<헤더 파일 및 전역 변수 등등>

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

#define MAX_ARGS 64
#define MAX_LINE 256
#define COLOR_GREEN "\033[0;32m"
#define COLOR_RED "\033[0;31m"
#define COLOR_BLUE "\033[0;34m"
#define COLOR_RESET "\033[0m"

char* home_dir;
int count = 1;

typedef struct {
    int job_id;
    pid_t pid;
    char command[MAX_LINE];
} Job;

Job jobs[64];
int job_count = 0;
```

명령어를 토큰화 해서 저장할 배열의 길이

입력받을 명령어를 저장하는 배열의 길이

사용사 인터페이스 색깔

홈 디렉토리 정의

백그라운드 갯수

백그라운드 사용시 출력할 정보들

<wordsep 함수> : 명령어를 파싱

```
void wordsep(char* line, char** args) {
        int i = 0;
        args[i] = strtok(line, " \t\n");
        while (args[i] != NULL && i <= MAX_ARGS - 2) {
                i++;
                args[i] = strtok(NULL, " \n\t");
        }
        args[i] = NULL;
}
```

→ 명령어에 " ","₩t","₩n"이 있으면 파싱해서 저장

→ 그 줄의 남은 명령어도 같은 방식으로 파싱해서 명령어를 다 파싱함(배열이 다 찰 것 같은 경우는 다 차기 직전까지만 파싱)

→ 명령어가 끝났음을 배열에 저장

<shell_prompt함수> : 사용자 인터페이스 구현

```
int  shell_prompt(){
        char cwd[PATH_MAX];
        char hostname[HOST_NAME_MAX];
        char* username = getlogin();

        home_dir = getenv("HOME");

        if(gethostname(hostname, sizeof(hostname)) != 0){
        perror("gethostname");
        return -1;
        }

        if(getcwd(cwd, sizeof(cwd)) == NULL){
        perror("getcwd");
        return -1;
        }

        if(strncmp(cwd, home_dir, strlen(home_dir)) == 0){
                cwd[0] = '~';
                memmove(cwd + 1, cwd + strlen(home_dir), strlen(cwd) - strlen(home_dir) + 1);
        }
        printf("[" COLOR_RED "%s" COLOR_RESET "@" COLOR_GREEN "%s" COLOR_RESET ":" COLOR_BLUE "%s" COLOR_RESET "]$ ",  username, hostname, cwd);
        fflush(stdout);

        return 0;
}
```

→ 디렉토리 위치 정의

→ 호스트 이름 정의

→ 유저 이름 정의 + 값 저장

→ 홈 디렉토리에 경로 값 저장

→ 호스트 이름 값 저장 + error시 error 메세지 출력

→ 현재 디렉토리 값 저장 + error시 error 메세지 출력

→ 홈 디렉토리나 홈 디렉토리 하위에 현재 주소가 있으면 홈 디렉토리를 ~로 바꿈

→ 사용자 인터페이스 출력

<command 함수>:일반적인 명령어 + &

```
int command(char* line){
    char* args[MAX_ARGS];
        int background_flag = 0;

    wordsep(line, args);
        if (args[0] == NULL) {
                    return 1;
                }
        int i = 0;
        while (args[i] != NULL) {
                i++;
                }
        if (i > 0 && strcmp(args[i - 1], "&") == 0) {
                background_flag = 1;
                args[i - 1] = NULL;
                }

                if (strcmp(args[0], "exit") == 0) {
                        exit(0);
                }

                if (strcmp(args[0], "cd") == 0) {

                        if (args[1] != NULL){
                 if (strcmp(args[1], "~") == 0) {
                 args[1] = home_dir;
                         }
                          if(chdir(args[1]) != 0){
                               perror("cd");
                                  return 1;
                           }
                         return 0;
                         }
                                 else{
                                 fprintf(stderr, "cd:missing operand\n");
                                 return 1;
                         }
                         }

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

백그라운드 실행 체크

아무것도 입력하지 않았을 때 구현(다음 프롬프트 진행)

i는 NULL 제외 파싱한 명령어 갯수

& 사용했는지 체크 + &를 NULL로 바꾸기(명령어를 실행시키기 위해)

exit 명령어 구현(프롬프트 종료)

명령어 cd를 사용한 경우

이동할 디렉토리를 쓴 경우

cd ~ 일때 홈 디렉토리로 이동

디렉토리 이동 + 실패시 error 메세지

이동할 디렉토리 안 쓴 경우 error 메세지

명령어 pwd를 쓴 경우

현재 디렉토리를 받아서 출력

현재 디렉토리를 못 받으면 error 메세지 출력

```
pid_t pid = fork();
```
→ 외부 명령어-fork

```
if(pid < 0){
        perror("Fork failded");
        return 1;
    }
```
→ fork 실패시 error 메세지 출력

```
else if(pid == 0) {
       if(execvp(args[0], args) < 0){
        perror("Execution failed");

        }
        _exit(1);
        }
```
→ 자식 프로세스-exec 실행 + exec 실패시 error 메세지 출력

```
        else{
```
→ 부모 프로세스

```
        if(!background_flag){
        int status;
        waitpid(pid, &status, 0);
        if(WIFEXITED(status)){
                return WEXITSTATUS(status);
        }
        }
```
→ 백그라운드 실행 X - 자식 프로세스가 종료될 때까지 기달리기 + 자식 프로세스에서 오류로 종료시 이를 부모 프로세스에게 전달

```
        else{
           jobs[job_count].job_id = count;
        jobs[job_count].pid = pid;
        snprintf(jobs[job_count].command, MAX_LINE, "%s %s", args[0], args[1]
        job_count++;
        printf("[%d] %d\n", count++, pid);
        }
        }
        return 1;
    }
```
→ 백그라운드 실행 O - 부모 프로세스는 자식 프로세스를 기달리지 않음 + & 사용시 출력되는 값 출력

<background 함수> : 백그라운드로 실행한 자식 프로세스가 종료될 때 하는 동작

```
void background(int sig){
        int status;
        pid_t pid;

        while((pid = waitpid(-1, &status, WNOHANG)) > 0) {
             for (int i = 0; i < job_count; i++) {
            if (jobs[i].pid == pid) {
                if (WIFEXITED(status)) {
                    printf("\n[%d]+ Done       %s\n", jobs[i].job_id, jobs[i].command);
                } else if (WIFSIGNALED(status)) {
                    printf("\n[%d]- Terminated %s\n", jobs[i].job_id, jobs[i].command);
                }
                break;
            }
        }
        shell_prompt();
        fflush(stdout);
        }
        }
```

현재 실행중인 모든 백그라운드 자식 프로세스 종료되었나 확인
어느 자식 프로세스가 종료 되었나 확인

자식 프로세스가 정상적으로 종료+그때 출력되는 값 출력

자식 프로세스가 오류로 종료 + 그때 출력되는 값 출력

자식 프로세스가 백그라운드에서 실행이 끝났으니 프롬프트 다시 출력

<pipeline_execute 함수> : 파이프라인 구현

```
int  pipeline_execute(char* line){
        char* commands[MAX_ARGS];
         int num_commands = 0;

         commands[num_commands] = strtok(line, "|");
        while (commands[num_commands] != NULL) {
                num_commands++;
                 commands[num_commands] = strtok(NULL, "|");
         }

        int i, in_fd = 0, status = 0;

        for(i = 0; i < num_commands; i++){
        int pipe_fd[2];
        if (pipe(pipe_fd) == -1) {
         perror("Pipe creation failed");
         return 1;
        }
        if(fork() == 0){
        dup2(in_fd, 0);

        if(i < num_commands - 1){
        dup2(pipe_fd[1], 1);
        }

        close(pipe_fd[0]);
        close(pipe_fd[1]);
        char* args[MAX_ARGS];
        wordsep(commands[i], args);

        if(execvp(args[0], args) < 0){
        perror("Execution failed");
        _exit(1);
        }
        }
        else{
        wait(&status);
        close(pipe_fd[1]);
        if(in_fd != 0){
                close(in_fd);
        }
        in_fd = pipe_fd[0];
        }
        }
        while (wait(&status) > 0) {
         if (WIFEXITED(status)) {
        if (WEXITSTATUS(status) != 0) {
            return WEXITSTATUS(status);
    } else {
                return 1;
        }
        }
        return 0;
}
```

→ | 를 기준으로 명령어 쪼개기

→ in_fd는 읽기 파이프의 역활을 같이 수행함(파이프는 초기화 되므로)

→ 다중 파이프라인에 있는 모든 명령어 수행

→ 파이프 생성 pipe[0]-읽는 파이프 + 파이프 생성 실패 error 메세지
pipe[1]-쓰는 파이프

→ fork 실행-자식 프로세스 내용

→ 입력에 읽기 파이프 대입(전 프로세스의 출력이 현 프로세스 입력)

→ 출력에 쓰기 파이프 대입(현 프로세스의 출력을 남겨 다음 프로세스의 입력으로 넘길 예정)

→ 읽기, 쓰기 파이프 다썼으니 닫기

→ 명령어 파싱해서 exec 실행 + 오류시 error 메세지 출력

→ 부모 프로세스 내용

→ 각 명령어의 자식 프로세스 종료될 때까지 기달림

→ 읽기, 쓰기 파이프 닫음(부모 프로세스는 파이프 필요 없음)

→ 모든 자식 프로세스 끝날때까지 기달림

→ 정상 종료

→ exit code가 비정상일때 그 값 반환

→ 비정상 종료시 반환값 반환

<multiple_commands 함수> : 다중 명령어 구현

```
void multiple_commands(char* line) {
    char* m_command;
    int prev_success = 1;
        char* save_ptr;


    m_command = strtok_r(line, ";",&save_ptr);

    while (m_command != NULL) {

        while (*m_command == ' ' || *m_command == '\t') {
            m_command++;
        }


        char* and_split = strstr(m_command, "&&");
        char* or_split = strstr(m_command, "||");

        if (and_split) {
            *and_split = '\0';
            prev_success = command(m_command);


            if (prev_success == 0) {
                prev_success = command(and_split + 2);
            }
        } else if (or_split) {
            *or_split = '\0';
            prev_success = command(m_command);

            if (prev_success != 0) {
                prev_success = command(or_split + 2);
            }
        } else {
                if(strstr(m_command, "|")){
                        prev_success = pipeline_execute(m_command);
                }
                else
                        prev_success = command(m_command);
                }
         m_command = strtok_r(NULL, ";",&save_ptr);

        }
        }
```

&&, || 을 위해 전 명령어가 실행됐는지 안됐는지 표현

; 구현을 위해 명령어를 ;를 기준으로 자르기

모든 명령어를 실행할 때까지

명령어에 빈칸 지우기

&&, ||이 있는 확인

&&가 있는 경우 첫번째 명령어 실행 후 실행 성공할 때만 두번째 명령어 실행

||가 있는 경우 첫번째 명령어 실행 후 실행 실패할 때만 두번째 명령어 실행

| 있으면 실행

;있으면 명령어 쪼개고 while문 이용해서 모두 실행
없으면 그냥 명령어 하나만 실행

<main 함수> : 지금까지 만든 함수 호출해서 셸 구현

```c
int main(void) {
        char line[MAX_LINE];
        char* args[MAX_ARGS];
          signal(SIGCHLD, background);

        while (1) {
                if(shell_prompt() < 0){
                fprintf(stderr, "Prompt failed.\n");
                break;
                }

                if (fgets(line, sizeof(line), stdin) == NULL) {
                        break;
                }
                multiple_commands(line);
        }
        return 0;
}
```

→ | 가 있을 때 자식 프로세스 실행 마칠 때 background 함수 호출

→ 프롬프트는 exit안하면 계속 반복

→ 사용사 인터페이스가 출력이 안될 때 error 메세지 출력 + 셸 종료

→ 명령어 입력 받기 + error 나 EOF면 종료

→ 무슨 명령어를 사용하였는지 확인하기 위해 함수 호출

<보안>

좀비 프로세스가 발생하지 않도록 자식 프로세스를 종료 시켰는지 확인하는 코드를 fork 함수를 호출할 때마다 사용함

오버플로우를 방지하는 코드는 구현하지 않았지만 오버 플로우가 발생하지 못하도록 프로세스 마다 할당된 메모리를 stack의 오버플로우로 인해 넘어가지 못하도록 현재 메모리가 할당된 메모리가 넘는지 않 넘는지 확인하고 넘는다면 프로그램을 강제 종료하는 반복문이 있으면 좋을 것 같다고 생각함