week2

- 현재 구현해야 하는 단계들을 정리해보았다
  1. 프롬프트 출력(현재 위치와 유저명 등 인터페이스)
  2. 문자열을 입력받아 버퍼로 넣기
  3. 입력받은 문자열에 ||, &&, ;가 있는지 확인하고, 각각에 대해 문자열을 나눠 명령어들을 저장, ||, &&, ;역시 순서대로 저장
  4. &가 있는지 검사 후, &따라 나눠 저장해줌
  5. 저장한 명령어들을 ||, &&, ;에 따라 처리해줌
  6. |이 있는지 확인 후, 있다면 파이프라인 처리
  7. 명령어 실행

  다음의 과정들을 구현해야 하며, 프롬프트 출력 함수, 명령어 실행 함수, 파이프라인 처리 함수, 그리고 다중 명령어 및 백그라운드 실행 처리 함수를 통해 구현하고자 한다.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <limits.h>
#include <fcntl.h>
#include <sys/wait.h>
```

- 먼저 구현에 사용한 라이브러리들이다.

sys.wait.h	stdlib.h	unistd.h	string.h	limits.h	fcntl.h
프로세스 종료 관련(waitpid() 등)	exit, malloc 등 함수	fork(), exec(), chdir() 등 UNIX 시스템 호출 라이브럴	문자열 처리	PATH_MAX 등 시스템 한계 값	파일 디스크립터 제어 라이브러리

- print_promt 함수

```c
void print_prompt() {
    char cwd[PATH_MAX];
    getcwd(cwd, sizeof(cwd));
    printf("[main:%s]$ ", cwd);
    fflush(stdout);
}
```

  이 함수에서는 현재 위치나 유저명 등을 출력해주며 쉘 프로그램의 인터페이스를 출력해주는 함수이다. getcwd()함수를 통해 현재 작업 디렉토라의 경로를 cwd변수에 받아온 뒤, 이를 출력해주며 쉘 프롬프트 창을 출력해주는 함수이다. 또한 현재 버퍼에 출력값이 남아있을 경우를 대비해 fflush(stdout)으로 버퍼에 남은 데이터들을 화면으로 출력해준다.

- main함수

```c
int main() {
    char buffer[MAX_INPUT];

    while (1) {
        print_prompt();
        if (!fgets(buffer, MAX_INPUT, stdin)) break;
        buffer[strcspn(buffer, "\n")] = 0;

        if (strcmp(buffer, "") == 0) continue;
        if (strcmp(buffer, "exit") == 0) break;

        char *cmds[MAX_ARGS];
        char *ops[MAX_ARGS - 1];
        int count = 0;
        char *tmp = buffer;
```

다른 함수를 살펴보기 전 main 함수를 살펴봐주었다. buffer라는 변수를 통해 사용자가 입력하는 값들을 받아주었고, while(1)문을 통해 exit가 입력되거나 오류가 발생할 때까지 입력 값을 받아주도록 했다.

받은 입력값의 마지막 부분에 있는 줄바꿈 문자를 삭제해준 뒤, 입력 받은 값이 exit와 동일하다면 while문을 break해주도록 만들어줬다. 추가로 입력값이 없어도 다시 프롬프트를 출력해 입력받도록 해주었다.

그 뒤 명령어들을 저장할 cmds, ;나|| && 등의 연산자들을 저장해줄 ops, 입력값에 들어있는 명령어의 개수를 저장할 count, 그리고 tmp에 입력값의 시작 주소를 저장해준다. tmp에 저장된 값은 입력값을 순회하며 연산자를 단위로 분할해주는 파싱을 수행하는 데 사용될 예정이다.

```c
while (*tmp) {
    while (*tmp == ' ') tmp++;

    char *start = tmp;
    while (*tmp && strncmp(tmp, "&&", 2) != 0 && strncmp(tmp, "||", 2) != 0 && *tmp != ';')
        tmp++;

    int len = tmp - start;
    while (len > 0 && start[len - 1] == ' ') len--;

    cmds[count] = strndup(start, len);

    if (strncmp(tmp, "&&", 2) == 0) {
        ops[count] = "&&";
        tmp += 2;
    } else if (strncmp(tmp, "||", 2) == 0) {
        ops[count] = "||";
        tmp += 2;
    } else if (*tmp == ';') {
        ops[count] = ";";
        tmp++;
    } else {
        ops[count] = NULL;
    }

    count++;
}
```

해당 부분에서는 입력값을 순회하며 연산자 단위로 분할해준다.

while(*tmp == ' ') tmp++부분을 통해 혹시 존재할 공백 부분을 제거해준다. start에는 현재 tmp의 값(시작 주소)를 저장해줘 현재 명령어 부분의 시작 주소를 저장해주고, tmp내에서 "&&", "||", ';'라는 부분이 나올때까지 tmp의 값을 올려주고, 더이상 올릴 수 없다면 그만해준다. 그 위, tmp - start를 통해 명령어의 길이를 구해준다. 그 뒤 혹시모를 공백을 제거해준 뒤, start의 주소부터 len만큼의 길이에 저장된 값들을 cmds[count]에 저장해준다. 즉, count번째 명령어가 저장되어지는 것이다.

그 뒤 찾은 연산자가 "||"인지, "&&"인지, ';'인지 아니면 없는지 판별해 해당 연산자를 ops에 저장해준다.

```
    int last_status = 0;

    for (int i = 0; i < count; i++) {
        char *command = cmds[i];
        while (*command == ' ') command++;

        char *subcmds[MAX_ARGS];
        int subcount = 0;

        char *sub = strtok(command, "&");
        while (sub && subcount < MAX_ARGS - 1) {
            while (*sub == ' ') sub++;
            int len = strlen(sub);
            while (len > 0 && sub[len - 1] == ' ') sub[--len] = '\0';
            subcmds[subcount++] = sub;
            sub = strtok(NULL, "&");
        }

        for (int j = 0; j<subcount;j++){
            int bg = (j < subcount - 1);
            if (i == 0 || strcmp(ops[i - 1], ";") == 0 || (strcmp(ops[i - 1], "&&") == 0 && last_status == 0) || (strcmp(ops[i - 1], "||") == 0 && last_status != 0))
            last_status = run(subcmds[j], bg);
        }

        free(cmds[i]);
    }
}
```

다음은 앞에서 cmds에 저장된 파싱된 명령어들을 &를 기준으로 백그라운드 명령을 처리한 뒤, ops에 저장된 연산자에 따라 run()함수를 통해 실행시켜주는 부분이다. 이때 다중 연산자 ||에서 사용할 last_status를 선언해준다.

for문을 통해 cmds에 저장된 명령어들을 command로 가지고 와줬다. 명령어 한 줄에서 여러 백그라운드 명령어들을 subcmds에 저장해주기 위해 변수를 선언해줬더, command를 ‘;’로 분할 한 뒤, 첫번째 부분을 char *sub에 저장해주었다.

그 뒤 분할된 부분들의 공백을 제거해주기 위해 while(sub && subcount < MAX_ARGS - 1)문을 통해 제거해주고, subcmd[]에 저장해주었다. 이때 strtok(NULL, ‘&”)을 통해 계속해서 ‘&’로 나눠주도록 했다. 그리고 subcount에는 subcmd의 개수를 저장해주었다.

subcount의 수에 따라 for문을 돌려주며

```
for (int j = 0; j< subcount;j++){
  int bg = (j < subcount - 1);
  if (i == 0 || strcmp(ops[i - 1], ";") == 0 || (strcmp(ops[i - 1], "&&") == 0 && last_status == 0) ||
    last_status = run(subcmds[j], bg);
}
```

위의 부분에서 j가 subcount -1보다 작다면 bg에 1을 저장해주어 아직 백그라운드 명령을 수행중인지 표시해주었고, ops에 저장한 연산자들을 우선순위에 맞추어 (‘;’ > “&&” > “||”) 연산해주었다. 이때 last_status변수에 초기값을 0을 넣어주고, run한 뒤 실패하면 1, 성공하면 0을 반환하게 만들어주어 ||, &&이 각각 앞의 명령어에 따라 수행될지 안될지 따져주었다.

last_status = run(subcmds[j], bg);부분을 통해 bg값과 명령어들을 모두 넘겨주었고, 반환값을 받아 last_status에 저장해 위의 과정이 가능하도록 해주었다.

그 뒤 cmd[i]의 모든 명령어들을 실행완료했다면 해당 부분을 free()를 통해 풀어주었다.

- run() 함수

```c
int run(char *cmd, int bg) {
    if (strchr(cmd, '|'))
        return pipeline(cmd);

    char *args[MAX_ARGS];
    parse_args(cmd, args);
    if (args[0] == NULL) return 1;

    if (strcmp(args[0], "cd") == 0) {
        if (args[1] == NULL || chdir(args[1]) != 0) {
            perror("cd");
            return 1;
        }
        return 0;
    }

    if (strcmp(args[0], "pwd") == 0) {
        char cwd[PATH_MAX];
        if (getcwd(cwd, sizeof(cwd))) {
            printf("%s\n", cwd);
            return 0;
        } else {
            perror("pwd");
            return 1;
        }
    }
```

해당 부분에서는 main함수에서 넘겨준 명령어들을 실행시켜주는 부분이다. 가장 먼저 '|'가 넘겨준 명령어에 존재하는지 알아보고, 존재한다면 이를 pipiline()함수로 넘겨줘 파이프라인 처리를 해주게 된다.

명령어를 파싱해준 뒤, 이를 저장할 args를 선언해준 뒤, 명령어를 파싱해주는 함수인 parse_args()함수에 넘겨 받아와준다. args[0]은 명령어의 파싱된 가장 첫번째 부분, 즉 명령어에서 함수의 역할을 하는 부분이다. 이 부분이 NULL이라는 것은 함수가 선언되지 않았다는 것이므로 1을 반환해준다.

먼저 args[0] 이 cd일 때의 경우 즉, cd 명령어 구현 부분이다. 이때에는 cd [경로]가 들어오게 되므로, args[1]에는 이동할 경로가 들어오게 된다. chdir을 통해 args[1]로 이동한 뒤, 이동이 안되거나 아예 NULL 값이 들어온다면 경로값이 들어오지 않거나 존재하지 않는 경로로 이동하게되는 것이므로 에러를 불러준 뒤, 1을 반환해주고, 아니라면 0을 반환해주었다.

다음은 pwd부분이다. pwd는 현재 디렉토리 위치를 출력해주는 명령어이므로 cwd라는 변수를 선언한 뒤, getcwd()를 통해 현재 위치를 받아와준다. 이때 현재 위치를 가져오지 못한다면 에러를 불러준 뒤 1을 반환해주고, 아니라면 현재 위치를 출력해준 뒤, 0을 반환해준다.

```c
    pid_t pid = fork();
    if (pid == 0) {
        execvp(args[0], args);
        perror("execvp");
        exit(127);
    } else if (pid > 0) {
        if (!bg) {
            int status;
            waitpid(pid, &status, 0);
            if (WIFEXITED(status))
                return WEXITSTATUS(status);
            else
                return 1;
        }
        return 0;
    } else {
        perror("fork");
        return 1;
    }
}
```

cd와 pwd 이외의 명령어들은 모두 execvp()를 통해 구현해주었다. 먼저 pid = fork()를 통해 자식 프로세스를 생성해준 뒤, execvp()를 통해 args에 저장된 명령어를 수행해준다. 자식 프로세스의 pid값은 0이므로 pid == 0일 때 실행되어진다.

pid > 0이라면 부모 프로세스이므로 먼저 bg값을 확인해준다. bg값이 0이라면 백그라운드 실행이 아니므로 자식 프로세스가 종료될 때까지 기다린 뒤, 끝났을 때 정상 종료값이 나오는지 확인해준다. 정상 종료값이 나온다면 정상적인 종료 상태값(0)을 반환해주고, 아니라면 1을 반환해준다. bg값이 0이 아니라면 백그라운드 실행이므로 waitpid()를 통해 자식 프로세스가 끝날 때까지 기다리지 않고 바로 프롬프트로 돌아간다. pid < 0 일 때는 에러가 났다는 뜻이므로 에러를 출력한 뒤 1을 반환해준다.

- parse_args() 함수

```c
void parse_args(char *cmd, char **args) {
    int i = 0;
    char *token = strtok(cmd, " ");
    while (token && i < MAX_ARGS - 1) {
        args[i++] = token;
        token = strtok(NULL, " ");
    }
    args[i] = NULL;
}
```

명령어를 파싱해주는 함수이다. cmd, arg를 통해 넘겨줄 주소를 받아준 뒤 cmd를 나눠준다. cmd를 “ “으로 나눠준 뒤 이를 각각 token에 저장시켜 args로 넘겨주는 방식이다. 모두 넘겨준 뒤 args의 마지막 부분에는 NULL을 저장시켜준다.

- pipeline() 함수

```c
int pipeline(char *cmd) {
    char *cmds[MAX_CMDS];
    int count = 0;

    char *token = strtok(cmd, "|");
    while (token != NULL && count < MAX_CMDS){
        while (*token == ' ') token++;
        cmds[count++] = token;
        token = strtok(NULL, "|");
    }

    if (count < 2){
        fprintf(stderr, "Invalid pipeline\n");
        return 1;
    }

    int fds[MAX_CMDS][2];
    for (int i = 0; i < count - 1; i++) {
        if (pipe(fds[i]) < 0) {
            perror("pipe");
            return 1;
        }
    }
```

run()함수에서 파이프라인을 처리하기 위해 만든 함수이다. 먼저 명령어들을 저장하기 위한 cmds 선언 뒤, 매개변수로 받아온 cmd를 strtok으로 ‘|’를 기준으로 나눠준 뒤 저장해준다. 나눈 값들은 token에 저장해주고 이를 cmd에 옮겨준다.

이때 명령어의 개수를 count에 저장해주는데, count값이 2보다 작다면 유효하지 않으므로 에러를 출력해준다.

그 뒤 파일 디스크립터들을 저장할 fds를 선언해준다. 이때 fds[i][0]은 읽기용, [1]은 쓰기용으로 사용한다. count개의 파이프가 존재하므로 count - 1개의 파일 디스크립터가 있어야 되고 pipe()함수를 통해 실제 파이프를 생성해준다. 이때 0보다 작은 값이 나오면 에러가 발생한 것이므로 에러를 출력하고 1을 반환해준다.

```
    int fds[MAX_CMDS][2];
    for (int i = 0; i < count - 1; i++) {
        if (pipe(fds[i]) < 0) {
            perror("pipe");
            return 1;
        }
    }

    for (int i = 0; i < count; i++) {
        char *args[MAX_ARGS];
        parse_args(cmds[i], args);

        pid_t pid = fork();
        if (pid == 0) {
            if (i > 0)
                dup2(fds[i - 1][0], STDIN_FILENO);

            if (i < count - 1)
                dup2(fds[i][1], STDOUT_FILENO);

            for (int j = 0; j < count - 1; j++) {
                close(fds[j][0]);
                close(fds[j][1]);
            }

            execvp(args[0], args);
            perror("execvp");
            exit(127);
        }
    }

    for (int i = 0; i < count - 1; i++) {
        close(fds[i][0]);
        close(fds[i][1]);
    }

    for (int i = 0; i < count; i++)
        wait(NULL);

    return 0;
}
```

명령어의 개수가 저장된 count만큼 for문을 돌려줘서 명령어 모두 파이프라인 과정을 수행해준다. 먼저 cmds[i]에 저장된 명령어들을 파싱해준 뒤 앞에서 선언한 args에 저장해준다. 그 뒤 fork()를 통해 자식 프로세스를 생성해준다. 그 뒤 각 각 프로세스별로 입출력을 리다이렉션한다. i번째 프로세스의 입력값은 i - 1번째 프로세스의 출력값이 들어가게 되고, i번째 프로세스의 출력값은 i + 1번째 프로세스의 입력값으로 들어가게 된다. 이를 통해 앞선 프로세스와 연결되어서 명령어를 실행할 수 있게 된다.

```
if (i > 0)
    dup2(fds[i - 1][0], STDIN_FILENO);
```

```
if (i < count - 1)
    dup2(fds[i][1], STDOUT_FILENO);
```

그 뒤 해당 출력값이 입력값으로 옮겨지고 난 뒤에는 이미 사용한 뒤 더 이상 사용하지 않는 프로세스들을아래의 코드를 통해 모두 닫아준다.

```
for (int j = 0; j < count - 1; j++) {
    close(fds[j][0]);
    close(fds[j][1]);
}
```

그 뒤 execvp를 통해 명령어를 실행시켜준다.

위의 자식 프로세스들에서의 과정이 모두 끝난다면 부모 프로세스에서도 역시 프로세스들을 모두 닫아준다.

- Makefile