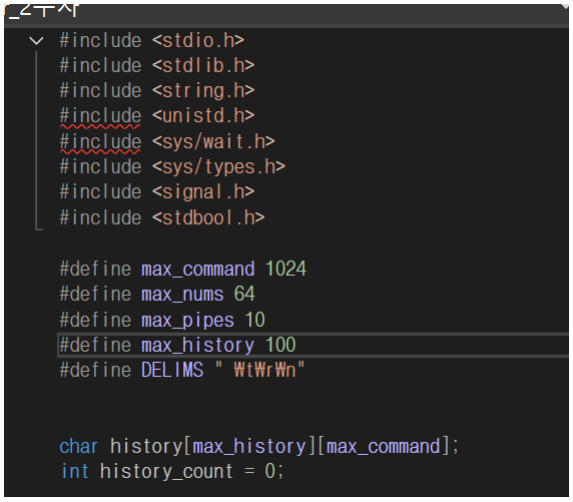
Cykor 과제 2주차 보고서

2024350223_원정재

이번 과제는 주어진 조건들을 만족하는 리눅스 쉘 프로그램 구현하기입니다. 저는 각각의 조건들을 visual studio 상에서 개별 함수들로 표현하였습니다.(물론 개별함수들 간에도 계층적으로 큰 함수 안에 작은 함수들이 구현됩니다. 과제의 조건들을 살펴보니 코드에 자주 반복될 일반적인 부분 그리고 다중 같은 특수한 상황등으로 코드들을 쪼개어 볼 수 있을 것같아서 크게 로드맵을 짜고 각각의 함수를 계층적으로 구현하여 다른함수에서 함수를 이용할 수 있게 만들었습니다) 먼저 위쪽에 부분적인 코드 사진을 보고 아래쪽에 그에 대한 설명을 하는 방식으로 각각의 조건마다 각각의 함수들을 하나씩 차례대로 자세하게 살펴보는 식으로 보고서를 작성하였습니다.



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <signal.h>
#include <stdbool.h>

#define max_command 1024
#define max_nums 64
#define max_pipes 10
#define max_history 100
#define DELIMS " WtWrWn"


char history[max_history][max_command];
int history_count = 0;
```

```c
int token(char* str, char* tokens[]);
void cd(char* args[]);
void pwd();
void pipeline(char* commands[], int cmd_count);
void command(char* tokens[], bool background, int* last_status);
void handle_multi_command(char* line);
void split_commands(char* line, char* commands[], char* operators[]);
void handle_history();

int last_command_status = 0;
```

우선 어떤 헤더파일을 왜 #include 했는지 그리고 어떤 전역변수가 왜 필요했는지는 아래의 개별함수들을 다루면서 그 함수들을 만나면 설명하도록 하겠습니다. 각각의 개별함수의 코드의 길이가 꽤 되므로 함수의 프로토타입을 먼저 선언하고 main함수부터 코드를 뜯어서 설명해보겠습니다.

```
int main() {
    char cmd[max_command];

    while (1) {

        char cwd[1024];
        if (getcwd(cwd, sizeof(cwd)) != NULL) {
            printf("₩033[1;32m%s₩033[0m$ ", cwd);
        }
        else {
            perror("getcwd() error");
            exit(EXIT_FAILURE);
        }
```

Main() 함수 내부에서 첫번째로 만나볼 의미블록 단위는 getcwd 함수를 이용한 현재 디렉토리 표시 입니다. 이를 위해 <unistd.h> 헤더파일이 필요했습니다. 그리고 일반적인 리눅스 bash쉘의 사용자 인터페이스 형태를 띄도록 하기 위해서 printf("₩033[1;32m%s₩033[0m$ ", cwd); 이렇게 표현했습니다 .이를 하나씩 뜯어보면 ₩033은 뒤에 오는 제어 시퀀스를 해석합니다 [는 시작을 의미하며 1과 32를 통해 일반적인 bash쉘의 진한 녹색글씨를 나타내었고 m을 통해 끝을 나타내어 지금 %s 자리에 오는 cwd가 굵은 녹색 글씨로 나타나게 됩니다. 하지만 다시 ₩033부터 m 이 나타나는데 이는 흔히 bash 쉘에서 $ 기호는 가장 기본 스타일로 표현되기 때문입니다. 그리고 <stdio.h>에 들어있는 perror 함수를 통해 오류 발생시 오류메세지를 출력하도록 하였습니다.

```
if (!fgets(cmd, sizeof(cmd), stdin)) break;

if (strcmp(cmd, "exit₩n") == 0) break;
```

이제는 위에서 정의한 main의 cmd지역변수에 명령어를 입력하는 부분입니다. Fgets 함수를 이용해 사용자의 표준입력인 stdin으로부터 입력을 받아 이를 cmd 버퍼에 받았습니다. 또한 일반적인 프로그램처럼 사용자가 exit()을 입력하면 쉘이 종료되도록 하였습니다.(물론 fgets 는 사용자의 엔터도 버퍼에 입력하므로 이를 포함하였습니다.

```
if (cmd[0] != '₩n') {
    if (history_count < max_history) {
        strncpy(history[history_count++], cmd, max_command - 1);
    }
    else {

        for (int i = 1; i < max_history; i++)
            strcpy(history[i - 1], history[i]);
        strncpy(history[max_history - 1], cmd, max_command - 1);
    }
}


if (strncmp(cmd, "history", 7) == 0 &&
    (cmd[7] == '₩n' || cmd[7] == ' ' || cmd[7] == '₩0')) {
    handle_history();
    continue;
}
```

이번 부분은 과제의 필수조건에는 해당하지 않으나 따로 간단하게 구현해볼 명령어를 생각하다가 history 명령어를 간단하게 구현해보았습니다. 그리고 그를 위한 main() 부분입니다.(나중에 history를 다루는 함수는 뒤에 있습니다.) 우선 위쪽 코드블록은 history 변수에 명령어들을 저장하는 부분인데 일반적인 리눅스 bash 쉘에서 명령어를 치지않고 단순히 엔터만 누르면 다음줄에 새로운 프롬프트창이 계속 뜨는데 이는 history 내부에 저장이 되지 않도록 if 분기 시작을 엔터가 아닐 때 라고 했고 <string.h> 헤더에 들어있는 strncpy 함수를 이용해 최대 명령어 길이보다 하나 작은(끝에는 널문자가 들어가야하기 때문에) 길이 만큼의 명령어를 history 2차원 배열에 넣었습니다. 이때 history[0] 부터 들어가도록 하기 위해 history의 개수를 세는 history_count 변수에 뒤쪽에 ++을 붙이는 후위증가연산자를 연산자 우선순위를 고려하여 사용하였습니다. 그리고 혹시 몰라서 만약 history 의 개수가 최대로 생각한 history의 개수보다 클 경우 제일 오래전 history요소인 history[0]이 삭제되도록 그 위에 계속 다음 history 요소를 덮어씌우고 나머지 history 요소들은 1칸씩 왼쪽으로 평행이동 되도록 하는 코드를 추가했습니다.

또한 <string.h>에 들어있는 strncmp 함수를 이용해 사용자가 프롬프트 창에 history 명령어와 그 뒤에 올수있는 모든 경우의 수 를 and 연산시켜서 history 명령어가 실행되도록 하였습니다.

```
        handle_multi_command(cmd);
    }
    return 0;
}
```

그리고 cmd를 위에서 받은 후 이를 처리하는 가장 큰 단위의 다중, 단일 모든 명령어 처리 함수입니다. 이는 내부의 세세한 구현함수들부터 전부 살펴본 후 다시 다루어 보겠습니다.

여기까지가 main() 함수의 의미블록단위마다 쪼개서 자세히 살펴본 결과입니다. 이제는

본격적으로 개별 구체적인 함수블록의 내부를 살펴보도록 하겠습니다.

```
int token(char* str, char* tokens[]) {
    int count = 0;
    char* token = strtok(str, DELIMS);

    while (token && count < max_nums - 1) {
        tokens[count++] = token;
        token = strtok(NULL, DELIMS);
    }
    tokens[count] = NULL;
    return count;
}
```

이번에 살펴볼 부분은 명령어 처리에서 굉장히 중요한 부분입니다. 왜냐하면 우리가 한 줄의 프롬프트에 명령어를 치는 것이 결국 전부라고 할 수 있는데 이 명령어가 단순히 1개가 아니라 옵션이 붙을 수도 있고 다중명령어가 될 수도있고 파이프라인, 멀티파이프라인 등 이번 과제의 조건에서 구현해야하는 것만 해도 엄청나게 중요합니다. 그래서 우선 긴 명령어의 각 부분을 각각의 토큰으로 쪼개는 함수를 구현해보았습니다. 이 함수는 <string.h>에 들어있는 strtok() 함수가 거의 모든 역할을 하고 나머지 변수들은 단지 strtok()가 처리한 토큰들을 저장하는 역할의 변수들입니다. 이 함수는 우선 긴 문자열과 토큰 배열을 매개변수로 입력받고 strtok() 함수를 통해 전체토큰의 개수를 반환해줍니다. **이 함수의 코드 자체는 그냥 단순하지만 이번과제를 하면서 strtok() 함수가 처음에만 긴 문자열을 파라미터로 입력받고 함수 내부의 '다음 토큰을 어디서부터 찾아야하는지'를 기억하는 내부 정적변수가 들어있어서 처음 이후에는 NULL을 파라미터로 넘겨서 다음 토큰의 포인포인포인터주 반환받는다는 것 그리고 두번째 파라미터로 넘겨주는 구분자 집합(DELIMS)를 만나면 이를 ₩0으로 바꾸고 토큰을 구분하면서 마지막에 구분자가 더 이상 없을 때는 NULL을 마지막 토큰으로 인식한다는 것이 신기했습니다.**

```
void cd(char* args[]) {
    if (args[1] == NULL) {
        chdir(getenv("HOME"));
    }
    else {
        if (chdir(args[1]) != 0) {
            perror("cd error");
            last_command_status = 1;
        }
        else {
            last_command_status = 0;
        }
    }
}
```

이번에는 쉘 내부에서 내부명령어로 구현해야하는 2가지 명령어 중 하나인 cd 명령어에 대해 다루는 함수이다. 우선 입력값으로 명령어들의 포인터 배열을 받는다, 첫번째 배열의 요소는 cd 일 것이므로 우리가 확인할 건 두번째 배열요소 즉 args[1]인데 만약 args[1]이 비어있다면 즉 사용자가 cd 만 쳤을경우 원래 리눅스 bash 쉘처럼 /home 디렉토리로 이동해야 하므로 <stdlib.h> 에 있는 getenv 함수를 이용해 환경변수의 경로를

가져오고 이를 <unistd.h> 헤더에 있는 chdir 함수를 통해 현재 프로세스의 작업 디렉토리를 변경한다. 이번엔 args[1]에 어떤 값이 있는 경우 chdir 함수를 통해 디렉토리를 변경하는데 이때 만약 제대로 동작하지않아서 0이 아닌 다른값을 반환하는 경우 오류 메세지를 출력한다. 그리고 last_command_status 라는 변수가 있는데 이는

```
void pwd() {
    char cwd[1024];
    if (getcwd(cwd, sizeof(cwd)) != NULL) {
        printf("%s\n", cwd);
        last_command_status = 0;
    }
    else {
        perror("pwd error");
        last_command_status = 1;
    }
}
```

이번에는 내부명령어 2번째인 pwd 명령어에 대해 다루는 함수이다. 이 부분은 위에서 main() 함수에서 사용하던 현재 디렉토리를 표시하는 getcwd 함수를 사용하면되기 때문에 간단하다. 단 이후를 위해 last_command_state 변수가 성공시 0 실패시 1을 가지도록 하였다.

```
void pipeline(char* commands[], int cmd_count) {
    int pipes[max_pipes][2];
    pid_t pids[max_pipes];
    int i, j;

    for (i = 0; i < cmd_count - 1; i++) {
        if (pipe(pipes[i]) == -1) {
            perror("pipe error");
            return;
        }
    }

    for (i = 0; i < cmd_count; i++) {
        pids[i] = fork();
        if (pids[i] == 0) {

            if (i > 0) {
                dup2(pipes[i - 1][0], STDIN_FILENO);
                close(pipes[i - 1][0]);
                close(pipes[i - 1][1]);
            }
            if (i < cmd_count - 1) {
                dup2(pipes[i][1], STDOUT_FILENO);
                close(pipes[i][0]);
                close(pipes[i][1]);
            }
```

```
            for (j = 0; j < cmd_count - 1; j++) {
                close(pipes[j][0]);
                close(pipes[j][1]);
            }

            char* args[max_nums];
            token(commands[i], args);
            execvp(args[0], args);
            perror("execvp error");
            exit(EXIT_FAILURE);
        }
    }


    for (i = 0; i < cmd_count - 1; i++) {
        close(pipes[i][0]);
        close(pipes[i][1]);
    }


    for (i = 0; i < cmd_count; i++) {
        waitpid(pids[i], NULL, 0);
    }
}
```

이번에 구현할 함수는 조건의 파이프라인 관련한 함수인데 변수들부터 차분히 살펴보면 각각의 파이프는 파이프 양쪽으로 읽기/쓰기 2개의 파일 디스크립터가 필요합니다, 즉 요소를 2개씩 가지는 파일 디스크립터 저장변수 입니다. 그게 pipes 라는 변수이고 나중에 fork() 함수를 부를 때를 위해 프로세스 ID를 저장할 배열로 pids 변수를 선언했습니다. 우선 첫번째 반복문은 파이프를 생성하는 반복문입니다.(당연히 파이프의 개수는 명령어의 개수-1입니다). 이는 <unistd.h> 헤더에 있는 pipe() 함수를 이용합니다. 그리고 두번째 반복문에서는 각각의 명령어마다 자식프로세스를 생성하여 실행하였습니다. 이때 <unistd.h>에 있는 fork() 함수를 사용했습니다. **헷갈렸던 부분이 바로 pid [i] = fork() 였는데 이때 분명 pid변수는 하나만 사전에 정의했는데 어떻게 부모프로세스의 반환값과 자식 프로세스의 반환값이 분명 다른데 pid 변수에 각각 저장될 수 있는가가 이상했었는데 Fork() 함수는 원리적으로 독립적인 새로운 프로세스가 만들어지는 것이기 때문에 메모리에서 pid 변수 마저 새로 복사된다는 것을 알게 되었습니다.** 이제 큰 if 분기에서 pid[i] == 0을 통해 자식 프로세스들만 보겠습니다. 이제 입력 리다이렉션을 살펴보겠습니다. 그 바로 뒤에 있는 if 문에서 i>0을 보았는데 이는 첫번째 명령어(i==0) 이외에 두번째 명령어 부터는 두번째 명령어의 입력을 파이프로부터 받아야 한다는 것을 의미합니다. 이를 위해 <unistd.h> 에 있는 dup2() 함수를 사용하였습니다. 즉 프로세스가 시작할 때 기본적으로 할당되는 파일 디스크립터를 이전 파이프의 읽기 디스크립터로 변경해야 한다는 것입니다. 그 후 close 함수를 통해 더 이상 사용하지 않는건 닫아주었습니다.

이제 출력 리다이렉션을 보겠습니다. 지금 if <층_count -1을 통해 마지막 명령어가 아닐 경우 출력을 파이프로 보내야합니다 그걸 다시 dup2 함수로 구현하였습니다. 그리고 원하는대로 복제가 끝났으니 이제 사용하지 않는 파이프들은 닫아주고 드디어 명령어 실행 부분입니다. 코드는 함수 2개가 전부고 이는 선배님의 강의설명이랑 똑같은데 여기서는

위에서 구현했던 토큰으로 명령어들을 나눠주는 함수를 실행한 후 <unistd.h> 에 있는 execvp() 함수를 실행했습니다. 그후 자원 누수 방지를 위해 부모 프로세스에서 안 쓰는 파이프를 닫아주고 <sys/wait.h> 헤더에 들어있는 waipid() 함수에 파라미터로 자신의 자식 프로세스 pid 를 넘겨서 자식프로세스가 끝날때까지 대기 하도록 만들었습니다.

```
/mnt/c/Users/rudy0/source/repos/Cykor_2주차/Cykor_2주차$ echo Hello
Hello
/mnt/c/Users/rudy0/source/repos/Cykor_2주차/Cykor_2주차$ ls | echo Hello
Hello
```

```
Try 'tr --help' for more information.
/mnt/c/Users/rudy0/source/repos/Cykor_2주차/Cykor_2주차$ echo hello world | tr 'a-z' 'A-Z'
HELLO WORLD
/mnt/c/Users/rudy0/source/repos/Cykor_2주차/Cykor_2주차$ echo hello world | tr 'a-z' 'A-Z' | rev
DLROW OLLEH
/mnt/c/Users/rudy0/source/repos/Cykor_2주차/Cykor_2주차$
```

추가로 지금 이걸 실행해보니 처음에 당황했었지만 명령어에 따라 파이프라인(멀티 파이프라인)이 안되는 경우도 있다는걸 알았다.

```
void command(char* tokens[], bool background, int* last_status) {
    int token_count = 0;
    while (tokens[token_count] != NULL) token_count++;
    if (token_count > 0 && strcmp(tokens[token_count - 1], "&") == 0) {
        tokens[token_count - 1] = NULL;
        background = true;
    }
    if (strcmp(tokens[0], "cd") == 0) {
        cd(tokens);
        return;
    }
    if (strcmp(tokens[0], "pwd") == 0) {
        pwd();
        return;
    }
    if (strcmp(tokens[0], "history") == 0) {
        handle_history();
        return;
    }
    pid_t pid = fork();
    if (pid == 0) {
        execvp(tokens[0], tokens);
        perror("execvp error");
        exit(EXIT_FAILURE);
    }
```

```
    }
    else if (pid > 0) {
        if (!background) {
            int status;
            waitpid(pid, &status, 0);
            *last_status = WIFEXITED(status) ? WEXITSTATUS(status) : 1;
        }
        else {
            printf("[백그라운드 PID: %d]₩n", pid);
            signal(SIGCHLD, SIG_IGN);
            *last_status = 0;
        }
    }
    else {
        perror("fork error");
        *last_status = 1;
    }
}
```

이번에는 입력받은 명령어를 실행하는 함수입니다. 당연히 명령어 토큰 배열을 인자로 받았고 지금 조건으로 백그라운드 실행이 있었기 때문에 이를 마치 플래그 처럼 백그라운드 실행 여부를 받을 변수를 받았고 마지막으로 명령 상태 저장 포인터까지 받았습니다. 우선 가장 먼저 백그라운드 실행여부를 확인하기 위해 tokens 를 처음부터 끝까지 한번 개수를 세주었고 strcmp로 마지막 & 여부를 확인했습니다.

이제 본격적으로 명령어 처리인데 가장 먼저 cd 처리는 전에 설정한 cd 함수를 불렀고 마찬가지로 pwd 함수도 pwd 함수를 불렀고 제가 직접 구현한 history 함수도 history 함수를 불렀습니다. 이제 그 외의 외부명령어들인데 이미 execvp 함수의 첫번째 파라미터가 환경변수 path 를 찾고 이것은 이미 운영체제에 환경변수 path 설정이 되어있기 때문에 fork함수로 부모,자식 프로세스를 불러주고 execvp 함수만을 호출해주면 끝났습니다. **이제 이 부분이 굉장히 중요한데 백그라운드를 신경써줄 필요가 없는 경우 위에서 한 것처럼 waitpid() 함수로 부모프로세스는 자식프로세스가 끝날때까지 기다리면 됩니다. 하지만 만약 실행하는 자식프로세스가 백그라운드 실행이라면 부모 프로세스는 자식을 기다리지 않고 바로 명령을 받습니다. 찾아보니 좀비 프로세스란 자식프로세스는 종료되었지만 부모 프로세스가 이를 확인하지 않아 프로세스 테이블에 남아있는 상태를 의미합니다. 특히 프로세스 ID를 점유하여 리소스 낭비를 일으킨다고 합니다. 그래서 signal(SIGCHLD, SIG_IGN) 부분을 추가해주었습니다. 이는 커널이 자동으로 자식의 종료 상태를 회수해서 좀비 프로세스가 생성되지 않도록 합니다.**

```c
void split_commands(char* line, char* commands[], char* operators[]) {
    int i = 0, j = 0, k = 0;
    while (line[i] != '\0') {
        if ((line[i] == '&' && line[i + 1] == '&') ||
            (line[i] == '|' && line[i + 1] == '|')) {
            line[i] = '\0';
            commands[k] = &line[j];
            operators[k] = (line[i + 1] == '&') ? "&&" : "||";
            k++;
            i += 2;
            j = i;
        }
        else if (line[i] == ';') {
            line[i] = '\0';
            commands[k] = &line[j];
            operators[k] = ";";
            k++;
            i++;
            j = i;
        }
        else {
            i++;
        }
    }
```

```c
    if (j < i) {
        commands[k] = &line[j];
        operators[k] = NULL;
        k++;
    }
    commands[k] = NULL;
    operators[k] = NULL;
```

이번에는 다중 명령어와 관련된 함수를 살펴보겠습니다. 앞에서 명령어 실행 함수에 대해서는 이미 짜두었기 때문에 이 부분은 말 그대로 명령어부분과 연산자 부분을 인식해서 구별하고 이들을 각각 명령어 목록과 연산자 목록에 저장하는 부분을 따로 떼둔 것입니다. 먼저 중요한 역할을 하는 3가지 변수에 대해 설명하겠습니다 ,각각 전체라인 상에서 현재 처리 중인 문자 위치를 나타내는 부분, 명령어와 연산자를 구분하는데 있어서 명령어의 시작위치, 그리고 명령어목록과 연산자 목록의 인덱스를 담당하는 변수 이렇게 3가지 입니다.

당연히 마지막 널 문자를 만나기 전까지 계속 while 문을 돌리는 전체 구조 안에서 먼저 2개가 연속해서 나오는 &&와 || 연산자인 경우를 분기하였고 1개만 단독으로 나오는 ; 의 경우도 위와 똑같이 코드를 짜주고 위에서 언급한 3개의 중요한 변수를 조정해주었습니다. 만약 연산자가 아닌 그냥 일반 문자라면 그냥 오른쪽으로 이동합니다. 그리고 최종적으로 더 이상 연산자는 없지만 마지막 명령어가 남는 경우 그때의 조건은 쉽게 j < I 임을 알 수 있기 때문에 저장해줍니다.

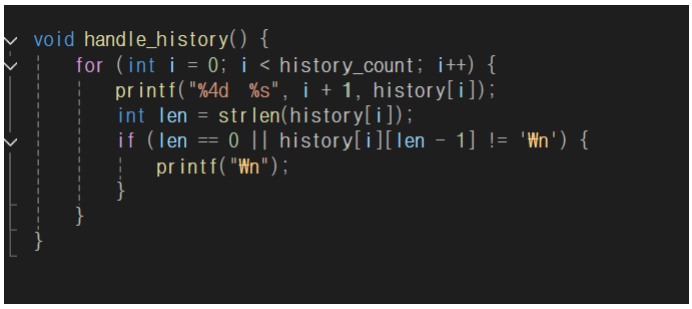
```
void handle_multi_command(char* line) {
    char* commands[max_nums];
    char* operators[max_nums];
    split_commands(line, commands, operators);

    for (int i = 0; commands[i] != NULL; i++) {

        if (strstr(commands[i], "|") != NULL) {
            char* pipe_commands[max_pipes + 1];
            int pipe_count = 0;
            char* token = strtok(commands[i], "|");
            while (token != NULL && pipe_count < max_pipes) {
                pipe_commands[pipe_count++] = token;
                token = strtok(NULL, "|");
            }
            pipe_commands[pipe_count] = NULL;
            pipeline(pipe_commands, pipe_count);
            last_command_status = 0;
            continue;
        }
```

```
        char* args[max_nums];
        int arg_count = token(commands[i], args);
        if (arg_count == 0) continue;
        if (i == 0 || (operators[i - 1] && strcmp(operators[i - 1], ";") == 0)) {
            command(args, false, &last_command_status);
        }
        else if (operators[i - 1] && strcmp(operators[i - 1], "&&") == 0) {
            if (last_command_status == 0)
                command(args, false, &last_command_status);
        }
        else if (operators[i - 1] && strcmp(operators[i - 1], "||") == 0) {
            if (last_command_status != 0)
                command(args, false, &last_command_status);
        }
    }
```

드디어 main() 함수의 마지막 의미블록에서 봤던 최종 집합판인 명령어를 다루는 함수입니다. 앞에서 만들었던 모든 함수들을 내부에서 사용할 것입니다. 가장 먼저 위에서 정의한 명령어와 연산자를 분리해주는 함수를 호출하여 명령어 목록과 연산자 목록을 각각 만들었습니다. 연산자를 처리했으니 명령어에는 파이프를 포함하거나 포함하지 않거나 하는 명령어들이 각각 하나씩 나뉘어져 들어가있을 겁니다. 이제 <string.h> 헤더에 들어있는 strstr 함수를 이용해 명령어 중 파이프 명령어가 있는 명령어들을 분리해내보았습니다. 그리고 이들 하위 명령어들을 저장할 배열을 선언하였고 이를 위에서 사용하였던 strtok 함수를 통해 전부 분리해서 위에서 선언한 배열에 저장하였습니다. 그 다음 이 항목을 파이프라인 명령어 실행 함수를 호출해 인자로 넘겼습니다. 그 다음 이제 파이프 부분은 처리가 끝났고 일반 명령어와 분리해놓은 연산자를 실행하는 부분으로 갔습니다.

즉 일반 명령어에서는 위에서 정의했던 토큰으로 나누어주는 함수를 호출하여 문자열을 각각 토큰화 시켰습니다. 이제 마지막으로 연산자에 따라 코드 실행 부분을 분기할건데 그동안 앞에 코드들에서 이후에 쓰임이 있다고 했던 이전 명령어의 성공,실패여부를 나타내던 변수가 &&와 || 같은 조건부 연산자 때문에 조건 분기문에 쓰입니다. 가장 먼저 첫번째 명령어이거나 그냥 무조건 실행하는 ; 연산자의 경우는 그냥 명령어 실행함수를 호출하고 &&연산자의 경우는 이전명령어가 성공(0)일때만 다음명령어를 실행하고 ||연산

자는 이전명령어가 실패(0이아닐 때)에만 다음 명령어를 실행합니다. **(참고로 지금 명령어 실행함수에서 백그라운드 여부를 나타내는 인자에는 그냥 false 를 항상 디폴트로 넘겨도 됩니다. 왜냐하면 이 실행함수에서 첫 의미블록 부분에서 자체적으로 백그라운드 여부를 확인하고 &가 있다면 True로 돌리기 때문입니다.)**



```
void handle_history() {
    for (int i = 0; i < history_count; i++) {
        printf("%4d  %s", i + 1, history[i]);
        int len = strlen(history[i]);
        if (len == 0 || history[i][len - 1] != '₩n') {
            printf("₩n");
        }
    }
}
```

이 함수는 제가 입력한 명령어들의 히스토리를 출력하는 함수입니다. 아까 main함수에서 저장한 history 배열들을 기록대로 출력하는 간단한 반복문 함수일 뿐입니다. 물론 2가지 정도를 살펴보자면 %4d를 이용해 이쁘게 history 명령어들을 4칸 오른쪽 정렬로 출력합니다. 그리고 fgets 에서 처음에 읽어 낼 때 이미 입력 끝에 ₩n (엔터)가 포함되어있어서 맨 처음에는 넣지 않았으나 생각해보니 다중명령어 같은경우엔에는 엔터가 없는 경우도 있을 것이므로 이를 고려하여 엔터 여부를 확인하여 history를 출력하도록 다시 수정하였다.