proj1 보고서

소프트웨어학부 2019007329 이지현

1. 함수 설명

• cmdexec 함수

기존의 cmdexec 스켈레톤 코드에서 파이프와 리다이렉션 기능을 추가하였다.

커맨드의 <, >, | 기호를 처리하기 위해 각 기호가 있는 인덱스를 저장하기 위한 변수를 선언하였다. 커맨드 전체를 p 변수에 할당하고 while문에서 p 변수를 파싱한다.

strsep 함수를 이용해서 커맨드를 파싱한 뒤, 만약 커맨드에 리다이렉션 기호인 <, >가 있다면 인덱스 저장용 변수인 l, r에 기호 위치를 저장한다. 파싱을 위한 인덱스 변수 argc는 파싱에 사용된 이후에 1씩 증가하기 때문에 리다이렉션 기호의 인덱스를 저장할 때는 -1을 하였다.

만약 | 기호를 발견하였다면 pp에 파이프 기호 인덱스를 저장한 후, break를 사용하여 while문을 빠져나온다. cmdexec 함수는 하나의 명령어만 처리할 수 있기 때문에 파이프 기호 전까지의 명령문만을 파싱한다.

while문을 빠져나온 뒤 명령 실행을 위한 조건문을 실행한다.

파이프 기호가 있다면 index는 0 초과일 것이므로 pp > 0 조건문을 작성하였고, 파이프 기호를 NULL 처리 해 주었다. pipe(fd)와 fork()를 사용하여 자식을 생성하고 pid == 0일 경우(자식일 경우) command_1(argv)를 실행하기 위해 파싱된 argv 배열을 문자열로 바꾸어 준다. sprintf를 사용하여 argv를 하나의 문자열로 만들어 buffer 변수에 저장해 주었고, dup2를 사용하여 표준 출력을 fd[1]로 바꾸어 주었다. 그 후 cmdexec(buffer)를 호출하며 command_1을 실행할 수 있도록 하였다. pid > 0 일 경우 부모 프로세스이므로 wait(NULL)을 이용하여 자식 프로세스가 끝날때까지 기다린 뒤 자식 프로세스의 값을 받기 위해 dup2를 사용하여 표준 입력 대신 fd[0]으로 입력값을 받도록 하였다. 그 후 while문 안에서 파싱되지 않은 파이프 이후의 커맨드 문자열인 p를 이용하여 cmdexec(p)를 호출해 command_x를 실행하도록 하였다. command_x는 파이프가 존재하지 않을 경우 재귀가 끝나게 된다.

파이프 조건문이 끝난 뒤에는 리다이렉션 조건문을 실행한다. l > 0 이라면 '<' 기호가 존재하기 때문에 NULL로 기호를 없애주고 읽어올 파일을 open 함수를 통해 열어준다. 읽어올 파일은 기호 바로 뒤에 위치해 있으므로 argv[l+1]을 열어주고 파일 생성과 읽고 쓸수 있는 권한을 주었다. dup2를 이용하여 표준 입력을 open을 통해 연 파일로 바꾼다.

r > 0 이라면 '>' 기호가 존재하기 때문에 NULL로 기호를 없애주고 저장할 파일을 생성한다. 표준 출력 대신 open 함수로 생성한 파일에 출력을 저장한다.

마지막으로 모든 조건문을 지나면 명령을 실행시켜주는 execvp 함수가 작동한다. cmdexec 함수는 한번에 하나의 명령어만 실행되기 때문에 execvp(argv[0], argv)를 사용하였다.

2. 컴파일 과정

```
os@os-VirtualBox:~/Documents/theory_assignment/proj1-1$ make
gcc -Wall -O      -c tsh.c
gcc -o tsh tsh.o
```

3. 실행 결과물

- 리다이렉션

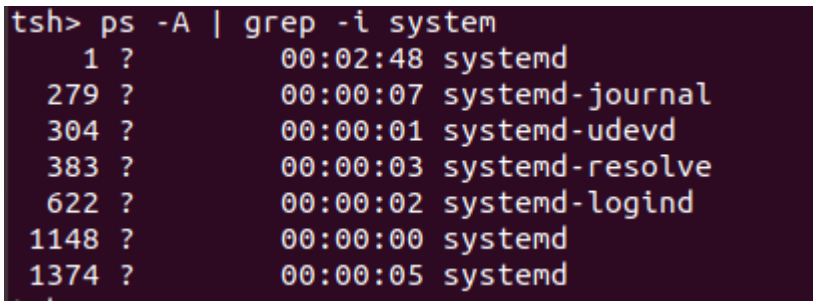
```
tsh> grep "int " < tsh.c
    int argc = 0;               /* 인자의 개수 */
    int l = 0;          /* '<' 체크용 위치 저장 변수 */
    int r = 0;          /* '>' 체크용 위치 저장 변수 */
    int pp = 0;         /* '|' 체크용 위치 저장 변수 */
            int fd[2];
             * 자식 프로세스. int main까지 합하면 손자.
                    int leng = 0;
                    for (int i = 0; i < argc; i++) {
             * 부모 프로세스. int main까지 합하면 자식.
            int fd_l = open(argv[l+1], O_RDWR | O_CREAT, 0644);     /* 읽어 올 파일 열기 */
            int fd_r = open(argv[r+1], O_RDWR | O_CREAT, 0644);     /* 저장할 파일 생성 */
int main(void)
    int len;                    /* 입력된 명령어의 길이 */
    int background;             /* 백그라운드 실행 유무 */
```

명령어 grep "int " < tsh.c를 실행시킨 결과이다. 파일 tsh.c에서 "int " 명령어를 포함하는 텍스트를 보여준다. int 뒤에 공백이 있기 때문에 printf와 같이 단순히 int만 포함된 텍스트는 출력되지 않는다.

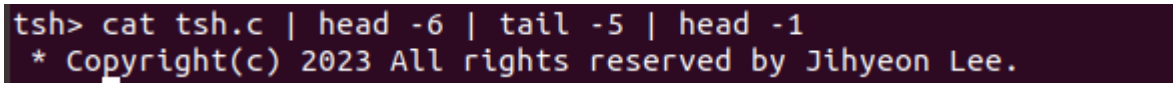
```
tsh> ls -l > delme
tsh> sort < delme > delme2
tsh> cat delme2
drwx------ 2 os os    4096  3월 16 15:31 __MACOSX
-rw-r--r-- 1 os os       0  3월 29 21:20 delme
-rw-rw-r-- 1 os os    5648  3월 29 21:14 tsh.o
-rw-rw-rw- 1 os os 2739828  3월 16 15:31 proj1.pdf
-rw-rw-rw- 1 os os     557  3월 16 15:39 Makefile
-rw-rw-rw- 1 os os    9371  3월 29 20:56 tsh.c
-rwxrwxr-x 1 os os   13448  3월 29 21:19 tsh
total 2720
```

먼저 ls -l > delme 명령어를 실행하였다. ls -l의 결과를 화면에 출력하지 않고 delme 파일을 생성한 후 저장했기 때문에 화면상에 나타나지 않는다. sort < delme > delme2 명령어는 delme 파일에서 읽어온 값을 sort하고, 그것을 delme2에 저장한다. 그 후 cat 명령어로 delme2 파일을 화면상으로 출력하면 ls -l이 정렬된 모습을 볼 수 있다.

- 파이프

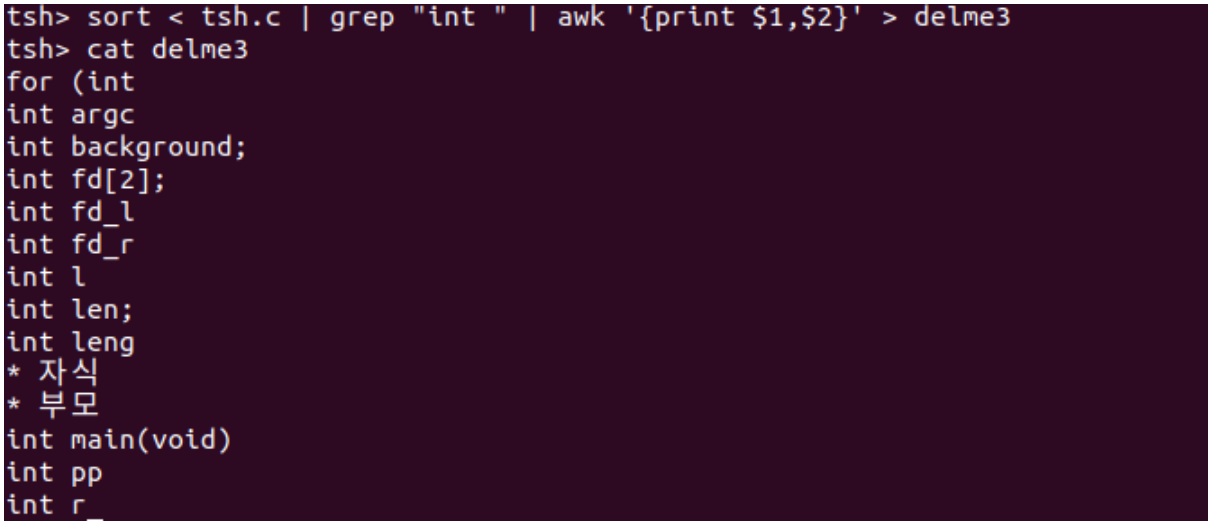


ps -A의 결과를 grep -i system의 입력으로 보내주었다. ps -A 결과 중 system을 포함하고 있는 정보만 출력된 것을 볼 수 있다.



tsh.c의 텍스트 중 위에서 6번째 줄까지, 그 중에서 아래에서 5번째 줄까지, 그 중에서 위에서 1번째 줄까지의 텍스트를 출력하는 명령어이다. 결국 파일의 위에서 두번째 줄만 출력되는 것을 볼 수 있다.

- 조합



리다이렉션과 파이프가 모두 등장한 명령어이다. 위의 명령어만 작성하였을 때는 출력이 되지 않는 상태이지만, delme3 파일에 저장된 값을 보기 위해 cat 명령어를 사용하면 sort된 tsh.c의 "int "가 포함된 행 중 1, 2열만 출력된 것을 볼 수 있다.

4. 문제점과 느낀점

파이프나 리다이렉션 단일 기능을 만드는 것은 의외로 간단하게 구현할 수 있었다. 그러나 다중 파이프, 파이프와 리다이렉션을 연결하는 부분에서 큰 어려움을 겪었다. 재귀를 이용하여 코드를 작성한다면 더욱 쉽게 구현할 수 있다는 교수님의 조언을 얻었지만 어느 부분에서 함수를 호출해야 할지, 재귀함수를 사용한다면 명령어 실행은 몇 번 해야 하는지 많은 의문점들이 생겼다. command_x만이 파이프를 가질 수 있고 command_1은 파이프가 없으니 부모 프로세스에서만 재귀함수를 사용하고 자식 프로세스에서 execvp를 사용해서 command_1을 실행할 계획이였지만 제대로 코드가 작동되지 않았기에 교수님께 위 문제에 대한 질문을 드렸다. 그 결과 자식 프로세스와 부모 프로세스 모두 재귀함수를 사용해야 한다는 답변을 얻을 수 있었다. 생각해 보니 두 프로세스 모두에서 재귀 함수를 사용하면 파이프가 없는 프로세스는 자동으로 코드 맨 마지막의 execvp를 실행하고, 파이프가 존재한다면 함수를 계속해서 반복한다는 아주 간단한 사실을 도출해 낼 수 있었다.

당연하게 생각했던 기능들이 작동하지 않아서 당황했던 경험도 있었다. 예를 들면 명령어를 작성하다 잘못된 부분이 있어 왼쪽 방향키를 눌러 이전 텍스트로 이동하려고 했을 때나 이전에 작성한 명령어를 그대로 불러오기 위해 위쪽 방향키를 눌렀을 때 모두 내가 원하는 대로 작동하지 않아 이러한 부분까지도 기능을 구현해야 사용할 수 있다는 것을 알게 되었다. 또한 기존의 쉘에서는 쉽게 명령어 출력 결과를 파악할 수 있도록 다양한 색상으로 텍스트가 출력되는 반면에 내가 만든 쉘에서는 같은 흰색 텍스트가 출력되어 가시성이 좋지 않았다. 이런 부분을 보완한다면 더욱 편리한 쉘을 만들 수 있을 것이라 생각한다.