2주차 과제

- 조건
  1. 현재 디렉토리 명을 쉘에 표시
  2. 일반적인 리눅스 bash 쉘의 사용자 인터페이스 형태를 띌 것
  3. cd, pwd 명령어를 반드시 구현 (상대경로 사용 가능해야 함, 단순히 exec을 사용해서는 안됨, 나머지 명령어는 exec 시스템콜을 이용해 구현해도 됨)
  4. 파이프라인 (멀티 파이프라인도 가능해야 함)
  5. 다중 명령어 (; , &&, ||)
  6. 백그라운드 실행(&)
  7. exit 입력 시 프로그램 종료
  8. 추가 명령어 및 옵션 직접 구현 시 가산점 부여
  9. 좀비 프로세스나 오버플로우 등 보안위협이 되는 요소들에 대한 고려가 있을 경우 가산점 부여 (보고서에 기록) ⇒ 고민의 흔적 자체를 긍정적으로 평가

1번부터 차례대로 구현하였습니다.

헤더 파일 분석

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

헤더	사용한 함수/기능
<pwd.h>	getpwuid
<unistd.h>	getcwd , gethostname , chdir , fork , execvp , dup2 , pipe
<string.h>	strtok , strchr , strcmp , strncmp , strlen , strcspn , strstr , strsep
<stdio.h>	printf , fgets , perror
<stdlib.h>	getenv , exit
<limits.h>	PATH_MAX

<sys/wait.h>	wait, waitpid

1. 현재 디렉토리 명을 쉘에 표시

```
void printUser(void) {
    char cwd[PATH_MAX];
    char hostname[_SC_HOST_NAME_MAX];
    char* username = getpwuid(getuid())→pw_name;

    gethostname(hostname, sizeof(hostname));
    getcwd(cwd, sizeof(cwd));

    const char* home = getenv("HOME");
    if (home && strstr(cwd, home) == cwd) {
        printf("[%s@%s:%s]$ ", username, hostname, cwd);
    } else {
        printf("[%s@%s:%s]$ ", username, hostname, cwd);
    }
}
```

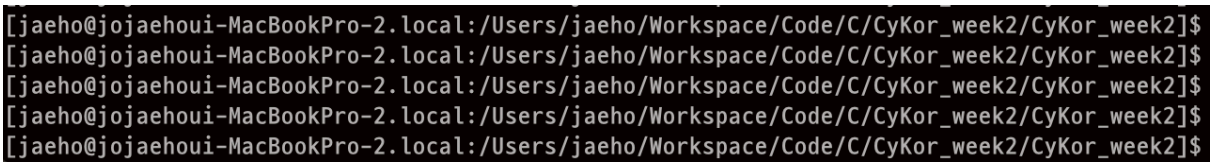
```
[jaeho@jojaehoui-MacBookPro-2.local:/Users/jaeho/Workspace/Code/C/CyKor_week2/CyKor_week2]$
[jaeho@jojaehoui-MacBookPro-2.local:/Users/jaeho/Workspace/Code/C/CyKor_week2/CyKor_week2]$
[jaeho@jojaehoui-MacBookPro-2.local:/Users/jaeho/Workspace/Code/C/CyKor_week2/CyKor_week2]$
[jaeho@jojaehoui-MacBookPro-2.local:/Users/jaeho/Workspace/Code/C/CyKor_week2/CyKor_week2]$
[jaeho@jojaehoui-MacBookPro-2.local:/Users/jaeho/Workspace/Code/C/CyKor_week2/CyKor_week2]$
```

printUser 함수 내에서 getcwd 함수를 통해 cwd 배열에 경로를 저장한 후, 현재 디렉토리 명을 쉘에 표시한다.

2. 일반적인 리눅스 bash 쉘의 사용자 인터페이스 형태를 띌 것

```
void printUser(void) {
    char cwd[PATH_MAX];
    char hostname[_SC_HOST_NAME_MAX];
    char* username = getpwuid(getuid())→pw_name;

    gethostname(hostname, sizeof(hostname));
    getcwd(cwd, sizeof(cwd));
```

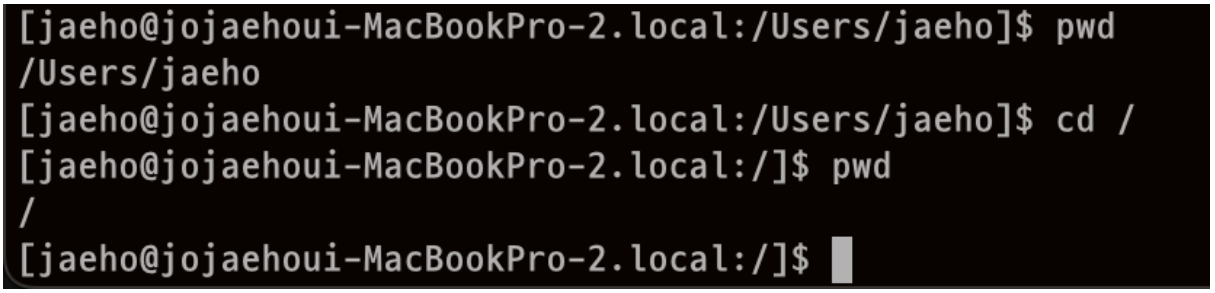
```
    const char* home = getenv("HOME");
    if (home && strstr(cwd, home) == cwd) {
        printf("[%s@%s:%s]$ ", username, hostname, cwd);
    } else {
        printf("[%s@%s:%s]$ ", username, hostname, cwd);
    }
}
```



- 1번과 마찬가지로, 리눅스 bash 쉘의 사용자 인터페이스는 [username@hostname:cwd]$ 형태를 띄고 있다.
- 따라서 `<unistd.h>` 헤더파일의 getcwd 함수를 이용해 username과 gethostname 함수를 이용해 username, hostname, cwd를 구할 수 있었다.

3. cd, pwd 명령어를 반드시 구현 (exec 를 제외한 구현)

```
    if (strcmp(options[0], "cd") == 0) {
        const char* target = options[1];
        if (target == NULL || strcmp(target, "~") == 0) target = getenv("HOME");
        if (target && chdir(target) != 0) printf("Cannot execute cd");
        return 0;
    }
    if (strcmp(options[0], "pwd") == 0) {
        char pwd[PATH_MAX];
        if (getcwd(pwd, sizeof(pwd)) != NULL) printf("%s\n", pwd);
        else printf("Cannot execute pwd");
        return 0;
    }
```



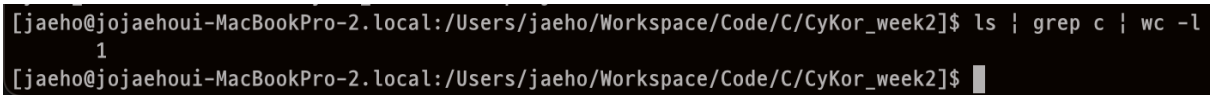
명령어를 options 배열의 0번째 인덱스로 저장해 cd, pwd를 구분하여 cd인 경우에 1번째 인덱스로 들어오는 경로를 chdir 함수를 이용하여 상대경로로 이동할 수 있도록 개발하였다.

또한 pwd인 경우에 PATH_MAX 상수를 이용하여 오버플로우를 방지하고 현재 위치를 출력할 수 있도록 개발하였다.

4. 파이프라인 (멀티 파이프라인도 가능해야 함)

```
        if (strchr(cmdLine, '|') != NULL) {
            pipelineFunc(cmdLine);
            continue;
        }
//main 함수의 코드 부분



void pipelineFunc(char* cmdLine) {
    char* commands[16];
    int num_cmds = 0;

    char* token = strtok(cmdLine, "|");
    while (token != NULL && num_cmds < 16) {
        while (*token == ' ') token++;
        commands[num_cmds++] = token;
        token = strtok(NULL, "|");
    }

    int prev_pipe[2] = {-1, -1};

    for (int i = 0; i < num_cmds; i++) {
        int pipefd[2];
```

```
        if (i < num_cmds - 1) pipe(pipefd);

        pid_t pid = fork();
        if (pid == 0) {
            if (i > 0) {
                dup2(prev_pipe[0], 0);
                close(prev_pipe[0]);
                close(prev_pipe[1]);
            }

            if (i < num_cmds - 1) {
                close(pipefd[0]);
                dup2(pipefd[1], 1);
                close(pipefd[1]);
            }

            char* args[MAX_ARGS];
            inputParser(commands[i], args);
            execvp(args[0], args);
            printf("Cannot execute execvp");
            exit(1);
        }

        if (i > 0) {
            close(prev_pipe[0]);
            close(prev_pipe[1]);
        }

        if (i < num_cmds - 1) {
            prev_pipe[0] = pipefd[0];
            prev_pipe[1] = pipefd[1];
        }
    }

    for (int i = 0; i < num_cmds; i++) {
        wait(NULL);
    }
}
```



- insertParser 함수가 실행되기 전에 입력한 문자열에서 파이프라인을 감지하여 먼저 수행할 수 있도록 개발하였다.
- 파이프라인을 기준으로 strtok 함수를 사용해 나눠준 뒤 while (*token == ' ') token++; 를 사용하여 빈칸을 없애주었다.
- prev_pipe로 이전 명령어의 출력값을 저장하고 `dup2()`를 사용해 현재 명령어의 표준 입력(STDIN)을 이 파이프에 연결할 수 있도록 준비했다.
- `i > 0`이면 앞에 명령어가 있는 상태이므로, 그 명령어의 출력을 **현재 명령어의 입력으로 연결하여** 파이프 쓰기 쪽을 `stdout`으로 바꾸는 방향으로 개발하였다.

5. 다중 명령어 (; , &&, ||)

```
while ((token = strsep(&rest, ";&|")) != NULL) {
    while (*token == ' ') token++;
    if (strlen(token) == 0) continue;

    int run = 1;
    if (op) {
        if (strcmp(op, "&&") == 0 && last_status != 0) run = 0;
        if (strcmp(op, "||") == 0 && last_status == 0) run = 0;
    }

    if (run) {
        last_status = execute_command(token);
    }

    if (rest) {
        op = rest;
        if (strncmp(op, "&&", 2) == 0 || strncmp(op, "||", 2) == 0) {
            rest += 2;
        } else {
            rest += 1;
        }
```

```
        }
    }
```

```
[jaeho@jojaehoui-MacBookPro-2.local:/Users/jaeho/Workspace/Code/C/CyKor_week2]$ false || echo failed
failed
[jaeho@jojaehoui-MacBookPro-2.local:/Users/jaeho/Workspace/Code/C/CyKor_week2]$
```

다중 명령어는 `cmdLine`을 `";&|"` 기준으로 자르는데,strcmp 함수를 통해 어떤 다중 명령어인지 파악하고, `strsep()` 함수를 통해 하나씩 나누며 rest를 앞으로 이동시켜 `cd test && ls || echo fail ; pwd` 와 같은 코드들을 순서대로 `cd test` , `ls` , `echo fail` , `pwd` 방식으로 추출할 수 있다.

6. 백그라운드 실행(&)

```
char* amp = strchr(cmdLine, '&');
if (amp) {
    background = 1;
    *amp = '\0';
}

    // 외부 명령 실행
    pid_t pid = fork();
    if (pid == 0) {
        execvp(options[0], options);
        printf("Cannot execute execvp");
        exit(1);
    } else {
        if (!background) waitpid(pid, NULL, 0);
        else printf("[BG PID %d started]\n", pid);
        return 0;
  }
```

```
[jaeho@jojaehoui-MacBookPro-2.local:/Users/jaeho/Workspace/Code/C/CyKor_week2]$ sleep 5 & echo hello
hello
[jaeho@jojaehoui-MacBookPro-2.local:/Users/jaeho/Workspace/Code/C/CyKor_week2]$
```

strchr 함수를 통해서 입력한 명령어 중 '&'기호를 찾아 문자열에 포함이 된다면 외부 명령 실행 중 background 변수에 걸려 백그라운드에서 실행할 수 있도록 개발하였다.

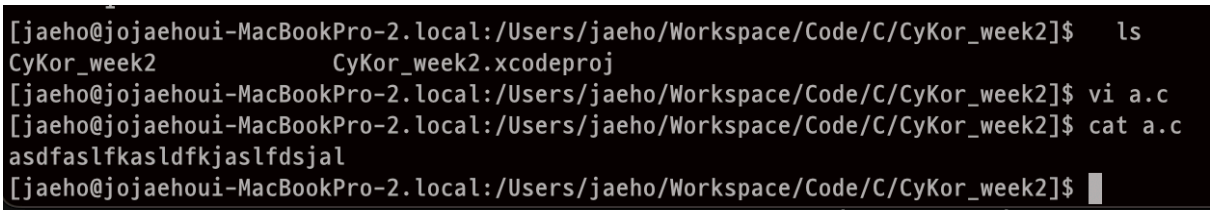
7. exit 입력 시 프로그램 종료

```
if (strcmp(options[0], "exit") == 0) exit(0);
```

- 파싱한 0번째 인덱스인 명령어가 exit이라면 exit 함수를 통해 프로그램이 종료할 수 있도록 개발하였다.

8. 추가 명령어 및 옵션 직접 구현 시 가산점 부여

```
pid_t pid = fork();
if (pid == 0) {
    execvp(options[0], options);
    printf("Cannot execute execvp");
    exit(1);
} else {
    if (!background) waitpid(pid, NULL, 0);
    else printf("[BG PID %d started]\n", pid);
    return 0;
}
```



- fork 함수를 사용하여 pid가 0이면 (자식 프로세스 실행 됨)options의 0번째 인덱스인 명령어와 명령어에 해당되는 옵션들 (options)을 인자로 넣어 다른 프로그램들을 실행시킬 수 있다.
- 이로서 추가 명령어들을 쉘 안에서 실행시킬 수 있다.

9. 좀비 프로세스나 오버플로우 등 보안위협이 되는 요소들에 대한 고려가 있을 경우 가산점 부여

```
#include <limits.h>
#define MAX_LINE 4096
#define MAX_ARGS 64
```

```
char cmdLine[MAX_LINE];
if (!fgets(cmdLine, MAX_LINE, stdin)) {
char cwd[PATH_MAX];
char hostname[_SC_HOST_NAME_MAX];
```

limits.h 파일을 사용하여 위와 같은 코드들의 제한을 걸어 오버플로우를 방지할 수 있다.