시스템 프로그래밍 과제 보고서

과제4: “mysh 작성”

32181854 박준영

1. 구현 설명

sys32181854@embedded: ~/myshell/complete

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>

void tokenize(char* string,char* tokv[]) //인자로 토큰화
{
        int i = 1;
        tokv[0] = strtok(string," ");
        while(1) {
                tokv[i] = strtok(NULL," ");
                if(tokv[i] == NULL) return;
                i++;
                }
}
int back_check(char* m_argv[]) //&여부를 판단해 bool로 바꾸고 바뀐 argv를 만듦
{
        int i;
        for(i = 0; m_argv[i] != NULL; i++)
        {
                if(strcmp(m_argv[i],"&") == 0) {
                        m_argv[i] = NULL;
                        return 1;
                }
        }
        return 0;
}

void redirect(char* m_argv[])
{
        int i = 0;
        int check = 0;  //redirection check
        int fd;
        for(i; m_argv[i] != NULL; i++)
        {
                if(strcmp(m_argv[i],">") == 0)
                {
                        check = 1;
                        break;
                }
        }
        if(check == 0) return;  //do not redirect

        fd = open(m_argv[i+1],O_RDWR | O_CREAT, 0641);
        if(!fd) {       //open failure
                printf("cannot open file\n");
                return;
--More--(47%)
```

1) tokenize 함수

-사용자의 input값을 받아 토큰화 합니다.

2) back_check 함수

-사용자가 백그라운드 프로세싱을 명령했는지 확인합니다.

-argv 함수 내에 &문자가 존재하는지 확인하고 있다면 해당 배열을 null로 채운 후 backg 플래그를 리턴합니다.

(background: 1, no: 0)

3) redirect 함수

-argv 함수 내에 〉 문자가 있다면 바로 다음 인수인 파일명으로 open합니다.

-dup2로 open된 파일의 fd가 표준출력 fd가 되게 합니다. (복제)

-파일 오픈이 실패하거나 dup이 실패한 경우 함수가 종료됩니다.

-마지막으로 execvp에는 〉 이전의 인자들(명령)만 들어가야하므로 〉와 파일명 인자는 null로 채웁니다.

sys32181854@embedded: ~/myshell/complete

```
        }
        if(dup2(fd,1) < 0) {    //duplicate failure
                printf("dup fail\n");
                return;
        }
        m_argv[i] = 0;
        m_argv[i+1] = 0;
}

int main() {
        char str[500];
        char* m_argv[1000];
        int backg = 0;  //background processing flag
        int pid_fork;

        printf("welcome to my shell <:3)~ I'm jerry\n");

        while(1)
        {
                fflush(stdout);
                fflush(stdin);

                printf(">> ");
                fgets(str,sizeof(str),stdin);
                if(strcmp(str,"exit\n") == 0) return 0;
                str[strlen(str)-1] = 0;

                tokenize(str,m_argv);   //tokenize

                backg = back_check(m_argv);     //background processing flag set

                if((pid_fork = fork()) < 0 ) {  //fork failure
                        printf("fork failure\n");
                        exit(1);
                        }
                if(pid_fork == 0)       //child process
                {
                        if(backg == 1) sleep(1); //background processing

                        redirect(m_argv);       //redirection

                        if(execvp(m_argv[0],m_argv))
                        {
                                fprintf(stderr,"program execution error : %s\n", strerror(errno));
                                return 1;
                        }
                        exit(1);
                }
                else //parent process
--More--(97%)
                    else //parent process
                    {
                            if(backg != 1) wait();

                    }
                    //parent exit
            }
    }
    sys32181854@embedded:~/myshell/complete$ whoami
    sys32181854
    sys32181854@embedded:~/myshell/complete$ date
    2020. 10. 30. (금) 01:57:38 KST
    sys32181854@embedded:~/myshell/complete$
```

〈main함수 수행 과정〉

1) main함수 시작 즉시 인사말 print (1회)

2) while문 시작. 입출력 버퍼에 문자열이 남아 다음 반복에서 출력되는 일을 방지하기 위해 fflush 수행

3) '〉〉 ' command interpreter의 입력 부분

4) white space를 기준으로 토큰화를 진행하기 때문에 fgets함수 사용. 마지막에 입력되는 '\n'를 지워줌

5) string을 argv로 토큰화

6) back_check 함수 호출

(리다이렉션 이전에 처리해야함. otherwise, 마지막 &문자가 제거되지 않은 상태로 redirect함수로 들어감)

7) fork() 수행. error handling 코드 수행

8) backg 플래그 값에 따라 다르게 동작

8-1) backg = 1 (background processing)

-부모 프로세스가 먼저 실행 → 자식프로세스를 wait하지 않고 프로세스 종료 → 고아가 된 자식 프로세스는 init프로세스의 자식 프로세스가 되어 exit_status를 return

부모프로세스는 종료됐으므로 자식 프로세스가 프로세싱 하는 동안 다시 쉘이 동작. '〉〉 '에 명령어 input 가능

-자식 프로세스가 먼저 실행 → 바로 실행하지 않고 sleep. 부모프로세스가 종료된 후 고아프로세스가 됨. 이 후는 위와 동일

8-2) backg = 0

-백그라운드 프로세싱이 아니므로 보통의 경우와 동일하게 synchronization

9) 자식 프로세스는 execvp에 파일명, 인자를 받아 실행

execvp의 return 값이 존재한다면 실행에 실패한 것이므로 종료

2. 수행 결과

(1) ls, ps, gcc 등의 명령어 구현

```
sys32181854@embedded:~/myshell/complete$ ./jerryshell
welcome to my shell <:3)~ I'm jerry
>> ls
jerryshell  jerryshell.c  jerryshell_developer.c
>> ls -l
total 20
-rwxrwxr-x 1 sys32181854 sys32181854 10252 10월  30 01:27 jerryshell
-rw-rw-r-- 1 sys32181854 sys32181854  1902 10월  30 01:18 jerryshell.c
-rw-rw-r-- 1 sys32181854 sys32181854  1992 10월  30 01:11 jerryshell_developer.c
>> ps
  PID TTY          TIME CMD
11464 pts/23   00:00:00 bash
27328 pts/23   00:00:00 jerryshell
27545 pts/23   00:00:00 ps
>> vi hello.c
>> ls
hello.c  jerryshell  jerryshell.c  jerryshell_developer.c
>> gcc -o hello hello.c
>> ls
hello  hello.c  jerryshell  jerryshell.c  jerryshell_developer.c
>> cat hello.c
#include <stdio.h>

int main() {
        printf("hello! this is jerry's shell! i'm cute mouse jerry. <:3)~~\n");
        printf("nice to meet you!\n");
}
>> ./hello
hello! this is jerry's shell! i'm cute mouse jerry. <:3)~~
nice to meet you!
>> whoami
sys32181854
>> date
2020. 10. 30. (금 ) 01:46:19 KST
```

(2) 리다이렉션 구현

```
>> cat hello.c > hello_jerry.c
>> ls
hello  hello.c  hello_jerry.c  jerryshell  jerryshell.c  jerryshell_developer.c
>> cat hello_jerry.c
#include <stdio.h>

int main() {
        printf("hello! this is jerry's shell! i'm cute mouse jerry. <:3)~~\n");
        printf("nice to meet you!\n");
}
>> whoami
sys32181854
>> date
2020. 10. 30. (금 ) 01:47:49 KST
>> ps
  PID TTY          TIME CMD
11464 pts/23   00:00:00 bash
27328 pts/23   00:00:00 jerryshell
32764 pts/23   00:00:00 ps
>>
```

(3) 백그라운드 프로세싱 구현

```
>> ps
  PID TTY          TIME CMD
11464 pts/23   00:00:00 bash
27328 pts/23   00:00:00 jerryshell
32764 pts/23   00:00:00 ps
>> ls -l &
>> ps
  PID TTY          TIME CMD
  660 pts/23   00:00:00 jerryshell
  665 pts/23   00:00:00 ps
11464 pts/23   00:00:00 bash
27328 pts/23   00:00:00 jerryshell
>> total 36
-rwxrwxr-x 1 sys32181854 sys32181854  4764 10월  30 01:45 hello
-rw-rw-r-- 1 sys32181854 sys32181854   140 10월  30 01:44 hello.c
-rw-r----x 1 sys32181854 sys32181854   140 10월  30 01:47 hello_jerry.c
-rwxrwxr-x 1 sys32181854 sys32181854 10252 10월  30 01:27 jerryshell
-rw-rw-r-- 1 sys32181854 sys32181854  1902 10월  30 01:18 jerryshell.c
-rw-rw-r-- 1 sys32181854 sys32181854  1992 10월  30 01:11 jerryshell_developer.c

>> ps
>>   PID TTY          TIME CMD
  746 pts/23   00:00:00 ps
11464 pts/23   00:00:00 bash
27328 pts/23   00:00:00 jerryshell
```

(4) mysh에서 gdb 실행

```
sys32181854@embedded:~/myshell/complete$ ./jerryshell
welcome to my shell <:3)~ I'm jerry
>> gdb hello
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/1-class/sys32181854/myshell/complete/hello
hello! this is jerry's shell! i'm cute mouse jerry. <:3)~~
nice to meet you!
[Inferior 1 (process 6409) exited with code 022]
(gdb)
```

(5) mysh를 디버깅

sys32181854@embedded: ~/myshell/complete

```
sys32181854@embedded:~/myshell/complete$ gdb jerryshell
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from jerryshell...done.
(gdb) run
Starting program: /home/1-class/sys32181854/myshell/complete/jerryshell
welcome to my shell <:3)~ I'm jerry
>> ls
hello  hello.c  hello_jerry.c  jerryshell  jerryshell.c  jerryshell_developer.c
>> ls -l
total 36
-rwxrwxr-x 1 sys32181854 sys32181854  4764 10월  30 01:45 hello
-rw-rw-r-- 1 sys32181854 sys32181854   140 10월  30 01:44 hello.c
-rw-r----x 1 sys32181854 sys32181854   140 10월  30 01:47 hello_jerry.c
-rwxrwxr-x 1 sys32181854 sys32181854 10252 10월  30 01:27 jerryshell
-rw-rw-r-- 1 sys32181854 sys32181854  1902 10월  30 01:18 jerryshell.c
-rw-rw-r-- 1 sys32181854 sys32181854  1992 10월  30 01:11 jerryshell_developer.c
```

sys32181854@embedded: ~/myshell/complete

```
[Inferior 1 (process 1897) exited normally]
(gdb) break 1
Breakpoint 1 at 0x8048748: file jerryshell.c, line 1.
(gdb) run
Starting program: /home/1-class/sys32181854/myshell/complete/jerryshell
welcome to my shell <:3)~ I'm jerry
>> cat hello.c > hello_jerry.c

Breakpoint 1, tokenize (string=0xffffd260 "cat hello.c > hello_jerry.c", tokv=0xffffc2c0) at jerryshell.c:10
10              int i = 1;
(gdb) n
11              tokv[0] = strtok(string," ");
(gdb) n
13                      tokv[i] = strtok(NULL," ");
(gdb) n
14                      if(tokv[i] == NULL) return;
(gdb) n
15                      i++;
(gdb) n
13                      tokv[i] = strtok(NULL," ");
(gdb) n
14                      if(tokv[i] == NULL) return;
(gdb) n
15                      i++;
(gdb) n
13                      tokv[i] = strtok(NULL," ");
(gdb) n
14                      if(tokv[i] == NULL) return;
(gdb) n
15                      i++;
(gdb) n
13                      tokv[i] = strtok(NULL," ");
(gdb) n
14                      if(tokv[i] == NULL) return;
(gdb) n
17      }
(gdb) n
main () at jerryshell.c:79
79                      backg = back_check(m_argv);     //background processing flag set
(gdb) n
81                      if((pid_fork = fork()) < 0 ) {  //fork failure
(gdb) n
85                      if(pid_fork == 0)       //child process
(gdb) n
100                             if(backg != 1) wait();
(gdb) n
69                      fflush(stdout);
(gdb) n
70                      fflush(stdin);
```

```
sys32181854@embedded: ~/myshell/complete
73                      fgets(str,sizeof(str),stdin);
(gdb) n
>> n
74                      if(strcmp(str,"exit\n") == 0) return 0;
(gdb) n
75                      str[strlen(str)-1] = 0;
(gdb) n
77                      tokenize(str,m_argv);   //tokenize
(gdb) n

Breakpoint 1, tokenize (string=0xffffd260 "n", tokv=0xffffc2c0) at jerryshell.c:10
10              int i = 1;
(gdb) n
11              tokv[0] = strtok(string," ");
(gdb) n
13                      tokv[i] = strtok(NULL," ");
(gdb) n
14                      if(tokv[i] == NULL) return;
(gdb) n
17      }
(gdb) n
main () at jerryshell.c:79
79                      backg = back_check(m_argv);     //background processing flag set
(gdb) n
81                      if((pid_fork = fork()) < 0 ) {  //fork failure
(gdb) n
program execution error : No such file or directory
85                      if(pid_fork == 0)       //child process
(gdb) n
100                             if(backg != 1) wait();
(gdb) n
69                      fflush(stdout);
(gdb) n
70                      fflush(stdin);
(gdb) n
72                      printf(">> ");
(gdb) n
73                      fgets(str,sizeof(str),stdin);
(gdb) n
>> n
74                      if(strcmp(str,"exit\n") == 0) return 0;
(gdb) n
75                      str[strlen(str)-1] = 0;
(gdb) n
77                      tokenize(str,m_argv);   //tokenize
(gdb) n

Breakpoint 1, tokenize (string=0xffffd260 "n", tokv=0xffffc2c0) at jerryshell.c:10
10              int i = 1;
```

3. discussion

수업을 들을 때는 shell을 구현하는 것인 논리적으로 그리 어려운지 의문을 품었었다. 왜냐하면 단순히 fork와 execvp 만을 구현하는 것이라면 전혀 어려울 것 없는 논리구조이기 때문이다. 또한 시험기간동안 하루정도 pipe에 빠져서 IPC 프로그래밍 강의까지 찾아보고 나니(지금 생각해보니 원리가 엄청 궁금했다 보다...) 전혀 어렵지 않을 것이라고 착각했다. 그러나 역시 생각으로만 논리를 그리는 것과 직접 코딩으로 작성하는 것은 완전히 다르다는 것을 다시 한 번 깨달았다.

교수님께서 보여주신 예제의 쉘에 준하는 멋진 과제물을 만들어보겠다고 포부 넘치게 jerryshell의 틀을 잡았다. 하지만 첫 컴파일에서 나를 반겼던 것은 넘쳐나는 syntax error였다. 또한 gets 함수를 쓰니 deprecated한 함수이므로 사용을 자제하라는 경고를 받았다. fgets함수의 사용법을 알아보며 무엇이 문제인지 간단히 찾아보았다.

```
sys32181854@embedded: ~/myshell
sys32181854@embedded:~/myshell$ vi jerryshell.c
sys32181854@embedded:~/myshell$ cp jerryshell.c test1.c
sys32181854@embedded:~/myshell$ ls
jerryshell.c  test1.c
sys32181854@embedded:~/myshell$ gcc -g test1.c
test1.c: In function `tokenize':
test1.c:9: error: syntax error before '=' token
test1.c:11: error: syntax error before ')' token
test1.c:13: error: `tok' undeclared (first use in this function)
test1.c:13: error: (Each undeclared identifier is reported only once
test1.c:13: error: for each function it appears in.)
test1.c:13: error: `i' undeclared (first use in this function)
test1.c: At top level:
test1.c:16: error: syntax error before '}' token
test1.c: In function `main':
test1.c:21: error: array size missing in `m_argv'
test1.c:30: warning: `gets' is deprecated (declared at /usr/include/stdio.h:638)
test1.c:32: error: syntax error before "if"
test1.c: At top level:
test1.c:44: error: syntax error before '}' token
sys32181854@embedded:~/myshell$
```

그림: syntax error

버퍼 오버플로

위키백과, 우리 모두의 백과사전.

다른 뜻에 대해서는 오버플로 문서를 참조하십시오.

버퍼 오버플로(영어: buffer overflow) 또는 **버퍼 오버런**(buffer overrun)은 메모리를 다루는 데에 오류가 발생하여 잘못된 동작을 하는 프로그램 취약점이다. 컴퓨터 보안과 프로그래밍에서는 프로세스가 데이터를 버퍼에 저장할 때 프로그래머가 지정한 곳 바깥에 저장하는 것을 의미한다. 벗어난 데이터는 인접 메모리를 덮어 쓰게 되며 이때 다른 데이터가 포함되어 있을 수도 있는데, 손상을 받을 수 있는 데이터는 프로그램 변수와 프로그램 흐름 제어 데이터도 포함된다. 이로 인해 잘못된 프로그램 거동이 나타날 수 있으며, 메모리 접근 오류, 잘못된 결과, 프로그램 종료, 또는 시스템 보안 누설이 발생할 수 있다.

버퍼 오버플로가 코드를 실행시키도록 설계되거나 프로그램 작동을 변경시키도록 설계된 입력에 의해 촉발될 수 있다. 따라서 이는 많은 소프트웨어 취약점의 근간이 되며 악의적으로 이용될 수 있다. 경계 검사로 버퍼 오버플로를 방지할 수 있다.

버퍼 오버플로는 보통 데이터를 저장하는 과정에서 그 데이터를 저장할 메모리 위치가 유효한지를 검사하지 않아 발생한다. 이러한 경우 데이터가 담긴 위치 근처에 있는 값이 손상되고 그 손상이 프로그램 실행에 영향을 미칠 수도 있다. 특히, 악의적인 공격으로 인해 프로그램에 취약점이 발생할 수도 있다.

흔히 버퍼 오버플로와 관련되는 프로그래밍 언어는 C와 C++로, 어떤 영역의 메모리에서도 내장된 데이터 접근 또는 덮어쓰기 보호 기능을 제공하지 않으며 어떤 배열에 기록되는 데이터가 그 배열의 범위 안에 포함되는지 자동으로 검사하지 않는다.

즉, 버퍼에 많은 데이터가 read되면 인접 메모리에 대한 검사나 보호조치 없이 값을 덮어쓰며 데이터를 훼손하는 것을 의미한다. 따라서 입력할 문자열의 사이즈를 정확히 받아 무한정으로 대용량 데이터를 버퍼에 입력할 수 없게 하는 것이다.

syntax에러와 버퍼 오버플로우를 해결하고 나니 segmentation fault가 등장했다.

```
welcome to my shell <:3)~ I'm jerry
 >> ls -l
Segmentation fault (core dumped)
```

그림: segmentation fault

이 문제 때문에 지금까지의 과제 중 gdb를 가장 많이 사용했다. run과 break, next 밖에 다룰 줄 몰랐어서 명령어를

layout src : 소스 코드와 명령 창을 보여주는 모드.
layout asm : 디스어셈블리와 명령 창을 보여주는 모드
layout split : 소스 코드, 디스어셈블리 그리고 명령 창을 보여주는 모드
layout resg : 레지스터 창을 보여주는 모드.
layout prev : 이전 모드로 이동.
layout next : 다음 모드로 이동.

더 찾아 사용했다. print [변수명]을 사용하면 디버깅 도중 변수값을 확인할 수 있다.(이 기능이 제일 유용하게 쓰였다.) 또한 소스 코드가 안보여서 너무 답답했는데 창을 split해서 사용할 수 있었다. LN2 즈음에 나왔던 기능이었는데 사용을 거의 안했었다. 이렇게 창을 나눠놓고 vim으로 코딩을 하니 조금 멋있어 보였지만 내 코드는 멋있지 않았다.

```
sys32181854@embedded:~/myshell$ ps
  PID TTY          TIME CMD
 8102 pts/41   00:00:00 bash
14512 pts/41   00:00:00 test2
14653 pts/41   00:00:00 test2
14672 pts/41   00:00:00 test2
14684 pts/41   00:00:00 test2
15111 pts/41   00:00:00 test2
16095 pts/41   00:00:00 test2
16173 pts/41   00:00:00 ps
20507 pts/41   00:00:00 bash
20508 pts/41   00:00:00 command-not-fou
sys32181854@embedded:~/myshell$
```

마구 실행을 해보다 보니 불현듯 jerryshell에서 명령어를 실행할 때 마다 바뀐 것은 하나도 없는데 (execvp가 동작하지 않고 있었다.) while문은 반복되고 있다는 것을 깨달았다. 불안한 마음에 ps를 실행해보니 역시나 쉘(test2)이 엄청나게 fork 돼 있었다. 로컬서버에서 진행하는 것이 아니라서 분명 문제가 생길 것 같아 얼른 지웠다. 이 과정에서 'killall -9

[process명]' 명령어를 처음 사용했다. 이 이후부터는 한 번의 테스트 후에는 반드시 ps에서 확인 후 프로세스를 지우는 습관을 들였다. 과제가 끝나갈 때 즈음엔 서버가 자주 터졌었는데 누군가 악의적으로 무한 fork한 것이 아니라면 이런 실수로 인해 서버가 마비된 것이 아닐까 추측했다. 기본 명령을 실행하기 위한 구현을 모두 마쳤다고 생각했는데 또 실행이 되지 않았다.

```
welcome to my shell <:3)~ I'm jerry
 >> ls
ls

프로그램 실행 error : No such file or directory
```

fgets함수가 문제였던 것이다. get 종류의 함수들은 scanf와 다르게 whitespace를 그대로 문자열로 입력해서 받아온다. 반면 scanf가 띄어쓰기나 '\n'와 같은 whitespace들을 기준으로 문자열을 분리한다. 따라서 이번의 경우도 개행문자가 변수 string에 그대로 남아 문제를 일으킨 것이다. 이를 제거해주고 나니 제대로 동작했다.

쉘을 구현했다는 기쁜 마음을 품고 백그라운드 프로세싱을 구현해봤다. 기본적인 논리는 역시 말로는 그다지 어렵지 않았다. 그냥 race condition을 조절해주는 synchronization을 조금 다르게 해주면 되는 것이다. &의 경우 문자열 맨 끝에 올 것이므로 찾아서 인식해주면 된다. 그러나 역시 또 문제를 일으키고 말았다. 백그라운드 프로세싱으로 실행시킨 자식

```
>>   PID TTY          TIME CMD
10349 pts/17   00:00:00 backtest4
10350 pts/17   00:00:00 ls <defunct>
10353 pts/17   00:00:00 ps <defunct>
10355 pts/17   00:00:00 cat <defunct>
10356 pts/17   00:00:00 ps
26219 pts/17   00:00:00 bash
```

프로세스들이 모두 defunct 그러니까 좀비 프로세스가 된 것이다. 고아 프로세스와 좀비 프로세스의 차이는 전자의 경우 부모가 자식보다 먼저 종료되기 때문에 exit_status를 보낼 곳이 없어 홀로 남게 된 자식 프로세스를 의미한다. 이 때 init프로세스가 이를 입양하여 wait함수를 호출해 exit_status를 받아 고아 프로세스를 종료시킨다. 반면 좀비 프로세스는 부모 프로세스가 자식 프로세스의 exit_status값을 제대로 처리하지 못해 발생한다. 이에 자식 프로세스는 실행은 끝났지만 프로세스 테이블에서 제거되지 않고 그대로 남아있게 된다. 즉 저 때 나의 코드가 그런 상태였다는 뜻이다. 디버깅 해보니 backg 플래그 값이 잘못 설정돼 있어서 나타난 일이었다. 수정을 하고 나니 백그라운드 프로세싱도 구현되었다.

리다이렉션을 구현할 때는 파싱된 argv를 제대로 활용하지 못해서 오류가 많이 났었다. 이 때부터는 서버가 갑자기 죽거나 정신없이 코딩했어서 스냅샷을 찍어두지 않았다. 오류가 발생했던 원인과 고친 방법을 이렇게 정리하고 나니 과제를 하며 찾아본 내용을 정리할 수 있어 좋은 것 같은데 이 부분은 사진이 없어서 정말 아쉽다. 어쨌든 과제를 마쳤다. 직접 만들어서 그런지 jerryshell에 애정이 생겼다. 참고로 jerryshell의 jerry는 톰과 제리의 jerry이다. 그래서 시작할 때 <:3)~~ 로 쥐 이모지를 넣어두었다. 가끔 jerryshell이 종료된 것을 모르고 bash에 복잡한 명령어를 실행해보면서 '아 내가 이렇게 잘 만들었었나?'하고 뿌듯해 하다 process목록에서 bash만 동작 중인 것을 보고 내심 실망한 적도 있었다. 어쩐지 안정적이고 오류도 적다 싶었다. 나도 이렇게 멋지고 안정적인 프로그램을 만들 수 있었으면 좋겠다는 생각을 했다. 시스템 프로그래밍 강의가 끝나고 나면 복습을 하면서 다시 shell을 만들어보고 싶다. 그 때는 제리가 아닌 tomshell로 만들어야겠다. 더 열심히 공부해야겠다!

p.s. 서버가 다운된 것을 계기로 우분투를 설치했고 앞으로 구현해보고 싶은 코드가 있을 때 유용하게 사용하며 리눅스 공부를 할 것이다. 그런데 지금처럼 서버로 접속하여 사용하는 방식의 리눅스는 host ip와 포트 번호 등을 알면 아이패드의 가상터미널 어플인 terminus에서도 사용이 가능하다. 그러면 내 노트북에도 리눅스 server용을 설치하면 아이패드에서

내 서버로 접속해 사용할 수 있는 것일까? 서버를 열기 위한 다른 작업은 어떤 것이 있을지 궁금해졌다. 이에 대해서도 찾아보고 공부해 봐야겠다.