2024 시스템 프로그래밍 과제 4

1분반 산업보안학과 박기쁨 32231594

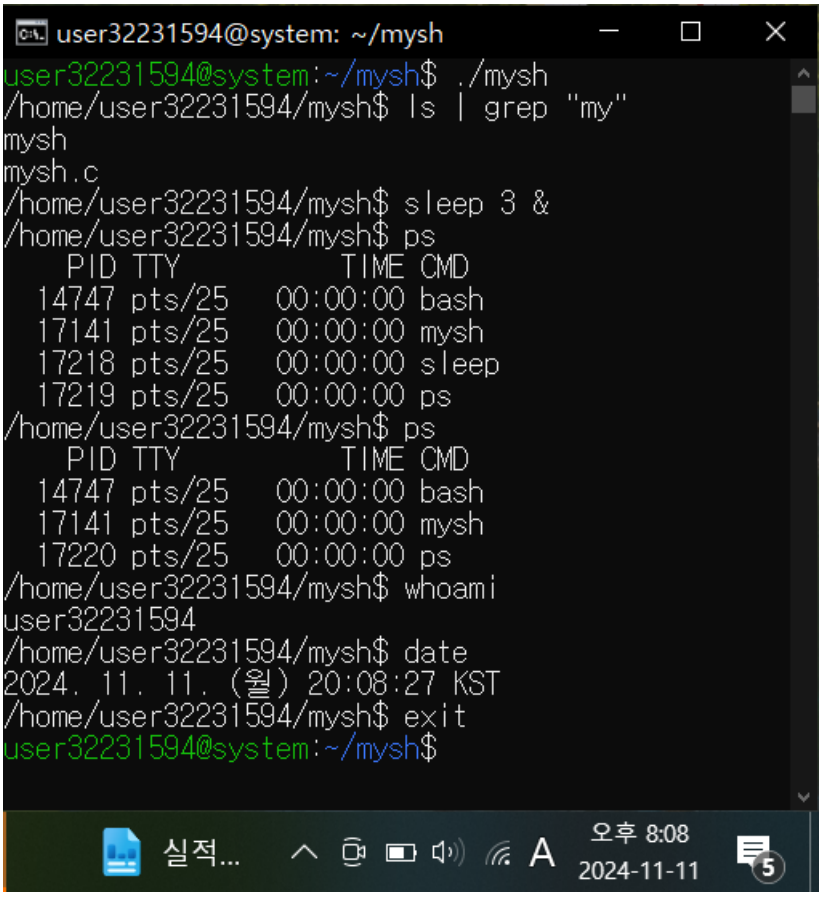
1. mysh 만들기 (필수)

** ssh로 서버에 접속하여 수행. 화면 캡쳐 시 user{학번}@{서버_이름} 출력*

** 쉘 실행 후 명령어 수행 결과 캡쳐*

** 수행 결과에 대해 간단한 설명*

** 추가 구현(pipe, redirection, background)에 대한 설명*



1. **./mysh** : mysh 실행

2. **Enter** : 아무 명령어도 입력하지 않아도 쉘 재실행

3. **help** : 도움말 출력

4. **ls** : 현재 directory의 list 출력

5. **echo "Hello" > test.txt** : "Hello"를 test.txt에 입력 (이후 ls를 통해 list 출력)

6. **cat test.txt** : test.txt의 내용을 출력

7. **echo "Hi" >> test.txt** : "Hi"를 test.txt에 이어서 입력 (이후 cat을 통해 내용 출력)

8. **rm test.txt** : test.txt 파일 제거 (이후 ls를 통해 list 출력)

9. **quit** : 쉘 종료

10. **ls | grep "my"** : list 중 'my'가 포함된 파일 출력

12. **sleep 3 &** : background에서 sleep 3초 수행

12. **ps** : background에서 수행 중인 sleep 확인, 3초 후 다시 ps하면 sleep 종료되어있음

13. **exit** : 쉘 종료

2. 코드 설명 (필수)

** 주요 함수 동작 과정 설명*

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

- **헤더** : 코드 실행에 필요한 헤더파일 포함

```c
bool cmd_help(int argc, char* argv[]) {
        printf("₩n        < HELP >₩n₩n");
        printf("* help : show this help₩n");
        printf("* exit : exit the shell₩n");
        printf("* quit : quit the shell₩n₩n");
        printf("* can use external commands₩n");
        printf("* can use |, >, >>, &₩n₩n");
        return true;
}
```

- **bool cmd_help** : 'help' 명령어를 입력했을 때 호출되어 문장을 출력하는 함수,

```c
int tokenize(char* buf, char* delims, char* tokens[], int maxTokens) {
        int token_count = 0;
        char* token = NULL;

        token = strtok(buf, delims);

        while(token != NULL && token_count < maxTokens) {
                tokens[token_count] = token;
                token = strtok(NULL, delims);
                token_count++;
        }

        tokens[token_count] = NULL;
        return token_count;
}
```

- **int tokenize** : buf의 내용을 delims를 기준으로 한 토큰 단위로 분할한 뒤 tokens 배열에 저장하는 함수, 배열의 마지막에 NULL 저장, token의 개수를 return

```c
bool run(char* line) {
        char* delims = " ₩"₩t₩n";
        char* tokens[64];
        int token_count = 0;

        token_count = tokenize(line, delims, tokens, sizeof(tokens) / sizeof(char*));
        if(token_count == 0) {
                return true;
        }
```

- **bool run** : 명령어를 수행하는 주요 함수, 토큰의 기준이 되는 delims는 공백/큰따옴표/탭/줄바꿈, tokenize 함수를 호출하여 return 값인 토큰 개수를 token_count 변수에 저장

```
        // help, exit, quit
        if(strcmp(tokens[0], "help") == 0) {
                return cmd_help(token_count, tokens);
        } else if(strcmp(tokens[0], "exit") == 0 || strcmp(tokens[0], "quit") == 0) {
                        return false;
        }
```

- tokens 배열의 첫 번째 토큰인 명령어가 help이면 **cmd_help** 호출, exit이거나 quit이면 false를 반환하여 이후 **main** 함수의 while문에서 break가 수행되어 프로그램 종료

```
        // &
        bool background = false;
        if(strcmp(tokens[token_count - 1], "&") == 0) {
                background = true;
                tokens[--token_count] = NULL;
        }
```

- 백그라운드를 판별할 bool 변수 선언, 마지막 토큰이 &이면 백그라운드 변수를 true로 변경한 후 마지막 토큰을 NULL로 변경

```
        // external
        pid_t pid;
        pid = fork();

        if(pid < 0) {
                perror("fork failed");
                return true;

        } else if(pid == 0) {
```

- 외부 명령어 실행 관련 코드 시작, fork()를 통해 자식프로세스 생성 후 정상적으로 fork()가 진행되지 않았을 시 에러 핸들링 (이후 코드는 자식프로세스가 정상적으로 생성되었을 때의 상황)

```
                // redirection
                int fd;

                for(int i = 0; i < token_count; i++) {
                        if(strcmp(tokens[i], ">") == 0) {
                                tokens[i] = NULL;
                                fd = open(tokens[i + 1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
                                if (fd < 0) {
                                        perror("open failed");
                                        exit(1);
                                }
                                dup2(fd, STDOUT_FILENO);
                                execvp(tokens[0], tokens);
                                close(fd);
                                break;
```

- '>'를 판별하기 위해 반복문 수행, '>' 발견 시 NULL로 변경한 후 '>' 이후 파일명의 파일 open, 파일이 있다면 덮어쓰기, open이 정상적으로 실행되지 않았을 시 에러 핸들링, 정상적으로 실행되었을 시 표준출력이 파일을 가리키도록 설정 후 명령 수행

```
                } else if(strcmp(tokens[i], ">>") == 0) {
                        tokens[i] = NULL;
                        fd = open(tokens[i + 1], O_WRONLY | O_CREAT | O_APPEND, 0644);
                        if(fd < 0) {
                                perror("open failed");
                                exit(1);
                        }
                        dup2(fd, STDOUT_FILENO);
                        execvp(tokens[0], tokens);
                        close(fd);
                        break;
                }
        }
```

- '>>'를 판별하기 위해 반복문 수행, '>>' 발견 시 NULL로 변경한 후 '>>' 이후 파일명의 파일 open, 파일이 있다면 이어서 쓰기, open이 정상적으로 실행되지 않았을 시 에러 핸들링, 정상적으로 실행되었을 시 표준출력이 파일을 가리키도록 설정 후 명령 수행

```
        //pipe
        int pipe_fd[2]; // [0]: read, [1]: write

        for(int i = 0; i < token_count; i++) {
                if(strcmp(tokens[i], "|") == 0) {
                        tokens[i] = NULL;

                        if(pipe(pipe_fd) == -1) {
                                perror("pipe failed");
                                exit(1);
                        }
```

- pipe 수행을 위한 전용 파일 디스크립터 선언, '|'를 판별하기 위해 반복문 수행, '|' 발견 시 NULL로 변경, pipe에 실패했다면 에러 핸들링

```
                        pid_t pid2;
                        pid2 = fork();

                        if(pid2 < 0) {
                                perror("fork failed");
                                exit(1);

                        } else if(pid2 == 0) {   // child of child (1st command)
                                dup2(pipe_fd[1], STDOUT_FILENO);
                                close(pipe_fd[0]);
                                close(pipe_fd[1]);
                                execvp(tokens[0], tokens);
                                exit(1);
```

- pipe 양쪽의 명령을 수행하기 위해 fork()를 통해 자식 프로세스 생성, 정상적으로 수행되지 않았다면 에러 핸들링, 정상적으로 생성된 자식 프로세스는 첫 번째 명령어 담당, 쓰기 전용 파일 디스크립터를 표준 출력과 연결, 파일 디스크립터 닫음, 첫 번째 토큰 명령어 수행하여 return 값을 pipe로 전달

```
                        } else {                // child (2nd command)
                                dup2(pipe_fd[0], STDIN_FILENO);
                                close(pipe_fd[1]);
                                close(pipe_fd[0]);
                                execvp(tokens[i + 1], &tokens[i + 1]);
                                perror("execvp failed");
                                exit(1);
                        }
                }
        }
```

- 수행되고 있던 프로세스(쉘의 자식 프로세스)는 읽기 전용 파일 디스크립터를 표준 입력과 연결, 파일 디스크립터 닫음, 첫 번째 명령어의 return 값을 가져와 '|' 이후 명령어 실행, 오류 발생 시 에러 핸들링

```
                execvp(tokens[0], tokens);
                perror("execvp failed");
                exit(1);

        } else {
                if(background == false)
                        wait(NULL);
        }
        return true;
}
```

- redirection이나 pipe가 아닌 외부 명령어를 execvp를 통해 수행, 쉘은 background 변수가 false 상태이면 자식 프로세스를 기다림, 만약 토큰 중 '&'를 발견한다면 background 변수가 true가 되어 자식 프로세스를 기다리지 않음으로써 자식 프로세스는 background에서 실행

```
int main() {
        char line[1024];
        char cwd[64];
        while(1) {
                printf("%s$ ", getcwd(cwd, 64));
                fgets(line, sizeof(line) -1, stdin);
                while(waitpid(-1, NULL, WNOHANG) > 0);   // 좀비프로세스 방지
                if(run(line) == false) break;
        }
        return 0;
}
                                                                140,9-16      Bot
```

- **int main** : while문을 통해 쉘 반복 실행, 기본적으로 현재 디렉토리와 $를 출력, 이후 입력한 문장은 line에 저장, while문의 WNOHANG 옵션으로 인해 종료된 자식 프로세스의 pid를 반환받아 대기 종료 (background의 좀비 프로세스 방지), run을 호출하여 return값이 false가 되면 while문이 종료되어 쉘 종료