Operating System (MS)

assignment #1: Simple MyShell

모바일시스템공학과 김태경 (32211203)

목차

Freeday used 0
Freeday left 5

2024.09.29.

1. 프로젝트 소개

1.1 프로그램 소개

개요:

SiSH(Simple Shell)는 사용자로부터 명령어를 입력받아 프로그램을 실행하는 쉘 프로그램입니다. 명령어 입력 후, 해당 프로그램이 종료되면 다음 명령어 입력을 대기하는 구조를 가지고 있습니다.

목적:

이 프로그램은 운영 체제의 시스템 호출을 활용해 쉘 프로그램의 기본 동작을 이해하는 데 목적이 있습니다. 이를 통해 프로세스 생성, 프로그램 실행 및 대기 메커니즘을 직접 학습할 수 있습니다.

1.2 프로젝트 목표

- 사용자가 입력한 명령어를 프로그램으로 해석하고 실행.
- 기본적인 시스템 호출을 사용한 프로세스 생성 및 관리.
- 환경 변수를 이용한 명령어 경로 탐색.
- 쉘 프롬프트에서 지속적으로 명령어를 입력받고 처리.

2. 프로젝트 세부 요구사항

2.1 프로그램 구현 기본 요구사항

2.1.1 Makefile 작성 및 컴파일

- make 명령어로 코드를 컴파일할 수 있도록 Makefile을 작성해야 합니다.

2.1.2 SiSH 시작 및 종료

- 실행 파일을 실행하면 SiSH(Shell 프로그램)가 시작됩니다.
- 사용자가 quit이라는 문자열을 입력하면 SiSH가 종료되어야 합니다.

2.1.3 SiSH의 동작

- 입력: 사용자가 프로그램 이름을 입력할 수 있어야 합니다.
- 실행: 입력된 프로그램이 파일 시스템에서 실행 가능한 경우, 그 프로그램을 실행할 수 있어야 합니다.
- 실행 경로: SiSH는 프로그램을 실행할 때, 실행 파일 경로를 절대 경로로 지정하지 않고, 환경 변수 PATH에 지정된 디렉터리에서 파일을 찾을 수 있어야 합니다.
- 프로세스 관리: 프로그램이 실행되는 동안에는 쉘이 비활성화되어야 하고, 프로그램 실행이 끝난 후에 다시 명령어를 입력받을 수 있어야 합니다.

- 반복: 프로그램이 완료된 후, 새로운 명령어를 받아서 다른 프로그램을 실행할 수 있어야 합니다.

2.1.4 **시스템 호출 사용**

- fork(), execve(), wait() 시스템 호출을 사용하여 프로세스를 생성하고 프로그램을 실행한 후, 종료 대기까지 수행해야 합니다.

2.2 프로그램 선택적 구현 요구사항

2.2.1 **쉘 프롬프트 사용자 정의**

- getenv()를 사용하여 현재 작업 디렉터리(PWD), 시간(TIME), 사용자(USER) 등을 프롬프트에 표시할 수 있습니다.

2.2.2 **추가 입력 인자 처리**

- 사용자가 실행할 프로그램에 추가 인자를 제공할 수 있어야 하고, 이 인자를 자식 프로세스에 전달해야 합니다. execve() 시스템 호출의 매뉴얼 페이지를 참고하여 구현할 수 있습니다.

3. 프로젝트에 사용된 개념

3.1 쉘(Shell)

쉘은 사용자와 운영 체제 간의 인터페이스로, 사용자가 명령어를 입력하면 이를 해석하여 해당 프로그램을 찾아 실행하고, 파일 및 디렉토리의 탐색 및 관리를 담당합니다. 그 외 여러 프로그램을 동시에 실행시키거나, 백그라운드에서 다른 작업들을 수행하는 역할을 합니다.

3.1.1 Shell의 동작과정

1). 사용자 입력 대기: 쉘은 터미널을 통해 사용자 입력을 대기합니다. 일반적으로 프롬프트(prompt)를 통해 입력 준비 상태를 표시합니다.

2). 명령어 분석: 사용자가 명령어를 입력하면, 쉘은 이를 분석하고 프로그램을 실행하기 위한 명령어와 인자를 추출합니다.

3). 명령어 실행: 쉘은 fork()를 통해 새로운 프로세스를 생성하고, execve()를 호출해 입력된 명령어에 해당하는 프로그램을 실행합니다.

4). 결과 반환 및 출력: 프로그램이 실행되고 나면 그 결과가 사용자에게 출력되며, 쉘은 다시 사용자 입력을 대기합니다.

3.2 프로세스

프로세스는 컴퓨터에서 실행 중인 프로그램의 인스턴스로, 디스크에 저장된 프로그램 파일이 메모리에 적재되어 CPU에서 실제로 실행될 때, 이를 프로세스라고 부릅니다. 프로그램은 정적인 코드이지만, 프로세스는 실행을 통해 동적으로 변화하는 개체입니다. 여기서 운영체제는 프로세스 관리를 통해 시스템 리소스를 효과적으로 분배하고, 안정적인 프로그램 실행을 보장합니다.

3.3 시스템 호출(System Calls)

- fork(): 부모 프로세스에서 자식 프로세스를 생성하는 시스템 호출입니다. 부모 프로세스는 자식 프로세스의 ID(PID)를 반환받고, 자식은 0을 반환받습니다.
- execve(): 자식 프로세스가 새로운 프로그램을 실행할 때 호출됩니다. 이 호출을 통해 자식 프로세스는 새로운 프로그램으로 대체됩니다.
- wait(): 부모 프로세스가 자식 프로세스가 종료될 때까지 기다립니다.
- getenv(): 환경 변수 값을 가져오며, PATH 변수를 참조하여 실행할 프로그램의 경로를 찾는 데 사용됩니다.

3.4 환경 변수(Environment Variables)

PATH: 실행할 프로그램의 경로가 저장된 환경 변수입니다. 쉘은 PATH 변수를 참조해 사용자가 입력한 명령어가 위치한 경로를 찾습니다.

3.5 문자열 처리

strtok_r(): 쉘에서 입력된 명령어를 프로그램 이름과 인자들로 분리하는 데 사용됩니다. 명령어를 공백으로 나누어 토큰화하는데 매우 유용합니다.

3.5 Makefile

Makefile은 Make라는 빌드 자동화 도구를 사용하기 위한 파일로, C 언어 프로젝트의 빌드 과정을 자동화하기 위해 작성되었습니다. Makefile은 Make 언어로 작성되었으며, 주로 gcc 같은 컴파일러를 이용해 소스 파일을 컴파일하고 링크하는 과정을 정의하는 역할을 합니다.

3.5.1 Makefile의 언어적 특징

- 명령어 자동화: 명령어를 미리 정의해두고, make 명령어 하나만 실행하면 빌드, 실행 파일 생성, 청소 등이 자동으로 이루어집니다.
- 패턴 규칙: 여러 파일에 대해 같은 규칙을 적용하기 위한 패턴 규칙을 지원합니다.
- 의존성 관리: 각 빌드 타겟에 필요한 파일들을 명시해 의존성을 관리합니다. 예를 들어, sish.o가 갱신되면 sish가 재생성됩니다.

4. 프로그램 구현

4.1 사용된 시스템 호출 및 API

```c
// fork를 통해 자식 프로세스 생성
pid = fork();

if (pid == 0)
{
    // 자식 프로세스
    if (find_executable(args[0], executable_path))
    {
        execve(executable_path, args, NULL);
        perror("execve failed");
        exit(EXIT_FAILURE);
    }
    else
    {
        printf("Command not found: %s\n", args[0]);
        exit(EXIT_FAILURE);
    }
}
else if (pid > 0)
{
    // 부모 프로세스: 자식 프로세스가 끝날 때까지 대기
    if (!background)
    {
        wait(NULL);
    }
}
else
{
    perror("fork failed");
}
```

4.1.1 **fork()**

는 부모 프로세스에서 자식 프로세스를 생성하는 시스템 호출입니다. 부모 프로세스는 이 호출을 통해 자신의 복사본인 자식 프로세스를 생성하고, 두 프로세스는 독립적으로 실행됩니다.

- 부모 프로세스는 자식 프로세스의 **프로세스 ID(PID)**를 반환받고, 자식 프로세스는 fork() 호출의 반환값으로 0을 받습니다.
- 부모와 자식 프로세스는 같은 코드와 데이터 영역을 공유하지만, 실행 시 서로 다른 경로를 따르게 됩니다.

4.1.2 **execve()**

는 자식 프로세스가 새로운 프로그램을 실행할 때 호출되는 시스템 호출입니다. 이 호출은 현재 프로세스를 완전히 새로운 프로그램으로 대체하며, 실행 중인 프로세스의 메모리, 코드, 데이터 영역을 초기화한 후, 새로운 프로그램을 처음부터 실행합니다.

- 실행 파일 경로와 인자 배열, 환경 변수 배열을 입력으로 받아 그 프로그램을 실행합니다.
- execve() 호출이 성공하면 기존 프로세스의 실행 흐름은 종료되고, 새로운 프로그램의 시작점에서부터 실행이 재개됩니다.

4.1.3 **wait()**

는 부모 프로세스가 자식 프로세스가 종료될 때까지 기다리는 시스템 호출입니다. 자식 프로세스가 종료되면 wait()는 자식 프로세스의 종료 상태를 반환하며, 부모 프로세스는 이후 실행을 재개합니다.

- wait 호출은 자식 프로세스가 정상적으로 종료되었는지 확인하고, 사용된 시스템 자원을 해제하는 데 사용됩니다.
- 자식 프로세스가 종료되기 전까지 부모 프로세스는 대기 상태에 머무릅니다.

4.1.4 **getenv()**

는 시스템의 환경 변수 값을 가져오는 함수입니다. SiSH에서는 특히 PATH 환경 변수를 사용하여 실행할 프로그램의 경로를 찾습니다.

- PATH 환경 변수는 시스템이 명령어를 찾는 디렉토리 경로 목록을 저장하는데, getenv()를 통해 이 정보를 읽어들여 사용자가 입력한 프로그램이 어느 경로에 있는지 확인할 수 있습니다. **4.2 프로그램 구조 및 동작 설명**

4.2.1 **Makefile 구성요소**

1) **변수 정의** :

- CC: 사용할 컴파일러를 지정합니다. 여기서는 gcc를 사용합니다.
- CFLAGS: 컴파일 시 사용할 플래그를 지정합니다. -Wall은 모든 경고를 표시하라는 의미입니다.

2) **빌드 대상 및 의존성 규칙** : `all: sish fork fork2 getenv stat`

- all: 기본 빌드 대상입니다. make 명령어만 실행할 때, all에 정의된 모든 타겟(sish, fork, fork2, getenv, stat)이 빌드됩니다.
- sish, fork, fork2, getenv, stat 각각은 실행 파일을 의미합니다.

3) **개별 타겟 규칙** :

```
sish: sish.o
    $(CC) $(CFLAGS) -o sish sish.o
```

- sish: sish.o 파일을 사용해 sish라는 실행 파일을 생성하는 규칙입니다.
- $(CC)와 $(CFLAGS)는 앞에서 정의한 변수로, gcc -Wall -o sish sish.o처럼 실행됩니다.
- 이 규칙은 sish.o 파일이 있을 때 sish라는 실행 파일을 만듭니다.

4) **오브젝트 파일 생성 규칙** :

```
%.o: %.c
    $(CC) $(CFLAGS) -c $<
```

- 이 규칙은 패턴 규칙으로, .c 파일을 .o 파일로 변환할 때 사용됩니다.
- %.o는 어떤 .c 파일이든 .o로 변환하는 패턴입니다. 예를 들어, sish.c 파일은 sish.o로 변환됩니다.
- $<는 의존성 목록에서 첫 번째 파일을 나타내며, 이를 통해 gcc -Wall -c sish.c처럼 실행됩니다.

5) **청소 규칙** :

```
clean:
    rm -f *.o sish fork fork2 getenv stat
```

- clean은 프로젝트에서 생성된 파일들을 삭제하는 규칙입니다.
- rm -f *.o sish fork fork2 getenv stat 명령어를 사용해 오브젝트 파일(*.o)과 실행 파일을 삭제합니다.
- make clean 명령을 실행하면 빌드 결과물들을 삭제할 수 있습니다.

4.2,2 **헤더파일**

<unistd.h> : 유닉스 시스템 호출(fork, execve, chdir 등)을 사용하기 위해 필요하다
<sys/types.h> : pid_t와 같은 자료형을 정의하기 위해 필요하다
<sys/wait.h> : 부모 프로세스가 자식 프로세스가 종료될 때까지 기다리는 wait() 함수를 사용하기 위해 필요하다

4.2.3 상수 정의

```
#define MAX_INPUT 1024
#define MAX_ARGS 64
```

: 사용자로부터 입력받는 명령어의 최대 길이 1024로 정의
: 명령어에 전달될 수 있는 최대 인자의 수를 64로 지정

4.2.4 변수 선언

```
char input[MAX_INPUT];
char *args[MAX_ARGS];
char executable_path[MAX_INPUT];
pid_t pid;
int status;
```

: 사용자가 입력한 명령어 전체를 저장하는 배열
: 명령어를 실행할 때 인자(argumant)들을 저장할 배열
: 명령어의 실행 파일 경로를 저장하는 배열
: 프로세스 ID를 저장하는 변수
: 자식 프로세스가 종료될 때의 상태를 저장하는 변수

4.2.5 기본 구조 (main)

```
while (1)
{
    // 프롬프트 출력
    printf("SiSH> ");
    fgets(input, MAX_INPUT, stdin);

    // 'quit' 입력 시 종료
    if (strncmp(input, "quit", 4) == 0)
    {
        break;
    }
```

: while으로 지속적인 명령어를 입력 받는 무한 루프
: fgets()는 사용자의 입력을 표준입력을 받아오며, 입력 버퍼에 들어오는 데이터를 input배열에 저장한다.
; strncmp로 명령어 'quit'가 입력되면 shell을 종료

4.2.6 명령어 입력 분리 함수 (parse_input)

```
// 명령어 입력을 분리하는 함수
void parse_input(char *input, char **args)
{
    char *token;
    int i = 0;

    // strtok을 이용하여 공백으로 분리
    token = strtok(input, " \n");

    while (token != NULL)
    {
        args[i] = token;
        i++;
        token = strtok(NULL, " \n");
    }
    args[i] = NULL;
}
```

parse_input 함수는 사용자가 입력한 명령어를 공백을 기준으로 나눠 인자들을 args[] 배열에 저장한다.

: args[i] = NULL은 execve()함수가 명령어 인자의 끝을 인식할 수 있도록, 배열의 마지막에 NULL을 추가

4.2.6 실행 파일 경로 찾기 함수 (find_executable)

```
// PATH 환경 변수를 사용해 실행 파일의 전체 경로를 찾는 함수
int find_executable(char *command, char *fullpath) {
    char *path = getenv("PATH");
    char *path_token;
    char path_copy[MAX_INPUT];

    strcpy(path_copy, path);
    path_token = strtok(path_copy, ":");

    while (path_token != NULL)
    {
        snprintf(fullpath, MAX_INPUT, "%s/%s", path_token, command);
        if (access(fullpath, X_OK) == 0) {
            return 1;
        }
        path_token = strtok(NULL, ":");
    }

    return 0; // 실행 파일을 찾지 못함
}
```

find_executable 함수는 사용자가 입력한 명령어 실행 파일을 PATH 환경병수에 정의된 디렉토리 경로에서 색출한다.

: getenv() 환경변수를 불러옴
: strtok()는 ':'문자를 기준으로 디렉토리 경로들을 추출
: access()는 fullpath 경로에 있는 파일이 실행가능한지 확인

4.2.7 파이프 처리를 위한 함수 (execute_pipe)

```c
// 파이프 처리를 위한 함수
void execute_pipe(char *cmd1, char *cmd2)
{
    int pipefd[2];
    pipe(pipefd);

    pid_t pid1 = fork();
    if (pid1 == 0)
    {
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[0]);
        close(pipefd[1]);
        char *args1[MAX_ARGS];
        parse_input(cmd1, args1);
        execvp(args1[0], args1);
        perror("execvp failed");
        exit(EXIT_FAILURE);
    }

    pid_t pid2 = fork();
    if (pid2 == 0)
    {
        dup2(pipefd[0], STDIN_FILENO);
        close(pipefd[0]);
        close(pipefd[1]);
        char *args2[MAX_ARGS];
        parse_input(cmd2, args2);
        execvp(args2[0], args2);
        perror("execvp failed");
        exit(EXIT_FAILURE);
    }

    close(pipefd[0]);
    close(pipefd[1]);
    wait(NULL);
    wait(NULL);
}
```

파이프의 생성
: pipfd[0]는 파이프의 읽기 끝(입력), pipfd[1]은 쓰기 끝(출력)
: pipe()로 파이프를 생성하고, 파이프의 두 끝을 배열에 저장

cmd1 실행 (첫 번째 자식 프로세스)
: dup2()는 자식 프로세스의 표준 출력을 파이프의 쓰기 끝에 연결. cmd1의 출력이 파이프를 통해 전달
: execvp()로 명령어를 실행하고, 실패시 오류 메시지 출력

cmd2 실행 (두 번째 자식 프로세스)
: dup2()는 자식 프로세스의 표준 출력을 파이프의 읽기 끝에 연결. cmd2의 출력이 파이프를 통해 전달

파이프 종료 및 부모 프로세스 대기

4.2.8 cd 명령어 처리 (main)

```c
// 'cd' 명령어 처리
if (strncmp(input, "cd", 2) == 0)
{
    char *dir = strtok(input + 3, " \n");
    if (dir == NULL)
    {
        fprintf(stderr, "No directory specified.\n");
    }
    else
    {
        if (chdir(dir) != 0)
        {
            perror("chdir failed");
        }
    }
    continue;
}
```

: strcmp()는 입력이 “cd”로 시작하는지 확인
: strtok()는 ‘cd’ 이후에 입력된 디렉토리를 추출하여 이후의 문자열을 공백과 줄바꿈 문자로 토큰을 분리
: chdir()는 추출한 디렉토리를 쉘의 부모 프로세스에서 직접호출하여 작업 디렉토리를 변경
: continue는 ‘cd’명령어를 처리한 후, 부모 프로세스에서 실행됨으로 자식 프로세스를 생성하지 않기 위해 이후의 반복을 진행

4.2.9 환경 변수 출력 처리 (main)

: input[0] ==’$’ 입력이 $로 시작하는 경우 환경변수를 출력하는 명령어로 인식
: getenv()는 $이후의 문자열의 환경변수 이름으로 받아 해당 변수의 값을 출력
; continue는 환경변수를 처리한 후, 이후의 명령어 입력을 기다림

// 작동 안함

```c
// 환경 변수 출력 처리 (예: echo $PATH)
if (input[0] == '$')
{
    char *env_var = getenv(input + 1);
    if (env_var)
    {
        printf("%s\n", env_var);
    }
    else
    {
        printf("Environment variable not found.\n");
    }
    continue;
}
```

4.2.10 **파이프 처리** (main)

```c
// 파이프 처리 (| 포함 명령어)
if (strchr(input, '|') != NULL)
{
    char *cmd1 = strtok(input, "|");
    char *cmd2 = strtok(NULL, "|");
    execute_pipe(cmd1, cmd2);
    continue;
}
```

: 입력 파이프에 '|'문자가 포함되어있는지 확인
: '|'문자를 기준으로 첫 명령어는 cmd1에, 두 번째 명령어는 cmd2에 저장
: // 위의 파이프 처리 함수 호출

4.2.11 **백그라운드 실행** (main) // 작동 안함

```c
// 백그라운드 실행 확인 (& 처리)
int i = 0;
while (args[i] != NULL) i++;
int background = 0;

if (i > 0 && args[i - 1][strlen(args[i - 1]) - 1] == '&')
{
    background = 1;  // 백그라운드 실행 플래그 설정
    args[i - 1][strlen(args[i - 1]) - 1] = '\0';
}
```

'&' 문자가 명령어 끝에 포함되어 있으면 백그라운드 실행 명령어로 간주

: background = 1 백그라운드 플래그로, 자식 프ㄹ세스를 생성 후 다음 명령어를 받음

: 백그라운드 기호 '&'를 제거

4.2.12 **&& 연산자 처리** (main) // 작동 안함

```c
// AND 연산자 처리 (&&)
if (strstr(input, "&&") != NULL)
{
    char *cmd1 = strtok(input, "&&");
    char *cmd2 = strtok(NULL, "&&");

    // 첫 번째 명령어 실행
    pid = fork();
    if (pid == 0)
    {
        parse_input(cmd1, args);
        if (find_executable(args[0], executable_path))
        {
            execve(executable_path, args, NULL);
            perror("execve failed");
            exit(EXIT_FAILURE);
        }
        else
        {
            printf("Command not found: %s\n", args[0]);
            exit(EXIT_FAILURE);
        }
    }
    wait(&status);
```

```c
    // 첫 번째 명령어가 성공한 경우에만 두 번째 명령어 실행
    if (WIFEXITED(status) && WEXITSTATUS(status) == 0)
    {
        pid = fork();
        if (pid == 0)
        {
            parse_input(cmd2, args);
            if (find_executable(args[0], executable_path))
            {
                execve(executable_path, args, NULL);
                perror("execve failed");
                exit(EXIT_FAILURE);
            }
            else
            {
                printf("Command not found: %s\n", args[0]);
                exit(EXIT_FAILURE);
            }
        }
        wait(NULL);
    }
    continue;
}
```

: strstr(input, "&&") 입력에 '&&'문자가 포함된 경우, 두 개의 명령어를 논리적 AND로 연결
: fork()를 통해 자식 프로세스를 생성하고 첫 번째 명령어를 실행
: wait()을 이용하여 첫 번째 명령어가 종료된 경우에만 두 번째 명령어르 실행

4.2.13 ; **순차 실행** (main) // 작동 안함

```c
// 순차 실행 처리 (; 포함 명령어)
if (strchr(input, ';') != NULL)
{
    char *cmd = strtok(input, ";");
    while (cmd != NULL)
    {
        pid = fork();
        if (pid == 0)
        {
            parse_input(cmd, args);
            if (find_executable(args[0], executable_path))
            {
                execve(executable_path, args, NULL);
                perror("execve failed");
                exit(EXIT_FAILURE);
            }
            else
            {
                printf("Command not found: %s\n", args[0]);
                exit(EXIT_FAILURE);
            }
        }
        wait(NULL);
        cmd = strtok(NULL, ";");
    }
    continue;
}
```

: strchr()로 ';' 문자가 포함여부 확인
: strtok()로 ';'문자를 기준으로 나누고 순차적으로 실행

4.2.14 **일반 명령어 실행** (main)

```c
if (pid == 0)
{
    // 자식 프로세스
    if (find_executable(args[0], executable_path))
    {
        execve(executable_path, args, NULL);
        perror("execve failed");
        exit(EXIT_FAILURE);
    }
    else
    {
        printf("Command not found: %s\n", args[0]);
        exit(EXIT_FAILURE);
    }
}
else if (pid > 0)
{
    // 부모 프로세스: 자식 프로세스가 끝날 때까지 대기
    if (!background)
    {
        wait(NULL);
    }
}
else
{
    perror("fork failed");
}
```

: fork()는 새로운 자식 프로세스를 생성하여 명령어를 실행
; find_executable()로 명령어가 실행 가능한 파일인지 확인
: execve()로 자식 프로세스에서 실행파일을 실행, 성공시 호출한 프로세스를 대체하므로 이후의 코드는 실행되지 않음

4.3 실행 예시

-- 실행 결과창에서 확인 --

5. 실행결과창

5.1 컴파일

1) gcc -o 컴파일명 컴파일할_파일 : 수동 컴파일

```
tg2002kim@BOOK-2I0K8A1RQ6:~/OpSys/2024-os-hw1$ gcc -o sish sish.c
tg2002kim@BOOK-2I0K8A1RQ6:~/OpSys/2024-os-hw1$ ./sish
SiSH>
```

2) make : Makefile로 컴파일

```
tg2002kim@BOOK-2I0K8A1RQ6:~/OpSys/2024-os-hw1$ make
gcc -Wall -c sish.c
gcc -Wall -o sish sish.o
gcc -Wall -c fork.c
gcc -Wall -o fork fork.o
gcc -Wall -c fork2.c
gcc -Wall -o fork2 fork2.o
gcc -Wall -c getenv.c
gcc -Wall -o getenv getenv.o
gcc -Wall -c stat.c
gcc -Wall -o stat stat.o
tg2002kim@BOOK-2I0K8A1RQ6:~/OpSys/2024-os-hw1$ ./sish
SiSH>
```

3) make clean : 컴파일 파일 제거

```
tg2002kim@BOOK-2I0K8A1RQ6:~/OpSys/2024-os-hw1$ make clean
rm -f *.o sish fork fork2 getenv stat
```

5.2 My_shell 실행 및 종료 (+ ctrl + C)

```
tg2002kim@BOOK-2I0K8A1RQ6:~/OpSys/2024-os-hw1$ ./sish
SiSH> quit
tg2002kim@BOOK-2I0K8A1RQ6:~/OpSys/2024-os-hw1$ ./sish
SiSH> ^C
tg2002kim@BOOK-2I0K8A1RQ6:~/OpSys/2024-os-hw1$
```

5.3 기본 명령어 실행

1) ls : 디렉토리의 파일과 디렉토리 목록 출력

```
SiSH> ls
Makefile   file    fork.o   fork2.o   getenv.o  sish.o  stat.o
README.md  fork    fork2    getenv    sish      stat
Sample     fork.c  fork2.c  getenv.c  sish.c    stat.c
```

2) pwd : 현재 작업중인 경로 출력

```
SiSH> pwd
/home/tg2002kim/OpSys/2024-os-hw1
```

3) echo "string" : 문자열 출력

```
SiSH> echo "Hello, world!"
"Hello, world!"
```

4) date : 현재 시스템의 날짜와 시간 출력

```
SiSH> date
Sun Sep 29 17:18:08 KST 2024
```

5) whoami : 현재 시스템에 로그인한 사용자의 이름 출력

```
SiSH> whoami
tg2002kim
```

: 각각의 명령어는 자식 프로세스를 생성(fork), 자식 프로세스에서 실행(execve), 부모 프로세스는 자식 프로세서가 종료되는 것을 기다립니다(wait).

5.4 명령어 인자 전달

1) echo string : 인자가 포함된 명령어 실행

```
SiSH> echo Hello World
Hello World
```

2) ls -1 : 인자와 옵션이 있는 명령어 실행 ->

```
SiSH> ls -1
Makefile
README.md
Sample
file
fork
fork.c
fork.o
fork2
fork2.c
fork2.o
getenv
getenv.c
getenv.o
sish
sish.c
sish.o
stat
stat.c
stat.o
```

3) cat sish.c : 파일이름을 인자로 전달하여 내용 출력

```
SiSH> cat sish.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```

4) echo $PATH : 환경변수 사용 : 작동안됨

```
SiSH> echo $PATH
$PATH
```

수정 후 ->

```
SiSH> $PATH
Segmentation fault
```

5.5 예외처리

1) 빈 명령어 입력

```
Command not found: Nope
SiSH> wrongcommand
```

2) 존재하지 않는 명령어 실행

```
SiSH> wrongcommand
Command not found: wrongcommand
```

3) 잘못된 경로 입력

```
SiSH>
Command not found: (null)
```

: execve에 존재하지 않는 명령어를 실행하려 하면 실패한다는 메시지가 출력

5.6 권한 문제

1) touch testfile : 파일 생성

```
SiSH> touch testfile
SiSH> ls
Makefile   file    fork.o   fork2.o   getenv.o   sish.o   stat.o
README.md  fork    fork2    getenv    sish       stat     testfile
Sample     fork.c  fork2.c  getenv.c  sish.c     stat.c
```

2) chmod 444 testfile : 읽기 전용 파일로 설정

```
SiSH> chmod 444 testfile
SiSH> ls -l testfile
-r--r--r-- 1 tg2002kim tg2002kim 0 Sep 29 17:57 testfile
```

3) ./testfile : 실행 권한이 없는 파일 실행 시도

```
SiSH> ./testfile
Command not found: ./testfile
```

4) rm testfile

```
SiSH> rm testfile
rm: remove write-protected regular empty file 'testfile'? y
SiSH> ls
Makefile   file    fork.o   fork2.o   getenv.o  sish.o  stat.o
README.md  fork    fork2    getenv    sish      stat
Sample     fork.c  fork2.c  getenv.c  sish.c    stat.c
```

5.7 자식 프로세스 종료 대기

1) sleep 5 : 5초간 대기한 후 종료 ->

```
SiSH> sleep 5
5
4
3
2
1
SiSH> Command not found: 5
SiSH> Command not found: 4
SiSH> Command not found: 3
SiSH> Command not found: 2
SiSH> Command not found: 1
SiSH>
```

5.8 파일 및 디렉토리 관리

1) mkdir testdir : 디렉토리 생성

```
SiSH> mkdir testdir
SiSH> ls
Makefile   file    fork.o   fork2.o   getenv.o  sish.o  stat.o
README.md  fork    fork2    getenv    sish      stat    testdir
Sample     fork.c  fork2.c  getenv.c  sish.c    stat.c
```

2) cd testdir : 생성된 디렉토리로 이동

3) touch testfile : 파일 생성

```
SiSH> cd testdir
SiSH> ls
SiSH> touch testfile
SiSH> ls
testfile
```

4) rm testfile : 파일 삭제

```
SiSH> rm testfile
SiSH> ls
Makefile   file    fork.o   fork2.o   getenv.o  sish.o  stat.o
README.md  fork    fork2    getenv    sish      stat    testdir
Sample     fork.c  fork2.c  getenv.c  sish.c    stat.c
```

5) cd .. : 상위 디렉토리로 이동

```
SiSH> cd ..
SiSH> ls
Makefile   file    fork.o   fork2.o   getenv.o  sish.o      stat.c
README.md  fork    fork2    getenv    sish      sish_pre.c  stat.o
Sample     fork.c  fork2.c  getenv.c  sish.c    stat        testdir
```

6) rmdir testdir : 디렉토리 삭제

```
SiSH> rmdir testdir
SiSH> ls
Makefile   file    fork.o   fork2.o   getenv.o  sish.o  stat.o
README.md  fork    fork2    getenv    sish      stat
Sample     fork.c  fork2.c  getenv.c  sish.c    stat.c
```

5.9 프로세스 관리 및 백그라운드 실행 (Optional)

1) sleep 5 & : 백그라운드 실행 : 작동안됨

```
SiSH> sleep 5 &
SiSH> sleep: invalid time interval ''
Try 'sleep --help' for more information.
```

2) ls & : 다른 명령어를 백그라운드에서 실행 : 작동안됨

```
SiSH> ls &
SiSH> ls: cannot access '': No such file or directory
```

5.10 특수 문자 처리

1) echo "Hello & World"

```
SiSH> echo "Hello & World"
"Hello & World"
```

2) ls | grep sish : 파이프 구현

```
SiSH> ls | grep sish
sish
sish.c
sish.o
sish_pre.c
```

3) ls && echo "Done" : AND 구현 : 작동안됨

```
SiSH> ls && echo "Done"
ls: cannot access '&&': No such file or directory
ls: cannot access 'echo': No such file or directory
ls: cannot access '"Done"': No such file or directory
```

4) ls; echo "Next" : 순자실행 : 작동안됨

```
SiSH> ls; echo "Next"
Makefile   file    fork.o   fork2.o   getenv.o  sish.o      stat.c
README.md  fork    fork2    getenv    sish      sish_pre.c  stat.o
Sample     fork.c  fork2.c  getenv.c  sish.c    stat
SiSH> echo "Next"
"Next"
```

6. 프로그램 빌드 환경

컴파일 환경: Ubuntu 22.04 -> VS Code 1.93
프로그래밍 언어: C언어
실행 방법: gcc -o sish sish.c -Wall /or/ make 이후 ./sish 으로 실행
실행 파일: sish.c / Makefile

7. 문제점과 해결방법

7.1 무한로딩 = 좀비 프로세스

현상 : shell을 실행하고서 일부 명령어를 작동시키던 중 다음 명령어 입력을 받지 않는 상태로 지속되는 현상이 발생

원인 : 자식 프로세스가 종료되었지만, 부모 프로세스가 자식 프로세스의 정보를 수집하지 않아 좀비 프로세스가 잔류하는 현상으로, 부모 프로세스가 wait()함수를 호출하여 해결할 수 있습니다.

```
else if (pid > 0)
{
    // 부모 프로세스: 자식 프로세스가 끝날 때까지 대기
    if (!background)
    {
        wait(NULL);
    }
}
```

해결 : 자식 프로세스의 pid를 반환 받아, 백그라운드 실행이 아닐 경우에만, 부모 프로세스가 자식 프로세스가 끝나는 것을 기다린다.

7.2 echo $PATH: 환경 변수 사용 문제 // 해결 안됨

```
if (strcmp(args[0], "echo") == 0)
{
    for (int i = 1; args[i] != NULL; i++)
    {
        if (args[i][0] == '$')
        {
            char *env_var = getenv(args[i] + 1);
            if (env_var)
            {
                printf("%s ", env_var);
            }
            else
            {
                printf("Environment variable not found ");
            }
        } else
        {
            printf("%s ", args[i]);
        }
    }
    printf("\n");
    continue;
}
```

현상 : echo $PATH 명령어 실행 시 환경변수 값이 출력되지 않고 '$PATH'가 출력됨

원인 : 환경 변수 처리 시 : '$'를 올바르게 처리 하지 않아 작동이 안됩니다. 'echo' 명령어 자체와 '$' 기호로 시작하는 환경 변수를 구분하여 처리해야 하는데, 이 부분이 누락된 상태 였습니다.

해결 : echo 명령어를 인식하고, 인자가 $로 시작하는 경우 환경 변수 값을 출력하는 로직을 추가해야 합니다.

7.3 파이프 구현 문제

현상 : ls | grep sish 명령어의 '|'를 인식하지 못해 ls 명령어만 실행한 후 '|'. grep와 sish의 실행에서 오류를 띄었음

원인 : 파이프의 구현 중 명령어의 출력을 다른 명령어의 입력으로 연결하는 과정에서 표준 입력과 쓰기를 제대로 이해하지 않아 연결을 잘못한 것이 문제였다.

해결 : GPT의 도움을 받음... //

7.4 **백그라운드 실행 &** **// 해결 안됨**

현상 : 백그라운드 실행 명령어 'sleep 5 &'가 작동을 하지않고, &에서 오류를 발생시킴

원인 : 백그라운드에서 실행된 프로세스가 다른 명령어들과 동시에 실행되는 경우가 생겨 여러 프로세스의 출력이 동시에 출력될 경우 출력이 섞이는 것을 원인으로 예상한다.

해결 : 1안으로는 waitpid() 호출 함수를 사용하여 좀비 프로세스를 정리하도록 하는 방법이 있다. 2안으로는 출력 관리로 실행된 프로세스는 출력을 버퍼링 하거나, 로그파일로 출력하여, 출력이 섞이지 않게 하는 것이다. (방법까지는 알아보았지만 구현에는 실패)

7.5 **순차 실행 ;** **// 해결 안됨**

현상 : 순차 실행을 테스트 하는 명령어 ' ls; echo "Next"가 실행되지 않았다.

원인 : ';'문자를 기준으로 첫 번째 명령어인 것을 식별 후 두 번째 명령어를 이어서 실행시키려 하였지만, 첫 번째 명령어가 종료되지 않은 상태로 두 번째 명령어가 실행된다.

해결 : wait()함수를 사용하여 자식 프로세스가 종료되는 것을 부모 프로세스가 기다리도록 코드를 수정한다. (수정하였으나 적절히 작동하지 않음)

7.6 **논리 AND 실행 &&** **// 해결 안됨**

현상 : AND 연산자를 테스트 하는 명령어 'ls && echo "Done""dl 실행되지 않았다.

원인 : 순차실행과 같은 첫 번째 명령어가 종료되지 않고, 두 번째 명령어가 실행된 것으로 보인다.

해결 : wait()함수를 적절히 사용하여 첫 번째 명령어의 종료를 확인 후 두 번째 명령어가 실행되도록 수정한다. (마찬가지로 수정하였으나 작동하지 않음..)

8. 느낀점

기본적인 fork(), execve(), wait() 등의 개념도 헷갈리는 상황에서 무작정 시작하려니 어려웠습니다. 시작하기 전에 과제의 예시로 제공된 Sample 코드들을 분석하고, 연습삼아 비슷한 코드들을 작성하면서 점차적으로 개념을 익혀나가며 공부했는데, 아직은 학기 초라 여유가 있어서 아슬아슬하게 마무리했다는 느낌이었습니다.

my_shell의 기본 틀도 인터넷에서 참고한 후, 하나하나 수정해보고, 오류가 발생할 때마다 GPT에게 문제점을 물어보며 반복적으로 개선해 나갔습니다. 이를 통해 추가해야 할 개념들을 하나씩 익혔고, 관련된 코드들을 찾아가며 프로젝트를 완성했습니다. 또한, 테스트해야 할 명령어가 많아서 하나하나 실행이 되는지 확인하는 과정이 힘들었지만, GPT에게 도움을 받아 작성한 테스트 시나리오 덕분에 Shell 기능을 테스트하는 시간을 크게 절약할 수 있었습니다.

가장 생소하게 느껴졌던 부분은 Makefile이었는데, 파일들을 일괄적으로 컴파일하는 방식이 신박하다고 느꼈습니다. 단순하게 컴파일하는 대신, 여러 파일을 한 번에 관리할 수 있다는 점이 인상 깊었습니다. 가장 구현하기 힘들었던 부분은 파이프였는데, dup2()라는 생소한

시스템 호출을 통해 표준 입력과 출력을 파이프의 읽기와 쓰기로 재지정하는 과정이 머리로 잘 그려지지 않아 개념을 이해하는 데 시간이 걸렸습니다. 결국 구현에는 GPT의 힘을 빌렸지만, 그 덕에 대략적인

가장 아쉬운 점은 PATH 출력 관련 부분입니다. 처음에는 명령어를 '$PATH'라고 바로 실행하게 했지만 작동하지 않았고, ‘echo $PATH‘를 실행하는 방식으로 바꿨지만 여전히 해결되지 않아 아쉬움이 남았습니다.

9. 참고자료

* 책 : Operating Systems: Three Easy Pieces (번역본)
- Remzi H. Arpaci-Dusseau & Andrea C. Arpaci-Dusseau

* 유튜브 개념 강의 / 프로세스의 생성과 복사 :
https://www.youtube.com/watch?v=RzN18na94Wc

* Makefile 만들기 :
https://velog.io/@hidaehyunlee/Makefile-%EB%A7%8C%EB%93%A4%EA%B8%B0

* Shell 구현 기본 틀 참고 :
https://velog.io/@32190192/%EC%89%98-%EA%B5%AC%ED%98%84%ED%95%98%EA%B8%B0

* Shell 명령어 구현 참고 : https://asidefine.tistory.com/108

* 파이프 개념 참고 : https://hump-mountain.tistory.com/4

* 순차실행 및 파이프 구현 참고용 사이트 모음 : https://mhwan.tistory.com/42

* 환경변수 출력 구현 참고 : https://cpm0722.github.io/system-programming/shell

* 그 외 운영체제 수업 학우들의 깃허브 :
https://github.com/mobile-os-dku-cis-mse/2024-os-hw1