Project#1 Simple Shell

2020062324 이은비

1. 함수 설명

> **int** main(**void**)

: 메인 함수. 명령어를 입력받고 절차에 따라 함수를 실행한다.

> **void** tokenize_cmd(**char** *line, **char** *args[])

: 받은 명령어(line)를 파싱해 포인트 배열(args)에 저장하는 함수.

> **void** exec_cmd(**char** **args)

: 명령어를 실행하는 함수. (만약 파이프 또는 리다이렉션이 포함되었다면 아래 함수를 다시 실행한다)

> **void** pipe_cmd(**char** **args, **int** point_idx)

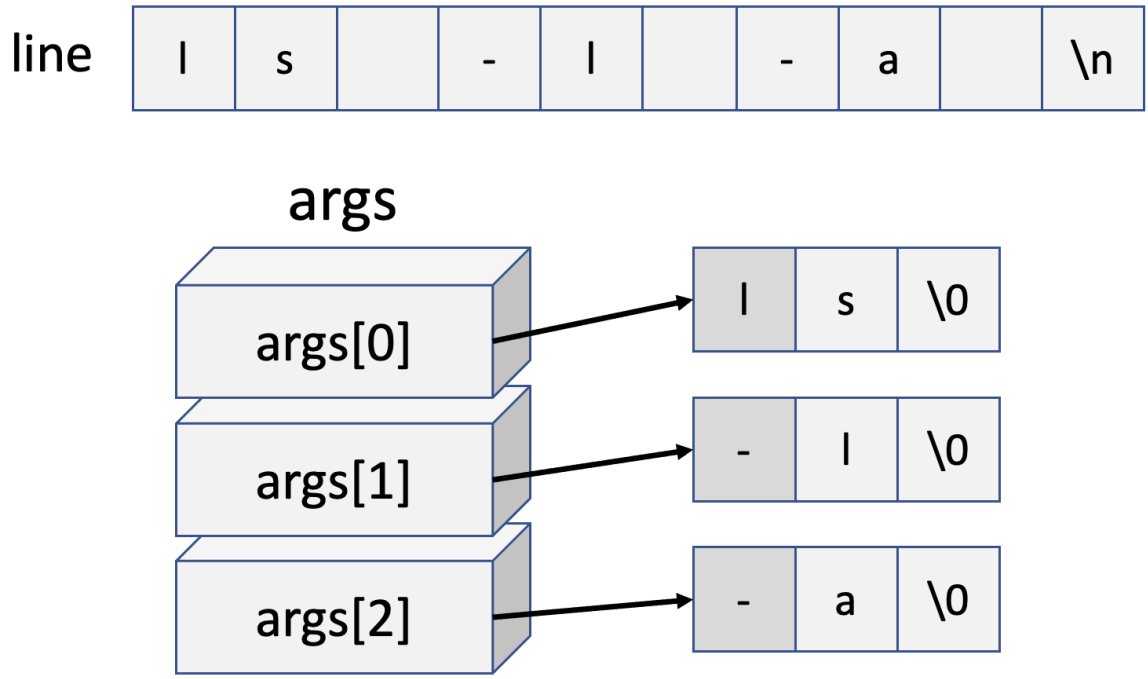
: 파이프가 있는 명령어를 실행하는 함수.

> **void** redirection_cmd(**char** **args, **int** point_idx, **int** is_left_redireciton)

: 리다이렉션이 있는 명령어를 실행하는 함수.

1) int main(void)

```
char  line[MAX_LINE];                   /* 명령어를 저장할 배열 */
char *args[MAX_LINE/2 + 1];             /* 명령어를 인수 단위로 저장할 배열 */
int should_run = 1;                     /* 프로그램의 종료를 나타낼 플래그 */
int has_bg = 0;                         /* '&'가 포함되어 있는지 나타낼 플래그 */
int i = 0;
```



ls -l -a 를 입력했을때 line, args의 모습

명령어를 받는 char 배열 line과 해당 명령어를 파싱하여 저장하는 args 변수를 선언한다.

```
while (should_run) {
    printf("osh>");

    fflush(stdout);                     /* 출력 버퍼를 비운다 */
    memset(line, 0, sizeof(line));      /* line 내에 남아 있는 데이터를 비운다 */


    scanf("%[^\n]", line);              /* 사용자에게 명령어를 입력 받는다 */
    getchar();
    line[strlen(line)] = '\n';

    tokenize_cmd(line, args);           /* 받은 명령어를 파싱한다 */
```

should_run 이 1일 동안 아래 해당 코드들을 반복한다.

사용자가 명령어(line)를 입력하기를 기다리고, 명령어를 받으면 line과 args 변수를 인자로한 tokenize_cmd 함수를 실행한다. 실행 후 line의 파싱 결과가 args에 저장된다.

```
    if (!args[0]) {                     /* 입력이 없을때 */
        continue;
    }

    if (strcmp(args[0], "exit") == 0) {     /* exit 명령을 받았을때 */
        should_run = 0;
        return 0;
    }

    while(args[i] != NULL) {
        i++;
    }
    if (strcmp(args[i-1], "&") == 0) {        /* '&'이 명령어에 포함되어 있는지 확인 */
        args[i-1] = NULL;
```

```
            has_bg = 1;
        }
```

만약 args가 공백(입력이 없다면)이라면 continue로 해당 while문의 처음으로 돌아간다. args가 공백이 아니고 첫번째 요소가 “exit”라면 should_run을 0으로 바꾸고 프로그램을 종료한다.

args의 요소가 NULL이 될때까지 i를 증가시켜 args의 마지막 index값을 알아낸다. 마지막 요소가 “&” 인지 조사하여 background 실행 명령어인지 확인한다.

```
        int pid;
        if ((pid = fork()) < 0) {           /* fork */
            fprintf(stderr, "Fork failed");
            return 1;
        }
        else if (pid == 0) {                /* 자식 프로세스인 경우 */
            exec_command(args);
        }
        else {                              /* 부모 프로세스인 경우 */
            if(!has_bg) {
                int status;
                waitpid(pid, &status, 0);   /* 백그라운드 실행이 아닐때, 실행이 끝나기를 기다림 */
                if (WIFSIGNALED(status)) {
                    fprintf(stderr, "Child exited by signal : %d\n", WTERMSIG(status));
                }
            }
            memset(args, 0, sizeof(args));
        }
    }
```

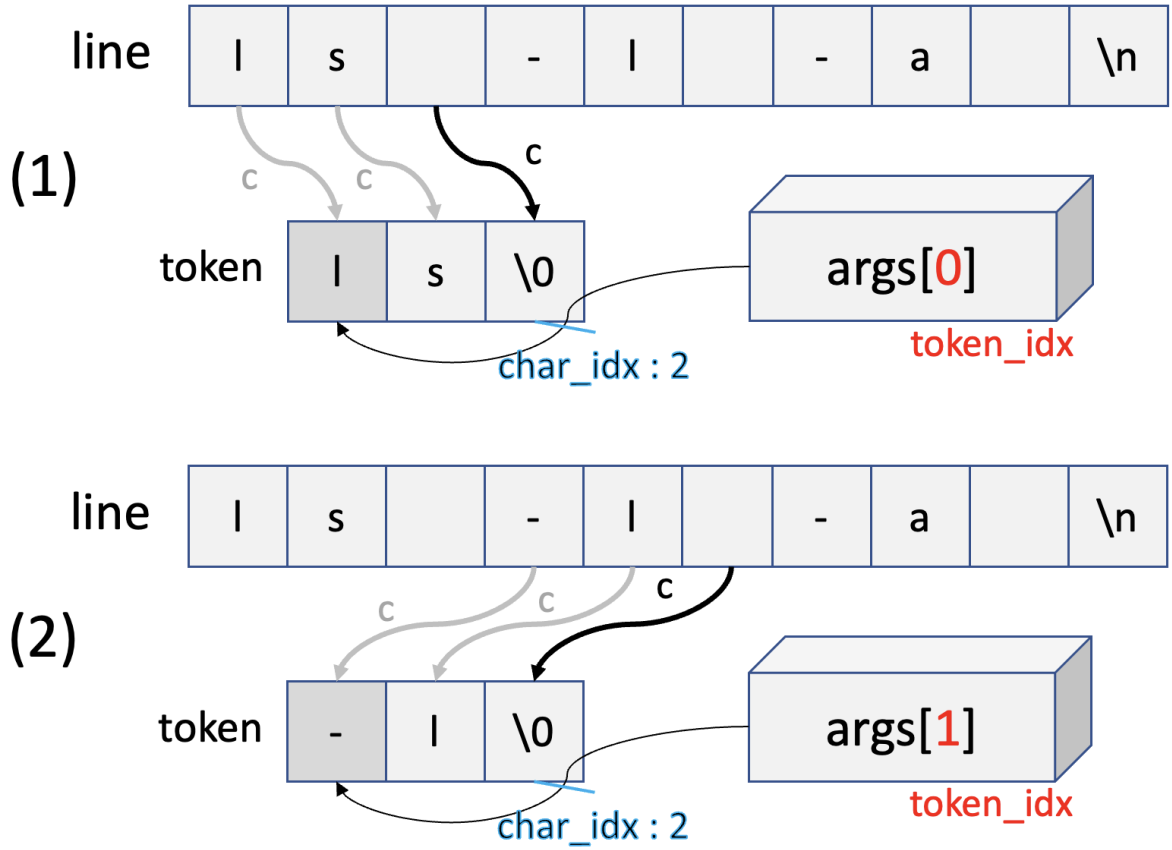
명령어를 실행하기 위해 fork로 자식 프로세스를 만든다. 만약 해당 프로세스가 부모 프로세스라면 앞서 조사한 background 여부에 따라 waitpid를 실행한다. 해당 프로세스가 자식 프로세스라면 args를 인자로 exec_command 함수를 실행한다.

해당 과정이 끝나면 다시 while문의 처음으로 돌아간다.

2) void tokenize_cmd(char *line, char *args[]) : 명령어 파싱 함수

```
    char *token = (char *)malloc(MAX_LINE * sizeof(char));  /* 공백 단위인 token을 저장할 char포인터 */
    int token_idx = 0;  /* token 개수에 관한 index */
    int char_idx = 0;   /* token 내의 char 개수에 관한 index */
```

```
    for (int i = 0; i < strlen(line); i++) {
        char c = line[i];
        if (c == ' ' || c == '\n' || c == '\0') {    /* 공백 문자일때 */
            if (char_idx != 0) {    /* token에 아무 것도 없지 않을때 */
                token[char_idx] = '\0';
                args[token_idx] = (char*)malloc(MAX_LINE * sizeof(char));   /* 다음 token을 저장하기 위한 char포인터 공간 할당 */
                strcpy(args[token_idx++], token);   /* token을 args에 저장 */
                char_idx = 0;   /* 새로운 token을 위해 idx 초기화 */
            }
        }
        else {  /* 공백 문자가 아닐때 */
            token[char_idx++] = c;  /* 해당 문자를 token에 저장 */
        }
    }
```



받은 명령어 line을 차례대로 읽으면서 해당 문자가 공백 문자가 아닌경우 token에 저장하고 공백 문자인 경우 해당 token을 args 요소로 할당한다.[그림(1)] 그리고 token을 초기화 시킨 후 다시 line을 읽는다. 다음 token은 할당된 args 요소의 다음 요소로 할당된다.[그림(2)]

3) void exec_cmd(char **args) : 명령어를 실행하는 함수

```
int has_pipe = 0;               /* 파이프 유무 플래그 */
int has_redirection = 0;        /* 리다이렉션 유무 플래그 */
int is_left_redireciton = 0;    /* 리다이렉션 방향 플래그 */
int point_idx = 0;              /* 파이프 또는 리다이렉션 위치 */
```

명령어 중 pipe(|), redirection(</>)이 있는지 확인하는 플래그, 해당 문자가 어디에 위치하는지 저장하는 int형 변수와 redirection 종류를 확인하는 변수를 선언, 초기화한다.

```
for (int i = 0; args[i] != NULL; i++) {
    if (strcmp(args[i], "|") == 0){    /* 파이프가 있다면 */
        args[i] = NULL;             /* | -> NULL로 수정, index 저장 */
        point_idx = i;
        has_pipe = 1;
    }
    else if (strcmp(args[i], "<") == 0 || strcmp(args[i], ">") == 0) {  /* 리다이렉션이 있다면 */
        is_left_redireciton = strcmp(args[i], ">") ? 1 : 0;     /* '<': 1, '>': 0 */
        args[i] = NULL;                                         /* </> -> NULL로 수정, index 저장 */
        point_idx = i;
        has_redirection = 1;
    }
}
```

args를 처음부터 읽으며 해당 문자가 “|”라면 해당 index 위치를 저장하고 NULL로 초기화시킨후 플래그를 수정한다.

해당 문자가 “<” 또는 “>”라면 redirection의 종류를 확인 후, 해당 index 위치를 저장하고 NULL로 초기화시킨후 플래그를 수정한다.

```
    if (has_pipe) { /* 파이프가 있다면 */
        pipe_cmd(args, point_idx);
    }
    else if (has_redirection) { /* 리다이렉션이 있다면 */
        redirection_cmd(args, point_idx, is_left_redireciton);
    }
    else{  /* 파이프, 리다이렉션이 없을때 */
        execvp(args[0], args);

        int status;
        wait(&status);
        if (WIFSIGNALED(status)) {
            exit(1);
        }
    }
```

문자에 “|”, “<”, “>”가 포함되어 있는지 확인후, 만약 “|”가 있다면 args와 point_idx를 인자로 하는 pipe_cmd 함수를 실행하고, “<” 또는 “>”가 있다면 args와 point_idx, is_left_redirection을 인자로 하는 redirection_cmd 함수를 실행한다.

만약 세 문자 모두 해당되지 않는다면 execvp를 통해 명령어를 바로 실행한다.

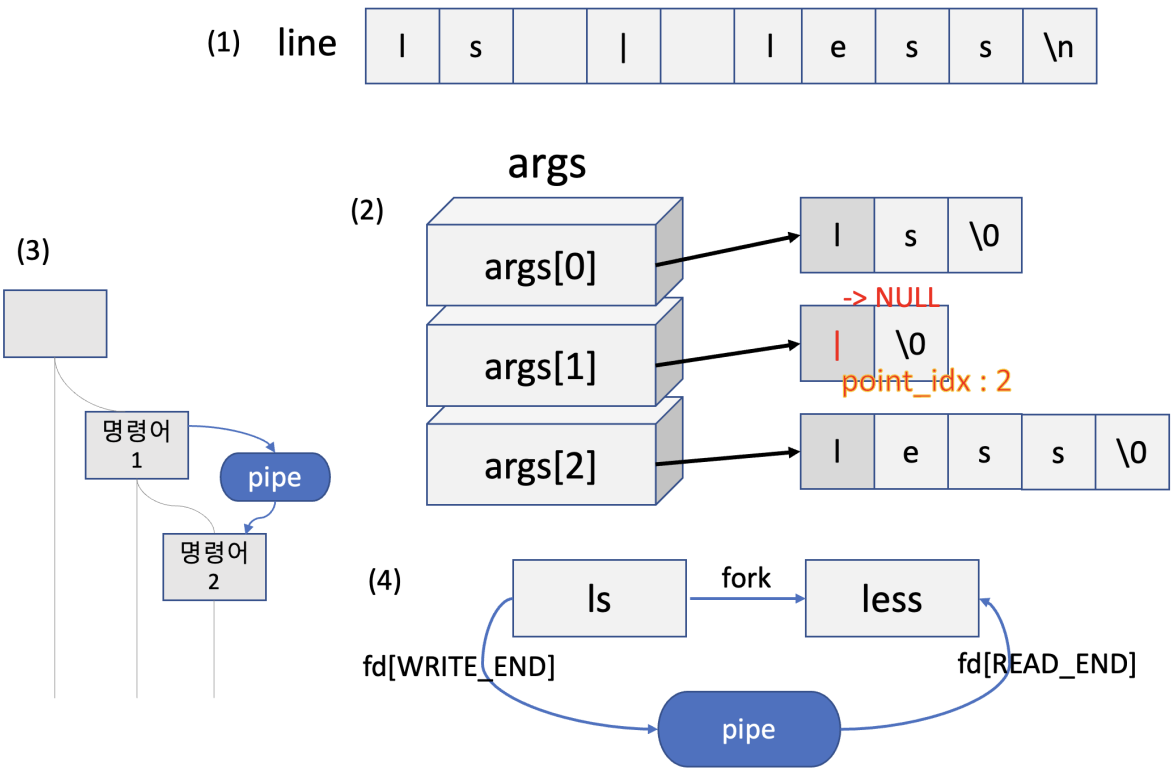
4) void pipe_cmd(char **args, int point_idx)

```
    int fd[2];
    int pid;
    int status;

    pipe(fd);    /* pipe create */
```

파이프(fd)로 사용할 변수를 만들고 파이프를 생성한다.

```
if ((pid1 = fork()) < 0) {
        fprintf(stderr, "Fork failed");
        exit(1);
    }
    else if (pid1 == 0) {
        close(fd[READ_END]);    /* READ pipe close */
        dup2(fd[WRITE_END], STDOUT_FILENO); /* standard output이 fd에 복사됨 */
        close(fd[WRITE_END]);   /* WRITE pipe close */

        execvp(args[0], &args[0]);  /* 명령어1 실행 */
    }
    else {
        if ((pid2 = fork()) < 0) {
            fprintf(stderr, "Fork failed");
            exit(1);
        }
        else if (pid2 == 0) {
            close(fd[WRITE_END]);   /* WRITE pipe close */
            dup2(fd[READ_END], STDIN_FILENO);   /* standard input에 대해서 fd에 복사됨 */
            close(fd[READ_END]);    /* READ pipe close */

            execvp(args[point_idx+1], &args[point_idx+1]);  /* 명령어2 실행 */
        }
        else {
            close(fd[READ_END]);
            close(fd[WRITE_END]);
            wait(NULL);
```

```
    }
}
```

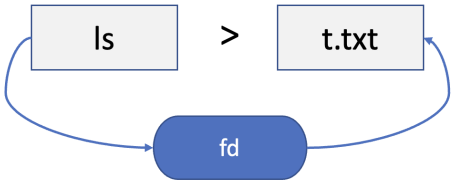


[그림(1)]의 명령어 line을 파싱한 최종 args의 형태가 [그림(2)]이다. [그림(3)]의 흐름을 따라 fork로 명령어1(ls)을 실행하고 해당 결과를 pipe를 통해 자식 프로세스인 명령어2(less)에게 input으로 전달한다. 명령어2는 받은 input을 통해 명령어를 수행한다.

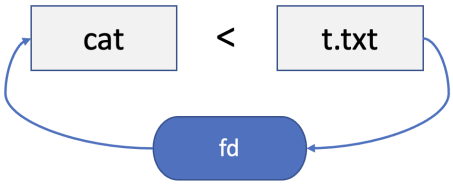
5) void redirection_cmd(char **args, int point_idx, int is_left_redireciton)

```
int fd;
int pid;
int status;

if ((pid = fork()) < 0) {               /* fork */
    fprintf(stderr, "Fork failed");
    exit(1);
}
else if (pid == 0) {                    /* 자식 프로세스인 경우 */
    fd = open(args[point_idx+1], O_RDWR | O_CREAT);  /* 읽기, 쓰기가 가능한 파일을 열기 */
    if (fd < 0) { perror ("Open error"); exit(-1); }

    is_left_redireciton ? dup2(fd, STDIN_FILENO) : dup2(fd, STDOUT_FILENO); /* '<' - standard input이 fd에 복사됨, '>' -
    close(fd);  /* pipe close */
    execvp(args[0], &args[0]);
}
wait(&status);
if (WIFSIGNALED(status)) {
    exit(1);
}
```

(1)



(2)



open을 통해 읽기/쓰기가 가능한 파일을 fd로 지정한다. (없으면 생성) 만약 “<” 리다이렉션이라면 모든 input이 fd에 복사되고, “>”이라면 모든 output이 fd에 복사된다. 전자의 경우(<) 파일의 내용이 복사되고, 후자의 경우(>) 명령어 실행 후 나온 output이 파일에 복사된다.

2. 컴파일 과정 (화면 캡쳐)

```
(base) eunbilee@eunbiui-MacBookPro 제출 % gcc osh.c
(base) eunbilee@eunbiui-MacBookPro 제출 % ./a.out
osh>
```

3. 실행 결과와 설명

a. **초기 화면**

```
(base) eunbilee@eunbiui-MacBookPro 제출 % ./a.out
osh>
```

프로그램 실행시 osh> 이 출력되고 입력을 기다리며 커서가 깜빡거린다.

b. **명령어**

i. 명령어 + 옵션

```
osh>ls -l
total 88
-rwxr-xr-x  1 eunbilee  staff  34754  3 31 15:36 a.out
-rw-rw-r--@ 1 eunbilee  staff   7164  3 31 15:32 osh.c
```

`ls -l` 명령어는 해당 디렉토리의 자세한 내용을 출력하는 명령어이다. 위는 `ls -l` 명령어를 실행하여 결과가 나온 모습이다.

c. **redirection**

i. 명령어 + 옵션 > 파일명

```
osh>ls -l > file.txt
```

`ls -l > file.txt` 명령어는 해당 디렉토리의 자세한 내용을 file.txt 파일에 출력하는 명령어이다. 위는 `ls -l > file.txt` 명령어를 실행하여 결과가 나온 모습이다. 자세한 결과는 아래 내용을 참고한다.

ii. 명령어 + 옵션 < 파일명

```
osh>cat -n < file.txt
     1  total 88
     2  -rwxr-xr-x  1 eunbilee  staff  34754  3 31 15:36 a.out
     3  -rwx--x---  1 eunbilee  staff      0  3 31 15:46 file.txt
     4  -rw-rw-r--@ 1 eunbilee  staff   7164  3 31 15:32 osh.c
```

cat -n < file.txt 명령어는 file.txt 파일의 내용을 표준 입력으로 사용해 앞의 명령어를 실행한다. 여기서는 cat -n 이므로 앞에 라인 번호를 단 텍스트 파일의 내용을 출력한다. 위는 cat -n < file.txt 명령어를 실행하여 결과가 나온 모습이다. 앞선 명령어를 통해 만든 file.txt 파일의 내용을 확인할 수 있다.

d. **pipe**

i. 명령어1 + 옵션 | 명령어2 + 옵션

```
osh>du -a | sort -n
8       ./file.txt
16      ./.DS_Store
16      ./osh.c
72      ./a.out
112     .
```

du -a | sort -n 명령어는 두개의 명령어로 이루어져있다. 첫번째 명령어 du -a 는 모든 파일의 디렉토리 용량 정보를 출력하는 명령어이고 sort -n 은 숫자를 비교해 텍스트를 정렬하는 명령어이다. 명령어 파이프라인을 통해 모든 파일의 디렉토리 용량 정보가 숫자를 기준으로 정렬되어 출력된다. 위는 du -a | sort -n 명령어를 실행하여 결과가 나온 모습이다.

e. **exit**

```
osh>exit
(base) eunbilee@eunbiui-MacBookPro 제출 %
```

해당 프로세스(simple shell)를 종료하는 명령어이다.

아래는 위에 대한 전체적인 흐름을 나타낸 결과이다.

```
(base) eunbilee@eunbiui-MacBookPro 제출 % ./a.out
osh>ls -l
total 88
-rwxr-xr-x  1 eunbilee  staff  34754  3 31 15:36 a.out
-rw-rw-r--@ 1 eunbilee  staff   7164  3 31 15:32 osh.c
osh>ls -l > file.txt
osh>cat -n < file.txt
     1  total 88
     2  -rwxr-xr-x  1 eunbilee  staff  34754  3 31 15:36 a.out
     3  -rwx--x---  1 eunbilee  staff      0  3 31 16:07 file.txt
     4  -rw-rw-r--@ 1 eunbilee  staff   7164  3 31 15:32 osh.c
osh>du -a | sort -n
8       ./file.txt
16      ./.DS_Store
16      ./osh.c
72      ./a.out
112     .
osh>exit
(base) eunbilee@eunbiui-MacBookPro 제출 %
```

4. 문제점과 느낀점

1) 문제점

- ‘&’ 백그라운드 프로세스

```
if(!has_bg) {
    int status;
    waitpid(pid, &status, 0);   /* 백그라운드 실행이 아닐때, 실행이 끝나기를 기다림 */
    if (WIFSIGNALED(status)) {
        fprintf(stderr, "Child exited by signal : %d\n", WTERMSIG(status));
    }
}
```

위 코드처럼 '&' 문자 유무에 따라 부모 프로세스가 자식 프로세스를 기다리는지에 대한 결정을 한다.

```
(base) eunbilee@eunbiui-MacBookPro 제출  % ./a.out
osh>sleep 3 &
osh>ls
zsh: segmentation fault  ./a.out
```

하지만 sleep 3 & 실행 후 (sleep은 백그라운드에서 돌아가고 있는 상태.) ls 명령어를 입력하면 segmentation fault 오류가 발생한다.

위 코드는 메인 함수에 있는 부모 프로세스 한 번만 적용되기 때문에, exec_cmd 함수에도 해당 조건을 적용하면 같은 예제에 대한 결과가 아래와 같이 나온다.

```
osh>sleep 3
osh>ps
  PID TTY           TIME CMD
73299 ttys000    0:00.24 -zsh
74924 ttys000    0:00.01 ./a.out
74950 ttys000    0:00.00 ./a.out
74951 ttys000    0:00.00 cat -n
78281 ttys000    0:00.01 ./a.out
78286 ttys000    0:00.00 (sleep)
osh>sleep 6 &
osh>usage: sleep seconds
ps
  PID TTY           TIME CMD
73299 ttys000    0:00.24 -zsh
74924 ttys000    0:00.01 ./a.out
74950 ttys000    0:00.00 ./a.out
74951 ttys000    0:00.00 cat -n
78281 ttys000    0:00.01 ./a.out
78286 ttys000    0:00.00 (sleep)
78319 ttys000    0:00.00 (sleep)
```

sleep 6 & 실행시 sleep 뒤의 인자가 읽혀지지 않은듯한 출력이 나타난다. (usage: sleep seconds). 명령어 토큰을 제대로 읽지 못하고, segmentation 오류가 발생하므로 args에 저장된 메모리를 접근할때 생기는 문제점인듯하다.

```
osh>ps
  PID TTY           TIME CMD
79117 ttys000    0:00.08 -zsh
79447 ttys000    0:00.01 ./a.out
osh>sleep 2
osh>ps
zsh: segmentation fault  ./a.out
```

```
osh>ls -l
total 96
-rwxr-xr-x  1 eunbilee  staff  34754  3 31 17:34 a.out
-rwx--x---  1 eunbilee  staff    177  3 31 16:07 file.txt
-rw-rw-r--@ 1 eunbilee  staff   7156  3 31 17:26 osh.c
osh>ls
zsh: segmentation fault  ./a.out
```

추가로 sleep 2 명령어 실행 후 ps 명령어를 실행해보고, ls -l 명령어 후 ls 명령어를 실행해보았다. 두 케이스는 앞선 명령어보다 args 길이가 적다. 따라서 이전에 남은 메모리에 대한 접근에 대한 오류라 판단하고 수정을 진행해보았지만 성공하지 못했다.

2) 느낀점

- 자식 프로세스와 부모 프로세스

: 각 명령어 실행을 위해 fork 한 자식 프로세스와 부모 프로세스 간의 흐름을 파악할 수 있었다. 또한 명령어 실행을 하나의 프로세스로 생각하고 다루는 방법을 배울 수 있었다.

- 프로세스 간 파이프 통신

  : 부모와 자식 프로세스 간의 통신을 관여하는 파이프에 대한 개념과 사용법을 익힐 수 있었다. 파이프는 부모 프로세스와 자식 프로세스의 통신을 위한 채널이다. 파이프는 양방향, 단방향 통신이 가능하다. 위 과제에서는 단방향 통신 파이프를 사용하였다. 단방향 파이프에서는 해당 자식 또는 부모 프로세스가 사용하지 않는 쪽을 꼭 닫아줘야 한다.

- 시스템콜과 리눅스 명령어

  : 리눅스 명령어가 어색했지만, 작성한 쉘의 테스트를 진행하면서 리눅스 명령어를 많이 접해볼 수 있었다. 리눅스 명령어에 대해 어려운 인식을 지니고 있었는데 과제를 진행하면서 해당 인식을 벗어날 수 있게 되었다.

  또한 시스템콜을 작성을 통해 사용법을 익힐 수 있었다.

- 포인터

  : 명령어를 파싱하고 실행하기 위해 포인터 개념이 적용되었다. 배열의 해당 포인터가 가리키는 값에 대한 수정을 위한 방법에 대해 배우게 되었다. 하지만 메모리 할당과 접근에 대해 미숙하여 발생한 오류를 쉽게 처리하지 못해 포인터에 대한 학습이 더 필요하다고 느끼게 되었다.