시스템프로그래밍(SW)

과제 4

이름: 정현우

학번: 32204236

목차

Ⅰ. 서론

이번 과제는 쉘을 구현하는 것이다. 쉘을 통해 명령을 받아 넘기고, 프로세스 생성 및 실행을 하고 Background processing을 구현하여 학번과 날짜를 출력하고, 추가로 입출력 재지정을 구현하는 요구 사항이 존재한다. 반복문을 통해 명령을 계속 입력 받고, fork()와 execve() 계열의 시스템 호출을 이용하면 기본 로직이 구현될 것이며, "&"가 끝에 붙었는지 확인하면 Background processing이 가능하다. 그 다음 whoami와 date 명령으로 학번과 날짜를 보이게 할 수 있으며, 입출력 재지정은 dup2()를 활용하면 구현할 수 있을 것이다.

Ⅱ. 본론

1. main() 함수와 사용 라이브러리, 추가 정보

```c
/* mysh: make a shell, by sys32204236 이름 : 정현우 학번 : 32204236, 최초 작성일 : 10월 24일 , 마지막 수정일 : 2021/11/3 nuna90@naver.com*/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
#include <fcntl.h>
#include <error.h>

// 메인 함수
int main(int argc, char *argv[]) {
        int bgCheck, token_size;
        char line[1024];
        char *tokens[100];

        if(argc != 1) {
                printf("USAGE: %s\n", argv[0]);
                return 0;
        }

        while(1) {
                bgCheck = 0;
                printf("%s $ ", get_current_dir_name());
                fgets(line, sizeof(line), stdin);

                if( strcmp(line, "\n") == 0 ) continue;

                line[strlen(line) - 1] = '\0';

                if( line[strlen(line) - 1] == '&' ) {
                        bgCheck = 1;
                }

                token_size = tokenizer(line, tokens);

                if( run(tokens, token_size, bgCheck) < 0 ) break;
        }

        return 0;
}
```

맨 위에 주석을 통해 만든 사람, 날짜, 학번, 이메일과 같은 추가 정보를 적어 두었으며 stdio.h, unistd.h, sys/types.h, sys/wait.h, string.h, fcntl.h, error.h 라이브러리들을 사용하기로 설정하였다. 메인

함수의 경우, 백그라운드 프로세싱인지 확인하는데 쓰일 bgCheck, 토큰의 개수를 저장할 token_size, 커맨드를 받을 line, 커맨드를 토큰으로 분리해 저장할 tokens를 선언한다. 그 다음, 쉘의 실행이 적절한지 체크하고 반복문을 통해 쉘을 작동한다. 반복의 시작 시마다 bgCheck를 0으로 설정한다. 명령을 사용자로부터 받은 다음, 입력 없이 엔터 키만 눌렸다면 다음 반복으로 즉시 넘어간다. 명령이 있었을 경우 fgets에 의해 생긴 '\n' 을 널 문자로 바꾼다. 그 다음 백그라운드 프로세싱인지 확인하기 위해 맨 끝에 '&'이 있는지 확인하고, 있으면 bgCheck를 1로 바꾼다. 그 다음 tokenizer() 함수를 실행하여 토큰 분리를 하고, 분리된 토큰과 토큰 크기, bgCheck를 run() 함수에 넘겨주고 실행한다. 이때 run() 함수의 리턴값이 음수이면 반복문에서 탈출한다.

2. tokenizer() 함수

```
// 명령을 토큰으로 분리
int tokenizer(char *line, char *tokens[]) {
        int count = 0;
        char *token;

        token = strtok(line, " ");
        while( token != NULL && count < 99 ) {
                tokens[count] = token;
                token = strtok(NULL, " ");
                count++;
        }
        if( strcmp(tokens[count - 1], "&") == 0 ) {
                count = count - 1;
        }
        tokens[count] = '\0';
        return count;
}
```

명령을 받은 다음 strtok() 함수를 통해 분리하고, 끝의 위치를 정하기 위해 count 변수를 사용한다. 만약 끝에 "&"가 있으면 백그라운드 프로세싱을 위해 붙은 문자이니 count를 1 감소시켜 끝으로 지정할 자리를 바꾼다. 토큰 배열의 끝에는 널 문자가 들어가고, 함수는 토큰의 사이즈를 리턴하고 끝난다.

3. run() 함수

```
// fork(), execvp(), redirection 확인, 백그라운드 프로세싱 확인, 내부 명령어 사용
int run(char *tokens[], int token_size, int bgCheck) {
        pid_t fork_return;
        int rd_check, cd_check;
        pid_t w;
        if( command_exit(tokens[0]) == 0 ) exit(0);

        cd_check = command_cd(tokens);
        if( cd_check > 0) {
                return 0;
        } else if ( cd_check < 0 ) {
                printf("이동 실패\n");
                return 0;
        }

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

        fork_return = fork();
        if( fork_return < 0 ) {
                printf("fork error\n");
                return -1;
        } else if( fork_return == 0 ) {
                rd_check = redirection(tokens, token_size);
                if( rd_check < 0 ) exit(1);

                execvp(tokens[0], tokens);
                printf("명령이 적절하지 않습니다.\n");
                exit(1);
        } else {
                if( bgCheck == 0 ) {
                        waitpid(fork_return, NULL, 0);
                }
        }

        return 0;
}
```

우선 내부 명령어들이 들어왔는지 확인하고, 그에 맞는 처리를 한다. 예를 들어, command_exit() 함수의 리턴 값이 0이면 쉘을 종료하고, 다른 내부 명령어들은 리턴하여 함수를 종료한다. 그 다음 fork()를 통해 프로세스를 생성하고, 자식 프로세스에서는 입출력 재지정을 하였는지 체크하고, 입출력 재지정 방식에 문제가 있으면 프로세스를 종료한다. 그 외의 경우 원하는 프로그램을 실행한다. 부모 프로세스의 경우 백그라운드 프로세싱일 경우 자식 프로세스의 종료를 기다리지 않으며, 그 외의 경우엔 자식 프로세스 종료까지 기다린다.

4. redirection() 함수

```
// 리다이렉션 처리
int redirection(char *tokens[], int token_size) {
        int fd, count = 0, idx = -1;

        while( count < token_size ) {
                if( strcmp(tokens[count], ">") == 0 ) {
                        idx = count;
                        break;
                }
                count = count + 1;
        }
        if( idx == 0 ) {
                printf("USAGE: input > output\n");
                return -1;
        } if( idx < 0 ) {
                return 0;
        } else if( idx > 0 ) {
                if( (count + 1) == token_size ) {
                        printf("USAGE: input > output\n");
                        return -1;
                }
                fd = open(tokens[idx + 1], O_RDWR | O_CREAT | O_TRUNC, 0641);
                if( fd < 0 ) {
                        perror("open error");
                        exit(1);
                }

                dup2(fd, STDOUT_FILENO);
                close(fd);
                tokens[idx] = '\0';
        }
        return 0;
}
```

입출력 재지정을 하기 위해 반복문을 통해 토큰에 “>”가 어디 있는지 확인한다. 없을 경우 idx 변수는 -1이다. 만약 idx가 0이면 첫 커맨드부터 “>”을 입력한 것이니 사용 방법을 알려주고 -1을 리턴하며 함수를 끝낸다. idx가 음수, 즉 -1을 유지한 경우 입출력 재지정 명령이 없는 것이니 0을 리턴하고 끝낸다. idx가 0보다 큰 경우, “>”가 맨 끝에 오면 안되기 때문에 확인해주고, 맨 끝에 왔으면 사용법을 알려주고 -1을 리턴하며 종료한다. 정상적인 형식인 경우 “>” 다음에 오는 파일 이름으로 open()한 파일 디스크립터에 표준 출력 (STDOUT_FILENO)를 dup2()를 통해 중복시킨다. 그 다음으로는 “>” 이후로는 토큰에서 없어야 하기 때문에 “>”자리에 널 문자를 넣어주고, 0을 리턴하며 함수를 종료한다.

5. Internal commands

```
//exit 내부 명령: 쉘 종료
int command_exit(char *token) {
        return strcmp(token, "exit");
}

// cd 내부 명령: 디렉토리 변경
int command_cd(char *tokens[]) {
        int result;
        if( strcmp(tokens[0], "cd") != 0 ) {
                return 0;
        }

        result = chdir(tokens[1]);
        if( result < 0 ) {
                return -1;
        } else {
                return 1;
        }
}

// help, ? 내부 명령: help 나 ? 입력 시 도움말 출력
int command_help(char *help) {
        if( strcmp(help, "help") == 0 || strcmp(help, "?") == 0 ) {
                printf("----------MY Shell----------\n");
                printf("This shell can be used like conventional shell\n\n");
                printf("Internal commands\n");
                printf("cd\t: change directory\n");
                printf("exit\t: exit this shell\n");
                printf("help\t: show this help\n");
                printf("?\t: show this help\n");
                printf("----------------------------\n");
                return 0;
        }
        return -1;
}
```

쉘에서 “exit”, “cd”, “help”, “?”를 입력했을 때의 처리를 위한 함수들이다. run()에서 이 함수들을 통해 내부 명령어 처리를 한다. command_exit() 함수의 경우 strcmp()를 통해 “exit”와 받은 문자열의 비교 결과를 리턴한다. 0일 경우, run() 에서 exit() 시스템 호출을 사용해야 한다. command_cd() 함수는 “cd”를 받은 경우 chdir()을 통해 디렉토리를 변경하는데, 첫 토큰이 cd가 아니면 0을 리턴, chdir()에서 에러 발생 시 -1 리턴, 정상적으로 디렉토리 변경 시 1을 리턴하고 run()에서 해당 값들에 맞는 처리를 한다. commend_help() 함수는 “?”나 “help”를 받으면 쉘에 대한 설명과 내부 명령어들을 보여주고 0을 리턴하여 함수를 종료한다. “?”나 “help”를 받은 게 아니면 -1을 리턴하여 함수를 종료한다.

Ⅲ. 결과 및 결론

```
[sys32204236@embedded:~$ ls
a.txt  examples.desktop  hello_attr.txt  hello_new.txt  mycp    my_favorite_poem.txt  mysh.c         new.txt  test1  test3   wc_man.txt
db     gdb_test.out      hello.c         hello.txt      mycp.c  mysh                  new_hello.txt  test     test2  test.c
[sys32204236@embedded:~$ gcc -o mysh mysh.c
[sys32204236@embedded:~$ ./mysh
[/home/1-class/sys32204236 $ ls
a.txt  examples.desktop  hello_attr.txt  hello_new.txt  mycp    my_favorite_poem.txt  mysh.c         new.txt  test1  test3   wc_man.txt
db     gdb_test.out      hello.c         hello.txt      mycp.c  mysh                  new_hello.txt  test     test2  test.c
[/home/1-class/sys32204236 $  gcc -o hello hello.c
[/home/1-class/sys32204236 $ ./hello
hello!
[/home/1-class/sys32204236 $ ?
----------MY Shell----------
This shell can be used like conventional shell

Internal commands
cd      : change directory
exit    : exit this shell
help    : show this help
?       : show this help
----------------------------
[/home/1-class/sys32204236 $ help
----------MY Shell----------
This shell can be used like conventional shell

Internal commands
cd      : change directory
exit    : exit this shell
help    : show this help
?       : show this help
----------------------------
[/home/1-class/sys32204236 $ ps
  PID TTY          TIME CMD
12732 pts/22   00:00:00 mysh
13042 pts/22   00:00:00 ps
15755 pts/22   00:00:00 bash
[/home/1-class/sys32204236 $ cat hello.txt > hellohello.txt
[/home/1-class/sys32204236 $ cat hellohello.txt
Hello!
My name is 정현우
[/home/1-class/sys32204236 $ ls &
/home/1-class/sys32204236 $ a.txt                gdb_test.out    hello.c         hello.txt  my_favorite_poem.txt  new_hello.txt  test1  test.c
db                hello          hellohello.txt  mycp       mysh                new.txt        test2  wc_man.txt
examples.desktop  hello_attr.txt  hello_new.txt   mycp.c     mysh.c              test           test3

[/home/1-class/sys32204236 $ cd ../
[/home/1-class $ pwd
/home/1-class
[/home/1-class $ whoami
sys32204236
[/home/1-class $ date
2021. 11. 03. (수 ) 01:40:54 KST
[/home/1-class $ exit
[sys32204236@embedded:~$ whoami
sys32204236
[sys32204236@embedded:~$ date
2021. 11. 03. (수 ) 01:41:02 KST
sys32204236@embedded:~$
```

우선, 쉘을 통해 명령어를 입력 받아 토큰으로 분리하고, 프로세스를 생성하고, 해당 프로세스를 통해 새 프로그램 수행을 하는 등, 기본 로직을 작성하는 첫번째 요구 사항을 충족하였다. ls, ps와 같은 다양한 명령어들을 받아들여 정상적으로 수행하는 것을 확인할 수 있다. 맨 끝에 “&”를 붙여 Background processing을 구현하는 두번째 요구 사항의 경우, Background processing을 하니 쉘이 먼저 돌아오는 것을 통해 만족한 것을 확인할 수 있다. whoami와 date를 통해 학번과 날짜를 보이게 하는 세 번째 요구 사항까지 만족하였다. 여기에, “>”을 통해 입출력을 재지정하는 것까지 구현하여 보너스 요구 사항까지 구현하는 것에 성공하였다. 또한, cd나 exit같은 내부 명령어까지 추가로 구현하였고, 실행 결과 잘 작동하는 것을 확인할 수 있다.

```
sys32204236@embedded:~$ ./mysh
/home/1-class/sys32204236 $ ls
a.out  examples.desktop  hello_attr.txt  hello.txt  my_favorite_poem.txt  new_hello.txt  test1  test.c
a.txt  gdb_test.out      hello.c         mycp       mysh                  new.txt        test2  wc_man.txt
db     hello             hello_new.txt   mycp.c     mysh.c                test           test3
27927 27927
/home/1-class/sys32204236 $
/home/1-class/sys32204236 $ ls &
/home/1-class/sys32204236 $ a.out  examples.desktop  hello_attr.txt  hello.txt  my_favorite_poem.txt  new_h
ello.txt  test1 test.c
a.txt  gdb_test.out      hello.c         mycp       mysh                  new.txt        test2  wc_man.txt
db     hello             hello_new.txt   mycp.c     mysh.c                test           test3

/home/1-class/sys32204236 $ ls
27971 27985
/home/1-class/sys32204236 $ a.out  examples.desktop  hello_attr.txt  hello.txt  my_favorite_poem.txt  new_h
ello.txt  test1 test.c
a.txt  gdb_test.out      hello.c         mycp       mysh                  new.txt        test2  wc_man.txt
db     hello             hello_new.txt   mycp.c     mysh.c                test           test3

/home/1-class/sys32204236 $ ls
27985 28075
/home/1-class/sys32204236 $ a.out  examples.desktop  hello_attr.txt  hello.txt  my_favorite_poem.txt  new_h
ello.txt  test1 test.c
a.txt  gdb_test.out      hello.c         mycp       mysh                  new.txt        test2  wc_man.txt
db     hello             hello_new.txt   mycp.c     mysh.c                test           test3
```

구현하며 가장 문제가 되었던 부분은 Background processing에서 발생하였다. 처음에 만들었던 쉘의 경우, 위와 같이 Background processing을 한 이후에는 Background processing을 하지 않아도 한 것처럼 쉘이 먼저 돌아오는 상황이 발생하였다. 처음에는 Background processing을 체크하는 부분에서 에러가 생긴 건가 하여 gdb를 통해 디버깅 해보았지만, info locals 명령으로 Background processing 유무를 체크한 변수의 내용을 확인해도 정상이었다. 그래서 run() 함수에서 wait() 하고 나서, wait()와 fork()의 리턴 값을 출력하였더니, wait()로 기다린 프로세스와 fork()로 생성한 프로세스가 다른 것을 확인할 수 있었다. 즉, 단순히 wait()를 하면 기다리는 프로세스가 실제로 기다리길 바란 프로세스와 다른 문제가 생기는 것을 간과한 것이다.

```
{
 if( bgCheck == 0 ) {
         waitpid(fork_return, NULL, 0);
 }
```

이 문제는 waitpid()를 통해 어떤 프로세스를 제거하고자 하는지 명확히 하여 해결할 수 있었다.

```
/home/1-class/sys32204236 $ ls &
/home/1-class/sys32204236 $ a.txt
db               hello          hellohe
examples.desktop  hello_attr.txt  hello_r

/home/1-class/sys32204236 $ cd ../
```

실행 결과 또한, Background processing을 한 이후에도 정상적으로 실행되는 것을 알 수 있다.