



<Homework 1>

운영체제(MS)

My Simple Shell 보고서

학 과 : 모바일시스템공학과

학 번 : 32164809

이 름 : 탁준석

담당교수 : 유시환

제출일자 : 21.09.30

남은 프리데이 : 5



목 차

1. 개요

- 가. 실험 목적
- 나. 실험 목표

2. 이론 및 적용

- 가. 이론
- 나. 적용

3. mysinh 프로그램

- 가. 개발 환경
- 나. 프로그램 분석

4. 문제 및 해결

- 가. 디버깅
- 나. 프로그램 구조화

5. 결과 및 분석

- 가. Makefile 실행
- 나. 리눅스 내외장 명령어 실행

6. 고찰 및 느낀 점

- 가. 고찰
- 나. 느낀 점

7. 참고 문헌

1. 개요

가. 실험 목적

이번 운영체제 과제1은 나만의 작은 셸(My simple shell)을 개발하는 것이다. 운영체제와 프로세스의 전반적인 개념을 살펴보고 운영체제 구성 요소인 커널과 셸이 무엇인지 알아본다. 커널과 셸의 역할을 이해했으면 유닉스 프로세스 상태에서 사용자 모드와 커널 모드의 차이를 구분하고 프로세스를 관리하는 시스템 호출을 학습한다. 프로세스의 시작 및 종료와 긴밀하게 연관된 시스템 호출을 사용하여 bash 셸처럼 동작하는 프로그램을 완성해 나가면서 사용자와 커널 사이에서 명령어 해석기의 역할을 담당하는 셸을 심도 있게 이해한다.

나. 실험 목표

1) 기본 목표

- 사용자와 상호작용할 수 있는 mysish 셸을 개발하여 리눅스의 기본적인 내외부 명령어를 수행한다.

2) 추가 목표

- 코드를 컴파일하여 오브젝트 파일을 생성하고, 오브젝트 파일을 링킹하여 실행 파일을 생성하는 Makefile을 만든다.
- 셸 프롬프트를 각색한다.
- 리눅스 내외부 명령어에 옵션과 인수를 제공하여 다양한 명령을 수행한다.

구현	예시
makefile	make, make clean
리눅스 내장 명령어	cd, history, exit(quit)
리눅스 외장 명령어(옵션과 인수)	ls, cat, man, vi, gcc 등
프롬프트	bash 프롬프트 각색

<표 1.1> 과제1 mysish 프로그램에서 구현할 목표와 예시.

2. 이론 및 적용

제2장 이론 및 적용은 이론과 적용으로 절을 나누어져 있다. 이론 절은 셸을 개발하기 전에 기초를 다질 수 있도록 운영체제, 프로세스, 그리고 시스템 호출에 대해 살펴본다.

가. 이론

1) 운영체제

가) 운영체제의 개념

운영체제는 사용자가 응용 프로그램을 실행할 수 있는 환경을 제공하여 컴퓨터를 편리하게 사용할 수 있도록 도와주고, 하드웨어를 효율적으로 사용할 수 있도록 다양한 기능을 제공하는 소프트웨어이다. 컴퓨터 시스템을 여러 사용자가 사용할 수 있으며, 다양한 응용 프로그램이 있을 수 있다. 운영체제는 다양한 응용 프로그램이 하드웨어, 즉 컴퓨터의 자원을 효율적으로 사용할 수 있도록 관리하고 조정하는 역할을 한다. 이에 따라 제공하는 기능도 무척 다양한데, 주요 기능을 크게 자원관리와 시스템 관리로 분류하여 살펴보자.

나) 운영체제의 기능

(1) 자원관리; 프로세스와 시스템 모드

컴퓨터 시스템의 메모리, 프로세스, 장치, 파일 등 구성 요소를 자원이라고 하는데 운영체제는 이 자원 관리한다. 특히 프로세스는 실행 중인 프로그램으로 보통 일괄 처리 작업 하나가 프로세스가 된다. 하나의 프로세스는 프로세서, 메모리, 파일, 입출력장치와 같은 자원이 있어야 업무를 수행할 수 있는데, 자원은 프로세스를 생성할 때 제공하거나 실행 중에 할당할 수 있다. 시스템은 이런 프로세스의 집합으로, 크게 시스템 코드를 수행하는 운영체제 프로세스와 사용자 코드를 수행하는 사용자 프로세스로 구분한다.

(2) 시스템 관리; 시스템 보호

운영체제는 추가로 시스템 보호, 네트워크, 명령 해석기 등 기능을 지원한다. 시스템 보호는 컴퓨터 자원에서 프로그램, 프로세서, 사용자의 접근을 제어하는 방법이다. 운영체제는 파일 사용 권한 부여, 데이터 암호화 등 서비스를 제공하여 데이터와 시스템을 보호한다. 컴퓨터 시스템에서는 여러 프로세스를 동시에 실행할 수 있으므로 서로 보호해야 한다. 그리고 네트워크로 파일 공유 사이트에 접속할 때는 다른 사용자의 프로그램에서 보호해야 한다.

다) 유닉스 운영체제의 구조

(1) 커널

유닉스는 크게 커널과 응용 프로그램으로 구성된다. 커널은 유닉스 운영체제의 핵심으로 프로세스 관리, 메모리 관리, 파일 시스템 관리, 장치 관리 등 컴퓨터의 모든 자원을 초기화하고 제어하는 기능을 수행한다. 커널은 파일 및 입출력을 담당하는 파일 서브 시스템과 프로세스 제어 서브 시스템으로 구성된다. 그리고 응용 프로그램은 시스템 호출로 커널에 컴파일과 파일 조작 등의 기능을 요청한다.

- 파일 서브 시스템

메모리, 외부 장치에 데이터를 전송하거나 블록 단위로 입출력을 실행할 때 메모리의 일부를 버퍼로 할당하고, 이것을 사용자의 주소 공간과 외부 장치 사이에 두는 디스크 캐시 방법으로 파일을 관리한다. 그리고 파일 공간 할당, 자유 공간 관리, 파일 접근 제어 등의 역할을 한다. 프로세스를 시스템 호출을 하여 파일 서브 시스템과 상호작용한다.

- 프로세스 제어 서브 시스템

프로세스 제어 서브 시스템은 프로세스의 동기화, 통신, 메모리 관리, 스케줄링을 담당한다. 프로세스를 제어하는 시스템 호출에는 fork, exec, wait, signal 등이 있다.

(2) 셸

셸은 사용자와 커널 사이의 중간자 역할을 담당하는 특별한 프로그램이다. 셸은 사용자가 입력한 명령을 해석하여 커널에 넘겨준다. 그러면 커널이 명령의 수행 결과를 돌려주는데, 셸은 이것을 다시 사용자가 이해할 수 있는 형태로 바꿔서 출력한다. 유닉스에서 셸은 여러 가지인데, 우리 모바일시스템공학과 서버에 설치된 셸은 배시 셸(bourne again shell)이다.

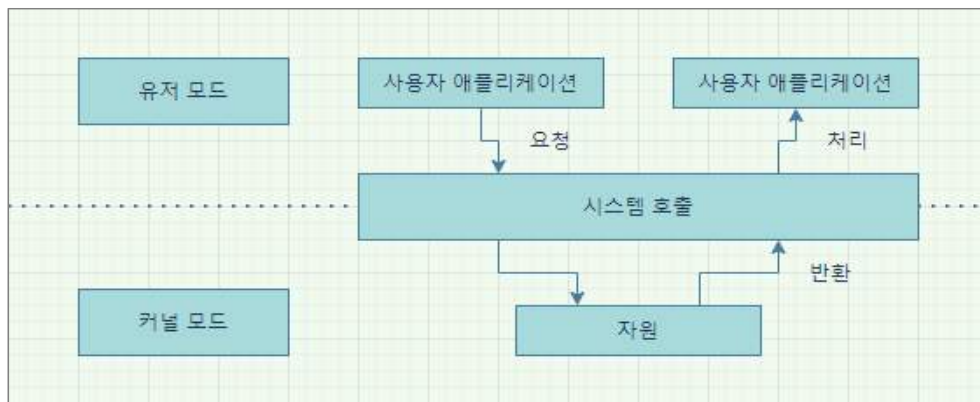
(3) 유틸리티와 파일 시스템

유닉스는 각종 개발도구, 문서 편집 도구, 네트워크 관련 도구 등 매우 다양한 유틸리티를 제공한다. 유틸리티는 사용자에게 편의를 제공하려고 준비한 시스템으로 유닉스 기본 명령어로 구성되고, 사용자가 작성한 응용 프로그램도 유틸리티로 취급한다. 그리고 유닉스는 계층적으로 구성된 파일 시스템을 사용하여 시스템 파일과 사용자 파일을 체계적으로 관리한다.

2) 유닉스 프로세스

가) 프로세스 상태

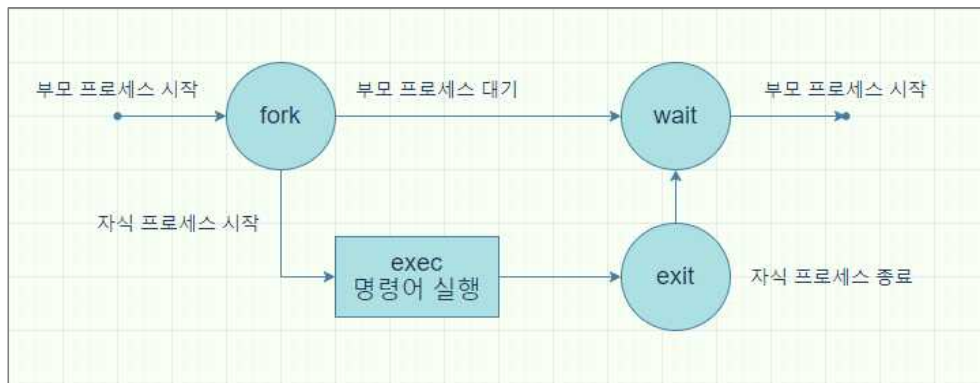
유닉스는 사용자 프로세스 환경에서 실행되므로 사용자 모드와 커널 모드가 있고, 시스템 프로세스와 사용자 프로세스를 사용한다. 사용자 모드는 사용자가 접근할 수 있는 영역을 제한하고 프로그램의 자원에 함부로 접근할 수 없는 모드로, 사용자 애플리케이션이 이곳에서 실행된다. 커널 모드는 컴퓨터의 CPU, 메모리 등 모든 자원에 접근할 수 있는 모드로, 사용자 모드는 시스템 호출, 인터럽트, 예외를 통해 커널 모드로 자원을 요청한다. 시스템 프로세스는 커널 모드에서 동작하며, 프로세스 할당, 스케줄링, 메모리 할당, 등 시스템 관리 작업을 수행하는 운영체제 코드를 실행한다. 사용자 프로세스는 사용자 모드에서 동작하며 사용자 프로그램이나 유틸리티를 실행한다.



<그림 2.1> 사용자 모드와 커널 모드 사이의 시스템 호출을 통한 요청과 반환.

나) 프로세스 제어

자식 프로세스는 fork 함수로 생성되어 부모 프로세스 주소 공간의 복사본으로 구성된다. 부모와 자식 프로세스는 fork 함수를 수행한 후에도 계속 실행되는데 새로운 자식 프로세스는 fork 함수의 반환 값으로 0을 돌려받고 부모 프로세스는 fork 함수의 반환 값으로 자식 프로세스의 프로세스 식별자(process identification, PID)를 돌려받는다. exec 함수는 가상 메모리에 새로운 프로그램을 바꾸어놓으며 부모 프로세스와 자식 프로세스 중 한 프로세스를 생성한 후에 사용한다. 프로세스는 exit 함수로 종료하고 부모 프로세스는 wait 함수로 자식 프로세스가 종료하기를 기다린다. 자식 프로세스가 올바르게 종료되면 부모 프로세스가 계속해서 실행되고, 부모 프로세스가 자식 프로세스보다 일찍 종료되면 자식 프로세스는 부모가 없는 좀비 프로세스가 된다.



<그림 2.2> 유닉스 fork 함수, exec 함수, 그리고 wait 함수와 프로세스와의 관계.

3) 사용할 시스템 호출

시스템 호출은 응용 프로그램의 요청에 따라서 운영체제 커널에 접근하기 위한 인터페이스이다. 우리는 myshell 셸을 개발하기 위해 리눅스에서 제공하는 시스템 호출을 더 자세하게 알아야 한다. 이 절에서는 myshell 셸을 개발하는데 사용되는 시스템 호출의 형식을 살펴보고 어떠한 기능을 하는지 알아보도록 한다.

가) 프로세스 제어

(1) fork()

형식	#include <unistd.h> pid_t fork(void);								
설명	fork는 새로운 프로세스를 생성하는 시스템 호출이다. fork를 호출한 프로세스는 부모 프로세스라고 하고, 새로 생성된 프로세스는 자식 프로세스라고 한다. 자식 프로세스는 부모 프로세스의 데이터를 복사하고 새로운 식별자(process identification, PID)를 가진다.								
반환	<table border="1"> <thead> <tr> <th>반환</th><th>설명</th></tr> </thead> <tbody> <tr> <td>양수</td><td>부모 프로세스에게 자식 프로세스의 PID를 반환한다.</td></tr> <tr> <td>0</td><td>자식 프로세스에게 0을 반환한다.</td></tr> <tr> <td>-1</td><td>오류가 발생하였을 때 -1을 반환한다.</td></tr> </tbody> </table>	반환	설명	양수	부모 프로세스에게 자식 프로세스의 PID를 반환한다.	0	자식 프로세스에게 0을 반환한다.	-1	오류가 발생하였을 때 -1을 반환한다.
반환	설명								
양수	부모 프로세스에게 자식 프로세스의 PID를 반환한다.								
0	자식 프로세스에게 0을 반환한다.								
-1	오류가 발생하였을 때 -1을 반환한다.								

(2) `execve()`

형식	<code>#include <unistd.h> int execve(const char* path, char* const argv[], char* const envp[])</code>				
설명	<code>exec</code> 는 프로세스 공간에 새로운 프로세스의 이미지를 덮어서 새로 실행하는 시스템 호출이다. 이전에 실행되던 프로그램은 없어지고 새 프로그램을 메모리에 올려 실행한다. <code>exec</code> 계열인 <code>execve</code> 시스템 호출은 <code>path</code> 에 지정한 경로의 파일을 실행하며 <code>argv</code> 에 마지막이 널 문자인 배열로 매개변수를 전달한다. <code>envp</code> 에 환경변수를 정의할 수 있다.				
반환	<table border="1"> <thead> <tr> <th>반환</th><th>설명</th></tr> </thead> <tbody> <tr> <td>-1</td><td>새로운 프로그램 교체에 실패하면 <code>errno</code>가 설정되어 오류를 표시한다.</td></tr> </tbody> </table>	반환	설명	-1	새로운 프로그램 교체에 실패하면 <code>errno</code> 가 설정되어 오류를 표시한다.
반환	설명				
-1	새로운 프로그램 교체에 실패하면 <code>errno</code> 가 설정되어 오류를 표시한다.				

(3) `wait()`

형식	<code>#include <sys/wait.h> pid_t wait(int *stat_loc);</code>						
설명	<code>wait</code> 는 자식 프로세스가 종료될 때까지 부모 프로세스를 기다리게 하는 시스템 호출이다. <code>wait</code> 를 호출한 프로세스는 신호를 받을 때까지 대기한다.						
반환	<table border="1"> <thead> <tr> <th>반환</th><th>설명</th></tr> </thead> <tbody> <tr> <td>양수</td><td>자식 프로세스가 정상적으로 종료되었을 때 자식 프로세스의 PID를 반환한다.</td></tr> <tr> <td>-1</td><td>자식 프로세스가 없거나 시스템 호출이 인터럽트 되었을 때, -1을 반환한다.</td></tr> </tbody> </table>	반환	설명	양수	자식 프로세스가 정상적으로 종료되었을 때 자식 프로세스의 PID를 반환한다.	-1	자식 프로세스가 없거나 시스템 호출이 인터럽트 되었을 때, -1을 반환한다.
반환	설명						
양수	자식 프로세스가 정상적으로 종료되었을 때 자식 프로세스의 PID를 반환한다.						
-1	자식 프로세스가 없거나 시스템 호출이 인터럽트 되었을 때, -1을 반환한다.						

4) 사용할 C 라이브러리 함수

사용자는 마치 셸과 문자로 대화하는 것처럼 명령을 내린다. 이것은 우리가 mysish 셸을 개발하기 위해 여러 문자열을 조작해야 하고 특히 스트링과 관련된 C 라이브러리의 함수를 자주 사용해야 한다는 것을 뜻한다. 이 절에서는 mysish 셸을 개발하는데 사용되는 몇 가지 C 라이브러리 함수의 형식을 확인하고 어떠한 기능을 하는지 살펴보도록 한다.

가) 스트링 조작

(1) strcpy() - 스트링 복사

형식	#include <string.h> char *strcpy(char *string1, const char *string2);
설명	strcpy 함수는 끝나는 널 문자를 포함해서 string2를 string1에서 지정한 위치로 복사한다. strcpy 함수는 널로 끝나는 문자열에서 작동하는데 이때 문자열은 인수는 스트링 끝에 널 문자(w0)를 포함해야 한다. 길이 검사는 수행하지 않는다.
반환	strcpy() 함수는 복사된 스트링에 대한 포인터를 반환한다(string1).

(2) strtok_r() - 스트링 토큰화

형식	#include <string.h> char *strtok_r(char *string, const char *seps, char **lasts);
설명	strtok_r 함수는 0개 이상의 토큰으로 string을 읽고 string에서 토큰 구분자 역할을 하는 문자로 seps를 읽는다. 인수 lasts는 사용자가 제공한 포인터를 가리키는데 이것은 strtok_r 함수가 같은 문자열을 계속 읽는데 필요한 정보를 가리킨다. string에서 다음 토큰을 읽으려면 NULL 인수와 함께 strtok_r() 함수를 호출한다. 이것으로 strtok_r 함수는 이전 토큰에서 다음 토큰을 찾는다. 원래 string의 구분자는 널 문자로 대체되며 lasts가 가리키는 포인터가 계속해서 갱신된다. string에 남은 토큰이 없으면 NULL 포인터가 반환된다.
반환	strtok_r 함수를 처음 호출할 때 string에서 첫 번째 토큰에 대한 포인터를 반환한다. 토큰 스트링이 같은 그다음 호출에서 strtok_r 함수는 스트림에서 다음 토큰을 가리키는 포인터를 반환한다. 토큰이 더 없으면 NULL 포인터가 반환된다.

나) 환경 상호작용

(1) getenv()

형식	#include <stdlib.h> char *getenv(const char *varname);
설명	getenv() 함수는 varname에 해당하는 항목에 대한 환경변수의 리스트를 검색한다.
반환	getenv() 함수는 현재 환경에 지정된 varname의 값을 포함하여 포인터를 문자열로 반환한다. getenv()가 환경 스트링을 찾을 수 없는 경우, 널이 반환되며 errno가 설정되어 오류를 표시한다.

(2) exit

형식	#include <stdlib.h> void exit(int status);
설명	exit() 함수는 프로그램에서 호스트 환경에 제어를 반환한다. 그리고 프로그램을 종료하기 전에 버퍼를 모두 삭제하고 열린 파일을 모두 닫는다.
반환	exit() 함수는 운영체제에 status 값과 제어를 둘 다 반환한다.

다) 문자 및 스트림 입출력**(1) getchar()**

형식	#include <stdio.h> int getc(FILE *stream); int getchar(void);
설명	getc() 함수는 현재 stream 위치에서 단일 문자를 읽고 stream 위치를 다음 문자로 이동한다. getchar() 함수는 getc(stdin)과 같다.
반환	getc() 및 getchar() 함수는 읽은 문자를 반환한다. EOF의 반환 값은 오류 또는 파일의 끝 조건을 표시한다.

나. 적용

이론 절에서 운영체제, 프로세스의 개념과 셸을 동작시키는데 사용되는 시스템 호출 함수, C 라이브러리 함수의 종류를 알아보았다. 적용 절은 이 이론을 바탕으로 myish 셸을 설계하기 위해 프로그램 디자인을 고안하고 이 디자인이 코드로 어떻게 적용될 것인지 알아본다. 프로그램은 절차 지향적으로 개발하였고 셸이 동작하는 기능 순서대로 최대한 묶어서 처리하였다.

1) 설계

셸을 설계하는데 한 가지 좋은 방법은 셸이 동작하는 기본적인 순서를 따라가는 것이다. 따라서 셸이 동작하는 순서마다 잠시 멈추어 역할을 먼저 파악하고 어떠한 행동을 해야 하는지 생각해보자. 그리고 큰 틀에서 그 행동마다 사용해야 할 언어 문법, 자료구조와 알고리즘, 시스템 호출, 그리고 C 라이브러리 함수를 써보자. <그림 2.3>는 절차적으로 설계한 myish 프로그램 디자인을 순서도로 나타낸 것이다.

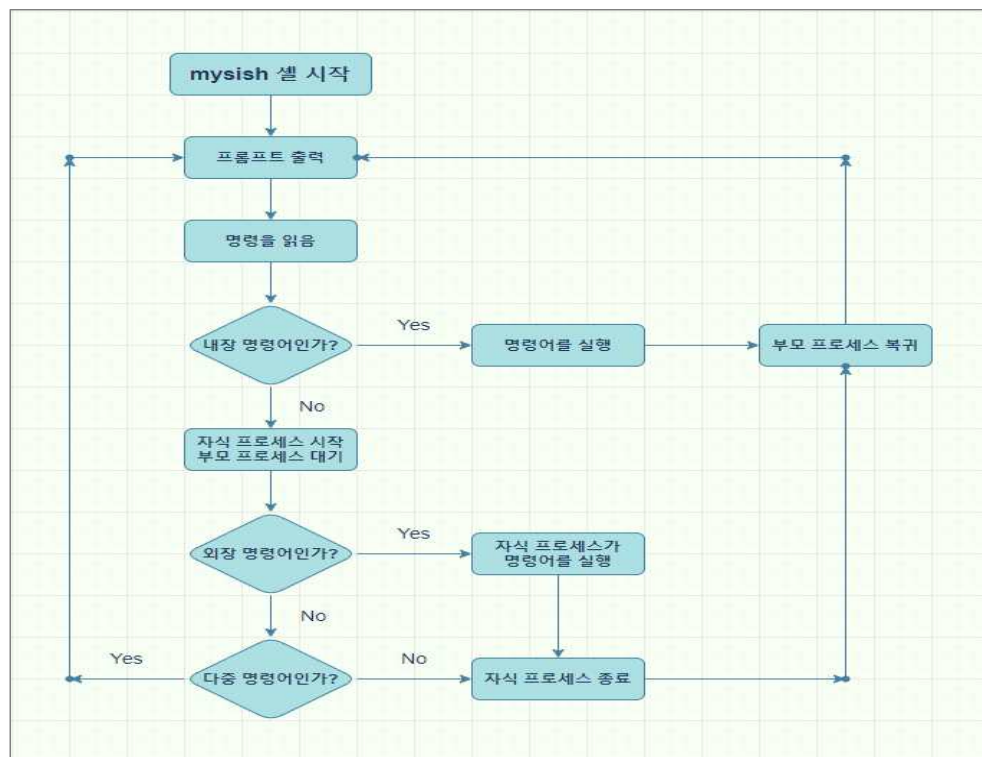
① 프롬프트를 출력한다.

셸은 프롬프트에서 시작한다. 사용자가 명령어를 입력하기 전, 그리고 명령어를 입력하고 실행한 후에 셸은 프롬프트를 출력하여 사용자가 명령을 입력하기를 기다려야 한다. 셸 프롬프트는 가장 바깥쪽에 있는 while 반복문이 시작할 때 가장 먼저 출력되어야 한다.

② 명령을 읽는다.

사용자가 터미널에 문자열을 입력하였을 때, 이 문자열을 읽어야 한다. getchar 함수로 문자열을 읽어 배열에 저장한다. 배열에 저장된 문자열은 strtok_r 함수로 명령, 옵션, 인수로 구분해야 한다.

- ③ **내장 명령어인지 확인한다.** 사용자가 입력한 문자열이 리눅스 내장 명령어인지 확인해야 한다. strcmp 함수로 내장 명령어와 문자열을 비교한다. 내장 명령어일 경우, 명령어에 대응하는 코드를 실행한다. 그리고 자식 프로세스를 생성하지 않고 다시 프롬프트를 출력한다.
- ④ **환경변수를 읽는다.** 외장 명령어를 실행하기 위해서 getenv 함수로 PATH 환경변수에 저장된 경로를 불러와 배열에 저장한다.
- ⑤ **자식 프로세스를 시작한다.** 외장 명령어는 자식 프로세스에서 실행된다. fork 시스템 호출로 자식 프로세스를 생성한다.
- ⑥ **외장 명령어인지 확인한다.** 사용자가 입력한 문자열이 리눅스 외장 명령어인지 확인해야 한다. strcpy 사용자가 입력한 문자열과 경로를 연결한다. 실행 파일일 경우, execve 함수에 실행 파일의 경로를 전달하고 외장 명령어일 경우, execve 함수에 PATH 환경변수의 경로를 전달한다. 명령어를 실행하고 exit 함수로 프로세스를 종료한다.
- ⑦ **부모 프로세스로 복귀한다.** wait 시스템 호출로 기다리던 부모 프로세스는 자식 프로세스가 exit 함수를 호출한 이후, 다시 프롬프트를 출력하여 사용자가 명령어를 입력하기를 기다린다.



<그림 2.3> myish의 초기 디자인 순서도. myish 셸의 생명 주기를 간단히 보여준다.

3. mysish 프로그램

제3장 mysish 프로그램은 제2장에서 학습한 내용을 바탕으로 설계한 mysish 프로그램을 분석한다.

가. 개발 환경

이번 과제1 mysish 프로그램은 C 언어로 Ubuntu 16.04.7 LTS에서 vi 편집기를 사용하여 개발되었다. mysish 프로그램의 소스 코드는 https://github.com/mobile-os-dku-cis-mse/2021_os_hw1/tree/32164809에서 확인할 수 있다. 다음 <표 3.1>은 32164809 브랜치에 업로드된 파일들을 클론하여 mysish.out 실행 파일을 생성하는 방법을 나타낸다.

명령어	설명
\$ git clone -b 32164809 git@github.com:mobile-os-dku-cis-mse/2021_os_hw1.git	mobile-os-dku-cis-mse/2021_os_hw1의 32164809 브랜치를 복제한다.
\$ ls	현재 디렉터리에 있는 디렉터리와 파일 목록을 보여준다.
function.c header.h main.c Makefile	
\$ make	main.c와 function.c를 컴파일하고 오브젝트 파일을 링킹하여 mysish.out 실행 파일을 생성한다.
gcc -c main.c	
gcc -c function.c	
gcc -o mysish.out main.o function.o	
\$./mysish.out	mysish.out 프로그램을 실행한다.
\$ make clean	main.o function.o 오브젝트 파일과 mysish.out 실행 파일을 삭제한다.
rm -f main.o function.o mysish.out	

<표 3.1> mysish 프로그램을 실행시키는 방법. 명령어를 입력하였을 때 터미널에 나타나는 결과는 파란색으로 표현되어 있다.

나. 프로그램 분석

mysish 프로그램은 header.h, function.c, 그리고 main.c로 파일이 분할되어 있다. header.h 파일은 mysish 프로그램에서 사용된 헤더 파일, 함수, 그리고 전역 변수가 선언되어 있고 function.c 파일은 사용자 명령을 입력받는 함수, 환경변수를 불러오는 함수, 프로세스를 실행하는 함수 등이 정의되어 있어 사용자 명령이 셸에서 어떠한 과정을 거치는지 구체적으로 확인할 수 있다. main.c 파일은 프롬프트를 출력하고 사용자로부터 명령을 입력받아 불러온 환경변수를 탐색하여 프로세스를 실행하는 function.c 파일의 여러 부분을 통합하여 셸의 전체적인 그림을 살펴볼 수 있다. 한편, mysish 프로그램 분석은 실험 목표를 고려하여 function.c 파일을 중심으로 설명한다. header.h 파일과 function.c 파일의 일부 코드는 주석을 함께 참고하여 읽으면 코드를 이해하는 데 어렵지 않으므로 프로그램 분석에서 제외하였다.

1) function.c

가) printPrompt 함수

printPrompt 함수는 myshish 셸에 프롬프트를 출력하고 사용자 입력을 기다리는 함수이다.

```
void printPrompt(void) {
    char curDir[512];
    char curTime[128];
    time_t t;

    // 현재 디렉터리를 배열에 저장합니다.
    if (getcwd(curDir, 512) == NULL) {
        perror("getcwd error.");
        exit(EXIT_FAILURE);
    }

    time(&t);
    strftime(curTime, 128, "%Y %B %dth %p %l:%M", localtime(&t));..... ①
    printf("\x1b[33m%s \x1b[0m", curTime);
    printf("%s@assam:~", getenv("USER"));
    printf("%s$ ", curDir);
    return;
}
```

① 각색한 프롬프트를 출력

myshish 셸은 프롬프트에서 출발한다. 사용자 입력을 기다리는 myshish 셸의 프롬프트는 일반적인 명령 프롬프트 형식을 따르되, 시간 정보를 추가하여 명령을 입력하는 시간을 확인할 수 있도록 하였다. 시간은 localtime 함수에서 반환된 현재 시각 문자열을 년, 월, 일, 현재 시각 순으로 curTime 배열에 저장한다. curTime 배열에 저장된 문자열은 bash 셸의 프롬프트와 혼동되지 않도록 노란색으로 시간을 강조하여 출력한다. 사용자 이름은 환경변수 USER에 저장된 사용자 이름을 getenv 함수를 사용하여, 반환된 사용자 이름을 문자열로 출력한다. 디렉터리 전체 경로는 getcwd 함수를 사용하여 디렉터리 전체 경로를 curDir 배열에 저장하고 이 배열에 저장된 문자열을 출력한다.

나) readCmd 함수

readCmd 함수는 사용자 입력 명령을 가장 정석인 형태로 정리하고 배열에 저장하는 함수이다.

```
int readCmd(char cmd[], int size) {
    int ch, i = 0;

    // 불필요한 공백을 제거합니다.
    while ((ch = getchar()) != '\n').....①
        if (i < size && (!isspace(ch) || ((i >= 1) && cmd[i - 1] != ' ')))
            cmd[i++] = ch;

    while (i > 0 && cmd[i - 1] == ' ')
        i--;
    // 문자열 가장 마지막에 널 문자를 삽입합니다.
    cmd[i] = '\0';

    // 개행 문자를 처리합니다.
    if (i == 0).....②
        return CONTINUE_SHELL;

    // histCmd 배열에 명령어를 저장합니다.
    strcpy(histCmd[histIndex++], cmd);.....③
    if (histIndex >= MAX_SIZE)
        histIndex = 0;
    return 0;
}
```

① 공백 문자를 제거하여 정석적인 형태로 가공

사용자가 잘못된 키보드 조작으로 인하여 명령어, 옵션, 그리고 인수 사이에 여러 개의 공백 문자를 입력하였을 때, 명령어를 실행하는 데 필요하지 않은 공백 문자를 제거한다. 첫 번째 while 반복문으로 명령어 앞에 입력된 공백 문자와 명령어, 옵션 그리고 인수 사이에 입력된 공백 문자를 제거한다. 두 번째 반복문으로 명령어 뒤에 입력된 공백 문자를 제거한다.

② 개행 문자는 프롬프트를 재출력

프롬프트에 개행 문자(enter)를 입력하는 경우를 생각해보자. 사용자가 명령어를 입력하고 개행을 하면 명령어가 실행된다. 하지만 사용자 어떠한 명령어도 입력하지 않고 개행을 하면 그저 프롬프트가 다시 출력된다. 명령어를 입력하고 개행을 했을 때, 0으로 초기화된 변수 i는 첫 번째 while 반복문을 거쳐 배열에 명령어를 저장하므로 반드시 0보다 큰 값을 가진다. 하지만 명령어를 입력하지 않고 개행을 했을 때, 첫 번째 while 반복문을 거치지 않으므로 변수 i는 0의 값을 가진다. 따라서 변수 i가 0일 때, 함수를 즉시 종료하고 CONTINUE_SHELL을 반환한다.

③ 입력한 명령어를 배열에 기록

개행 문자를 제외한 모든 사용자 명령은 histCmd 배열에 저장된다. histCmd 배열에 저장된 문자열은 사용자가 history 명령어를 입력하였을 때 터미널에 출력된다. strcpy 함수를 사용하여 사용자 명령을 histCmd 배열에 복사하고, 배열의 인덱스로 사용되는 변수 histIndex의 값을 1만큼 증가시킨다. 변수 histIndex의 값이 배열의 크기인 MAX_SIZE보다 크면 값을 0으로 초기화한다.

다) parseCmd 함수

parseCmd 함수는 사용자 입력 명령에서 명령어, 옵션, 인수를 분리하여 배열에 저장하는 함수이다.

```
void parseCmd(char* newCmd[], char cmd[]) {
    int innerLoopIndex = 1;
    char* savePtr = NULL;
    newCmd[0] = strtok_r(cmd, " ", &savePtr);..... ①

    while (1) {
        newCmd[innerLoopIndex] = strtok_r(NULL, " ", &savePtr);
        if (newCmd[innerLoopIndex] == NULL)
            break;
        innerLoopIndex++;
    }
    return;
}
```

① 명령을 명령어, 옵션, 인수로 분리

사용자는 명령어에 옵션과 인수를 추가로 입력하여 구체적인 명령을 내릴 수 있다. 공백 문자로 구분된 명령어, 옵션, 그리고 인수는 strtok_r 함수로 문자열을 분리하여 newCmd 배열에 저장한다. 첫 번째 strtok_r 함수에서 반환된 문자열은 명령어로서 newCmd[0]에 저장된다. 옵션과 인수는 while 반복문에서 두 번째 strtok_r 함수에서 반환된 문자열은 옵션과 인수로서 newCmd[1], newCmd[2] 등에 저장된다.

라) execBuiltIn 함수

execBuilt 함수는 PATH 환경변수에 저장되지 않은 리눅스 내장 명령어를 실행하는 함수이다.

```
int execBuiltIn(char* newCmd[]) {
    // cd 명령어를 실행합니다.
    if (strcmp(newCmd[0], "cd") == 0) {..... ①
        if (newCmd[1] == NULL)
            chdir(getenv("HOME"));
        else
            if (chdir(newCmd[1]) == -1)
                printf("-mysish: %s: %s: No such file or directory\n", newCmd[0], newCmd[1]);
            return CONTINUE_SHELL;
    }

    // history 명령어를 실행합니다.
    if ((strcmp(newCmd[0], "history") == 0)) {..... ②
        for (int i = 0; i < histIndex; i++)
            printf(" %d %s\n", i, histCmd[i]);
        return CONTINUE_SHELL;
    }

    // exit, quit 명령어를 실행합니다.
    if ((strcmp(newCmd[0], "exit") == 0) ||..... ③
        (strcmp(newCmd[0], "quit") == 0)) {
        closeSh();
        exit(EXIT_SUCCESS);
    }
}
(중략)
```

① cd 명령어를 실행

cd(change directory) 명령어는 사용자가 위치한 디렉터리에서 다른 디렉터리로 이동하는 명령어이다. strcmp 함수를 사용하여 사용자가 입력한 명령어 newCmd[0]이 "cd" 문자열과 서로 일치할 때 옵션 문자열 newCmd[1]을 chdir 함수의 인수로 전달한다. 사용자로부터 올바른 디렉터리 경로를 옵션으로 전달 받은 chdir 함수는 디렉터리의 위치를 변경시킨다. 한편, 올바른 디렉터리 경로가 전달되지 않아 chdir 함수에서 -1을 반환하였을 때, "그러한 디렉터리가 없다."라는 문장을 터미널에 출력한다.

② history 명령어를 실행

history 명령어는 사용자가 입력한 명령어를 터미널에 모두 출력하는 명령어이다. strcmp 함수를 사용하여 사용자가 입력한 명령어 newCmd[0]이 "history" 문자열과 서로 일치할 때 for 반복문으로 그동안 histCmd 배열에 저장했던 모든 사용자 명령을 터미널에 출력한다.

③ exit · quit 명령어를 실행

exit 명령어는 mysh 셸을 종료하는 명령어이다. 마찬가지로 strcmp 함수를 사용하여 사용자가 입력한 명령어 newCmd[0]이 "exit" 또는 "quit" 문자열과 서로 일치할 때, 종료 프롬프트를 출력하고 exit 함수로 mysh 셸 프로그램을 종료한다.

마) readPath 함수

readPath 함수는 PATH 환경변수에 저장된 경로를 배열에 저장하는 함수이다.

```
void readPath(char* newPath[], char path[]) {
    int innerLoopIndex = 1;
    char* tempPath;
    char* savePtr = NULL;

    strcpy(path, getenv("PATH"));..... ①
    newPath[0] = strtok_r(path, ":", &savePtr);

    while (1) {
        newPath[innerLoopIndex] = strtok_r(NULL, ":", &savePtr);
        if (newPath[innerLoopIndex] == NULL)
            break;

        innerLoopIndex++;
    }
    return;
}
```

① PATH 환경변수 경로를 배열에 저장

셸 외장 명령어는 PATH 환경변수의 여러 가지 경로에 저장된 프로그램이다. 사용자가 입력한 모든 명령어는 PATH 환경변수에 저장된 프로그램과 비교하여 서로 일치하는지 탐색하는 과정을 거쳐야 한다. mysh 셸은 PATH 환경변수의 여러 가지 경로를 우선순위에 맞추어 배열에 저장하여 사용자가 입력한 명령어 연결할 수 있도록 한다. strtok_r 함수를 사용해 첫 번째 경로를 newPath[0]에 저장한다. 이때 경로들은 콜론 문자(:)로 서로 구분되어 있으므로 콜론 문자를 구분자로 사용한다. 마찬가지로 while 반복문과 strtok_r 함수를 사용하여 남은 경로들도 newPath[1], newPath[2] 등에 순서대로 저장한다. strtok_r 함수에서 구분할 경로가 더 없으면 NULL을 반환하고 while 반복문을 빠져 나와 함수를 종료한다.

바) execObj 함수

execObj 함수는 실행 파일을 실행하는 함수이다.

```
void execObj(char* newCmd[], char** environ) {
    char curPath[MAX_SIZE];

    strcpy(curPath, newCmd[0]);
    if (execve(curPath, newCmd, environ) != -1)..... ①
        exit(EXIT_SUCCESS);
    return;
}
```

① 실행 파일을 실행

사용자가 컴파일하고 링킹한 실행 파일을 자식 프로세스에서 실행하기 위해 execve 함수에 인수로 path, argv[], 그리고 envp[]를 전달해야 한다. 사용자가 위치한 디렉터리에서 ./filename 파일을 실행한다고 가정할 때, newCmd[0] 문자열에 저장된 경로를 curPath 배열에 복사하고 path 인수로 전달한다. argv[] 인수는 사용자가 입력한 명령어, 옵션, 인수가 저장된 newCmd 포인터 배열을 전달하고, envp[] 인수는 extern으로 선언된 environ 전역 변수를 전달한다. 사용자가 위치한 디렉터리에 해당하는 실행 파일이 있을 때, 파일을 실행하고 자식 프로세스를 종료한다.

사) execCmd 함수

execCmd 함수는 PATH 환경변수에 저장된 리눅스 외장 명령어를 실행하는 함수이다.

```
void execCmd(char* newPath[], char* newCmd[], char** environ) {
    char curPath[MAX_SIZE];
    int innerLoopIndex = 0;

    do {
        if (newPath[innerLoopIndex] == NULL) {
            printf("No command '%s' found.\n", newCmd[0]);
            break;
        }
        else {
            strcpy(curPath, newPath[innerLoopIndex]);
            strcat(curPath, "/");
            strcat(curPath, newCmd[0]);
            strcat(curPath, "");
        }
        innerLoopIndex++;
    } while (execve(curPath, newCmd, environ) == -1);..... ①

    exit(EXIT_SUCCESS);
    return;
}
```


① 외장 명령어를 실행

위의 `execObj` 함수와 마찬가지로 `PATH` 환경변수에 저장된 외장 명령어를 자식 프로세스에서 실행하기 위해서 `execve` 함수에 `path`, `argv[]`, 그리고 `envp[]`를 전달해야 한다. `newPath` 포인터 배열에 저장한 `PATH` 환경변수의 여러 가지 경로에 사용자가 입력한 명령어 `newCmd[0]` 문자열을 `strcat` 함수로 연결하여 `curPath` 문자열 만든다. 이 `curPath` 문자열을 `path` 인수로 전달한다. `argv[]` 인수는 사용자가 입력한 명령어, 옵션, 인수가 저장된 `newCmd` 포인터 배열을 전달하고, `envp[]` 인수는 extern으로 선언된 `environ` 전역 변수를 전달한다. `curPath` 배열에 저장된 경로에 사용자가 입력한 명령어를 실행시키는 파일이 없을 때 `execve` 함수가 -1을 반환하므로, 다음 `newPath` 포인터 배열에 저장된 환경변수에 사용자가 입력한 명령어를 `strcat` 함수로 연결하여 두 번째 `curPath` 문자열을 만든다. 위와 같은 작업을 while 반복문으로 반복하다가 `curPath` 배열에 저장된 경로에 사용자가 입력한 명령어를 실행시키는 파일이 있을 때, 파일을 실행시키고 while 반복문을 빠져 나와 자식 프로세스를 종료한다. 이와 반대로, `PATH` 환경변수에 저장된 경로를 모두 탐색하였는데도 명령어를 실행시키는 파일이 없을 때, "그러한 명령이 존재하지 않습니다."라고 터미널에 출력하고 break로 while 반복문을 빠져 나와 자식 프로세스를 종료한다.

4. 문제 및 해결

제4장 문제 및 해결은 `mysish` 프로그램을 개발하는 동안 발생했던 몇 가지 문제와 해결하는 과정을 서술했다. 프로그램 실행 중에 문제가 되었던 부분과 잘못된 결과를 먼저 확인하고 디버깅을 통해 문제를 분석하여 고쳐나가는 과정을 살펴본다.

가. 디버깅

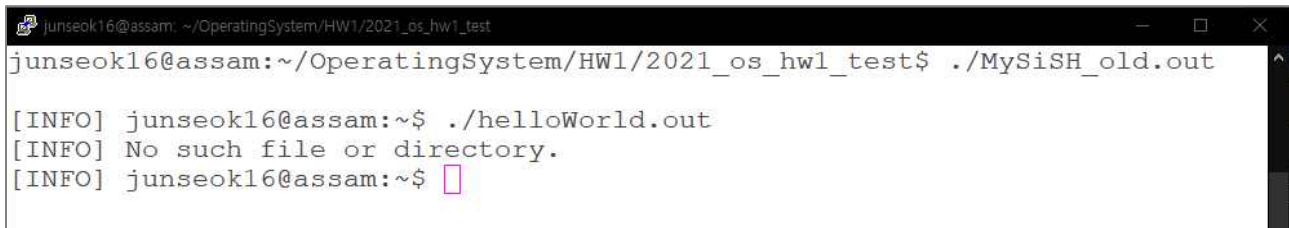
1) 사용자 실행 파일

"Hello, world!"를 출력하는 `./helloWorld.out` 실행 파일을 실행하지 못하는 오류가 발생했다. 이 파일은 리눅스가 제공하는 내외장 명령어가 아닌 사용자가 직접 파일을 컴파일하고 링킹한 실행 파일이었기 때문에 `PATH` 환경변수에 저장된 여러 가지 경로를 전달한 `execv` 함수에서 실행되지 못하였다.

```
int process(void) {
    (중략)
    else if (pid == 0) {
        printf("[INFO] Child process with pid %d has been created.\n", getpid());
        while (execv(newPath, newArgv) == -1) {..... ①
            if (!(tempPath = strtok_r(NULL, ":", &savePathPtr))) {
                printf("[INFO] No such file or directory.\n");
                break;
            }
            strcpy(newPath, tempPath);
            strcat(newPath, "/");
            strcat(newPath, cmd);
            strcat(newPath, "");
        }
        (중략)
    }
```

① PATH 환경변수를 경로로 취하는 `execv` 함수

위의 코드는 수정 전 `MySiSH_old.c` 코드의 일부이다. `process` 함수는 `strcpy` 함수와 `strcat` 함수로 편집된 `PATH` 환경변수의 경로를 `newPath` 문자열에 저장하고 그 배열을 인수로 취할 뿐, 사용자가 입력한 `/` 디렉터리와 실행 파일에 전혀 대응할 수 없는 코드이다. `helloWorld.out` 파일은 `PATH` 환경변수에 존재하지 않으므로 <그림 4.1>에서 그러한 파일이나 디렉터리가 없다는 문장만 출력한다는 것을 알 수 있다.



```

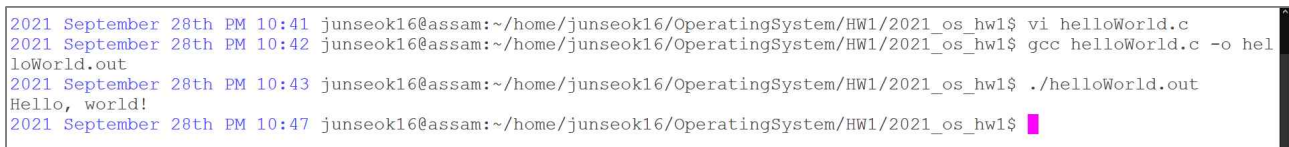
junseok16@assam: ~/OperatingSystem/HW1/2021_os_hwl_test$ ./MySiSH_old.out

[INFO] junseok16@assam:~$ ./helloWorld.out
[INFO] No such file or directory.
[INFO] junseok16@assam:~$ 

```

<그림 4.1> 초기 `MySiSH_old` 프로그램의 `helloWorld.out` 실행 결과. 초기 `mysish` 프로그램은 사용자가 컴파일한 실행 파일에 대응하는 코드가 없다.

사용자가 컴파일하고 링킹한 실행 파일은 `PATH` 환경변수 경로가 아닌 사용자가 위치한 디렉터리의 경로를 `execve` 함수의 `path`로 전달해야 했다. 따라서 제3장 `execObj` 함수에서 사용자가 위치한 디렉터리의 경로를 전달하는 `execve` 함수를 새로 작성하고 남은 인수들에 올바른 값을 각각 전달하여 사용자 파일을 실행시킬 수 있었다.



```

2021 September 28th PM 10:41 junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hwl$ vi helloWorld.c
2021 September 28th PM 10:42 junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hwl$ gcc helloWorld.c -o helloWorld.out
2021 September 28th PM 10:43 junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hwl$ ./helloWorld.out
Hello, world!
2021 September 28th PM 10:47 junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hwl$ 

```

<그림 4.2> 수정한 `mysish` 프로그램의 `helloWorld.out` 실행 결과. 사용자가 컴파일한 `helloWorld.out` 실행 파일이 올바르게 실행되는 것을 확인할 수 있다.

2) history 명령어

`history` 명령어를 입력하였을 때, 사용자가 셸을 실행하는 동안 입력했던 이전의 명령어가 번호 순서대로 출력되는 것을 기대하였지만, 방금 입력한 "history" 명령어만 반복해서 출력되는 오류가 발생하였다.

```

int main(void) {
    char* histCmd[MAX_SIZE];
    int histIndex = 0;
    int* histPtr = &histIndex;

    while(1) {
        char cmd[MAX_SIZE];
        (중략)
    }
}

int readCmd(char cmd[], int size, char* histCmd[], int* histPtr) {
    (중략)
}

```

```

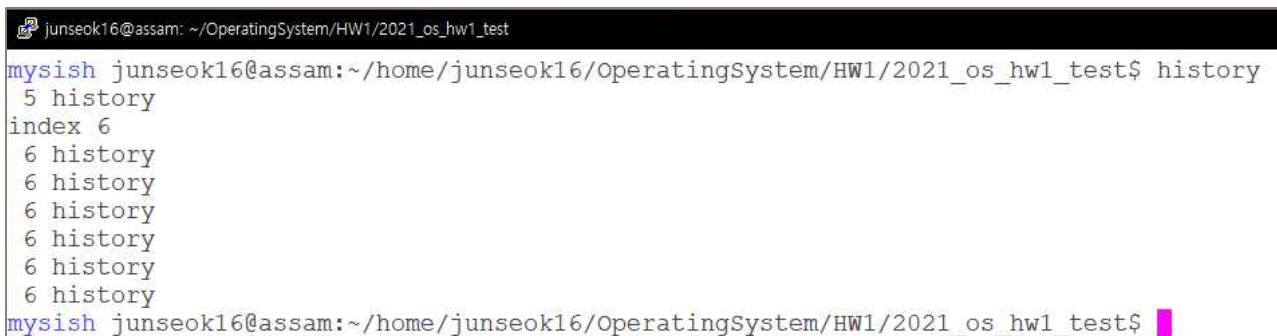
histCmd[*histPtr] = cmd;..... ①
printf(" %d %s\n", *histPtr, histCmd[*histPtr++]);
if (*histPtr >= MAX_SIZE) *histPtr = 0;
return 0;
}

int execCmd(char* newCmd[], char* histCmd[], int* histPtr) {
    if (strcmp(newCmd[0], "history") == 0) {
        printf("index %d\n", *histPtr);
        for (int i = 0; i < *histPtr; i++)
            printf(" %d %s\n", *histPtr, histCmd[i]);
        return CONTINUE_SHELL;
    }
    (종락)
}

```

① 지역 변수의 유효 범위

마찬가지로 위의 코드는 수정 전 MySiSH_old.c 코드의 일부이다. 초기 MySiSH_old 프로그램은 history 명령어를 처리하기 위해 main 함수에 char 포인터 배열 histCmd와 int 포인터를 사용하였다. readCmd 함수에서 histCmd 포인터 배열에 cmd 배열을 저장하고 이것을 execCmd에서 for 반복문을 통해 저장된 cmd 배열을 출력하려고 하였지만, 셸이 while 반복문에서 재시작할 때마다 cmd 배열이 새로 선언되어 histCmd 포인터 배열에 저장된 이전의 cmd 포인터 배열이 사라졌다.



```

junseok16@assam: ~/OperatingSystem/HW1/2021_os_hw1_test
mysish junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hw1_test$ history
5 history
index 6
6 history
6 history
6 history
6 history
6 history
6 history
mysish junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hw1_test$

```

<그림 4.3> 초기 MySiSH_old 프로그램의 history 명령어 실행 결과. 사용자가 입력한 명령어의 목록 대신 방금 입력한 "history" 명령에 해당하는 문자열만 출력되고 있다.

따라서 지역 변수의 유효 범위를 우회하는 방법으로 2차원 histCmd 배열과 histIndex 변수를 전역 변수로 선언하였고 readCmd 함수와 execBuiltIn 함수에 사용되었던 복잡한 인수들을 없앴다. 제3장에서 살펴본 것처럼 readCmd 함수에서는 사용자가 입력한 명령을 포인터가 아닌 문자열로 저장하였고 이것을 execBuiltIn 함수에서 for 반복문으로 출력하도록 하였다.



```

2021 September 28th PM 10:33 junseok16@assam:~/home/junseok16/OperatingSystem/HW1$ history
0 cd .
1 cd ..
2 cd
3 cd OperatingSystem/HW1
4 ls
5 clear
6 history
2021 September 28th PM 10:33 junseok16@assam:~/home/junseok16/OperatingSystem/HW1$

```

<그림 4.4> 수정한 mysish 프로그램의 history 실행 결과. 사용자가 입력한 명령어가 순서대로 올바르게 출력되는 것을 확인할 수 있다.

5. 결과 및 분석

가. Makefile 실행

1) make, make clean

```
junseok16@assam: ~/OperatingSystem/HW1/2021_os_hw1/Source
junseok16@assam:~/OperatingSystem/HW1/2021_os_hw1/Source$ ls
function.c header.h main.c Makefile
junseok16@assam:~/OperatingSystem/HW1/2021_os_hw1/Source$ make
gcc -c main.c
gcc -c function.c
gcc -o mysish.out main.o function.o
junseok16@assam:~/OperatingSystem/HW1/2021_os_hw1/Source$ ls
function.c function.o header.h main.c main.o Makefile mysish.out
junseok16@assam:~/OperatingSystem/HW1/2021_os_hw1/Source$ make clean
rm -f main.o function.o mysish.out
junseok16@assam:~/OperatingSystem/HW1/2021_os_hw1/Source$ ls
function.c header.h main.c Makefile
junseok16@assam:~/OperatingSystem/HW1/2021_os_hw1/Source$
```

<그림 5.1> bash 셸에서 make, make clean을 입력한 모습. 터미널에 ls, make, ls, make clean, ls 순으로 입력하였다.

나. 리눅스 내외부 명령어 실행

1) 내부 명령어

가) cd

```
junseok16@assam: ~/OperatingSystem/HW1/2021_os_hw1/Source
junseok16@assam:~/OperatingSystem/HW1/2021_os_hw1/Source$ ./mysish.out

  MYSISH
  by junseok16

  Now starting...

2021 September 28th PM 10:31 junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hw1/Source$ cd .
2021 September 28th PM 10:31 junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hw1/Source$ cd ..
2021 September 28th PM 10:31 junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hw1$ cd
2021 September 28th PM 10:31 junseok16@assam:~/home/junseok16$ cd OperatingSystem/HW1
2021 September 28th PM 10:32 junseok16@assam:~/home/junseok16/OperatingSystem/HW1$ ls
2021 os_hw1  2021 os_hw1 test
2021 September 28th PM 10:32 junseok16@assam:~/home/junseok16/OperatingSystem/HW1$
```

<그림 5.2> mysish 셸에서 cd 명령어를 입력한 모습. 터미널에 cd ., cd .., cd, cd OperatingSystem/HW1을 입력하였다.

나) history

```
junseok16@assam: ~/OperatingSystem/HW1/2021_os_hw1/Source
2021 September 28th PM 10:33 junseok16@assam:~/home/junseok16/OperatingSystem/HW1$ history
0 cd .
1 cd ..
2 cd
3 cd OperatingSystem/HW1
4 ls
5 clear
6 history
2021 September 28th PM 10:33 junseok16@assam:~/home/junseok16/OperatingSystem/HW1$
```

<그림 5.3> mysish 셸에서 history 명령어를 입력한 모습. 터미널에 history를 입력하였다.

2) 외부 명령어

가) ls

```

junseok16@assam: ~/OperatingSystem/HW1/2021_os_hw1/Source
2021 September 28th PM 10:35 junseok16@assam:~/home/junseok16/OperatingSystem/HW1$ ls
2021_os_hw1 2021_os_hw1_test
2021 September 28th PM 10:35 junseok16@assam:~/home/junseok16/OperatingSystem/HW1$ ls -F
2021 os_hw1/ 2021 os_hw1_test/
2021 September 28th PM 10:35 junseok16@assam:~/home/junseok16/OperatingSystem/HW1$ ls -a
. . . 2021_os_hw1 2021_os_hw1_test
2021 September 28th PM 10:35 junseok16@assam:~/home/junseok16/OperatingSystem/HW1$ ls -al
total 16
drwxrwxr-x 4 junseok16 junseok16 4096 9월 19 10:42 .
drwxrwxr-x 4 junseok16 junseok16 4096 9월 16 20:10 ..
drwxrwxr-x 5 junseok16 junseok16 4096 9월 26 23:13 2021_os_hw1
drwxrwxr-x 2 junseok16 junseok16 4096 9월 26 10:24 2021 os_hw1_test
2021 September 28th PM 10:35 junseok16@assam:~/home/junseok16/OperatingSystem/HW1$ ls -a -l
total 16
drwxrwxr-x 4 junseok16 junseok16 4096 9월 19 10:42 .
drwxrwxr-x 4 junseok16 junseok16 4096 9월 16 20:10 ..
drwxrwxr-x 5 junseok16 junseok16 4096 9월 26 23:13 2021_os_hw1
drwxrwxr-x 2 junseok16 junseok16 4096 9월 26 10:24 2021_os_hw1_test
2021 September 28th PM 10:35 junseok16@assam:~/home/junseok16/OperatingSystem/HW1$

```

<그림 5.4> myish 셸에서 ls 명령어를 입력한 모습. 터미널에 ls, ls -F, ls -a, ls -al, ls -a -l을 입력하였다.

나) man

```

junseok16@assam: ~/OperatingSystem/HW1/2021_os_hw1/Source
2021 September 28th PM 10:55 junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hw1$ man -k passwd
checkPasswdAccess (3) - query the SELinux policy database in the kernel
chgrp (8) - update group passwords in batch mode
chpasswd (8) - update passwords in batch mode
fgetpwent_r (3) - get passwd file entry reentrantly
getpwent_r (3) - get passwd file entry reentrantly
gpasswd (1) - administer /etc/group and /etc/gshadow
grub-mkpasswd-pbkdf2 (1) - generate hashed password for GRUB
htpasswd (1) - Manage user files for basic authentication
mkpasswd (1) - Overfatured front end to crypt(3)
mksmbpasswd (8) - formats a /etc/passwd entry for a smbpasswd file
pam_localuser (8) - require users to be listed in /etc/passwd
passwd (1) - change user password
passwd (1ssl) - compute password hashes
passwd (5) - the password file
passwd2des (3) - RFS password encryption
selinux_check_passwd_access (3) - query the SELinux policy database in the kernel
smbpasswd (5) - The Samba encrypted password file
smbpasswd (8) - change a user's SMB password
SSL_CTX_set_default_passwd_cb (3ssl) - set passwd callback for encrypted PEM file handling
SSL_CTX_set_default_passwd_cb_userdata (3ssl) - set passwd callback for encrypted PEM file handling
update-passwd (8) - safely update /etc/passwd, /etc/shadow and /etc/group
2021 September 28th PM 10:55 junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hw1$

```

<그림 5.6> myish 셸에서 man 명령어를 입력한 모습. 터미널에 man -k passwd를 입력하였다.

다) vi, gcc, 실행 파일

```

junseok16@assam: ~/OperatingSystem/HW1/2021_os_hw1/Source
2021 September 28th PM 10:41 junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hw1$ vi helloWorld.c

1 #include <stdio.h>
2
3 int main(void) {
4     printf("Hello, world!\n");
5     return 0;
6 }

2021 September 28th PM 10:41 junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hw1$ vi helloWorld.c
2021 September 28th PM 10:42 junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hw1$ gcc helloWorld.c -o helloWorld.out
2021 September 28th PM 10:43 junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hw1$ ./helloWorld.out
Hello, world!
2021 September 28th PM 10:47 junseok16@assam:~/home/junseok16/OperatingSystem/HW1/2021_os_hw1$

```

<그림 5.7> myish 셸에서 vi, gcc 명령어를 입력한 모습. 터미널에 vi helloWorld.c, gcc helloWorld.c -o helloWorld.out, ./helloWorld.out을 입력하였다.

6. 고찰 및 느낀 점

가. 고찰

우리는 학습한 내용을 바탕으로 이번 운영체제 첫 번째 과제 `mysish` 셸 프로그램을 개발하여 `bash` 셸이 어떻게 동작하는지 자세히 알게 되었다. 우리는 이 실험에서 직접 작성한 몇 가지 내장 명령어와 외장 명령어를 `mysish` 셸에서 실행시키고 그 올바른 결과를 얻어냈다. 이것으로 미루어보았을 때, 셸은 사용자가 명령을 내렸을 때 시스템 호출을 통해 자식 프로세스를 생성하고 부모 프로세스에 돌아가는 일련의 과정을 반복하며, 이 과정에서 사용자와 컴퓨터가 상호작용하는 중간 역할을 한다는 이론을 결론적으로 확인할 수 있었다. 한편, 문자열을 저장할 수 있는 선형 연결 리스트 등의 자료구조를 도입하여 사용자가 입력한 명령어를 조금 더 세밀하고 다룰 수 있게 하였다면 메타 문자를 사용하는 다중 명령어를 구현하거나, 전역 변수를 사용하지 않은 `history` 명령어를 구현하여 조금 더 다채로운 실험 결과를 도출해낼 수 있을 것으로 생각한다. 또한, `PATH` 환경변수에서 명령어를 탐색하는데 선형 검색 알고리즘보다 이진 검색 알고리즘을 도입하였다면 프로그램 성능을 조금 더 높일 수 있었을 것으로 예상된다.

나. 느낀 점

이번 운영체제 과제는 첫 번째 과제였지만 시작부터 많은 것을 배우고 느꼈다. 그동안 여러 개발 언어로 프로그램을 작성할 때, 주로 Visual Studio 2019, Visual Studio Code, Android Studio 같은 개발도구를 사용하였다. 이렇게 윈도우 운영체제에서 GUI를 제공하는 개발도구를 사용하다가 CLI로 상호작용하는 Assam 서버의 우분투 리눅스에서 작업하는 것은 다소 생소한 경험이었다. 인터넷에 올라온 우분투 리눅스에 대한 정보는 단편적이고 얇은 것들이 너무 많아 결국 '사용자와 프로그래머를 위한 UNIX 활용' 서적을 살펴보고 파일, 디렉터리, vi 편집기, gcc 등 리눅스에서 제공하는 기본적인 명령어의 사용 방법을 익힐 수 있었다. 한편, 이번 과제는 셸 코드를 작성하기 전에, 저번 컴퓨터구조와 모바일프로세서 프로젝트에서 어설프게 하고 넘어갔던 코드 설계에 신경을 많이 썼다. 처음 셸을 디자인할 때 프로세스의 시작과 종료에 초점을 맞추어 셸이 생명 주기를 그림으로 그렸고, 각 순서도의 부분마다 어떠한 행동을 취해야 하는지 고민하는 시간을 가진 다음, 이것을 코드로 옮기려고 하였다. 특히, 코드를 작성할 때는 작성한 설계를 참고하여 흐름을 파악하려고 노력했고, 특히 사용자 명령과 `PATH` 환경변수를 배열로 담아내고 이것을 시스템 호출 함수에 올바르게 전달하는 데 집중하였다. 결과적으로 셸의 기본적인 모습을 갖춘 프로그램을 만들 수 있었다. 셸의 다양한 기능을 프로그램으로 개발하는데 시간이 조금 부족하여 더 깊은 연구를 하지 못한 것이 아쉽지만 기초적인 운영체제와 프로세스의 개념을 다지는 기회가 되어 기쁘다.

7. 참고 문헌

구현회.(2018) 그림으로 배우는 구조와 원리 운영체제 개정 3판. 한빛미디어
 권기현, 오승준.(2002) 사용자와 프로그래머를 위한 UNIX 활용 개정판. 사이텍미디어
 "IBM Documentation." IBM i 7.3. 2021-09-27 접속, <https://www.ibm.com/docs/ko/i/7.3>.