[i] = token3;

    //(1) fork a child process using fork()
    pid = fork();

    if(pid < 0)
    {
        printf("Fork Failed\n");
        return 1;
    }
    else if(pid == 0){
        // CHILD EXECUTES

        // Use the command before it was altered
        // which is prevCommand
        if(strchr(prevCommand, '>'))
        {
            FILE* writeToFile = fopen(args[--i], "w");
            int fd = fileno(writeToFile);
            if(writeToFile < 0)
            {
                printf("ERROR in opening file\n");
                return 1;
            }
            dup2(fd, 1);
            if((execvp(copyArgs[0], copyArgs)) < 0)
            {
                printf("The child did not execute correctly\n");
                return 1;
            }
            close(fd);
            fclose(writeToFile);

            return 0;
        } else if(strchr(prevCommand, '<')) {
            FILE* readFromFile = fopen(args[--i], "r");
            int readFd = fileno(readFromFile);
            if(readFromFile < 0)
            {
                printf("ERROR in opening file\n");
                return 1;
            }
            dup2(readFd, 0);

            if((execvp(copyArgs[0], copyArgs)) < 0)
            {
                printf("The child did not execute correctly\n");
                return 1;
            }
            close(readFd);
            fclose(readFromFile);

            return 0;
        }
    }
}

```

Figure 3: Redirecting Input and Output

```

if(strchr(command, '|'))
{
    if(pipe(mypipe))
    {
        printf("Pipe creation failed\n");
        return 1;
    }

    token4 = strtok(command, " \\n");
    while(token4 != NULL)
    {
        args[i] = token4;
        token4 = strtok(NULL, " \\n");
        if((strchr(args[i], '|')) || nullifyCopyArgs)
        {
            nullifyCopyArgs = true;
            copyArgs[i] = NULL;
        }else {
            copyArgs[i] = args[i];
            indexOfPipeInArgs = i;
        }
        i++;
    }
    // Assign argument to NULL
    args[i] = token4;
    copyArgs[i] = token4;

    //int count = i - 1;
    int count = indexOfPipeInArgs + 2;
    int j;
    // Continue while the args array has items left
    for(j = 0; count < (MAX_LINE/2 + 1); j++)
    {
        if(args[count] == NULL || nullifyCopyArgs2)
        {
            nullifyCopyArgs2 = true;
            copyArgs2[j] = NULL;
        }else {
            copyArgs2[j] = args[count];
        }
        count++;
    }
}

```

Figure 4: Communication via Pipe

Contributor: Kefil Tonouewa & Ravyn Lynn Dickinson