Operating Systems

& Adv. Mobile project HW1

: simple Shell (SiSH)

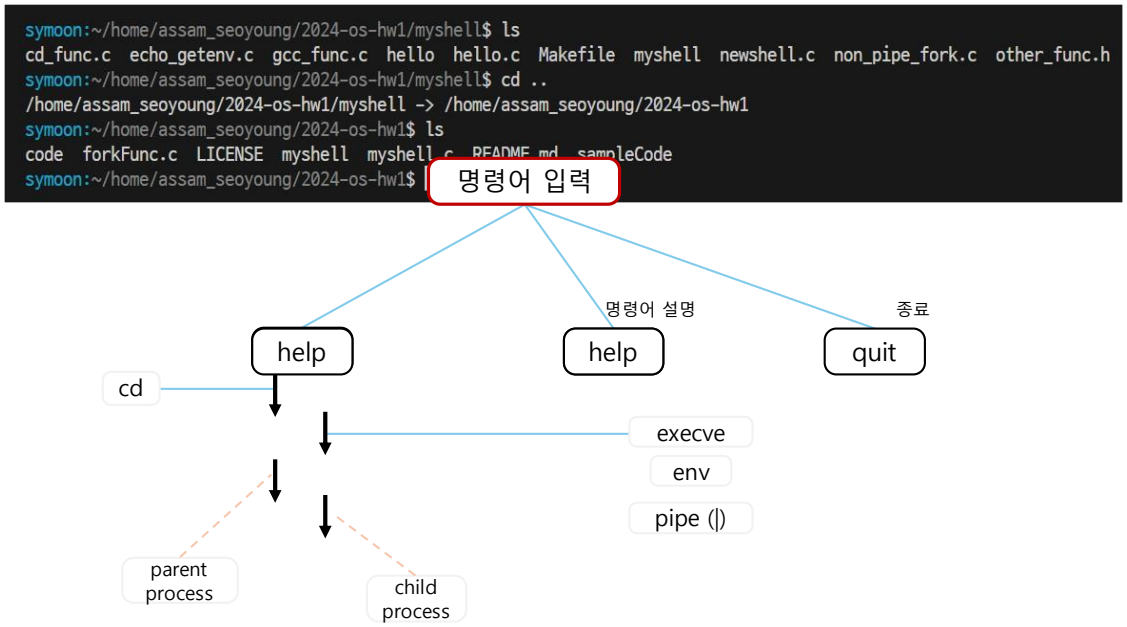
모바일시스템공학과 32217259 문서영

[목차]

1. 개요

이번 운영체제 첫 번째 과제는 사용자가 명령어를 입력하여 실행할 수 있는 **쉘(Shell) 프로그램**을 작성하는 것입니다. 쉘은 **운영체제의 커널과 사용자 사이의 인터페이스** 역할을 하며, 사용자가 명령을 입력하면 이를 처리하고 실행 결과를 출력하는 기능을 제공합니다.

과제에서 쉘의 기본적인 동작을 구현하기 위해 **fork() 함수**를 사용하여 쉘 자체가 종료되지 않도록 하였으며, **execve() 함수**를 사용하여 사용자 명령어를 시스템 호출 방식으로 직접 실행하였습니다. 이 과정에서 **환경 변수를 extern char* 변수를 사용**하여 각 명령어가 필요한 실행 환경을 제공하였습니다. 또한 **getenv() 함수**를 사용하여 환경변수를 직접 확인할 수 있도록 하였습니다. 이러한 접근 방식은 쉘이 시스템 명령어를 처리하고 독립적으로 실행할 수 있도록 합니다.



2. 배경지식

1. shell 이란?

shell 은 리눅스 커널과 사용자를 연결해주는 인터페이스로, 사용자의 명령을 받아 해석한 후 프로그램을 실행시키는 역할을 합니다. 주요 기능은 명령어 해석 기능, 프로그래밍 기능, 사용자 환경 설정 기능을 갖고 있습니다. 기본적으로 리눅스는 /bin/bash 쉘을 사용합니다. 자신이 어떤 쉘을 사용하는지 궁금하다면 "echo $SHELL"을 입력하면 확인할 수 있습니다. 이런 쉘은 터미널 창에서 실행되는데, 터미널 하나를 프로세스라고 말합니다.

운영체제 별로 shell 의 종류가 다른데, 윈도우에서는 cmd(command shell), PowerShell 을 사용하고, 리눅스에서는 Bourne Shell, C shell, TC shell, Korn shell, Bourne Again shell, Z shell 이 있습니다.

Shell 의 주요 기능 (by ChatGPT)

1. **명령어 해석**: 사용자가 입력한 명령어를 해석하고 적절한 작업을 수행할 수 있도록 해야 합니다.
2. **프로세스 관리**: 명령어를 실행하고, 프로세스를 생성하며, PID(프로세스 ID)를 관리하는 기능이 필요합니다.
3. **입출력 리다이렉션**: >와 <를 사용하여 파일로의 출력 및 파일에서의 입력을 처리할 수 있어야 합니다.
4. **파이프**: 여러 명령어를 연결하여 데이터 흐름을 처리하는 기능(|)을 구현해야 합니다.
5. **환경 변수 관리**: 환경 변수를 설정하고 조회할 수 있는 기능이 필요합니다.
6. **배시 스크립트 지원**: 쉘 스크립트를 작성하고 실행할 수 있도록 지원하는 기능.
7. **명령어 기록**: 사용자가 입력한 명령어를 기록하고 이전 명령어를 쉽게 재사용할 수 있는 기능.
8. **자동 완성**: 사용자가 명령어를 입력할 때 자동으로 제안해주는 기능.

2. 개념 설명

fork() 함수

fork 함수는 fork 로 호출한 프로세스(컴퓨터에서 연속적으로 실행되고 있는 프로그램, 운영체제에서 자원을 할당 받아 실행하는 작업의 단위)를 복제하는 기능을 갖습니다. 이때 부모 프로세스, 자식 프로세스가 있는데, 원래 진행하던 프로세스를 부모 프로세스, 복제된 프로세스를 자식 프로세스라고 부릅니다. fork() 함수는 프로세스 id(pid)를 반환하게 되는데, 이때 부모 프로세스에서는 자식 pid 가 반환되고, 자식 프로세스에서는 0 이 반환됩니다. 만약 fork() 함수 실행이 실패하면 -1 을 반환합니다.

즉, 명령어 실행은 터미널 정보를 기준으로 백그라운드에 자식 프로세스가 fork 되어 명령어를 실행합니다.

fork 사용 이유 (by ChatGPT)

1. **병렬 처리**: 여러 작업을 동시에 처리할 수 있게 해 주어, 멀티태스킹 환경을 지원합니다. 예를 들어, 웹 서버는 각 클라이언트 요청을 처리하기 위해 새로운 프로세스를 생성할 수 있습니다.
2. **독립성**: 자식 프로세스는 부모와 독립적으로 실행되며, 서로 다른 메모리 공간을 가집니다. 이는 안정성과 보안성을 높이는 데 도움을 줍니다.
3. **프로세스 간 통신**: 자식 프로세스는 부모 프로세스와 데이터를 교환하거나 통신할 수 있어, 복잡한 작업을 나누어 처리할 수 있습니다.
4. **리소스 관리**: 리눅스의 유닉스 계열 시스템에서는 fork()와 함께 exec()를 사용하여 다른 프로그램을 실행할 수 있습니다. 이는 기존 프로세스의 환경을 유지하면서 새로운 프로그램을 실행할 수 있게 해줍니다.

pipe(파이프): "|"

파이프는 두 프로세스 간의 통신을 위해서 동작합니다.

예를 들어 "cat Makefile | grep shell"와 같이 파이프로 연결된 명령어들은 독립적인 주소공간을 갖는 프로세스이므로, 메모리로 통신이 불가능합니다. 그러므로 기본적으로 어느 한 프로세스가 가지고 있는 정보를 다른 프로세스에서 알 수가 없습니다(cat Makefile 과 grep shell 은 서로 가지고 있는 정보를 확인할 수 없음). 이때 프로세스 간 정보를 전달하기 위해서 IPC 방법을 사용하는데 shell 에서는 파일을 이용하거나 위의 명령어처럼 파이프를 이용할 수 있습니다.

pipe() 함수의 원형을 보면 int pipe(int pipefd[2])와 같이 생겼습니다. 프로세스가 파이프 객체 생성을 요구하면 커널은 프로세스가 접근할 수 있는 메모리 공간을 할당(버퍼)하고 그 버퍼에 접근할 수 있는 두 개의 파일 디스크립터(pipefd[2] → pipefd[0]: 읽기 영역, pipefd[1]: 쓰기 영역, 읽기/쓰기 서로 혼용 불가능)를 프로세스에게 알려줍니다.

이 때 프로세스끼리 통신을 하기 위해선 fork 를 사용해야 하는데 부모 프로세스에서 파이프가 생성되고(디스크립터 중 쓰기 영역 닫음) 자식 프로세스에서 파이프에 접근할 수 있는 파일 디스크립터가 복사됩니다(디스크립터 중 읽기 영역을 닫음).

PID (프로세스 ID)

PID 는 운영체제에서 프로세스를 식별하기 위해 프로세스에 부여하는 번호를 의미합니다. 여기서 프로세스란 실행 중인 프로그램을 말하며, 특정 시점에서 유일한 값을 갖고 있습니다. 부호형 16bit 를 사용했던 오래된 유닉스 시스템과의 호환성 때문에 보통 최대 PID 값은 327268 입니다.

Signal Handler(시그널 핸들러): ctrl + c

Signal Handler 는 프로세스가 시그널을 받았을 때, 그 시그널을 처리하는 함수입니다. 시그널은 통신 방법 중 하나로, 어떠한 이벤트가 발생했을 때 그 이벤트의 종류를 시그널 신호로 보내지는 것입니다. 만약 "ctrl + C"라는 이벤트가 발생했을 때, SIGINT 라는 시그널이 보내지고, 이를 프로그램에서 처리할 수 있습니다.

C 언어에서는 signal 함수를 사용하여 시그널 핸들러를 등록할 수 있습니다. signal 함수는 두 개의 인자를 받는데, 그 중 첫 번째 인자는 처리하고자 하는 시그널 번호(SIGINT, SIGKILL 등)를 받습니다. 그리고 두 번째 인자는 해당 시그널이 발생했을 때 호출될 함수(시그널 핸들러)를 지정합니다.

exec 함수들

exec 계열 함수는 현재 실행되고 있는 프로세스를 다른 프로세스로 대신하여 새로운 프로세스를 실행하는 함수입니다. 만약 ls 명령어를 실행하기 위해 exec 계열 함수를 호출하면, 현재 쉘 프로세스는 ls 프로그램으로 교체되고, ls 명령어의 코드가 실행됩니다. Exec 가 성공했을 경우, 쉘은 더 이상 실행되지 않고 ls 의 실행 흐름이 시작됩니다. 하지만 프로세스 ID 가 바뀌진 않고 그대로 유지됩니다.

그 중에서 execve 랑 execvp 차이를 찾아보았습니다. 둘 다 현재 프로세스를 새로운 프로그램으로 교체하는 데 사용됩니다. 하지만 execve 함수는 int execve(const char *pathname, char *const argv[], char *const envp[])의 형태로, 실행할 프로그램의 경로가 필요하고 이 지정된 경로에서 프로그램을 실행합니다. 또한 환경 변수를 직접 설정할 수 있습니다. 이 함수는 exec 계열의 함수 중 가장 낮은 수준의 함수로 다른 exec 함수들은 execve 함수를 기반으로 추가적인 편의성을 가진 기능을 더하여 만들어지게 됩니다. 또한 execvp 함수는 int execvp(const char *file, char *const argv[])의 형태를 갖고 있으며, 실행할 프로그램의 전체 경로는 몰라도 괜찮습니다. 실행할 프로그램의 이름을 안다면 PATH 환경 변수를 사용하여 직접 검색해줍니다. execve 함수보다 편리하지만, 환경변수를 직접 지정할 수 없고 부모 프로세스의 환경 변수를 상속받습니다.

3. 코드 설명(작성 과정 및 오류 설명)

Shell 의 기본 틀은 ChatGPT 의 도움을 받아 작성하였습니다. 하지만 실행하였을 때, 모든 명령어가 작동하지는 않았고, 그에 대한 명령어는 새로 만들었습니다.

[기본 틀에서의 추가 작업]

- cd

cd 명령어가 셸에서 제대로 작동하지 않는 이유는 주로 프로세스의 작업 디렉터리 변경 방식과 관련이 있습니다. cd 명령어는 현재 프로세스의 작업 디렉터리를 변경하지만, 자식 프로세스에서 이를 호출할 경우 부모 프로세스(즉, 셸)의 작업 디렉터리는 변경되지 않기 때문입니다.

그렇기에 cd_command라는 함수를 만들어서 cd를 직접 생성하였습니다. chdir() 함수를 사용하여 디렉토리가 바뀌는 것을 확인할 수 있었으나 셸에서 적용이 되지는 않았습니다.

```
assam_seoyoung@assam:~/2024-os-hw1$ gcc newshell.c
assam_seoyoung@assam:~/2024-os-hw1$ ./a.out
symoon$ ls
a.out  code  forkFunc.c  hello  hello.c  LICENSE  myshell.c  newshell.c  README.md  sampleCode
symoon$ cd code
/home/assam_seoyoung/2024-os-hw1 -> /home/assam_seoyoung/2024-os-hw1/code
symoon$ ls
a.out  code  forkFunc.c  hello  hello.c  LICENSE  myshell.c  newshell.c  README.md  sampleCode
symoon$
```

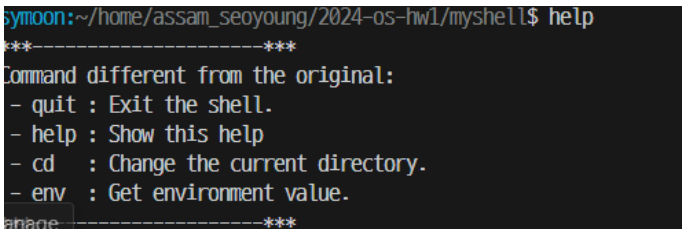
(↑ 여전히 같은 디렉토리에 위치하는 걸 확인할 수 있음)

작동이 되지 않았던 이유는 자식 프로세스에서 cd 명령어를 수행했기 때문이었습니다. fork로 프로세스 종류를 판별하기 전에 main 함수에서 cd 명령어를 직접 수행하니 동작이 잘 되었습니다.

```
assam_seoyoung@assam:~/2024-os-hw1/myshell$ ./myshell
symoon$ ls
cd_func.c  cd_func.h  Makefile  myshell  newshell.c
symoon$ cd ..
/home/assam_seoyoung/2024-os-hw1/myshell -> /home/assam_seoyoung/2024-os-hw1
symoon$ ls
code  forkFunc.c  hello  hello.c  LICENSE  myshell  myshell.c  README.md  sampleCode
symoon$ cd myshell
/home/assam_seoyoung/2024-os-hw1 -> /home/assam_seoyoung/2024-os-hw1/myshell
symoon$ ls
cd_func.c  cd_func.h  Makefile  myshell  newshell.c
symoon$ cd ~
/home/assam_seoyoung/2024-os-hw1/myshell -> /home/assam_seoyoung
```

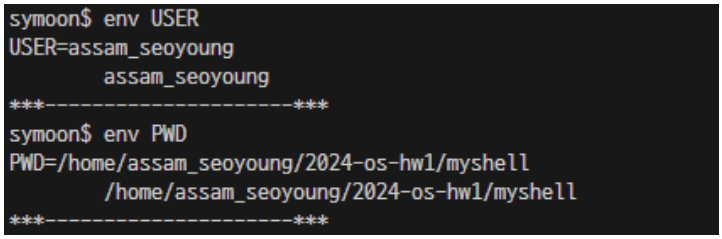
- help 작성하기

help 명령어를 작성함으로써 지정되지 않은 명령어의 사용법을 작성하였습니다.



- 환경 변수

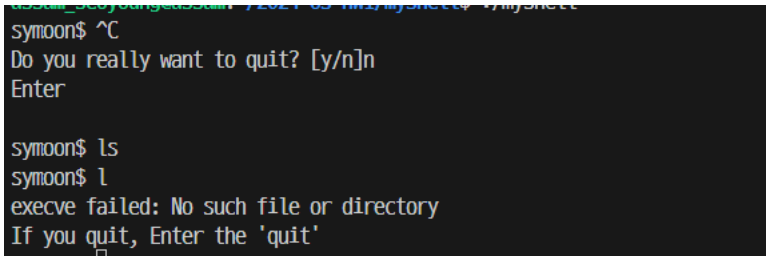
getenv.c 함수를 살짝만 바꿔 자식 프로세스에서 **env {환경변수}**를 입력하면 출력하도록 하였습니다.



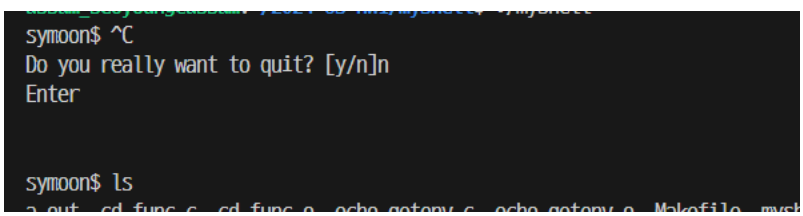
- ctrl + C 누를 시, 강제 종료

Signal 함수를 사용하여 ctrl+c를 눌렀을 때, 종료할 지 선택하는 문구를 넣었습니다. 만약 y를 입력하면 종료되고, 그렇지 않으면 종료되지 않도록 하였습니다.

그런데 n을 입력하고 shell로 바로 돌아오는 게 아니라 버퍼가 남아있었기에 ChatGPT의 도움을 받아 shell로 돌아오고 명령어를 작성할 때 문제가 되는 경우는 해결하였습니다.



하지만 enter를 2번 입력하여야 다시 shell로 빠져나오는 상황은 계속되었습니다.



다시 수정하여 quit에 대해 묻지 말고, enter하면 바로 ^C에서 빠져나오도록 하였습니다.

```
symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$ ^C
If you want to quit, Enter 'quit'.

symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$ ls
cd_func.c  cd_func.o  echo_getenv.c  echo_getenv.o  gcc_func.c  hello.c  hello.s  Makefi
symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$ ^C
If you want to quit, Enter 'quit'.

symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$
```

- exec 함수들 (feat. gcc 구현 오류)

execvp 함수를 사용하였다가, 다시 execve 함수로 쉘을 구현하였습니다. execve 함수에서는 execvp와 달리 환경변수를 지정해주어야 했습니다. ChatGPT의 도움을 받아 환경변수를 직접 작성하였고, ls나 pwd, clear와 같은 작업은 잘 동작하였으나, gcc를 실행하였을 때 cc1이 없다는 오류가 나면서 실행이 되지 않았습니다.

```c
// gcc와 cc1의 경로 설정
char *gcc_path = "/usr/lib"; // gcc가 설치된 경로
char *cc1_path = "/usr/lib/gcc/x86_64-linux-gnu/11"; // cc1이 있는 경로

char *new_path = malloc(strlen(path) + strlen(gcc_path) + strlen(cc1_path) + 3); // gcc_path의 길이도 포함
// sprintf(new_path, ".:%s", path);
sprintf(new_path, ".:%s:%s:%s", gcc_path, cc1_path, path); // gcc_path를 포함한 새로운 경로 생성
// sprintf(gcc_path, ".:%s", path);
char *dirToken = strtok(new_path, ":"); // 첫 번째 디렉토리 경로
char *fullPath = NULL;
char *env[] = {"TERM=xterm", "LANG=en_US.UTF-8", "PATH=/usr/lib/gcc/x86_64-linux-gnu/11", NULL}; // 환경 변

while (dirToken != NULL) {
    fullPath = malloc(strlen(dirToken) + strlen(args[0]) + 2); // 경로 + 명령어 + '/'
    sprintf(fullPath, "%s/%s", dirToken, args[0]);
```

다른 친구의 도움을 받아 extern **char environ의 변수를 사용하면 변수 자체에서 환경변수를 가져올 수 있다는 걸 알았습니다. 그에 맞게 수정을 하니 잘 동작되는 모습을 볼 수 있었습니다.

```
symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$ gcc hello.c
symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$ ls
a.out  cd_func.c  cd_func.o  echo_getenv.c  echo_getenv.o  gcc_func.c  hello.c  hello.s  Makefile  myshell  newshe
symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$ ./a.out
Hello World! I'm symoon!
symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$
```

environ 변수는 extern 변수이기 때문에 외부 라이브러리를 사용하여 현재 프로세스의 환경 변수 목록을 가져오는 배열을 의미합니다.

- 파이프 구현

파이프를 구현하기 위해서 "|"가 있는지 확인하고 없으면 pipe가 없는 함수의 fork함수(fork_command)로 가도록 하였습니다.

```
while(pipe_token != NULL){
    // printf("cmd: %s\n", pipe_token);
    cmds[num_pipe++] = pipe_token;
    pipe_token = strtok(NULL, "|"); // 다음 명령어
}
cmds[num_pipe] = NULL;

int fd[2 * (num_pipe - 1)]; // pipe 위한 디스크립트 배열

// 파이프 생성
for(int i=0;i<num_pipe-1;i++){
    if(pipe(fd + i * 2) == -1){ // 각 프로세스마다 파이프 생성
        perror("pipe");
        exit(1);
    }
}

for(int i=0;i<num_pipe;i++){
    pid_t pid = fork();
```

입력 받은 명령어를 '|' 기준으로 잘랐고, 새로운 버퍼에 넣었습니다. 처리해야 할 명령어 개수의두 배만큼의 파일 디스크립터를 생성해주었습니다. 명령어 개수만큼 돌면서 fork를 처리하도록 하였고, pipe가 없는 fork와 겹치는 부분이기 때문에 execve_command 함수로 빼서 자식 프로세스의 처리를 분리하였습니다.

```
symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$ ls | grep hello
hello
hello.c
symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$ ls
cd_func.c  echo_getenv.c  gcc_func.c  hello.c   myshell     newshell.o      non_pipe_fork.o  pipe_fork.c  test
cd_func.o  echo_getenv.o  hello       Makefile  newshell.c  non_pipe_fork.c  other_func.h     pipe_fork.o  test.c
```

위의 결과처럼 pipe가 잘 동작하는 것을 확인할 수 있었습니다.

4. 출력 결과

출력 방법

```
assam_seoyoung@assam:~/2024-os-hw1/myshell$ make
gcc -Wall -c -o cd_func.o cd_func.c
gcc -Wall -c -o newshell.o newshell.c
gcc -Wall -c -o echo_getenv.o echo_getenv.c
gcc -Wall -c -o non_pipe_fork.o non_pipe_fork.c
gcc -Wall -c -o pipe_fork.o pipe_fork.c
gcc -o myshell cd_func.o newshell.o echo_getenv.o non_pipe_fork.o pipe_fork.o
assam_seoyoung@assam:~/2024-os-hw1/myshell$ make clean
rm -f cd_func.o newshell.o echo_getenv.o non_pipe_fork.o pipe_fork.o
assam_seoyoung@assam:~/2024-os-hw1/myshell$ l
cd_func.c  echo_getenv.c  gcc_func.c  hello*  hello.c  Makefile  myshell*  newshell.c  non_pipe_fork.c  other_func.h  pipe_fork.c  test*  test
assam_seoyoung@assam:~/2024-os-hw1/myshell$ ./myshell
symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$
```

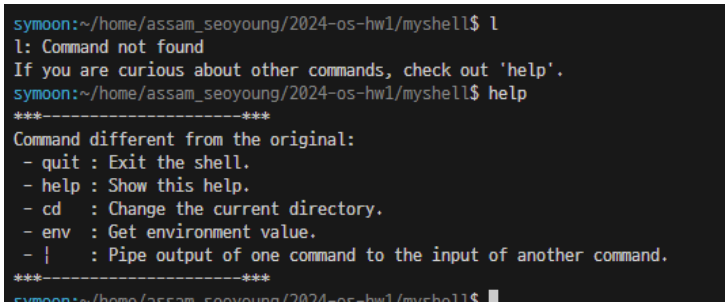
Makefile을 사용하여 myshell 파일을 실행해주었습니다. Make를 사용해 실행파일을 생성하고, make clean으로 사용하지 않는 파일들을 삭제해주었습니다. (make clean은 입력하지 않아도 코드는 제대로 작동함)

```
symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$ ls
cd_func.c  echo_getenv.c  gcc_func.c  hello  hello.c  Makefile  myshell  newshell.c  non_pipe_fork.c  other_func.h  pipe_fork.c  test
symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$ pwd
/home/assam_seoyoung/2024-os-hw1/myshell
symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$ df -h
Filesystem      Size  Used Avail Use% Mounted on
tmpfs           4.8G  3.8M  4.8G   1% /run
/dev/sda2       422G  337G   65G  84% /
tmpfs            24G     0   24G   0% /dev/shm
tmpfs           5.0M  4.0K  5.0M   1% /run/lock
/dev/sda1       511M  7.4M  504M   2% /boot/efi
/dev/sdb1       470G  266G  180G  60% /home3
/dev/sdd1       470G  404G   42G  91% /home4
/dev/sdc1       470G  207G  239G  47% /home2
/dev/sde1       3.6T  1.2T  2.3T  35% /home5
tmpfs           4.8G   92K  4.8G   1% /run/user/1234
tmpfs           4.8G   92K  4.8G   1% /run/user/1477
```

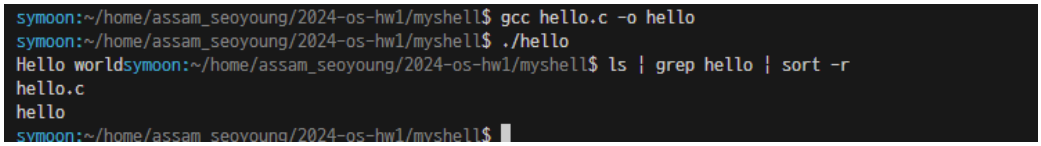
[직접 명령어를 구현하지 않아도 되는 명령어 예시: clear 또한 잘 구현됨]

```
symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$ env HOME
HOME=/home/assam_seoyoung
        /home/assam_seoyoung
***----------------------***
symoon:~/home/assam_seoyoung/2024-os-hw1/myshell$ env USER PATH
USER=assam_seoyoung
        assam_seoyoung
***----------------------***
PATH=/home/assam_seoyoung/.vscode-server/cli/servers/Stable-38c31bc77e0dd6ae88a4e9cc93428cc27a56ba40/server/bin/remote-cli:/usr/local/sbin:/usr/
:/usr/games:/usr/local/games:/snap/bin
        /home/assam_seoyoung/.vscode-server/cli/servers/Stable-38c31bc77e0dd6ae88a4e9cc93428cc27a56ba40/server/bin/remote-cli
        /usr/local/sbin
        /usr/local/bin
        /usr/sbin
        /usr/bin
        /sbin
        /bin
        /usr/games
        /usr/local/games
        /snap/bin
***----------------------***
```

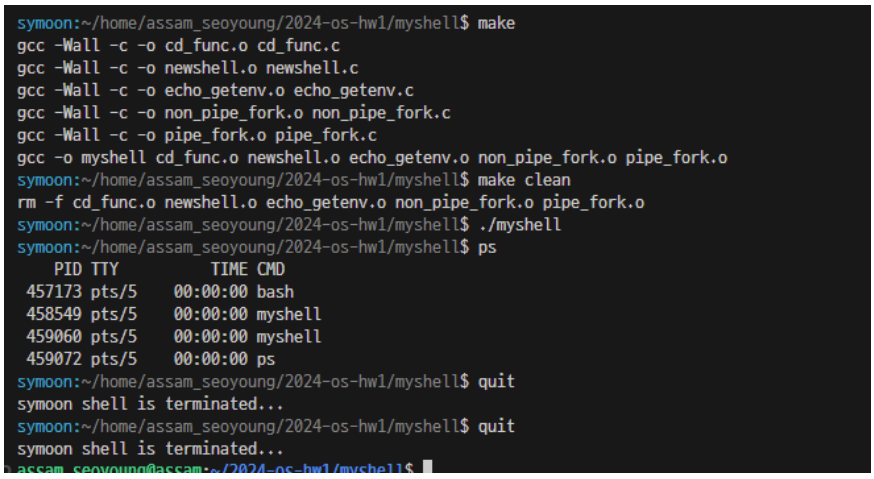
[환경변수 출력 결과: 여러 개 입력해도 잘 동작하는 걸 확인할 수 있음]



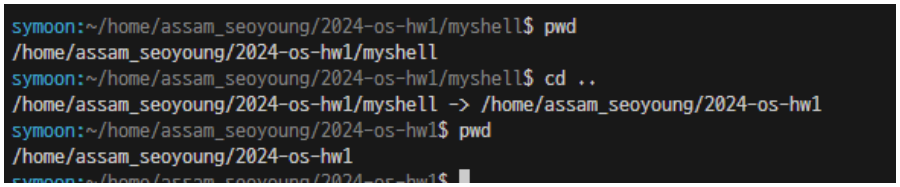
[명령어를 잘못 입력했을 때 찾을 수 없다고 나오고 help를 확인하라는 문구가 뜸 + help 출력 결과]



[파이프를 여러 개 사용한 결과 + 컴파일 동작 화면]



[myshell 안에서 myshell을 실행했을 때, 잘 동작하는 걸 볼 수 있음. (+quit 누르면 종료)]



[디렉토리 이동 명령어(cd)]

5. 어려웠던 점 및 후기

이번 과제에서는 먼저 쉘이 종료되지 않도록 fork() 함수를 사용해 프로세스를 복제하여 부모 프로세스와 자식 프로세스로 나누었습니다. 자식 프로세스에서는 환경 변수를 직접 설정할 수 있는 execve() 함수를 사용하여 명령어를 실행했습니다. 또한, getenv() 함수를 사용해 환경 변수를 출력할 수 있도록 구현했습니다. cd 명령어는 현재 프로세스에서만 유효하며, 현재 쉘의 작업 디렉터리를 변경하는 것이기 때문에 fork()를 사용하지 않고 부모 프로세스에서 직접 수행하도록 하였습니다.

1 학기 컴퓨터 구조의 MIPS 과제에서는 구조를 손으로 그리며 직접 확인할 수 있었기 때문에 비교적 쉽게 이해할 수 있었지만, 이번 과제는 마음에 와닿지 않아 구현하는 데 어려움이 있었습니다. 하지만 평소 아무 생각 없이 사용하던 쉘을 직접 만들어 보면서 쉘의 내부 처리 방식뿐만 아니라 프로세스의 동작 방식에 대해서도 조금이나마 이해할 수 있는 좋은 경험이었습니다.