시스템 프로그래밍 쉘 구현 보고서

2018037356 안동현

1. 기본 쉘 구현

```
// 시스템 변수
char *args[MAX_LEN / 2 + 1];
int should_run = 1;
int background = 0;

// 명령 변수
char *input;
int status;

// 파이프 변수
int pipe_idx[TOKEN_CNT] = {0,};
int pipe_cnt = 0;
int has_pipe = 0;
```

args는 입력받은 문자열을 잘랐을 때, 잘린 조작들을 저장하는 변수입니다. should_run은 프로그램이 종료하지 않는 이상 무한으로 돌아가도록 하기 위해서 작성한 변수입니다. background는 프로그램 실행 후, 명령어를 실행할 때 &를 통해서 백그라운드 모드를 활성화 했는지, 안했는지를 확인해주는 변수입니다. 이러한 시스템 변수들로 프로그램의 현 상황을 인지하고 그에 맞게 동작하도록 작성합니다.

또한 명령어를 받기 위한 변수인 input 변수와, 자른 문자열의 인덱스 관리를 해주는 status 변수가 있습니다. pipe 변수는 이따가 설명드리겠습니다.

```
// 무한 반복
while (should_run) {
    // 인터페이스 출력
    printf("my_shell> ");

    // 출력 버퍼 비워두기
    fflush(stdout);
```

이런식으로 반복문을 사용하며, 루프의 첫번째 동작은 my_shell> 인터페이스를 출력하고, 혹시나 모를 출력 버퍼를 비워 두어 버그를 방지합니다.

```c
// 기본적으로 백그라운드, 파이프 비활성
background = 0;
has_pipe = 0;
pipe_cnt = 0;

// 명령어 입력
input = (char*)malloc(MAX_LEN*sizeof(char));
fgets(input, MAX_LEN, stdin);

// 그냥 엔터만 치면 반복문 맨 끝으로 점프
if (strcmp(input, "\n") == 0){
    goto next_time;
}
// exit 치면 쉘 나가기
else if (strcmp(input, "exit\n") == 0){
    break;
}
```

이후에는 초기화 해줘야할 변수들을 초기화해주고 명령어를 입력받습니다.

입력은 fgets와 표준 입력 스트림을 통해서 받아내고, 만약 엔터만 친다면

```c
187
188
189         }
190
191 next_time:
192         free(input);
193     }
194     return 0;
195 }
```

바로 이렇게 goto문을 통해서 할당한 input을 해제해주면서 반복문의 끝으로 갑니다.

또한 exit을 입력한다면, my shell 프로그램을 종료합니다.

```
// toknize
char *p;
p = strtok(input, " \n");

status = 0;
while(p != NULL){
    args[status++] = p;
    p = strtok(NULL, " \n");
}

// 마지막에 널문자
args[status] = NULL;
```

이들 모두에 해당되지 않는다면, 다른 명령어를 작성했다는 소리이니 문자열을 자릅니다. strtok를 통해서 공백 문자와 개행 문자를 기준으로 문자열을 자르며 p에 저장하고, 이때 임시로 저장된 포인터이기에, status를 통해서 args에 하나씩 저장해가며 strtok(NULL, " ₩n"); 으로 p가 NULL이 될 때 까지 포인터를 이동합니다. 이후 마지막에는 끝을 알려주는 NULL을 추가해줍니다.

이제 args 안에는 저희가 입력했던 명령어들이 공백이나, 개행문자를 기준으로 잘려서 하나씩 들어가게 되었습니다.

```
// 문자열 비교 -> 백그라운드, 파이프 확인
for (int i = 0; i < status; i++){
    if (strcmp(args[i], "&") == 0){
        background = 1;
        args[status - 1] = NULL;
    }
    else if (strcmp(args[i], "|") == 0){
        args[i] = NULL;
        pipe_idx[++pipe_cnt] = i+1;
        has_pipe = 1;
    }
}
```

이제 자른 문자열들을 확인해가며, 모드를 사용했는지 학인해줍니다. &가 있다면 프로그램 실행을 백그라운드 모드로 하겠다는 것을 의미하고, |가 있다면 파이프가 존재하므로 파이프 기능을 사용하겠다는 것을 의미합니다.

```
// 자식 프로세스 생성
pid_t pid = fork();
if (pid < 0) {
    perror("Fork error");
    exit(0);
}
// 자식 프로세스이면, 파일 실행
else if (pid == 0){
    execvp(args[0], args);

    // 이 밑은 프로그램이 정상 작동하지 않았을때 실행
    printf("command not found | invalid option\n");
    exit(0);
}
// 부모 프로세스이면
else{
    // 자식 프로세스 끝나길 대기
    if (background == 0){
        waitpid(pid, NULL, 0);
    }
    // 백그라운드 모드
    else{
        printf("background process\n");
    }
```

이제 프로그램을 실행하기 위해서, 자식 프로세스를 포크합니다. 기본적인 프로세스 처리를 하고, 만약 자식 프로세스이면 파일을 execvp를 통해서 실행시킵니다. 이때 프로그램이 정상 작동된다면 저 밑의 코드는 실행되지 않으나, 없는 명령어 등 이상 상황이 있다면 밑의 코드가 실행됩니다.

부모 프로세스는 기본적으로 자식 프로세스가 끝나기를 기다리나, 만약 백그라운드, 모드일 시 기다리지 않고 독립적으로 작동합니다. 여기까지가 기본 구현입니다.

2. 파이프 기능 추가

파이프 기능은 A | B 가 있다고 할 때, A의 실행 결과를 B의 입력으로 주는 기능입니다. 여기에서 A | B | C | D .... 계속 간다면, 연쇄적으로 A실행되고 그 결과가 B의 입력으로 가고 B가 실행되고 그 결과가 C의 입력으로 가고...를 반복합니다. 이것을 구현하기 위해서 변수를 설정합니다.

```
// 파이프 변수
int pipe_idx[TOKEN_CNT] = {0,};
int pipe_cnt = 0;
int has_pipe = 0;
```

pipe_idx는 args에서 "|" 가 등장한 후, 바로 다음의 명령어의 인덱스를 가리키는 배열입니다. 이를 트레이스해서 연속적인 파이프가 가능하도록 합니다.

pipe_cnt 는 파이프의 개수입니다. 이를 통해서 pipe_idx에 값을 저장하고, 후의 파이프 함수에서 반복문의 횟수를 결정합니다.

has_pipe는 파이프가 존재하는지에 대한 변수입니다. 해당 변수가 1이라면 일반적인 명령어 실행으로 가지 않고 do_pipe라는 함수에 진입합니다.

```
// 문자열 비교 -> 백그라운드, 파이프 확인
for (int i = 0; i < status; i++){
    if (strcmp(args[i], "&") == 0){
        background = 1;
        args[status - 1] = NULL;
    }
    else if (strcmp(args[i], "|") == 0){
        args[i] = NULL;
        pipe_idx[++pipe_cnt] = i+1;
        has_pipe = 1;
    }
}
```

이렇게 "|"를 찾아내면 일단 해당 부분을 NULL로 바꾸고, pipe_idx에 다음에 올 명령어의 인덱스를 넣습니다. 그리고 파이프가 존재한다는 플래그를 넣습니다. ++pipe_cnt 인 이유는 처음에 변수 설정을 할 때 pipe_idx를 전부 0으로 초기화했고, 첫번째 명령어는 인덱스 0에 있기에 바꿀 필요가 없기 때문입니다.

```
// 파이프 하나라도 존재하면, 파이프 모드로 실행
if (has_pipe == 1){
    do_pipe(args, pipe_idx, pipe_cnt, background);
    goto next_time;
}
```

이제 has_pipe를 체크하고 do_pipe에 진입합니다. 백그라운드 함수도 지원하기에 인자에 같이 넣습니다. do_pipe를 끝내고 빠져나오면 다시 입력을 처음부터 받기 위해서 goto문을 이용합니다.

do_pipe함수를 설명드리기 이전 구조부터 말씀드리겠습니다. 해당 함수의 구조는

1. 첫번째 명령어 실행

2. 반복문으로 중간 부분 명령어 실행

3. 마지막 명령어 실행

으로 이루어져있으며, 파이프 배열을 통해서, 하나하나씩 결과를 옮겨줍니다.

```
// 파이프 라인
void do_pipe(char *args[], int pipe_idx[], int pipe_cnt, int background)
{
    int pipes[TOKEN_CNT][2] = {0, };  // 파이프 목록
    int pid;
    int status;
    /*****  1번째 명령어 실행  *****/
    pipe(pipes[0]);         // 파이프 생성
    if ((pid = fork()) < 0) {
        perror("Fork Error");
        exit(0);
    }
    else if (pid == 0) {
        close(STDOUT_FILENO);  // stdout close
        dup2(pipes[0][1], STDOUT_FILENO);   // 0번째 pipe stdout에 복사
        close(pipes[0][1]);    // pipe close
        execvp(args[pipe_idx[0]], &args[pipe_idx[0]]);  // exec
        printf("command not found | invalid option\n");
        exit(0);
    }
    close(pipes[0][1]);     // pipe close
    wait(&status);          // exec 종료까지 대기
```

여기 함수가 시작되고, 첫번째 명령어를 실행하는 부분이 있습니다. 먼저 pipes 이차원 배열을 선언해서 파이프로 사용합니다.

바로 pipe(pipes[0])을 통해서 처음에 쓸 파이프를 열어줍니다. 이후 자식 프로세스를 포크해주고, 자식 프로세스에서 표준 출력 fd를 닫아주고, dup2를 통해서 저희가 만든 파이프의 출력 부분에

복사해줍니다. 이후 파이프를 닫아주고, 프로그램을 실행시킵니다. pipe_idx[0] = 0입니다. 첫번째 명령어를 실행할 것이고, 결과가 파이프로 전해질 것입니다. 만약 프로그램이 제대로 실행되지 않았다면 밑부분의 코드가 실행되고 프로그램에서 강제로 나가게 합니다.

이후 부모 프로세스에서 안쓰는 출력 부분 파이프를 닫아주고 자식 프로세스가 끝나기를 기다립니다.

이제 첫번째 명령어가 실행이 완료되었고, 그 결과가 파이프를 통해 있을 것입니다.

```
/*****  마지막 명령어 제외 모두 실행  *****/
for (int i = 0; i < pipe_cnt - 1; i++) {
    pipe(pipes[i+1]);      // 파이프 생성
    if ((pid = fork()) < 0) {
        perror("Fork Error");
        exit(0);
    }
    else if (pid == 0) {
        close(STDIN_FILENO);  // stdin close
        close(STDOUT_FILENO); // stdout close
        dup2(pipes[i][0], STDIN_FILENO);    // pipe stdin에 복사
        dup2(pipes[i+1][1], STDOUT_FILENO); // pipe stdout에 복사
        close(pipes[i][0]);   // pipe close
        close(pipes[i+1][1]); // pipe close
        execvp(args[pipe_idx[i+1]], &args[pipe_idx[i+1]]);  // exec
        printf("command not found | invalid option\n");
        exit(0);
    }
    close(pipes[i+1][1]); // pipe close
    wait(&status);        // exec 종료까지 대기
}
```

이제 마지막 명령어를 제외하고 모두 순차적으로 실행시켜줍니다. i = 0부터니 사용하는 파이프는 1부터입니다. 파이프를 열어주고 똑같이 자식 프로세스를 포크해줍니다.

자식프로세스에서는 이전 결과에서 들어오는 결과값을 입력으로 사용할 것이기에 이번에는 표준 입력 fd도 닫아주고 복사를 해줍니다. dup2(pipes[i][0], STDIN_FILENO) 는 이전 명령에서의 파이프에 표준 입력 fd를 복사해주어, 이전 명령의 출력값을 가져옵니다.

그리고 dup2(pipes[i+1][1], STDOUT_FILENO) 을 통해 이번 명령어의 출력을 저장할 준비를 합니다. execvp를 통해서 프로그램을 실행시켜줍니다.

이후 부모 프로세스에서는 똑같이 파이프를 닫아주고, 자식 프로세스가 끝나기를 기다립니다.

이 과정에서 연속적으로 A | B | C ..... 가 진행될 것입니다.

```
/*****  마지막 명령어 실행  *****/
if ((pid = fork()) < 0) {
    perror("Fork Error");
    exit(0);
}
else if (pid == 0) {
    close(STDIN_FILENO);  // stdin close
    dup2(pipes[pipe_cnt-1][0], STDIN_FILENO); // pipe stdin에 복사
    close(pipes[pipe_cnt-1][0]);  // pipe close
    close(pipes[pipe_cnt-1][1]);  // pipe close
    execvp(args[pipe_idx[pipe_cnt]], &args[pipe_idx[pipe_cnt]]);  // exec
    printf("command not found | invalid option\n");
    exit(0);
}
close(pipes[pipe_cnt-1][1]);
if (background == 0)
    wait(&status);  // exec 종료까지 대기
else printf("background process\n");
return;
```

이제 마지막으로 마지막 명령어를 실행시킵니다. 앞으로 넘겨줄건 없으니, 출력 파이프는 닫아버리고, 입력 파이프를 이용해서 이전 명령어의 결과값을 가져오고 프로그램을 실행시킵니다.

정상적으로 실행되었다면, 밑의 코드가 실행이 되지 않았을 것이고 부모 프로세스는 파이프를 닫아주고 만약 백그라운드 모드면 자식을 기다리지 않고, 백그라운드 모드가 아니면 자식을 기다립니다.

이렇게 파이프의 모든 과정이 끝나게 됩니다.

```
my_shell> ls
a  proj  proj.c  test.txt
my_shell> ls | sort
a
proj
proj.c
test.txt
my_shell> ls | sort | grep p
proj
proj.c
my_shell> ls | sort | grep p &
background process
my_shell> proj
proj.c

my_shell>
```

이런식으로 실행이 잘 됩니다.

3. 기타 기능 추가

```
// 내부 명령어 cd 추가 구현
if (strcmp(args[0], "cd") == 0){
    if (chdir(args[1]) == -1){
        perror("cd");
        goto next_time;
    }
    else{
        goto next_time;
    }
}
```

cd로 인한 디렉토리 이동은 기본 쉘 구현으로는 구현이 되어있지 않기에 내부 명령어로 구현해보았습니다. 명령어로 cd가 주어진다면 chdir로 경로를 통해서 디렉토리를 이동합니다.

```
my_shell> pwd
/home/wollong/sysProj
my_shell> mkdir b
my_shell> ls
a  b  proj  proj.c  test.txt
my_shell> cd b
my_shell> pwd
/home/wollong/sysProj/b
my_shell> ls
my_shell>
```

이런식으로 작동이 잘 됩니다.

4. 테스트

이제 여러가지 테스트를 해보겠습니다.

```
my_shell> dcdcddg
command not found | invalid option
my_shell>
```

이상한 입력이면 이렇게 나오게 됩니다.

```
my_shell> ls
a  b  proj  proj.c  test.txt
my_shell> ls -l
total 40
drwxrwxr-x 2 wollong wollong  4096 12월  4 17:32 a
drwxrwxr-x 2 wollong wollong  4096 12월 11 23:22 b
-rwxrwxr-x 1 wollong wollong 17600 12월  5 21:17 proj
-rw-rw-r-- 1 wollong wollong  4881 12월 11 23:22 proj.c
-rw-rw-r-- 1 wollong wollong    23 12월  5 21:12 test.txt
my_shell>
```

기본적인 명령어들이 잘 됩니다.

```
my_shell> ls -l &
background process
my_shell> total 40
drwxrwxr-x 2 wollong wollong  4096 12월  4 17:32 a
drwxrwxr-x 2 wollong wollong  4096 12월 11 23:22 b
-rwxrwxr-x 1 wollong wollong 17600 12월  5 21:17 proj
-rw-rw-r-- 1 wollong wollong  4881 12월 11 23:22 proj.c
-rw-rw-r-- 1 wollong wollong    23 12월  5 21:12 test.txt

my_shell>
```

백그라운드 모드도 문제 없습니다.

```
my_shell> vi test.txt
my_shell>
```

파일 편집도 가능합니다.

또한 위에서 보았듯, cd와 파이프도 잘 작동합니다.

5. 느낀점

프로젝트를 진행하면서 직접 쉘을 구현하는 점이 꽤 재밌었습니다. 실제로 동작하는 것을 보니 뿌듯함이 들기도 했습니다.

기본 구현은 어려움 없이 금방 진행했지만, 파이프 구현은 시간이 꽤나 걸렸었고 그만큼 구현했을 때 아직 부족한 점은 많지만 보람을 느끼기도 했고 시스템 프로그래밍의 구조에 대해서 좀 더 잘 알 수 있는 계기가 되었습니다.