

Cykor_2

이번 과제의 경우 상당히 막막하여 우선 현재의 디렉토리명을 셸에 표시하는 첫번째 조건부터 마지막 조건까지 만족해 나갔습니다. `getcwd`를 활용하여 표시했는데 이 코드를 작성할 때 순간 현재 디렉토리명을 표시하는 것으로 받아들여 코드를 짰고 에러가 나는 경우 에러를 표시하도록 코드를 짰습니다. 이때 `PATH_MAX`라는 최대 경로의 길이가 있어 그대로 사용하였습니다. 이제 메인 함수에서 `grtline`함수를 사용하기 위해 버퍼 포인터와 변수를 준비하여 `while`문 안에서 아까 만든 함수를 호출하여 현재 디렉토리가 출력이 되도록 했고 우선 명령어에 반응은 안 하더라도 명령어는 쳐질 수 있도록 `getline`명령어를 사용을 하였습니다. 또한 엔터로 입력을 하므로 개행 문자를 `'\0'`으로 변경하여 명령어만 들어가도록 하였습니다. 빈 입력의 경우 다시 맨위로 돌아가도록 하였고 빈칸이 아니라면 입력이 잘들어가는지 확인하기 위해 입력을 다시 출력하도록 하였습니다. 마지막에 `getline`에서 `line`에 할당을 하였으므로 `free`를 해주었습니다.

```
#include <stdio.h>
#include <unistd.h>
#include <limits.h>
#include <libgen.h>
#include <string.h>
#include <stdlib.h>

void print_prompt() {
    char cwd[PATH_MAX];
    if (getcwd(cwd, sizeof(cwd))) {
        printf("%s$ ", basename(cwd));
    } else {
        perror("getcwd");
    }
    fflush(stdout);
}

int main(void) {
    char *line = NULL;
    size_t len = 0;

    while (1) {
```

```

    print_prompt();
    if (getline(&line, &len, stdin) == -1) {
        printf("\n");
        break;
    }

    if (line[strlen(line)-1] == '\n')
        line[strlen(line)-1] = '\0';

    if (strlen(line) == 0)
        continue;

    printf("%s\n", line);
}

free(line);
return 0;
}

```

이제 일반적인 리눅스 bash 셸의 사용자 인터페이스 형태를 띄는 조건을 수행하려다가 전체를 보이고

A terminal window showing a custom prompt. The prompt is enclosed in square brackets and contains the text 'chk_pass@BOOK-4GNLV3MI3H:~' followed by a dollar sign '\$'. The text is color-coded: 'chk_pass' is red, '@BOOK-4GNLV3MI3H' is green, and ':~\$' is white.

와 같은 형태로 출력하도록 만들어야 한다는 것을 알았고 이 부분은 뭔가 재미있어 보여 이후에 수행하기로 하고 넘어갔습니다.

이제 3번째 cd와 pwd명령어를 구현하였는데 우선 strtok와 "\t"를 사용하여 공백이거나 탭의 경우에 구분자로 지정을 하였습니다. 이렇게 되면 구분자 다음을 포인터로 가르키게 되어 구분자의 역할을 잘 할 수 있게 됩니다. 이제 token이 0이 아니고 argc < MAX_ARGS - 1일 때 얻어낸 주소를 배열에 저장하고 다시 strtok함수에 NULL을 넘겨 다시 찾고를 반복하는데 argv[0]부터 argv[argc-1]가 토큰이 됩니다. 이제 NULL로 배열을 마무리하는데 이전에 argc < MAX_ARGS - 1로 한 이유가 NULL을 넣을 공간을 만들기 위해서 입니다. cd의 경우 처음에 빈칸으로 넣으면 HOME 디렉토리로 가게 했는데 이후에 ~를 넣어도 HOME으로 가게 하였습니다. pwd는 pwd를 입력하면 getcwd함수로 현재 경로를 출력하도록 하였습니다. 이 둘도 오류가 나는 경우 오류를 출력하도록 하였습니다. main함수는 개행을

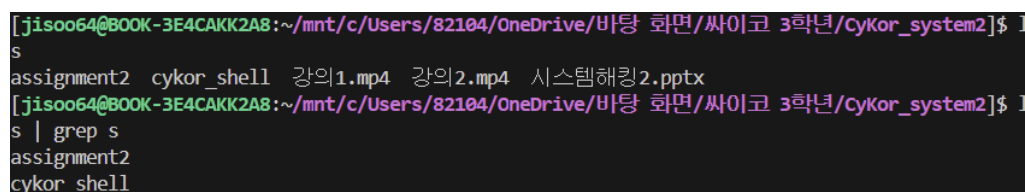
'\0'으로 바꾸는 형태만 바꾸었고 7번 조건인 exit입력시 프로그램을 종료하는 것도 추가시켰습니다. 이후에 다른 명령어들도 실행이 가능하도록 execvp를 사용할 예정입니다.

다음으로 파이프라인을 만들었습니다. 이제 한 명령어에 들어갈 수 있는 인자의 개수가 아닌 파이프 라인의 경우 파이프로 연결이 가능한 다른 변수가 필요할 것으로 생각되어 이전처럼 토큰을 생성해 줄 것인데 이번에는 '|' 로 구분자를 만들어 나누도록 하고 나머지는 같습니다. (이때 변수를 좀 잘 설정했다라면 뒤에 구분자가 섞여 문제가 일어나지 않아 시간이 단축이 됐을것으로 예상이 됩니다. 이때 ncmd == 0인 경우에는 그냥 반복이 되고, ncmd == 1인 경우 단일 명령이므로 위에서 실행이 되었던 것처럼 그냥 그대로 넣어주었습니다. 그러나 이제 그 이상인 경우 ncmd개의 명령어 연결을 위해 파이프는 ncmd-1개가 필요하므로 파이프를 생성하는데 pipes[i][0]는 읽기, pipes[i][1]는 쓰기를 위한 것이고

```
int pipes[MAX_CMDS - 1][2];
for (int i = 0; i < ncmd - 1; i++) {
    pipe(pipes[i]);
}
```

와 같이 파이프를 생성하였습니다. 이제 각각의 명령을 다시 공백과 탭으로 구분하여 토큰화를 해주었고, 이제 fork를 통해 자식 프로세스를 생성을 하였습니다. 이제 예를 들어 3개의 명령이 있는 경우 입력은 그대로, 출력은 pipes[0][1]를 복제하고 다음 명령은 입력을 이전 파이프에서 읽어오고, 출력을 다음 파이프로 보내줍니다. 마지막으로 입력은 받아오지만 출력은 터미널로 보이도록 해 줍니다. 더 늘어날 경우 가운데는 동일하고 처음과 마지막 파이프라인만 터미널에서 입력을 받고 터미널로 출력을 한다는 것만 주의하여 만들었습니다. 이제 dup2를 하고 남은 원본 파이프 fd가 남아있을 수 있으므로 부모와 자식 프로세스에서 파이프를 모두 닫아줍니다. 이제 명령어를 실행하고 부모에서 파이프를 모두 닫는데 자식 프로세스들이 모두 종료가 될 때까지 기려주도록 waitpid를 사용하였습니다. 이를 통해 부모는 사라졌는데 자식은 존재하는 좀비 프로세스를 방지하였습니다.

아래는 파이프라인의 실행 모습입니다.



```
[jisoo64@BOOK-3E4CAKK2A8:~/mnt/c/Users/82104/OneDrive/바탕 화면/싸이코 3학년/CyKor_system2]$ s
assignment2 cykor_shell 강의1.mp4 강의2.mp4 시스템해킹2.pptx
[jisoo64@BOOK-3E4CAKK2A8:~/mnt/c/Users/82104/OneDrive/바탕 화면/싸이코 3학년/CyKor_system2]$ s | grep s
assignment2
cykor_shell
```

이제 &&, ||, ;를 실행하는 코드를 만드려 하는데 생각보다 잘 안되고 우선 echo가 안되어 검증이 잘 안되어

```
pid_t pid = fork();
if (pid < 0) {
```

```

        perror("fork");
        return 1;
    }
    if (pid == 0) {
        execvp(argv[0], argv);
        perror(argv[0]);
        exit(EXIT_FAILURE);
    } else {
        int st;
        waitpid(pid, &st, 0);
        return WEXITSTATUS(st);
    }
}

```

를 통해 cd나 pwd가 아닌 경우 실행이 가능하도록 시스템콜을 사용해주었습니다. 그 후에 프롬프트 출력도 다음과 같이 구현하였습니다.

```

void print_prompt() {
    char user[30];
    char host[50];
    char cwd[PATH_MAX];
    char buf[PATH_MAX + 80 + 100];

    struct passwd *pw = getpwuid(getuid());
    if (pw) {
        strncpy(user, pw->pw_name, sizeof(user));
    } else {
        strcpy(user, "unknown");
    }

    if (gethostname(host, sizeof(host)) < 0) {
        strcpy(host, "unknown");
    }

    if (!getcwd(cwd, sizeof(cwd))) {
        strcpy(cwd, "unknown");
    }

    char super = (getuid() == 0) ? '#' : '$';
    snprintf(buf, sizeof(buf), "[%01e[1;32m\002%s@%s\001e[0m\002:~\001e

```

```
fputs(buf, stdout);
fflush(stdout);
}
```

이 코드는 주어진 ppt에 나와있던 renkangchen의 코드를 참고하였으며, user, host의 버퍼크기를 만들었고 super, 그러니까 root인 경우, sudo 명령어로 실행하면 #이, 평소에는 \$가 뜨도록 만들었으며 색을 변형하는 방법도 알아내어 색 또한 변경하였습니다. 버퍼의 크기는 색의 생성을 위해 넉넉하게 잡았습니다.

그 후에 false && echo 2가 출력이 된다거나 true || cd ..이 실행이 된다거나 하는 문제점의 이유를 알아낼 수 있었고 그 이유는 return을 대부분 0으로 해놓아서 반환값이 0이므로 제대로 이전의 명령어가 실행되었는지 아닌지 알 수 없었기 때문입니다. 따라서 전부 정상적인 경우 0, 에러의 경우 1을 반환하도록 하였고

```
[jisoo64@BOOK-3E4CAKK2A8:~/mnt/c/Users/82104/OneDrive/바탕 화면/싸이코 3학년/CyKor_system2/assignment2]$ false || cd ..
[jisoo64@BOOK-3E4CAKK2A8:~/mnt/c/Users/82104/OneDrive/바탕 화면/싸이코 3학년/CyKor_system2]$ true && cd ..
[jisoo64@BOOK-3E4CAKK2A8:~/mnt/c/Users/82104/OneDrive/바탕 화면/싸이코 3학년]$ true || cd ..
[jisoo64@BOOK-3E4CAKK2A8:~/mnt/c/Users/82104/OneDrive/바탕 화면/싸이코 3학년]$ false && cd ..
[jisoo64@BOOK-3E4CAKK2A8:~/mnt/c/Users/82104/OneDrive/바탕 화면/싸이코 3학년]$
```

```
[jisoo64@BOOK-3E4CAKK2A8:~/mnt/c/Users/82104/OneDrive/바탕 화면/싸이코 3학년]$ echo 1; echo 2
; echo 3
1
2
3
```

다음과 같이 잘 수행이 됩니다.

이 실행 코드의 경우 우선 이전의 구분자를 저장할 수 있는 sep를 만들어놓았고 이때 최대길이 &&2개에 널문자 까지 길이는 3으로 하였습니다. 이제 명령의 상태를 저장하기 위한 status도 만들어 놓았습니다. 다음으로 while문을 통해 다음 구분자를 찾을 것인데 이때는 found를 만들었는데 sep와 동일하게 길이는 3으로 하였습니다. 이때 next는 첫 구분자의 시작주소를 가르키게 했고 구분자가 &&인지, ||인지, ;인지 인식합니다. 이때 만약 next가 NULL이 아니라면, 즉 구분자를 찾은 상태라면 이 주소에 널 함수를 넣어줍니다. 이제 구분자에 따른 실행여부를 넣어주었습니다. 이렇게 다음 구분자가 없을 때까지 반복해주고 len을 모두 써놓았으므로 포인터가 그대로 구분자를 뛰어넘어 다음 명령어를 가리키도록 작성하였습니다.

마지막으로 백그라운드를 적용시키는데 여기에서 위에 말했던 오류가 터졌습니다. 구분자 구분 없이 진행하여 sleep 5 &를 입력하면 &는 메인에서 제거가 되는 것이 맞지만 공백이 영향을 미쳐서 sleep에서 필요한 인자가 없다고 나오는데 다른 인자가 필요한 명령어들, ls -l같은 명령어를 입력해도 똑같이 ls명령어와 동일하게 실행되는 것을 보고 문제를 느껴 각

strtok함수들을 strtok_r함수를 사용하여 포인터에 저장을 하였고 이렇게 구분이 되도록 구분자들을 설정하여주었습니다.

백그라운드를 위해 메인 함수를 변경하였는데 위까지는 이전과 같고

```
while (l > 0 && isspace((unsigned char)line[l-1])) {  
    line[--l] = '\0';  
}
```

를 사용해 공백 제거를 해주었습니다. 기본적으로 백그라운드는 false로 해두었으나 만약 맨 끝이 &인데 이것이 &&가 아닌 경우에 true로 바꾸어주고 &를 제거해주는 함수를 작성해주었고 이 앞의 공백 또한 isspace를 통해 제거하였습니다. 이제 백그라운드가 아니라면 이전과 동일하게 실행이 되도록 하였고, 백그라운드의 경우 fork를 해주고 자식 프로세스는 명령 실행후에 바로 exit을 하고 부모는 pid만 보여주게 됩니다. 다만 여기서 자식이 좀비 프로세스가 되는 위험이 있는데

```
void reap_children(int signum) {  
    int status;  
    pid_t pid;  
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {  
        printf("child %d terminated\n", pid);  
    }  
}
```

를 사용하여 방지하였습니다.

```
[jisoo64@BOOK-3E4CAKK2A8:~/mnt/c/Users/82104/OneDrive/바탕 화면/싸이코 3학년/cykor_system2/assignment2]$ sleep 5 &  
[377030]  
[jisoo64@BOOK-3E4CAKK2A8:~/mnt/c/Users/82104/OneDrive/바탕 화면/싸이코 3학년/cykor_system2/assignment2]$ sleep 4 &  
[377104]  
[jisoo64@BOOK-3E4CAKK2A8:~/mnt/c/Users/82104/OneDrive/바탕 화면/싸이코 3학년/cykor_system2/assignment2]$ child 377030 terminated  
child 377104 terminated
```