Tiny Shell

컴파일 과정

```
os@os-VirtualBox:~/Documents/proj1$ make
gcc -Wall -O -c tsh.c
gcc -o tsh tsh.o
os@os-VirtualBox:~/Documents/proj1$
```

기능 설명

가장 먼저 프로그램이 **표준 입출력 리다이렉션('<', '>')** 명령어와 **파이프('|')** 명령어를 파싱할 수 있도록 아래와 같이 `strpbrk 함수` 인자에 각 문자를 추가하였습니다.

```
44          /*
45           * 공백문자, 큰 따옴표, 작은 따옴표, <, >, |가 있는지 검사한다.
46           */
47          q = strpbrk(p, " \t\'\"<>|");
```

0. 변수

기존에 있던 변수들을 제외하고, 기능을 추가하면서 새롭게 추가한 변수들은 아래와 같습니다. 주석으로 설명을 추가하긴 했지만 더 자세한 내용은 각 기능에 대해 설명하면서 보충하도록 하겠습니다.

```
34      int fd, fd2;                /* 파일 처리를 위한 변수 */
35      bool is_input = 0, is_output = 0;   /* 입력 리다이렉션 | 출력 리다이렉션 여부 */
36      int input_argc, output_argc;        /* 리다이렉션 기호 입력 당시의 명령어 갯수 */
```

1. 표준 입출력 리다이렉션

표준 입출력 리다이렉션을 구현하기 위해서는 '<' 문자와 '>' 문자를 추출하고, 각 문자에 따라서 알맞은 리다이렉션 기능을 수행시켜야 합니다.

- 리다이렉션 문자를 찾는 부분은 앞서 `strpbrk 함수`를 사용하여 이미 구현을 완료했습니다.
- 따라서 q가 가리키는 문자가 '<' 혹은 '>' 중 무엇인지에 따라서 명령어를 처리하기 위해, 기존에 있던 if-else if-else문에 아래와 같이 조건문을 추가하였습니다.

```
55          /*
56           * 입력 리다이렉션이 필요한 경우
57           * 파일명을 입력받기 위해 입력 리다이렉션 표시를 남기고,
58           * 현재 명령어 갯수를 저장한다.
59           */
60          else if (*q == '<') {
61              // ? "grep int<tsh.c" 와 같이, 띄어쓰기 없이 명령어를 입력한 경우를 대비
62              q = strsep(&p, "<");
63              if (*q) argv[argc++] = q;
64
65              input_argc = argc;
66              is_input = true;
67          }
68          /*
69           * 출력 리다이렉션이 필요한 경우
70           * 파일명을 입력받기 위해 출력 리다이렉션 표시를 남기고,
71           * 현재 명령어 갯수를 저장한다.
72           */
73          else if (*q == '>') {
74              q = strsep(&p, ">");
75              if (*q) argv[argc++] = q;
76
77              output_argc = argc;
78              is_output = true;
79          }
```

표준 입출력 리다이렉션의 명령어 형태는 `command < file`, `command > file`과 **같이 리다이렉션 명령어 문자 뒤에 파일명**이 등장합니다. 즉, 입출력 리다이렉션 문자를 파싱한 다음에는 **파일명을 파싱**해야 합니다. 따라서 파일명이 argv 배열에 추가되었는지 확인할 수 있도록 `input_argc`, `output_argc` 변수를 추가하였고, 해당 변수들은 리다이렉션 처리 여부를 판단할 때 사용하였습니다. 실제로 리다이렉션을 처리하는 부분은 while문 마지막 부분에 아래 코드를 추가하여 구현하였습니다.

```
150         /*
151          * 명령어를 한번 파싱할 때마다 리다이렉션을 처리할 준비가 되었는지 확인하기.
152          * 리다이렉션 기호를 찾은 시점을 기준으로 명령어 수가 1만큼 증가했다는 것은
153          * 파일명이 argv에 입력되었다는 의미이므로 리다이렉션을 처리한다.
154          */
155         if (is_input && argc == input_argc + 1) {
156             /* 파일을 처리함과 동시에 파일명은 명령어 배열에서 삭제한다. */
157             if ((fd = open(argv[--argc], O_RDONLY, 0)) > 0) {
158                 /* 아직 출력 리다이렉션이 처리되지 않은 경우, output_argc 값도 1 감소시킨다. */
159                 if (is_output) output_argc--;
160                 dup2(fd, STDIN_FILENO);
161                 close(fd);
162             } else {
163                 printf("file error\n");
164                 return ;
165             }
166             /* 리다이렉션을 모두 처리하면 관련 flag 값을 갱신한다. */
167             is_input = false;
168         }
169         else if (is_output && argc == output_argc + 1) {
170             if ((fd2 = open(argv[--argc], O_WRONLY | O_TRUNC | O_CREAT, 0644)) > 0)
                {
171                 dup2(fd2, STDOUT_FILENO);
172                 close(fd2);
173             } else {
174                 printf("file error\n");
175                 return ;
176             }
177             /* 리다이렉션을 모두 처리하면 관련 flag 값을 갱신한다. */
178             is_output = false;
179         }
180     } while (p);
```

우선 위 코드에서 사용된 `is_input`, `is_output`, `input_argc`, `output_argc` 변수에 대해 설명을 드리겠습니다.

- `is_input`, `is_output` 변수는 모두 bool 변수로, 각각 입력 / 출력 리다이렉션을 처리해야 함을 의미하며 입/출력 리다이렉션을 구분하기 위해 사용했습니다.
- `input_argc`, `output_argc` 변수는 리다이렉션 문자('<' | '>')가 파싱될 때까지 argv에 명령어가 총 몇 개 저장되어 있는지 기억하는 변수로, 파일명이 argv 배열에 입력되었는지 확인하기 위해 사용했습니다.

2. 파이프

파이프 기능도 마찬가지로 '|' 문자를 파싱한 후 파이프 기능을 수행시켜야 합니다.

- 파이프 문자를 찾는 부분은 앞서 `strpbrk 함수`를 사용하여 이미 구현을 완료했습니다.
- 따라서 q가 가리키는 문자가 '|' 일 때 파이프 기능을 수행할 수 있도록, 기존에 있던 if-else if-else문에 아래와 같이 조건문을 추가하였습니다.

```
85          else if (*q == '|') {
86              q = strsep(&p, "|");
87
88              if (*q) argv[argc++] = q;
89
90              /*
91              * 자식-손자를 연결할 파이프를 생성
92              */
93              int fd_pipe[2];
94              if (pipe(fd_pipe) == -1) {
95                  printf("PIPE ERROR\n");
96                  return ;
97              }
98
99              pid_t pid;
100             if ((pid = fork()) < 0) {
101                 printf("FORK ERROR\n");
102                 return ;
103             }
104             /*
105              * 손자 프로세스는 WRITE_END 파이프를 출력과 연결시킨 후,
106              * 현재까지의 명령어를 실행한다.
107              */
108             if (pid == 0) {
109                 close(fd_pipe[READ_END]);
110                 dup2(fd_pipe[WRITE_END], STDOUT_FILENO);
111                 close(fd_pipe[WRITE_END]);
112
113                 argv[argc] = NULL;
114                 execvp(argv[0], argv);
115             }
116             /*
117              * 자식 프로세스는 READ_END 파이프를 입력과 연결시킨 후,
118              * 파이프(|) 이후 명령어부터 저장하기 위해 argc 값을 초기화한다.
119              */
120             else {
121                 close(fd_pipe[WRITE_END]);
122                 dup2(fd_pipe[READ_END], STDIN_FILENO);
123                 close(fd_pipe[READ_END]);
124                 argc = 0;
125             }
126         }
```

우선 파이프는 앞 명령어의 실행 결과를 또 다른 명령어의 입력으로 전달할 때 사용하는 기능입니다. 이러한 파이프 명령어를 구현하기 위해서는 우선 '|' 문자를 추출하고, '|' 문자를 기준으로 앞/뒤 명령어를 나누어서 실행해야 합니다.

즉, **앞 명령어를 먼저 실행**한 후, 그 결과를 **다음 명령어의 입력으로 전달**해야하기 때문에 앞의 명령어를 실행하여 그 결과를 전달해줄 자식*(이 프로젝트에서는 손자)* 프로세스를 생성해야 합니다.

- **(L93 - L103)** 손자 프로세스를 생성(fork)하기 전, 자식 프로세스는 다음 명령어의 입력을 손자 프로세스로부터 전달받아야 하기 때문에 둘이 정보를 주고받을 수 있는 `파이프`를 먼저 생성했습니다.
- **(L104 - L125)** `손자 프로세스`는 파이프의 **WRITE_END**를 열고, 현재까지 입력받은 명령어를 실행합니다. `자식 프로세스`는 파이프를 통해 손자 프로세스의 명령어 실행 결과를 받아와야 하므로, 파이프의 **READ_END**를 열고 이를 통해 손자 프로세스가 보낸 실행 결과를 받습니다.
- **(L124)** 파이프 문자(|) 이전의 명령어들은 `손자 프로세스`를 통해 이미 실행을 마쳤기 때문에, 자식 프로세스에서는 더이상 쓸 일이 없습니다. 또한 `자식 프로세스`는 이제 파이프 뒤에 오는 명령어를 순차적으로 실행해야 하므로, argc 값을 0으로 초기화하여 파이프 문자(|) 뒤에 오는 명령어를 처음부터 파싱합니다.

Test Case 실행 결과

기능을 추가한 후 각 테스트 케이스를 입력한 결과는 다음과 같습니다.

- `grep int tsh.c` - tsh.c 파일 내에서 int 라는 글자가 포함된 행을 모두 출력

```
tsh> grep int tsh.c
    int argc = 0;               /* 인자의 개수 */
    int fd, fd2;                /* 파일 처리를 위한 변수 */
    int input_argc, output_argc;        /* 리다이렉션 기호 입력 당시의 명령어
갯수 */
            // ? "grep int<tsh.c" 와 같이, 띄어쓰기 없이 명령어를 입력한 경우를
 대비
            int fd_pipe[2];
                printf("PIPE ERROR\n");
                printf("FORK ERROR\n");
                printf("file error\n");
                printf("file error\n");
int main(void)
    int len;                    /* 입력된 명령어의 길이 */
    int background;             /* 백그라운드 실행 유무 */
            printf("[%d] + done\n", pid);
        printf("tsh> "); fflush(stdout);
```

- `grep "if.*NULL" tsh.c &` - (백그라운드 실행) tsh.c 파일 내에서 if로 시작하고 NULL이 포함된 행을 모두 출력

```
tsh> grep "if.*NULL" tsh.c &
tsh>         if (q == NULL || *q == ' ' || *q == '\t') {
        if (p != NULL) {

[5312] + done
tsh> ps
```

- `ps` - 현재 실행중인 프로세스 출력

```
tsh> ps
  PID TTY          TIME CMD
 4361 pts/0    00:00:00 bash
 5300 pts/0    00:00:00 tsh
 5313 pts/0    00:00:00 ps
```

- `grep "int " < tsh.c` - tsh.c 파일에서 “int “ 패턴이 있는 행을 모두 출력

```
tsh> grep "int " <tsh.c
    int argc = 0;               /* 인자의 개수 */
    int fd, fd2;                /* 파일 처리를 위한 변수 */
    int input_argc, output_argc;        /* 리다이렉션 기호 입력 당시의 명령어
갯수 */
            int fd_pipe[2];
int main(void)
    int len;                    /* 입력된 명령어의 길이 */
    int background;             /* 백그라운드 실행 유무 */
```

- `ls -l > delme` - `ls -l` 실행 결과를 delme 파일에 작성

```
tsh> ls -l > delme
tsh> cat delme
total 48
-rw-r--r-- 1 os os     0  3월 30 16:02 delme
-rw-r--r-- 1 os os   240  3월 26 22:29 delme2
-rw-r--r-- 1 os os    75  3월 28 14:56 delme3
-rw-rw-rw- 1 os os   408  3월 26 22:28 Makefile
-rwxrwxr-x 1 os os 13240  3월 30 15:59 tsh
-rw-rw-rw- 1 os os 10853  3월 30 15:59 tsh.c
-rw-rw-r-- 1 os os  5440  3월 30 15:59 tsh.o
tsh> sort < delme > delme2
tsh> cat delme2
-rw-r--r-- 1 os os     0  3월 30 16:02 delme
-rw-r--r-- 1 os os   240  3월 26 22:29 delme2
-rw-r--r-- 1 os os    75  3월 28 14:56 delme3
-rw-rw-r-- 1 os os  5440  3월 30 15:59 tsh.o
-rw-rw-rw- 1 os os 10853  3월 30 15:59 tsh.c
-rw-rw-rw- 1 os os   408  3월 26 22:28 Makefile
-rwxrwxr-x 1 os os 13240  3월 30 15:59 tsh
total 48
```

- `ps -A | grep -i system` - 모든 프로세스 출력 결과에서 대소문자 구분 없이 system 문자열을 포함하는 행을 출력
- `ps -A | grep -i system | awk '{print $1,$4}'` - 위의 결과에서 각 행에 대해 1열(프로세스 id)과 4열(cmd)에 있는 값들만 출력한다.

```
tsh> ps -A | grep -i system
    1 ?        00:00:15 systemd
  278 ?        00:00:02 systemd-journal
  299 ?        00:00:01 systemd-udevd
  390 ?        00:00:00 systemd-resolve
  570 ?        00:00:00 systemd-logind
 1147 ?        00:00:00 systemd
 1366 ?        00:00:00 systemd
 4637 ?        00:00:00 apt.systemd.dai
 4641 ?        00:00:00 apt.systemd.dai
 5418 ?        00:00:00 systemd-timedat
tsh> ps -A | grep -i system | awk '{print $1,$4}'
1 systemd
278 systemd-journal
299 systemd-udevd
390 systemd-resolve
570 systemd-logind
1147 systemd
1366 systemd
4637 apt.systemd.dai
4641 apt.systemd.dai
5418 systemd-timedat
```

- `cat tsh.c | head -6 | tail -5 | head -1`
  1. tsh.c 파일 내용을 읽어온다.
  2. tsh.c 파일 내용을 처음부터 6개의 줄로 잘라낸다.
  3. 6개의 줄에서 마지막 줄부터 5개의 줄로 잘라낸다.
  4. 5개의 줄에서 처음부터 1개의 줄로 잘라낸다. (즉, 소스 코드의 두 번째 행만 남은 상태)

```
tsh> cat tsh.c | head -6 | tail -5 | head -1
/*
```

++ 뼈대 코드 주석의 경우 아래와 같이 출력되는 것을 확인할 수 있다.

```
tsh> cat tsh.c | head -6 | tail -5 | head -1
 * Copyright(c) 2023 All rights reserved by Heekuck Oh.
```

- `sort < tsh.c | grep “int “ | awk ‘{print $1,$2}’ > delme3`
    1. tsh.c 파일의 내용을 오름차순 정렬한다.
    2. 1의 결과에서 `int`가 포함된 행을 모두 찾는다.
    3. 2의 결과의 각 행에 대해 1번 열과 2번 열만 출력한다.
    4. 3의 결과를 delme3 파일에 저장한다.

```
tsh> sort < tsh.c | grep "int " | awk '{print $1,$2}' > delme3
tsh> cat delme3
int argc
int background;
int fd,
int fd_pipe[2];
int input_argc,
int len;
int main(void)
tsh>
```

문제점 및 느낀점

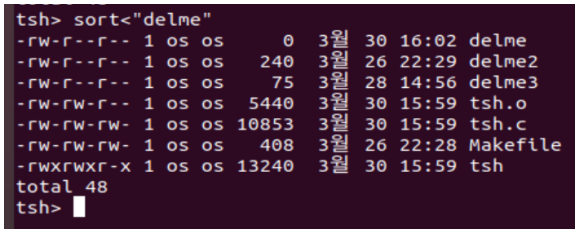
리다이렉션 기능 구현 과정

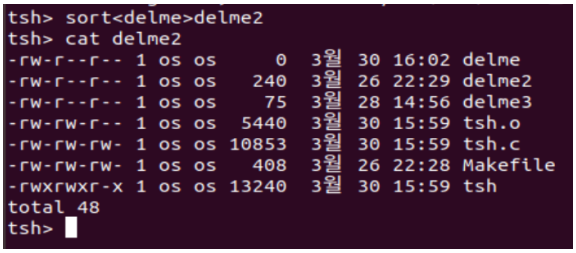
리다이렉션 기능을 구현한 코드의 경우, 처음에는 리다이렉션 문자를 처리하면서 바로 파일 입출력까지 처리하는 방식으로 작성했습니다. 다시 말해, 조건문을 따로 분리하지 않고 리다이렉션 문자(‘<’ | ‘>’)를 파싱하는 조건문 내에서 파일명까지 파싱한 후 파일 입출력까지 바로 처리했습니다.

하지만 여러번 테스트 과정을 거치다보니 해당 방법대로 구현하게 되면 `ls - l > “delme”` 와 같이 **파일명을 따옴표로 감싼 경우** 제대로 동작하지 않는 것을 확인할 수 있었습니다. 이는 리다이렉션 조건문 내에서 공백 및 따옴표에 대한 파싱을 진행하지 않았기 때문입니다.

하지만, 리다이렉션 조건문 내에서 공백 및 따옴표에 대한 파싱을 따로 진행하기에는 코드가 비효율적이라고 판단했습니다. 왜냐하면 기존에 **이미 관련 기능이 구현**되어 있기 때문입니다. 똑같은 코드를 추가하는 것보다 기존에 구현된 기능을 활용하는 것이 더 효율적일 것이라고 판단하여, 작성했던 코드를 모두 과감하게 지우고, 새로운 접근 방식을 떠올리며 더 완성도 높은 결과물을 얻을 수 있었습니다.

기존 코드에서는 `sort<delme>delme2` 와 같이 **띄어쓰기를 하지 않고 리다이렉션 명령어를 조합한 경우**에도 제대로 동작하지 않았지만, 수정 과정을 거치면서 위와 같은 명령어도 제대로 동작하는 것을 확인할 수 있었습니다.





파이프 기능 구현 과정

리다이렉션 기능을 구현하고 파이프 기능의 동작 방식을 이해한 상태에서 진행하여 파이프 기능은 금방 구현할 수 있었습니다. 특히 이를 구현하는 과정에서 프로세스끼리 통신하는 메커니즘에 대해 더 잘 이해할 수 있었습니다.

프로젝트를 진행하며 느낀점

항상 프로젝트를 진행할 때는 코드의 **일관성**과 **효율성**을 고려해야 한다고 생각합니다. 하지만 이번 프로젝트를 처음 시작할 때는 기존의 코드를 이해하는 것보다 기능을 추가하는데만 급급하여 결국 첫 번째 결과물에서는 제대로 된 결과를 얻지 못했습니다. 가령, 기존 뼈대 코드에서는 명령어를 하나씩 파싱하는 방식을 추구한 반면, 제 코드에서는 ***한번에 여러 개의 명령어를 파싱****(ex. 입출력 리다이렉션에서 파일명까지 한번에 파싱)*하다보니 문제가 발생하는 것을 발견할 수 있었습니다. 저는 이러한 문제점을 인식하고 다시 원점으로 돌아가 뼈대 코드를 정확히 이해하는 데 더 많은 시간을 투자했습니다. 그 결과 오히려 코드를 작성하는 데 소요되는 시간이 현저히 줄어들었고, 더 효율적이고 일관적인 코드로 재완성시킬 수 있었습니다. 저는 이 과정을 통해 프로젝트를 빠른 시간 내에 완성시키는 것도 물론 중요하지만, 해당 프로젝트를 얼마나 잘 이해하고 있는가가 가장 중요하다고 느꼈습니다.

그리고 프로젝트에서 처음부터 완벽한 코드를 완성시키려 하기보다는, 처음 작성한 코드가 미완성된 코드일지라도 문제점을 하나씩 보완하면서 완성도를 높여가는 방법이 더 효율적이라고 느꼈습니다. 프로젝트를 시작했을 때는 처음부터 완벽한 결과물을 내려다보니 오히려 코드가 잘 짜여지지 않았습니다. 초기에는 작은 기능만 수행하도록 완성시켜놓고, 그 다음에 기능을 확장하고 문제점을 보완해나가며 수정하다보니 프로그램에 대한 이해도도 더 높아지고 시간도 더 효율적으로 사용할 수 있었습니다.

마지막으로 현재까지 완성한 프로젝트에서 아쉬움이 남는 부분은 cmdexec 파일 내의 기능이 분리되어있지 않다는 점이었습니다. 각 명령어를 처리하는 부분을 따로 분리하여 관리하면 재사용 측면에서 더 좋을 수 있을 것 같다는 생각이 들었으나, 이번 프로젝트에서는 시간이 부족하여 관련 내용을 따로 진행하지 못한 점이 아쉽습니다.