2주차 과제 탐구 보고서

목차

가. 사용된 헤더파일(1~7)

작성 코드

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

설명

1. #include <stdio.h>
: 표준 입출력 함수들, 예: printf, scanf, fgets 등

2. #include <stdlib.h>
: 메모리 할당, 프로그램 종료, 문자열 변환 함수들 ex) malloc, free 등

3. #include <string.h>
: 문자열 처리 함수들, 예: strcpy, strlen, strcmp, strtok 등

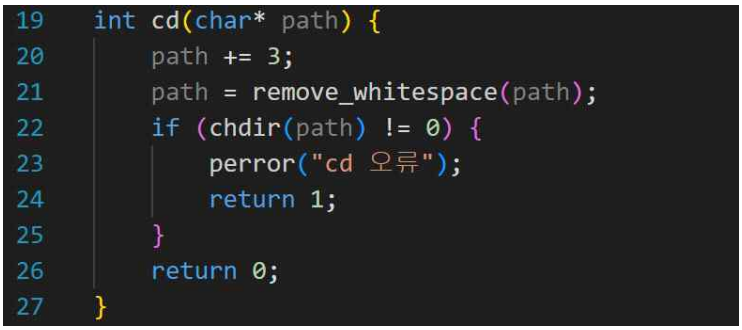
4. #include <unistd.h>
: 유닉스/리눅스 시스템 호출 함수들, 예: fork, exec, chdir 등

5. #include <sys/wait.h>
: 자식 프로세스 관련 처리 함수들, 예: wait, waitpid 등

6. #include <signal.h>
: 시그널 처리 관련 함수 및 상수 백그라운드를 위해서 사용, 예: signal 등

7. #include <errno.h>
: 에러 번호를 정의한 헤더, 예: errno 전역 변수 같은 에러 코드

나. cd 함수(19~27)

작성 코드	
설명	20~21: path에 저장된 문자열에서 "cd " 부분을 뛰어넘기 위해서 +3을 해주고 오류가 생기지 않기 위해서 remove_whitespace를 이용해서 양끝의 공백을 제거하여 이동하기 위한 주소만 남겨둔다. 22~26: chdir(path)를 통해 현재 작업 디렉토리를 path로 변경한다. chdir()이 성공하면 0, 실패하면 –1을 반환하기 때문에 실패했을 때 perror를 통해 오류를 출력하고 1을 반환하고, 성공하면 0을 반환한다.

```
19  int cd(char* path) {
20      path += 3;
21      path = remove_whitespace(path);
22      if (chdir(path) != 0) {
23          perror("cd 오류");
24          return 1;
25      }
26      return 0;
27  }
```

다. pwd 함수(29~38)

작성 코드	
설명	31~37: getcwd()를 통해서 cwd에 현재 작업중인 디렉토리 주소를 불러온다. 저장에 성공하면 printf를 통해 주소를 출력하고, 실패하면 null을 반환하여 else문안에 있는 perror를 출력한다. 또한 각각 호출한곳에 성공유무를 알려주기 위해 성공하면 0, 실패하면 1을 반환한다.

```
29  int pwd() {
30      char cwd[100];
31      if (getcwd(cwd, sizeof(cwd))) {
32          printf("%s\n", cwd);
33          return 0;
34      } else {
35          perror("pwd 오류");
36          return 1;
37      }
38  }
```

라. 파이프라인(41~93)

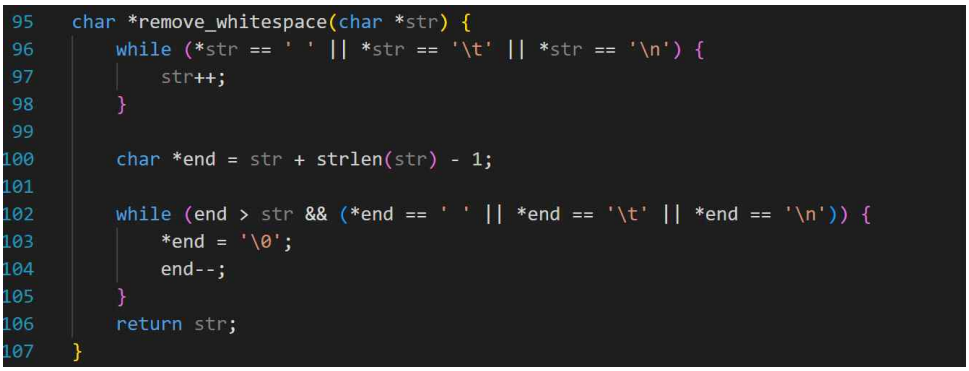
작성 코드

```c
void multi_pipe(char *command) {
    char *cmds[10];
    int num = 0;

    char *token = strtok(command, "|");
    while (token != NULL && num < 10) {
        token = remove_whitespace(token);
        if (strlen(token) > 0) {
            cmds[num++] = token;
        }
        token = strtok(NULL, "|");
    }

    int prev_fd = -1;

    for (int i = 0; i < num; i++) {
        char *args[100];
        split_args(cmds[i], args);

        int pipe_fd[2];
        if (i < num - 1 && pipe(pipe_fd) < 0) {
            perror("pipe 실패");
            exit(1);
        }
```

```c
        pid_t pid = fork();
        if (pid == 0) {
            if (prev_fd != -1) {
                dup2(prev_fd, 0);
                close(prev_fd);
            }
            if (i < num - 1) {
                close(pipe_fd[0]);
                dup2(pipe_fd[1], 1);
                close(pipe_fd[1]);
            }

            execvp(args[0], args);
            perror("명령 실행 실패");
            exit(1);
        }
        else {
            if (prev_fd != -1) {
                close(prev_fd);
            }
            if (i < num - 1) {
                close(pipe_fd[1]);
                prev_fd = pipe_fd[0];
            }
            waitpid(pid, NULL, 0);
        }
    }
}
```

설명	45~52: "|"을 기준으로 명령어들을 쪼개고 remove_whitespace를 이용해서 양쪽 공백을 제거한다. 그 다음 각각을 순서대로 cmd 배열에 저장한다. 54: 이전 명령어의 읽기 파이프 끝을 저장하기 위한 변수. 56: 파이프에 있는 명령어들을 실행하기 위한 반복문. 57~58: cmds에서 명령어를 불러와 split_args()를 통해서 실행가능한 형태로 만든다. 60~64: 마지막 명령어가 아니라면, 새로운 파이프를 생성하여 읽기/쓰기 파일 디스크립터 생성. 만약 실패 시 perror 출력 후 종료. 66~81: 만약 자식 프로세스라면, 이전 명령어의 출력이 현재 명령어의 입력으로 들어오도록 dup2(prev_fd, 0), 다음 명령어가 존재한다면, 현재 명령어의 출력을 파이프로 보내기 위해 dup2(pipe_fd[1], 1)을 사용한다. 마지막으로 execvp()로 명령어를 실행하고, 실패 시 perror 출력 후 종료. 82~91: 만약 부모 프로세스라면 자식프로세스가 끝날때까지 waitpid()기다린다. 종료후에는 필요한 읽기 파이프만 유지한다.

마. 공백 제거 함수(95~107)

작성 코드	
설명	96~98: while문을 통해 받은 명령어의 문자열 앞부분의 공백을 제거. 100: 문자열의 마지막 주소를 변수에 저장. 102~105: while문을 통해 받은 명령어의 문자열 뒷부분의 공백을 제거.

```
95  char *remove_whitespace(char *str) {
96      while (*str == ' ' || *str == '\t' || *str == '\n') {
97          str++;
98      }
99
100     char *end = str + strlen(str) - 1;
101
102     while (end > str && (*end == ' ' || *end == '\t' || *end == '\n')) {
103         *end = '\0';
104         end--;
105     }
106     return str;
107 }
```

바. 다중 연산자(109~164)

작성 코드

```
109  void logical_command(char *logic) {
110      char *now = logic;
111      int last = 0;
112      int run_next = 1;
113
114      while (*now != '\0') {
115          char *next_and = strstr(now, "&&");
116          char *next_or  = strstr(now, "||");
117          char *next_semi = strchr(now, ';');
118
119          char *next = NULL;
120          int type = 0;
121
122          if (next_and && (!next || next_and < next)) {
123              next = next_and;
124              type = 1;
125          }
126
127          if (next_or && (!next || next_or < next)) {
128              next = next_or;
129              type = 2;
130          }
131
132          if (next_semi && (!next || next_semi < next)) {
133              next = next_semi;
134              type = 3;
135          }
136
137          char *segment = now;
138          if (next != NULL) {
139              *next = '\0';
140          }
141
142          segment = remove_whitespace(segment);
143
144          if (run_next && strlen(segment) > 0) {
145              last = single_command(segment);
146          }
147
148          if (next != NULL) {
149              if (type == 1) {
150                  run_next = (last == 0);
151              }
152              else if (type == 2) {
153                  run_next = (last != 0);
154              }
155              else {
156                  run_next = 1;
157              }
158              now = next + (type == 3 ? 1 : 2);
159          }
160          else {
161              break;
162          }
163      }
164  }
```

설명	110: 현재 처리 중인 명령어의 위치를 저장하는 포인터 변수. 111: 명령어의 실행 결과를 저장하는 변수. 논리 연산자에 따라 다음 명령어의 실행 여부를 결정하는 데 사용한다. 112: run_next는 다음 명령어를 실행할지 말지를 결정하는 변수. 초기값은 1로 설정되어 있고, 기본적으로 첫 번째 명령어는 실행된다. 114: now의 위치가 명령어의 끝까지 갈 때 동안 반복문을 실행한다. 115~117: strstr(curr, "&&"), strstr(curr, "||"), strchr(curr, ';')를 이용해 각각 &&, ||, ;가 현재 위치 이후에 있는지 찾는다. 그 후 찾은 위치를 next_and, next_or, next_semi에 저장한다. 119: 가장 먼저 나오는 논리 연산자 위치를 변수에 저장한다. 120: 발견된 연산자의 종류를 &&는 1, ||는 2, ;는 3으로 type 변수를 설정한다. 122~135: next_and가 존재하고 next가 아직 설정되지 않았거나, next_and가 next보다 더 앞에 있으면 next를 next_and로 설정하고, type을 1로 설정한다. next_or와 next_semi도 마찬가지 방식으로 작동한다. 137~140: 현재 명령어를 segment 변수에 저장한다. next가 존재하면, next 위치에 '₩0'을 넣어 segment가 논리 연산자 앞까지의 명령어만 포함하게 만든다. 142: remove_whitespace를 통해 명령어 앞뒤의 공백을 제거한다. 144~146: run_next가 1일 경우, 즉 이전 명령어가 실행되어야 한다면 single_command(segment)를 통해 명령어를 실행하고, 그 결과를 last에 저장한다. 148~161: next가 존재하면, 해당 논리 연산자에 맞춰 run_next 값을 설정한다. - type == 1 (&&): last == 0이면 다음 명령어를 실행하도록 run_next를 1로 설정. - type == 2 (||): last != 0이면 다음 명령어를 실행하도록 run_next를 1로 설정. - type == 3 (세미콜론): 항상 run_next = 1로 설정하여 무조건 다음

	명령어가 실행되도록 한다. now를 다음 명령어로 이동하고 type == 3일 경우 next + 1로 이동하여 ; 다음부터 시작하고, 그렇지 않으면 next + 2로 이동하여 논리 연산자 다음부터 시작한다. 만약 next가 NULL이면 반복문을 종료시킨다.

사. 백그라운드(166~177)

작성 코드

```
166  void background(int check) {
167      int saved_errno = errno;
168      pid_t pid;
169      int status;
170
171      while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
172          printf("[백그라운드 종료] pid: %d\n", pid);
173          fflush(stdout);
174      }
175
176      errno = saved_errno;
177  }
```

설명

167: errno 값을 백업해둔다. waitpid 실행 중 오류가 발생해도 기존의 오류 상태를 보존하기 위해서이다.

168: 자식 프로세스의 pid를 저장할 변수 pid 선언.

169: 자식 프로세스 종료 상태를 저장할 변수 status 선언.

171~174: waitpid(-1, &status, WNOHANG)을 사용하여 좀비 프로세스를 회수한다. WNOHANG은 종료된 자식이 없으면 기다리지 않고 바로 반환하게 한다. 자식 프로세스가 종료되었다면 해당 pid를 출력한다. 출력 후 fflush(stdout)으로 출력 버퍼를 비워 즉시 출력되도록 한다.

176: 함수 시작 시 백업해둔 errno 값을 복원하여, 이 함수에서의 waitpid 호출이 외부 오류 상태에 영향을 주지 않도록 한다.

아. 문장 분리 함수(179~188)

작성 코드

```
179  void split_args(char *input, char **args) {
180      int i = 0;
181      char *token = strtok(input, " ");
182
183      while (token != NULL && i < 99) {
184          args[i++] = remove_whitespace(token);
185          token = strtok(NULL, " ");
186      }
187      args[i] = NULL;
188  }
```

설명

180~181: 입력 문자열을 strtok 함수를 사용하여 공백 기준으로 나누어 첫 번째 토큰을 추출한다.

183~186: token이 NULL이 아니고 개수가 99 미만일 때까지 반복하여 토큰을 하나씩 remove_whitespace를 사용하여 각 토큰의 앞뒤 공백을 제거한 후 저장하고, 이후 strtok으로 다음 토큰을 얻는다.

187: 마지막에 args[i]를 NULL로 설정하여 명령어 함수들이 이 배열의 끝을 인식할 수 있도록 한다.

자. 단일 명령어 실행 함수(190~235)

작성 코드

```c
int single_command(char *command) {
    command = remove_whitespace(command);

    int background = 0;
    size_t len = strlen(command);
    if (len > 0 && command[len - 1] == '&') {
        background = 1;
        command[len - 1] = '\0';
        command = remove_whitespace(command);
    }

    if (strcmp(command, "exit") == 0) {
        exit(0);
    }
    if (strncmp(command, "cd ", 3) == 0) {
        return cd(command);
    }
    if (strcmp(command, "pwd") == 0) {
        return pwd();
    }
    if (strchr(command, '|')) {
        multi_pipe(command);
        return 0;
    }

    char *args[100];
    split_args(command, args);
```

```c
    pid_t pid = fork();
    if (pid == 0) {
        execvp(args[0], args);
        perror("실행 실패");
        exit(1);
    }
    else {
        if (!background) {
            int status;
            waitpid(pid, &status, 0);
            return WEXITSTATUS(status);
        }
        else {
            printf("[백그라운드 실행] pid: %d\n", pid);
            return 0;
        }
    }
}
```

설명

191: 받은 문자열의 앞뒤 공백을 제거하여 명령어를 만든다.

193~199: 명령어 끝이 &이면 백그라운드 실행임을 의미한다. 그러므로 background를 1로 설정하고, &를 제거한 후 다시 공백을 제거해 명령어를 정리한다.

201~203: strcmp를 이용해 명령어의 가장 앞단어가 "exit"이라면 프로그램을 종료시킨다.

	204~206: strcmp를 이용해 명령어의 가장 앞부분 3개가 "cd"라면 cd 함수를 호출한다. cd 함수가 성공하면 0을 반환하고 실패하면 1을 반환한다. 207~209: strcmp를 이용해 명령어의 가장 앞단어가 "pwd"라면 pwd 함수를 호출한다. pwd 함수가 성공하면 0을 반환하고 실패하면 1을 반환한다. 210~213: strchr를 이용해 명령어에 | 문자가 포함되어 있는지 판단하고, 포함되어 있다면 multi_pipe() 함수를 호출한다. 215~216: 명령어를 split_args를 통해 실행할 수 있는 형태로 쪼갠다. 218: 자식 프로세스를 생성한다. 219~223: 만약 자식 프로세스라면 execvp()를 이용해서 cd,pwd를 제외한 나머지 명령어를 실행한다. 224~234: 만약 부모 프로세스라면 백그라운드 실행중이 아니라면 waitpid로 자식 프로세스의 종료를 기다리고 종료 상태를 반환한다. 백그라운드가 실행중이면 메시지만 출력하고 기다리지 않는다.

차. main(237~258)

작성 코드

```
237  int main() {
238      signal(SIGCHLD, background);
239
240      char command[100];
241
242      while (1) {
243          char cwd[100];
244          getcwd(cwd, sizeof(cwd));
245          printf("%s$ ", cwd);
246          fflush(stdout);
247
248          if (!fgets(command, sizeof(command), stdin)) {
249              break;
250          }
251
252          command[strcspn(command, "\n")] = 0;
253
254          logical_command(command);
255      }
256
257      return 0;
258  }
```

설명

238: 자식 프로세스가 종료될 때, SIGCHLD 시그널 발생하면 background 함수를 호출한다.

244~246: getcwd()를 통해 현재 디렉토리의 경로를 문자열로 cwd에 저장하고 출력한다. 현재 디렉토리 경로를 출력하고 $를 붙여 입력을 유도한다. fflush(stdout)를 이용해 출력 버퍼를 비워 즉시 화면에 출력되도록 보장한다.

248~250: 사용자로부터 한 줄 입력을 받아 command에 저장한다. fgets()가 NULL을 반환하면 입력 오류이므로 루프를 종료한다.

252: 입력된 명령어의 끝에 있는 '₩n'를 '₩0'로 바꿔 문자열을 끝을 프로그램이 인식할 수 있도록 지정한다.

254: command를 logical_command 함수로 전달하여 명령어들을 실행시킨다.

카. 보안 위협을 고려한 요소들

1. 좀비 프로세스 제거

가) 좀비 프로세스란?

: 실행은 끝났지만, 부모가 수거하지 않아 시스템에 남아 있는 자식 프로세스의 껍데기.

나) 좀비 프로세스의 문제점

: 좀비가 쌓이면 PID를 점점 잡아먹고, 스텍 오버플로우와 같이 시스템에서 더 이상 프로세스를 만들 수 없는 상황도 생긴다. 또한, 좀비 프로세스를 일부러 쌓아서 시스템을 느려지게 만드는 간단한 DoS 공격 수단이 되기도 한다.

다) 내가 적용한 해결 방법

: signal(SIGCHLD, handler)를 사용해, 자식 종료 시 자동으로 waitpid(-1, ..., WNOHANG)를 통해 좀비 프로세스 수거.

2. 파일 디스크립터(FD) 및 리소스 누수

가) 파일 디스크립터란?

: 열어놓은 파일 디스크립터를 제대로 닫지 않아서, 사용하지도 않으면서 시스템 자원을 점점 차지하는 현상.

나) 파일 디스크립터(FD) 및 리소스 누수의 문제점

: pipe()나 dup2()로 생성된 FD를 닫지 않으면 리눅스의 FD 제한을 초과해, 입출력 오류나 프로세스 고장 발생 가능성이 있다.

다) 내가 적용한 해결 방법

: 파이프 사용 후 불필요한 FD는 항상 close()로 정리, dup2() 이후에도 원본 FD를 닫아 리소스 누수를 방지하였다.

3. 개행 파싱 오류

가) 파싱의 중요성

: 파싱을 통해 명령어 조작 공격을 예방하고 입력 오류로 인한 오동작을 차단, 내부/외부 명령의 안전한 구분 실행 가능하다.

나) 내가 적용한 해결 방법

: remove_whitespace()와 split_args() 함수를 만들어 모든 명령어와 인자에서 정확하게 분리하고 앞뒤 공백 제거함으로써 파싱 안정성과 명령 실행의 정확도 향상.