

Cykor 2week 과제

1. 헤더파일

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

unistd.h, pwd.h 는 쉘에서의 프로세스 실행에 관련해 필요한 함수들 fork(), execvp() 등, 을 사용하기 위해서 가져왔다. 또한

sys/types,와 sys/wait는 각각 pid를 선언하기 위해, 부모 프로세스가 자식프로세스를 기다리기 위해 필요한 wait, waitpid함수를 사용하기 위해서 가져왔다. 나머지 stdio.h, string.h, stdlib함수는 문자열, 기본적인 입출력 함수를 사용하기 위해서이다.

2. 매크로

```
#define MAX_INPUT_SIZE 1024
#define MAX_TOKEN_SIZE 64
#define HOST_NAME_MAX 64
#define PATH_MAX 1024
#define TOKEN_CNT 64
```

INPUT SIZE 명령어 입력

TOKEN_SIZE 토큰 사이즈

HOST_NAME_MAX 인터페이스 구현에 필요한 HOSTNAME 사이즈

PATH_MAX 인터페이스 구현에 필요한 경로 사이즈

TOKEN_CNT 토큰 수

3. tokenize 함수

```
char** tokenize(char* line, int* token_count) //입력한 문자열을 공백단위로 나눠 토큰화 한 후 return하는 함수 -> 이러면 입력을 배열로 받아야 함.
{
    char** tokens = malloc(sizeof(char*) * MAX_TOKEN_SIZE);
    char buffer[MAX_TOKEN_SIZE];
    int tokenindex = 0, charindex = 0;

    for (int i = 0; ; i++) {
        char c = line[i];
        if (c == ' ' || c == '\n' || c == '\t' || c == '\0') {
            if (charindex > 0) {
                buffer[charindex] = '\0';
                tokens[tokenindex] = strdup(buffer);
                tokenindex++;
                charindex = 0;
            }
            if (c == '\0') break;
        }
        else {
            buffer[charindex++] = c;
        }
    }
    tokens[tokenindex] = NULL;
    *token_count = tokenindex;
    return tokens;
}
```

일단 셸에서 명령어를 인식 어떻게 인식을 하는지 생각해보니 우리는 ls -al와 같이 공백을 기준으로 명령어, 조건, 경로 순으로 입력을 하게 된다. 이것을 어떻게 구현을 할지 구글링을 통해 다른 사람들이 만든 셸 구현 코드를 보니 토큰을 통해 우리가 입력한 명령어를 쪼개서 (토큰화) 하여 배열처럼 저장하여 execvp 함수로 실행되도록 구현을 하였다. 그럼 여기서 execvp 함수에 알아보았더니.

```
int execvp(const char *file, char *const argv[]);
```

이렇게 첫번째 인자는 명령어, 두번째 인자는 명령어와 인자를 포함한 리스트이면 된다.

그러면 이중 배열 즉 list[0]="명령어" list[1]="인자".... 이렇게 토큰화해서 저장한 후 execvp(list[0], list) 이런식으로 실행을 하면 되겠다. token 배열을 선언하고 token_index와 charindex를 선언한 후 line(입력값)을 순회하면서 공백이 나오기 전까지는 buffer에 문자 하나씩 담으면서 charindex를 증가시키다가, 공백을 만난 경우, token에 buffer에 담긴 문자열을 복사한 후, token_index를 증가시키고 다시 charindex 0으로 초기화 하는 구조로 코드를 짰다. 그리고 마지막 문자열에 NULL을 추가해줘야 한다. (execvp 함수가 NULL 문자열 전까지만 인식하므로)

그 다음으로는 내장 함수 pwd, cd를 구현하였다.

4.pwd 함수

내장함수의 특징은 셸에 직접적인 영향을 끼친다는 점으로 그래서 자식프로세스를 따로 만들지 않고 부모 프로세스에서 바로 실행한다. 그 중 pwd는 현재 디렉토리를 출력해주는 함수이다.

```
void pwd(void)
{
    char cwd[PATH_MAX];
    if (getcwd(cwd, sizeof(cwd)) != 0) {
        printf("%s\n", cwd);
    }
    else {
        printf("Error getting current directory\n");
    }
}
```

pwd.h에 있는 현재 디렉토리 경로를 가져오는 함수인 getcwd를 사용해서 구현하였다.

5. cd 함수

cd 함수는 디렉토리를 이동하는 경로이다. chat gpt한테 물어보니 chdir함수를 사용해야한다는 것을 알게 되었다.

```

int cd(char* path) {
    static char prev_cwd[PATH_MAX] = "";
    char curr_path[PATH_MAX];
    // 현재 경로 저장
    if (getcwd(curr_path, sizeof(curr_path)) == NULL) {
        perror("getcwd");
        return -1;
    }

    if (path == NULL || strcmp(path, "") == 0 || strcmp(path, "~") == 0) { //cd, cd ~ -> 홈 디렉토리로 이동하는 명령어 구현
        path = getenv("HOME");
        if (path == NULL) {
            fprintf(stderr, "cd: HOME not set\n");
            return -1;
        }
    }
    //cd - 구현
    else if (strcmp(path, "-") == 0)
    {
        if (strlen(prev_cwd) == 0)
        {
            fprintf(stderr, "cd: OLDPWD not set\n");
            return -1;
        }
        path = prev_cwd;
        printf("%s\n", path);
    }

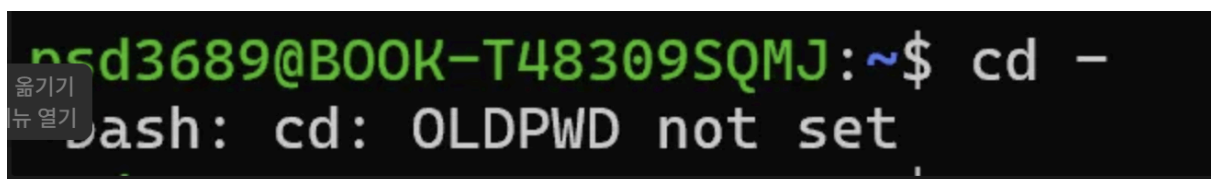
    if (chdir(path) != 0) {
        perror("cd");
        return -1;
    }

    strcpy(prev_cwd, curr_path); //다음 cd호출 때는 curr_path가 이전 경로가 되기 때문에 갱신
    return 0;
}

```

처음에 getcwd를 사용해서 현재 경로를 저장하는데 이는 cd - 를 구현하기 위해서이다. chdir 함수의 인자는 cd 다음에 오는 인자인데 ex) cd - 이면 "-" 이 때 -는 인지하지 못하는 문제가 생겨서 따로 구현을 해야한다. 그 이유는 찾아보니 -는 symbol이 아니라 그냥 쉘 문법이기 때문에 인식하지 못한다고 한다. 그리고 cd만 입력 할 때 cd ~ 와 같은 경우도 마찬가지로 home 디렉토리로 이동하는 것인데, 인식을 하지않아 따로 구현해야 한다. 그래서 if문으로 분기를 통해 cd, cd~,cd- cd 경로 일 경우 를 나눠서 코드를 구현하였다. 이때 home 디렉토리로 이동하는 경우 getenv() 함수를 사용했는데 이 함수는 환경변수를 특정 symbol로 입력하는 것인데

여기서는 HOME을 인자로 저장하게 했다. 그리고 만약 cd - 일 경우, 이전 경로로 갱신하여 이전 경로로 이동하게 하였다. 여기서 든 의문점은 쉘 명령어 처음으로 cd - 을 할 경우 이전 경로로 들어간다고 하면 코드상 현재 경로로 이동하는 문제점이 발생하므로 애를 조금 먹었지만 직접 wsl에서 실행해보니



nsd3689@BOOK-T48309SQMJ:~\$ cd -
bash: cd: OLDPWD not set

이렇게 이전 경로가 set되지 않는다고 출력되니 예외 사항을 만들어두면 되겠다.

그리고 경로 이동 후 strcpy함수를 사용하여 이전 경로에 현재 경로가 저장되도록 하였다.

여기서 fprintf , stderr 는 표준에러출력 스트림이다. 이전 경로가 저장되었지 않을 때 출력되도록 하였다.

6. 인터페이스 구현

pwd.h에 있는 getpwuid, gethostname, getcwd함수를 사용하여 구현

```
void print_prompt() { //인터페이스 구현
    char hostname[HOST_NAME_MAX];
    char cwd[PATH_MAX];
    char* username;

    username = getpwuid(getuid())->pw_name;
    gethostname(hostname, sizeof(hostname));
    getcwd(cwd, sizeof(cwd));
    printf("%s@%s:%s$ ", username, hostname, cwd);
}
```

7. 멀티 파이프 구현

파이프와 다중 명령어는 공백 뿐만 아니라 특정 문자로도 token을 나눠야 하기 때문에 복잡했다.

처음에 생각 한 것은 2차원 배열을 선언하여 첫번째 명령어는 tokens[0][0]="문자열" tokens[0][1]="인자"... 두번째 명령어("|"을 기준으로 나뉘어짐)는 tokens[1][0].... 이런 식으로 하려고 했다. 이렇게 했더니 make할 때 오류가 생겼는데 보니까 execvp 함수의 인자타입이 이중포인터여야 한다는 것이다. 그래서 2차원 배열인데 row열이 이중 포인터 즉 char *list[] 형태가 되도록 코드를 구현했다.

```

void pipee(char** token, int token_count, char* line) {
    int pipes[TOKEN_CNT][2] = { 0, };
    int pid = 0;
    int status;

    char** node_token[MAX_TOKEN_SIZE];
    int node = 0;
    int mi_token = 0;

    node_token[0] = malloc(sizeof(char*) * MAX_TOKEN_SIZE);
    for (int i = 0; token[i] != NULL; i++) {
        if (strcmp(token[i], "|") == 0) {
            node_token[node][mi_token] = NULL;
            node++;
            mi_token = 0;
            node_token[node] = malloc(sizeof(char*) * MAX_TOKEN_SIZE);
        }
        else {
            node_token[node][mi_token++] = token[i];
        }
    }
    node_token[node][mi_token] = NULL;
}

```

tokenize 함수와 비슷한 맥락으로 토큰화하는 코드를 짰다. 파이프 기호 "|"를 기준으로 이차원 배열의 열이 바뀌고 아니면 행을 동적 할당을 해서 token의 요소들(명령어, 인자...)를 저장하였다.

```

// 첫 번째 명령어
if (strcmp(node_token[0][0], "cd") == 0 || strcmp(node_token[0][0], "pwd") == 0) {
    if (strcmp(node_token[0][0], "cd") == 0) {
        if (node_token[0][1] == NULL) {
            cd(NULL);
        }
        else if (node_token[0][2] != NULL) {
            fprintf(stderr, "cd: too many arguments\n");
        }
        else {
            cd(node_token[0][1]);
        }
    }
    else {
        pwd();
    }
}
else {
    pipe(pipes[0]);
    pid = fork();

    if (pid < 0) exit(1);
    else if (pid == 0) {
        dup2(pipes[0][1], STDOUT_FILENO);
        close(pipes[0][0]);
        close(pipes[0][1]);
        execvp(node_token[0][0], node_token[0]);
        perror("execvp");
        exit(1);
    }
    close(pipes[0][1]);
    wait(&status);
}

```

이제 파이프 명령어를 구현할 차례이다. 다른 사람이 짠 코드를 참고하여 구현을 하였는데 첫 번째 명령어, 2번째 마지막 -1 번째 명령어, 마지막 명령어 이렇게 단계별로 구현을 하였다. 그 이유는 첫번째 명령어는 그 첫번째 명령어의 출력(STDOUT)만 파이프에 연결해주면 되기 때문이다. 그래서 코드상으로 내장명령어와 외장명령어의 구현 방식을 다르게 하였으니, 조건문을 통해 내장, 외장인지 구분을 한 후 외장일 경우에만 fork, execvp 함수를 통해 자식 프로세스 생성 → 명령어 실행 방식으로 구현해주면 되겠다. 그래서 pid에 fork()함수 값을 받고 만약 pid==0(자식 프로세스) 인 경우, dup2함수를 통해 모든 출력(STDOUT_FILENO)를 파이프에 연결해주면 되겠다.

dup2함수는 어떤 프로세스의 출력을 다른 프로세스의 입력으로 사용할 수 있게 만드는 함수이다.

기본적으로 pipe()함수를 사용하는데 pipefd[2]를 선언하여 pipe[0]:읽기전용, pipe[1]:쓰기 전용으로 한 후

dup2(pipefd[1],1) (쓰기 전용을 출력통로로 바꿈) ⇒ dup2(pipefd[0],0) (읽기 전용을 입력통로로 바꿈)을 통해서 만약 a|b가 있을 때 a의 출력(쓰기 전용)을 pipe를 통해서 b의 입력(읽기 전용)으로 바꾸는 것이다.

그럼 전체적인 단계는 pipe 통로 설치, fork()→ 자식 프로세스 생성, stdout, stdin 파이프 연결후 각각을 exec함수를 통해 실행, 이때 안쓰는 파이프는 close를 해줘야하는게 핵심이다. (dup2 이후 자식 프로세스에서 pipe[0],[1]을 닫고, 부모 프로세스에서 먼저 [0](읽지 않으므로) 그다음 [1] (쓰기) 닫는 형식)

이때 pipes 변수를 이중 배열로 선언을 하여 다수의 명령어를 파이프로 연결할 수 있게 하였다.

첫번째 명령어 출력[0][1]====파이프====[0][0] 입력 두번째 명령어에 해당한다.

그리고 여기서 exit와 같은 내장함수를 실행해도 쉘이 꺼지지 않는데 그 이유는 파이프로 연결되어 있어 exit도 자식 프로세스에서 실행되기 때문이다.

```
// 중간 명령어 처리
for (int i = 0; i < node - 1; i++) {
    if (strcmp(node_token[i + 1][0], "cd") == 0 || strcmp(node_token[i + 1][0], "pwd") == 0) {
        if (strcmp(node_token[i + 1][0], "cd") == 0) {
            if (node_token[i + 1][1] == NULL) {
                cd(NULL);
            }
            else if (node_token[i + 1][2] != NULL) {
                fprintf(stderr, "cd: too many arguments\n");
            }
            else {
                cd(node_token[i + 1][1]);
            }
        }
        else {
            pwd();
        }
        continue;
    }

    pipe(pipes[i + 1]);
    pid = fork();
    if (pid < 0) exit(1);
    else if (pid == 0) {
        dup2(pipes[i][0], STDIN_FILENO);
        dup2(pipes[i + 1][1], STDOUT_FILENO);
        close(pipes[i][0]);
        close(pipes[i + 1][1]);
        execvp(node_token[i + 1][0], node_token[i + 1]);
        perror("execvp");
        exit(1);
    }
    close(pipes[i][0]);
    close(pipes[i + 1][1]);
    wait(&status);
}
```

여기서 첫번째 명령어와 다른 점은 dup2함수의 사용법이다. 중간명령어는 입출력 둘다 dup2함수로 파이프를 연결해줘야 하기 때문이다.


```

// 마지막 명령어 처리
if (strcmp(node_token[node][0], "cd") == 0 || strcmp(node_token[node][0], "pwd") == 0) {
    if (strcmp(node_token[node][0], "cd") == 0) {
        if (node_token[node][1] == NULL) {
            cd(NULL);
        }
        else if (node_token[node][2] != NULL) {
            fprintf(stderr, "cd: too many arguments\n");
        }
        else {
            cd(node_token[node][1]);
        }
    }
    else {
        pwd();
    }
    return;
}

pid = fork();
if (pid < 0) exit(1);
else if (pid == 0) {
    dup2(pipes[node - 1][0], STDIN_FILENO);
    close(pipes[node - 1][0]);
    execvp(node_token[node][0], node_token[node]);
    perror("execvp");
    exit(1);
}
close(pipes[node - 1][0]);
wait(&status);
}

```

마지막 명령어도 dup2함수의 인자가 다르다.

그리고 각 코드마다 wait(&status)라는 코드가 항상 있는데 wait함수는 이름을 보면 알 수 있듯이 부모 프로세스가 자식 프로세스가 종료될 때 까지 대기하도록 하는 함수이고 status는 자식 프로세스를 종료 상태를 저장할 수 있는 변수이다. 그래서 이 코드가 없으면 부모 프로세스는 자식프로세스를 기다려주지 않는다. → &(백그라운드 명령어) 구현시 활용할 수 있다는 생각이 들었다.

```

// 메모리 해제
for (int i = 0; i <= node; i++) {
    free(node_token[i]);
}

```

그리고 마지막에는 동적할당한 메모리를 해제해주었다.

8. 다중 명령어 구현

8.1;

```

void multi1 (char* line, char** token, int token_count) { // 다중 명령어 ; 처리
    char** node_token[MAX_TOKEN_SIZE];
    int node = 0;
    int mi_token = 0;
    int pid = 0; //fork() 사용 위함
    node_token[0] = malloc(sizeof(char*) * MAX_TOKEN_SIZE);
    for (int i = 0; token[i] != NULL; i++) {
        if (strcmp(token[i], ";") == 0) {
            node_token[node][mi_token] = NULL;
            node++;
            mi_token = 0;
            node_token[node] = malloc(sizeof(char*) * MAX_TOKEN_SIZE);
        }
        else {
            node_token[node][mi_token++] = token[i];
        }
    }
    node_token[node][mi_token] = NULL;
    for (int i = 0; i <= node; i++)
    {
        pid = fork(); //부모 프로세스를 자식 프로세스로 대체해야 반복문 계속 진행
        if (pid == 0) {
            execvp(node_token[i][0], node_token[i]);
            perror("execvp fail");
            exit(1);
        }
        else {
            wait(NULL);
        }
    }

    for (int i = 0; i <= node; i++) {
        free(node_token[i]);
    }
    return;
}

```

일단 앞에서 tokenize함수 구현, 멀티 파이프 구현과 마찬가지로, 다중명령어 또한 그 문자열 기준으로 token화를 해야하므로 그 역할을 수행하는 코드를 짰고, ; 명령어는 그냥 명령어들을 실행하는 것이므로 반복문을 통해 execvp함수를 반복해서 실행되도록 하였다.

이때 wait(NULL)은 자식 프로세스의 종료 결과의 관계없이 부모 프로세스가 기다린다는 것으로, 단순히 부모 프로세스가 자식 프로세스를 기다리게 하고 좀비 프로세스가 발생하는 것을 방지하기 위해서이다.

8.2 &&

```

void multi2(char* line, char** token, int token_count) { //다중명령어 && 처리
    char** node_token[MAX_TOKEN_SIZE];
    int node = 0, mi_token = 0;
    node_token[0] = malloc(sizeof(char*) * MAX_TOKEN_SIZE);

    for (int i = 0; token[i] != NULL; i++) {
        if (strcmp(token[i], "&&") == 0) {
            node_token[node][mi_token] = NULL;
            node++;
            mi_token = 0;
            node_token[node] = malloc(sizeof(char*) * MAX_TOKEN_SIZE);
        }
        else {
            node_token[node][mi_token++] = token[i];
        }
    }
    node_token[node][mi_token] = NULL;

    for (int i = 0; i <= node; i++) {
        int pid = fork();
        if (pid == 0) {
            execvp(node_token[i][0], node_token[i]);
            perror("execvp fail");
            exit(1);
        }
        else {
            int status;
            waitpid(pid, &status, 0);
            // 앞 명령어 실패하면 다음 명령어는 실행 안 함
            if (!WIFEXITED(status) || WEXITSTATUS(status) != 0) //부모가 자식의 종료상태 체크
                break;
        }
    }

    for (int i = 0; i <= node; i++)
        free(node_token[i]);
}

```

구현하면서 상당히 애를 먹었는데 처음에는 wait(status)의 return값을 보고 다음 명령어를 실행시킬지 안시킬지 결정하는 코드로 구현하려고 했으나, execvp함수의 특징 상 만약 execvp() 함수가 실행될 경우, 다음 코드는 실행되지 않기 때문에 어떻게 return 가지고 하는지 애를 먹고 구글링을 통해서 wait함수에 관련된 매크로들을 알게 되었다.

WIFEXITED(status) // 자식이 정상 종료했는지 확인

WEXITSTATUS(status) // 자식의 반환값(리턴값)을 확인

WIFSIGNALED(status) // 시그널로 인해 종료됐는지

WTERMSIG(status) // 종료시킨 시그널 번호

이러한 매크로를 이용해서 부모 프로세스 구현 코드에서 조건문을 구현했다.

```

if (!WIFEXITED(status) || WEXITSTATUS(status) != 0) //
    break;

```

정상 종료안됨(return 값이 0이 아님)이 되면, 반복문은 break하는 코드다.

8.3 ||

```
void multi3(char* line, char** token, int token_count) { //다중 명령어 || 처리
    char** node_token[MAX_TOKEN_SIZE];
    int node = 0, mi_token = 0;
    node_token[0] = malloc(sizeof(char*) * MAX_TOKEN_SIZE);

    for (int i = 0; token[i] != NULL; i++) {
        if (strcmp(token[i], "||") == 0) {
            node_token[node][mi_token] = NULL;
            node++;
            mi_token = 0;
            node_token[node] = malloc(sizeof(char*) * MAX_TOKEN_SIZE);
        }
        else {
            node_token[node][mi_token++] = token[i];
        }
    }
    node_token[node][mi_token] = NULL;

    for (int i = 0; i <= node; i++) {
        int pid = fork();
        if (pid == 0) {
            execvp(node_token[i][0], node_token[i]);
            perror("execvp fail");
            exit(1);
        }
        else {
            int status;
            waitpid(pid, &status, 0);
            // 앞 명령이 성공하면 다음 명령어는 실행 안 함
            if (!WIFEXITED(status) || WEXITSTATUS(status) == 0)
                break;
        }
    }

    for (int i = 0; i <= node; i++)
        free(node_token[i]);
}
```

|| 명령어는 &&와 반대기 때문에, 조건문만 #에서 =으로 바꿔주면 된다.

즉 정상종료시 break

8.4 & 백그라운드 실행 함수

```

void multi4(char** line, int line_index) {
    for (int i = 0; i < line_index; i++) {
        int token_count = 0;
        char** tokens = tokenize(line[i], &token_count);

        int pid = fork();
        if (pid == 0) {
            // 자식 프로세스
            execvp(tokens[0], tokens);
            fprintf(stderr, "KU Shell: %s: command not found\n", tokens[0]);
            exit(1);
        }
        // 부모는 wait() 없이 다음 명령으로 넘어감 (백그라운드 실행)

        // 메모리 해제
        for (int j = 0; j < token_count; j++) {
            free(tokens[j]);
        }
        free(tokens);
    }
}

```

앞서 언급했듯이 부모 프로세스가 자식 프로세스를 기다리지 않게 하면 되겠다.

그리고 bash 셸에서 sleep 10 & 와 같은 & 백그라운드 명령어를 입력시 job number와 실행 순서를 부여해서 출력하고 백그라운드 실행이 끝나면 끝났다는 done 메시지를 출력해주는데 시간이 부족해 이 부분을 구현하지 못했다..

근데 이 코드를 짰 후 알아보니 명령어 한줄에 여러 다중명령어 그리고 하나의 명령어 여러 개를 사용할 수 있었다. 내가 구현한 코드는 만약 ls ; cat ~/t.txt || grep ~~~ 면 ;을 기준으로 ls ,cat ~/t.txt || ~~~을 기준으로 ls ;cat~/ 이렇게 나뉘어서 인식이 잘못되는 문제가 발생한다.

그래서 pipe처럼 구현하는 것이 아닌 다중 명령어가 있을 경우 명령 문자열을 다중 명령어 기준으로 나눠주는 형태로 코드를 구현하려고 했다. 찾아보니 strtok라는 훌륭한 함수가 있었다. 문자열을 어떤 문자 기준으로 나누고 문자 기준 왼쪽 문자열을 반환하는 함수다.

```

char** mul_multi(char* line, const char* delim) {
    char** result = malloc(sizeof(char*) * 64);
    int index = 0;

    char* token = strtok(line, delim);
    while (token != NULL) {
        result[index++] = strdup(token);
        token = strtok(NULL, delim);
    }
    result[index] = NULL; // 마지막은 NULL로

    return result;
}

```

delim 함수 인자는 다중 command이다. 그래서 result ** 이중포인터를 선언해서 문자열을 저장하였다.

그리고 execc 함수 내에 다중 명령어 조건문을 대폭 수정했다.

help_line 선언하여 line을 command를 기준으로 split한 후, 다중 명령어 구현 함수를 호출 하였다.

그러면 help_line에는 {명령어 하나, 명령어 둘...} 이런 식으로 되있기 때문에 각각의 index를 또 토큰화 해줘야 하므로 tokens 이중 포인터를 선언해서 각 index를 토큰화 → 각 인덱스의 명령어 실행 → free → 다음 인덱스 이런 알고리즘으로 구현을 하였다.

```

void multi1(char** line, int line_index) { // 다중 명령어 ; 처리
    for (int i = 0; i < line_index; i++) {
        int token_count = 0;
        char** tokens = tokenize(line[i], &token_count); // 명령어 하나씩 토큰화

        pid_t pid = fork();
        if (pid == 0) {
            // 자식 프로세스
            execvp(tokens[0], tokens);
            perror("execvp failed");
            exit(1);
        }
        else if (pid > 0) {
            // 부모 프로세스
            wait(NULL);
        }
        else {
            perror("fork failed");
        }

        // 메모리 정리 (strdup 사용 시)
        for (int j = 0; j < token_count; j++) {
            free(tokens[j]);
        }
        free(tokens);
    }
}

```

```

//다중 명령어 기호 탐색
for (int i = 0; line[i] != '\0'; i++)
{
    if (line[i] == '&'amp; && line[i + 1] == '&') {
        help_line = mul_multi(line, "&&");
        int count = 0;
        while (help_line[count] != NULL) count++;

        multi4(help_line, count);

        return;
    }
    else if (line[i] == '|' && line[i + 1] == '|') {
        help_line = mul_multi(line, "||");
        int count = 0;
        while (help_line[count] != NULL) count++;

        multi3(help_line, count);
        return;
    }
    else if (line[i] == ';')
    {
        help_line = mul_multi(line, ";");
        int count = 0;
        while (help_line[count] != NULL) count++;

        multi1(help_line, count);
        return;
    }
    else if (line[i] == '&')
    {
        help_line = mul_multi(line, "&");
        int count = 0;
        while (help_line[count] != NULL) count++;

        multi4(help_line, count);

        return;
    }
}

```

9. execc함수 구현(명령어 구현 사용자 정의 함수)

여태까지 구현했던 기능들은 결국 사용자가 입력한 명령어 줄을 실행시키는 것이다. 이것을 한꺼번에 모아 특정 명령어의 조건에 만족하는 함수가 실행되도록 하는게 중요하므로 이 함수를 만들었다.

함수의 인자인 line은 사용자가 입력한 명령어 줄이다. 이것을 처음에 토큰화 한 후 line의 요소를 순회하며 특정 문자가 있는지 없는지 판단한다.

파이프 명령어, 다중 명령어가 아닌 경우 단일 명령어 이고, 단일 명령어에는 내장, 외장 명령어로 나뉘지기 때문에 이렇게 조건문을 설정 하였다.

```
// 파이프 기호 탐색
for (int i = 0; line[i] != '\0'; i++) {
    if (line[i] == '|') {
        pipp = 1;
        break;
    }
}

if (pipp) {
    pipee(tokens, token_count, line);
}
else {
    // 내부 명령어 처리
    if (strcmp(tokens[0], "cd") == 0 || strcmp(tokens[0], "pwd") == 0) {
        if (strcmp(tokens[0], "cd") == 0) {
            if (token_count == 1) {
                cd(NULL);
            }
            else if (token_count == 2) {
                cd(tokens[1]);
            }
            else {
                fprintf(stderr, "cd: too many arguments\n");
            }
        }
        else {
            pwd();
        }
    }
    else {
        pid = fork();
        if (pid < 0) {
            perror("fork fail");
            return;
        }
        else if (pid == 0) {
            execvp(tokens[0], tokens);
            perror("execvp fail");
            exit(1);
        }
        else {
            int status;
            waitpid(pid, &status, 0);
        }
    }
}
```

10. main 함수

```

int main(int argc, char* argv[])
{
    char line[MAX_INPUT_SIZE];
    print_banner();

    while (1) {
        print_prompt();

        char* result = fgets(line, sizeof(line), stdin);
        if (result == NULL) {
            break;
        }
        line[strcspn(line, "\n")] = '\0';
        if (strlen(line) == 0) continue;
        if (strcmp(line, "exit") == 0) break; //exit 기능 구현
        execc(line);
    }
}

```

마지막으로 main 함수이다.

line 배열을 선언한 후, while문을 통해서 계속해서 print_prompt 함수를 실행하고 result 에 사용자 입력을 받는다. 그리고

사용자가 명령어 줄을 입력하고 누른 개행까지 문자열로 받기 때문에 개행을 널 문자로 바꿔 주는 작업을 하고,

이러한 exit기능 구현과 예외 처리까지 한 후,

line을 execc 함수의 인자로 전달하면 되겠다.

++ 좀비 프로세스와 버퍼 오버플로우 방지

좀비 프로세스란 프로세스가 종료되고 리소스는 모두 회수되었지만, 시스템 프로세스 테이블에 남아있는 프로세스를 말한다. 프로세스 과정에서 exit() 함수를 호출 함으로서 운영체

제에게 자신의 종료를 요청한다. 이때, 자식프로세스를 기다려주는 부모 프로세스가 wait 함수를 호출해 자식 프로세스의 종료상태를 확인해야하는데 wait을 호출하지 않아 계속 프로세스 테이블에 남게 된다.

그래서 프로세스 실행과정 시 부모 프로세스 코드 구현 과정에서 wait 함수를 넣는 것이 중요하다.

그럼 만약 wait() 함수를 호출해도 좀비 프로세스가 남아있다면 어떨까?

SIGCHLD는 자식 프로세스가 자신의 종료상태를 부모프로세스에게 알리는 시그널이다.

이 시그널을 보내는 코드를 직접 구현해서 하면 좀비 프로세스를 방지할 수 있을 거 아닌가 라고 생각해봤다.

또한 버퍼 오버플로우에 대해서 생각 해보았다.

내가 짠 코드는 매크로로 정의 된 정해진 메모리 크기 안에서 명령어를 받고 처리하고 있다. 만약 정해진 크기 이상의 명령어 줄이 들어온다면 다른 메모리까지 침범해 오버플로우를 발생시킬 수 있다.

```

char buffer[MAX_TOKEN_SIZE];
int tokenindex = 0, charindex = 0;

for (int i = 0; ; i++) {
    char c = line[i];
    if (c == ' ' || c == '\n' || c == '\t' ||
        if (charindex > 0) {
            buffer[charindex] = '\0';
            tokens[tokenindex] = strdup(buffer);
            tokenindex++;
            charindex = 0;
        }
        if (c == '\0') break;
    }
    else {
        buffer[charindex++] = c;
    }
}

```

buffer의 크기를 MAX_TOKEN_SIZE로 제한하고 있지만 반복문을 통해서 charindex를 증가시켜서 메모리에 값을 저장하고 있다. 만약 charindex가 TOKEN_SIZE를 넘어가면 전혀 알 수 없는 메모리에 값을 저장시키게 된다.

```

if (charindex < MAX_TOKEN_SIZE - 1) {
    buffer[charindex++] = c;
}

```

그래서 코드를 이렇게 수정해줬다.

또한 메모리 낭비가 심하지 않기 위해 malloc()을 이용한 동적할당을 할 경우 free로 메모리 해제를 해주었다.