

2024 시스템 프로그래밍 과제 4

1분반 산업보안학과 김가람 32230324

1. mysh 만들기 (필수)

- * ssh로 서버에 접속하여 수행. 화면 캡처 시 user{학번}@{서버_이름} 출력
- * 셸 실행 후 명령어 수행 결과 캡처
- * 수행 결과에 대해 간단한 설명
- * 추가 구현(pipe, redirection, background)에 대한 설명

```
user32230324@system:~/mysh$ ./mysh
/home/user32230324/mysh $
/home/user32230324/mysh $ ls
mysh  mysh.c
/home/user32230324/mysh $ echo "Hello" > test.txt
/home/user32230324/mysh $ cat test.txt
Hello
/home/user32230324/mysh $ rm test.txt
/home/user32230324/mysh $ ls
mysh  mysh.c
/home/user32230324/mysh $
/home/user32230324/mysh $ help
/*****simple shell*****/
<Internal command>
quit : quit the shell
exit : quit the shell
help : Show the help of the commands
<External command>
ls : Display a list of files and folders in the directory
cat : Print the contents of a file or combine files
rm : Delete a file or directory
There are a variety of other external commands.
/*****/
/home/user32230324/mysh $ whoami
user32230324
/home/user32230324/mysh $ data
execvp failed
/home/user32230324/mysh $ date
2024. 11. 11. (월) 02:27:06 KST
/home/user32230324/mysh $ quit
```

```
user32230324@system:~/mysh$ ./mysh
/home/user32230324/mysh $ echo "Hello" > test.txt
/home/user32230324/mysh $ cat test.txt
Hello
/home/user32230324/mysh $ echo "Dankook" >> test.txt
/home/user32230324/mysh $ cat test.txt
Hello
Dankook
/home/user32230324/mysh $ ls
mysh  mysh.c  test.txt
```

```

/home/user32230324/mysh $ ls | grep "test"
test.txt
/home/user32230324/mysh $ sleep 5 &
[1] 1341868
/home/user32230324/mysh $ ps
  PID TTY          TIME CMD
 700125 pts/19    00:00:00 bash
1035498 pts/19    00:00:00 mysh
1341868 pts/19    00:00:00 sleep
1349277 pts/19    00:00:00 ps
/home/user32230324/mysh $ ps
  PID TTY          TIME CMD
 700125 pts/19    00:00:00 bash
1035498 pts/19    00:00:00 mysh
1390159 pts/19    00:00:00 ps
/home/user32230324/mysh $ whoami
user32230324
/home/user32230324/mysh $ date
2024. 11. 11. (월) 02:28:58 KST
/home/user32230324/mysh $ exit
user32230324@system: ~ /mysh$

```

<수행 결과에 대한 간단한 설명>

1. ./mysh 실행 : 셸이 실행되고, 프롬프트(현재 디렉토리 \$)가 나타난다.
2. ls, cat, rm 등과 같은 외부 실행 명령어 실행 : 각 명령어에 맞는 결과가 나옴
3. echo "Hello" > test.txt , echo "Dankook" >> test.txt 등으로 redirection(>, >>) 사용 시 "Hello"를 test.txt라는 파일에 쓰거나(파일이 없을 경우 새로 생성) test.txt 파일에 "Dankook"을 추가함
4. ls | grep "test"로 pipe() 사용 시 ls의 결과를 grep "test"를 통해 필터링 하여 파일명에 test가 들어간 경우만 출력된다.
5. sleep 5 &로 background(&) 사용 시 sleep 5 명령어를 백그라운드에서 실행한다. 이때 백그라운드 프로세스 아이디가 [1] pid 형식으로 출력된다.
6. ps를 사용하여 5번에서 실행한 sleep이 백그라운드에서 실행 중인 것을 확인할 수 있으며, 5초 후엔 사라진 것을 확인할 수 있다.
7. whoami를 통해 현재 사용자의 이름을 출력한다.
8. date를 통해 시간 정보를 출력한다.
9. exit이나 quit을 통해 셸을 종료한다.

<추가 구현(pipe, redirection, background)에 대한 설명>

1. pipe()

- 한 명령의 출력을 다른 명령의 입력으로 연결하는 기능
- ex) `ls | grep "test"` 는 `ls` 명령어의 출력을 `grep "test"`로 넘겨 `test`가 포함된 파일만 출력
- 구현 : `tokenize()` 함수에서 `|`를 처리하여 파이프 앞뒤 명령어를 분리하고, 이를 `pipepipe()` 함수에서 파이프를 이용해 처리하도록 함, `pipepipe()` 함수에서는 두 개의 프로세스를 `fork()`하여 첫 번째 프로세스의 출력은 파이프에 쓰고 두 번째 프로세스는 그 파이프에서 읽어 실행하도록 함

2. redirection(>, >>)

- 명령어의 출력을 `>`는 파일에 덮어쓰고, `>>` 파일에 덧붙이는 기능
- ex) `echo "Hello" > test.txt`는 `echo` 명령어의 출력을 `test.txt` 파일에 저장(`test.txt` 파일이 없으면 새로 생성)
- ex) `echo "Dankook" >> test.txt`는 `echo` 명령어의 출력을 `test.txt` 파일에 덧붙임
- 구현 : `redirection` 함수에서 `open` 시스템 호출을 통해 파일을 열고, `dup2()`를 사용하여 표준 출력을 파일로 변경한 뒤, `execvp`로 명령어를 실행하도록 함

3. background(&)

- 명령어 뒤에 `&`를 붙여 해당 명령어를 백그라운드에서 실행하게 하는 기능
- ex) `sleep 5 &` 는 `sleep 5`를 백그라운드에서 실행
- 구현 : 명령어에 `&`가 있을 경우 자식 프로세스를 `fork()`한 후, 부모 프로세스가 `wait()`를 호출하지 않고 바로 새로운 명령을 받을 수 있도록 함.

2. 코드 설명 (필수)

** 주요 함수 동작 과정 설명*

1. 헤더 파일

- 헤더 파일 추가

2. sigchld 함수

- 자식 프로세스 종료 시 신호 처리 함수
- waitpid를 사용하여 종료된 자식 프로세스가 있으면 해당 프로세스의 상태를 확인하고, 종료된 자식 프로세스를 제거함
- WNOHANG 플래그 : 자식 프로세스가 종료되지 않았을 때 대기하지 않고 바로 반환
- sleep과 같은 명령어 사용 후 쉘의 프롬프트가 제대로 출력되지 않는 문제를 해결하기 위해 추가함. (자식 프로세스가 종료되었을 때, 부모 프로세스가 자식 프로세스를 기다리고 처리하는 과정에서 자식 프로세스의 종료 처리가 늦어지거나, 제대로 처리되지 않은 것이 원인이었기에 이를 해결하기 위해 이 함수를 통해 자식 프로세스를 처리하고자 함)

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

void sigchld(int sig) {
    while(waitpid(-1, NULL, WNOHANG) > 0);
}
```

(다음 페이지에 이어서)

3. quotes 함수

- 따옴표 제거 함수
- 문자열이 큰따옴표로 감싸져 있으면 그 따옴표를 제거하고, 따옴표 없이 문자열을 반환
- 예를 들어 `echo "Hello" > test.txt` 를 했을 때 test 파일에 Hello가 아닌 "Hello"가 그대로 저장되는 문제를 해결하고자 추가함 / 또 다른 예로는 `ls | grep "test"`를 했을 땐 아무 결과가 출력되지 않고 `ls | grep test`를 해야 원하는 결과가 추가되는 문제를 해결하고자 추가함

```
char* quotes(char* str) {  
    if(str[0] == '"' && str[strlen(str) - 1] == '"') {  
        str[strlen(str) - 1] = '\0';  
        return str + 1;  
    }  
    return str;  
}
```

4. tokenize 함수

- 명령어가 들어오면 공백 및 특정 구분자를 기준으로 파싱하여 명령어와 인자를 분리하는 함수
- buf : 입력 문자열(명령어) / delims : 구분자(공백, 줄바꿈 등) 지정 / tokens[] : 구분자로 분리된 각 토큰을 저장할 배열 / maxTokens : 최대 토큰 개수 / pipe1[], pipe2[] : 파이프의 좌측과 우측 부분을 저장할 배열
- count : 토큰의 개수를 세는 변수 / token : strtok 함수로 분리된 현재 토큰을 저장, type : 명령어의 유형을 나타내는 변수(기본 : 0, 파이프 : 1, 리다이렉션 : 2 or 3, 백그라운드 : 4)
- while문 : token이 NULL이 아니고 count가 maxTokens보다 작은 동안 반복한다. 먼저 quotes(token)을 호출하여 현재 토큰에서 따옴표를 제거한다. 이후 tokens[count]에 저장한다.
- tokens[count]가 |인 경우 type을 1로 설정하고 pipe1[]에 파이프 왼쪽 부분, pipe2[]에 파이프 오른쪽 부분을 저장한다. 각 배열의 끝에는 NULL을 넣어 배열의 끝을 표시한다.
- tokens[count]가 > 또는 >>인 경우 type을 각각 2와 3으로 설정한다.
- tokens[count]가 &인 경우 type을 4로 설정하고 tokens[count]를 NULL로 설정하여 &

를 명령어의 끝으로 처리함

- count++을 통해 토큰 배열의 다음 위치를 가리키게 함.
- strtok 함수로 NULL을 전달하여 이전 호출에서 이어서 다음 토큰을 추출
- 파이프, 리다이렉션, 백그라운드 등에 해당하지 않을 경우 type이 기본값 0을 가지고 tokens[count]를 NULL로 설정하여 토큰 배열의 끝을 표시 후 type 값을 반환

```
int tokenize(char* buf, char* delims, char* tokens[], int maxTokens, char* pipe1[], char* pipe2[]) {
    int count = 0;
    char *token = strtok(buf, delims);
    int type = 0;

    while(token != NULL && count < maxTokens) {
        token = quotes(token);
        tokens[count] = token;

        if(strcmp(tokens[count], "|") == 0) {
            type = 1;
            int i, j = 0;
            for(i = 0; i < count; i++) {
                pipe1[i] = tokens[i];
            }
            pipe1[count] = (char *) 0;
            while(i++ < maxTokens && tokens[i] != NULL) {
                pipe2[j++] = tokens[i];
            }
            pipe2[j] = (char *) 0;
        }
        else if(strcmp(tokens[count], ">") == 0) {
            type = 2;
        }
        else if(strcmp(tokens[count], ">>") == 0) {
            type = 3;
        }
        if(strcmp(tokens[count], "&") == 0) {
            type = 4;
            tokens[count] = NULL;
        }
        count++;
        token = strtok(NULL, delims);
    }
    tokens[count] = NULL;
    return type;
}
```

(다음 페이지에 이어서)

5. cmd_help 함수

- help 명령어를 처리하는 함수
- 내부 명령어와 외부 명령어에 대한 도움말 출력

```
bool cmd_help() {
    printf("/*****simple shell*****/\n");
    printf("<Internal command>\n");
    printf("quit : quit the shell\n");
    printf("exit : quit the shell\n");
    printf("help : Show the help of the commands\n");
    printf("<External command>\n");
    printf("ls : Display a list of files and folders in the directory\n");
    printf("cat : Print the contents of a file or combine files\n");
    printf("rm : Delete a file or directory\n");
    printf("There are a variety of other external commands.\n");
    printf("/*****/\n");

    return true;
}
```

6. redirection 함수

- redirection을 처리하는 함수
- type : redirection 종류(2는 >, 3은 >>) / tokens[] : 분리된 명령어와 인자들을 저장한 배열
- fd : 파일 디스크립터 / i : >이나 >>가 있는 위치를 찾을 때 사용할 변수
- for문 : tokens[i]가 > 또는 >>인 위치(i)를 찾고, 찾으면 루프 종료
- 리다이렉션 위치를 기준으로 명령어를 두 부분으로 나누기 위해 tokens[i]를 NULL로 설정
- fork()를 사용해 자식 프로세스를 생성
- 자식 프로세스에서 리다이렉션 수행 -> type이 2면 >을 수행해야 하는데 >는 덮어쓰는 기능이므로 O_WRONLY | O_CREAT(파일 없으면 생성) | O_TRUNC(기존 파일 있으면 덮어쓰) 플래그 사용하여 open하고 >>는 덧붙이는 기능이므로 O_WRONLY | O_CREAT | O_APPEND(기존 내용 끝에 덧붙이기) 플래그를 사용하여 open한다. 이때 0644는 파일의 권한 설정으로 소유자는 읽기/쓰기 권한, 그룹과 다른 사용자들은 읽기만 허용한다.
- 파일 열기를 실패할 경우 오류 메시지를 출력하고 프로그램을 종료한다.
- dup2를 사용해 fd를 표준 출력 STDOUT_FILENO로 복제 -> 이후의 출력이 파일로 리다이렉션 됨

- close로 파일 닫기
- execvp()로 실제 명령어를 실행, 이때 tokens[0]은 명령어이고 tokens 배열은 명령어와 인수들이 포함된 배열이며, 실행을 실패할 경우 오류 메시지를 출력하고 프로세스 종료
- 부모 프로세스는 자식 프로세스가 종료될 때까지 기다림 (wait(NULL))

```
void redirection(int type, char *tokens[]) {
    int fd;
    int i;
    for(i = 0; tokens[i] != NULL; i++) {
        if(strcmp(tokens[i], ">") == 0 || strcmp(tokens[i], ">>") == 0) {
            break;
        }
    }

    if(tokens[i] == NULL) return;
    tokens[i] = NULL;

    pid_t pid = fork();
    if(pid == 0) {
        if(type == 2) {
            fd = open(tokens[i + 1], O_WRONLY | O_CREAT | O_TRUNC, 0644);
        }
        else if(type == 3) {
            fd = open(tokens[i + 1], O_WRONLY | O_CREAT | O_APPEND, 0644);
        }

        if(fd < 0) {
            printf("Can't open %s file\n", tokens[i + 1]);
            exit(1);
        }

        dup2(fd, STDOUT_FILENO);
        close(fd);
        execvp(tokens[0], tokens);
        printf("execvp failed\n");
        exit(1);
    }
    else {
        wait(NULL);
    }
}
```

7. pipepipe 함수

- pipe를 처리하는 함수로 첫 번째 명령어의 출력을 두 번째 명령어의 입력으로 전달
- pipe1[] : 파이프의 왼쪽 명령어와 인수를 저장한 배열 / pipe2[] : 파이프의 오른쪽 명령어와 인수를 저장한 배열
- fd[2] : 파이프의 읽기 끝과 쓰기 끝을 저장할 파일 디스크립터 배열(fd[0]은 읽기 끝, fd[1]은 쓰기 끝) / pid1, pid2 : 첫 번째와 두 번째 자식 프로세스의 ID를 저장할 변수
- pipe(fd)로 파이프를 생성, 실패하면 오류 메시지를 출력하고 함수 종료
- pid1 = fork()로 파이프 왼쪽 명령어를 실행할 자식 프로세스 생성 후 자식 프로세스에서 dup2를 통해 파이프의 쓰기 끝(fd[1])을 표준 출력(STDOUT_FILENO)에 복제함으로써

파이프 왼쪽 명령어의 출력을 파이프로 전달, 이후 파일 디스크립터를 모두 닫고 `execvp`를 통해 실행(실패 시 오류 메시지 출력 후 자식 프로세스 종료)

- `pid2 = fork()`로 파이프 오른쪽 명령어를 실행할 자식 프로세스 생성 후 자식 프로세스에서 `dup2`를 통해 파이프의 읽기 끝(`fd[0]`)을 표준 입력(`STDIN_FILENO`)에 복제함으로써 파이프 오른쪽 명령어를 통해 왼쪽 명령어의 출력을 읽을 수 있게 함, 이후 파일 디스크립터를 모두 닫고 `execvp`를 통해 실행(실패 시 오류 메시지 출력 후 자식 프로세스 종료)

- 부모 프로세스에서는 더 이상 파이프가 필요 없으므로 파일 디스크립터를 닫은 후 `waitpid()`를 통해 두 자식 프로세스가 종료될 때까지 대기

```
void pipepipe(char* pipe1[], char* pipe2[]) {
    int fd[2];
    pid_t pid1, pid2;
    if(pipe(fd) == -1) {
        printf("pipe failed\n");
        return;
    }

    pid1 = fork();
    if(pid1 == 0) {
        dup2(fd[1], STDOUT_FILENO);
        close(fd[0]);
        close(fd[1]);
        execvp(pipe1[0], pipe1);
        printf("execvp failed\n");
        exit(1);
    }

    pid2 = fork();
    if(pid2 == 0) {
        dup2(fd[0], STDIN_FILENO);
        close(fd[1]);
        close(fd[0]);
        execvp(pipe2[0], pipe2);
        printf("execvp failed\n");
        exit(1);
    }

    close(fd[0]);
    close(fd[1]);
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);
}
```

(다음 페이지에 이어서)

8. run 함수

- 사용자가 입력한 명령어를 파싱 후 적절한 기능을 수행하는 함수로 성공적으로 실행되었는지 나타내기 위해 bool 타입 값 반환
- line : 사용자로부터 입력받은 명령어 문자열
- tokens[] : 명령어를 토큰화한 후 각각을 저장할 배열 / pipe1[], pipe2[] : 파이프 왼쪽, 오른쪽 명령어와 인수를 각각 저장할 배열 / type : 명령어 타입을 저장할 변수로, tokenize() 함수에서 반환된 값이 저장됨(0은 기본, 1은 파이프, 2와 3은 리다이렉션, 4는 백그라운드)
- tokenize() 함수는 사용자로부터 입력받은 명령어(line)을 공백과 개행 문자를 기준으로 나누어 tokens[]에 저장 후 명령어 타입에 따른 값 반환
- tokens[0] 즉, 사용자가 입력한 명령어가 NULL일 경우(없을 경우) true를 반환하며 함수 종료, exit나 quit일 경우(strcmp를 통해 비교) false를 반환하며 함수를 종료, help일 경우 cmd_help() 함수를 호출하여 명령어 도움말을 출력한 후 true를 반환하며 함수 종료 (이때 false는 셸 종료를 의미함)

```
bool run(char* line) {
    char* tokens[100];
    char* pipe1[100];
    char* pipe2[100];
    int type = tokenize(line, " \n", tokens, 100, pipe1, pipe2);

    if(tokens[0] == NULL) {
        return true;
    }

    if(strcmp(tokens[0], "exit") == 0 || strcmp(tokens[0], "quit") == 0) {
        return false;
    }

    if(strcmp(tokens[0], "help") == 0) {
        cmd_help();
        return true;
    }
}
```

- type이 0일 경우 파이프, 리다이렉션, 백그라운드 없는 단순한 명령어이므로 fork를 통해 자식 프로세스를 생성하고 execvp()를 통해 명령어를 실행, 실패할 경우에는 오류 메시지를 출력 후 자식 프로세스를 종료 / 부모 프로세스는 wait(NULL)을 통해 자식 프로세스가 종료될 때까지 대기
- type이 1일 경우 파이프를 처리해야 하므로 pipepipe() 함수를 호출. 이때, pipepipe 함수는 파이프 왼쪽 명령어의 출력을 오른쪽 명령어의 입력으로 연결하여 파이프를 처리

```

if(type == 0) {
    pid_t pid = fork();

    if(pid == 0) {
        execvp(tokens[0], tokens);
        printf("execvp failed\n");
        exit(1);
    }

    else {
        wait(NULL);
    }
}

else if(type == 1) {
    pipepipe(pipe1, pipe2);
    return true;
}

```

- type이 2 또는 3일 경우 리다이렉션을 처리해야 하므로 redirection() 함수를 호출.
- type이 4일 경우 백그라운드 처리를 수행. 먼저 fork()를 통해 자식 프로세스 생성 후 execvp()를 통해 명령어를 실행(실패 시 오류 메시지 출력 후 자식 프로세스 종료), 백그라운드를 구현하기 위해선 wait()을 사용하지 않아야 하기에 부모 프로세스에서는 '[1] 자식 프로세스 pid' 형식의 메시지를 출력 후 대기(wait)하지 않고 종료.
- 모든 작업이 성공적으로 완료되면 true를 반환

```

else if(type == 2 || type == 3) {
    redirection(type, tokens);
    return true;
}

else if(type == 4) {
    pid_t pid = fork();

    if(pid == 0) {
        execvp(tokens[0], tokens);
        printf("execvp failed\n");
        exit(1);
    }
    else {
        printf("[1] %d\n", pid);
    }
}

return true;
}

```

9. main 함수

- 사용자에게 명령어를 지속적으로 입력 받음
- signal()을 이용해 자식 프로세스가 종료 될 때 호출되는 신호(SIGCHLD)를 sigchld 함수로 설정 -> 자식 프로세스를 정리해주지 않으면 좀비 프로세스가 남기에 프롬프트가 정상적으로 출력되지 않는 등의 문제가 발생하여 이를 해결하고자 추가하게 됨
- line[] : 입력된 명령어를 저장할 공간과 cwd[] : 현재 작업 디렉토리를 저장할 배열 / cwd[] : 현재 작업 디렉토리 경로를 저장하는 문자 배열(이때 PATH_MAX는 시스템에서 허용하는 경로의 최대 길이를 나타내는 매크로 상수로 일반적으로 4096)
- while문 : break 문이 없다면 프로그램은 무한히 반복
- getcwd()를 통해 현재 작업 디렉토리를 cwd 배열에 저장
- printf()를 통해 '현재 디렉토리 \$ '와 같은 프롬프트를 표시하고 명령어 입력을 기다림
- fgets()를 통해 표준 입력으로부터 한 줄의 명령어를 line 배열에 입력받고 run() 함수를 실행. 이때 run() 함수의 반환값이 false라면 break하여 쉘을 종료함

```
int main()
{
    signal(SIGCHLD, sigchld);

    char line[1024];
    char cwd[PATH_MAX];
    while(1) {
        getcwd(cwd, sizeof(cwd));
        printf("%s $ ", cwd);
        fgets(line, sizeof(line) - 1, stdin);
        if(run(line) == false) break;
    }

    return 0;
}
```