Simple Implementation of Shell CLI in Linux

차 호 현(#32224560), [outcider112@dankook.ac.kr](mailto:outcider112@dankook.ac.kr)[[1]](#footnote-1)  
Undergraduate Student in Mobile Systems Engineering, Dankook University

**Index**

1. Introduction
2. Build
3. Usage
4. Implementation
5. Limitation
6. Files

**0. Introduction**

운영체제와 사용자 간의 상호작용을 위한 인터페이스로 사용되는 CLI는 터미널 환경에서 문자열을 입력하여 프로그램을 실행하거나, 컴퓨터 시스템을 구성할 수 있다. 이러한 CLI는 컴퓨터 시스템에 접근하기 위한 껍데기라는 의미로서, Shell이라고 칭하며 운영체제에서 제공하는 시스템 콜 및 라이브러리를 통해 상호작용을 구현할 수 있다. 이러한 리눅스에서 가장 흔하게 사용되는 Bash부터 시작하여 Sh, Zsh, Cshell 등의 다양한 Shell 프로그램이 존재하며, 기본으로 제공되는 쉘이 아닌 사용자가 직접 Shell을 구현하여 사용할 수 있다. 따라서 본 보고서에서는 리눅스 운영체제에서 자주 사용되는 Bash Shell을 모방하여 구현할 것이며, Shell을 통해서 어떻게 프로그램을 실행시킬 수 있는지 관찰할 것이다.

**1. Build**

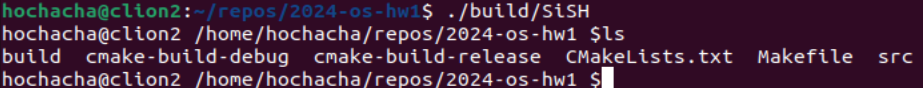
구현된 SiSH 프로그램은 Cmake를 통해 빌드할 수 있으며, Makefile은 cmake를 호출하는 wrapper로서 동작한다.

해당 명령어를 통해, 프로젝트 루트 경로 하위 build 디렉터리(2024-os-hw1/build)에서 생성된 프로그램을 확인할 수 있다.

~/2024-os-hw1/$ make

make

~/2024-os-hw1/$ ./build/SiSH

****

**2. Usage**

SiSH에서 지원하는 명령어는 내장 명령어와, 프로그램 실행 명령어로 구분된다.

**내장 명령어**

- cd

- echo

- pwd

hochacha@any\_hostname /home/hochacha/anywhere/ $ cd ..

> hochacha@any\_hostname /home/hochacha/ $

hochacha@any\_hostname /home/hochacha/ $ echo “hello world”

> hello world

hochacha@any\_hostname /home/hochacha/ $ echo $PWD

> /home/hochacha

hochacha@any\_hostname /home/hochacha/ pwd

> /home/hochacha

**프로그램 실행 명령어**

hochacha@any\_hostname /home/hochacha/ $ [프로그램 바이너리] [인자 1] [인자 2]

**명령어 파이프**

hochacha@any\_hostname /home/hochacha/ $ [명령어 1] | [명령어 2]

hochacha@any\_hostname /home/hochacha/ $ [명령어 1] | [명령어 2] | … | [명령어 n]

**출력 리다이렉션**

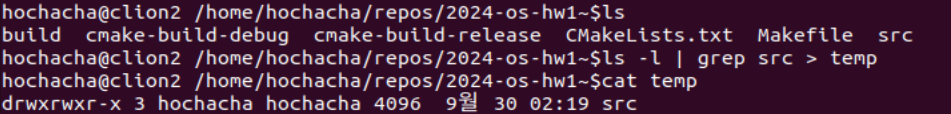
hochacha@any\_hostname /home/hochacha/ $ [명령어 1] > [파일명]

**파이프 및 리다이렉션 복합 사용**

hochacha@any\_hostname /home/hochacha/ $ [명령어 1] | [명령어 2] > [파일명]

**사용 예시**

hochacha@any\_hostname /home/hochacha/ $ [명령어 1] | [명령어 2] > [파일명]

****

**3. Implementation**

프로그램의 전반적인 흐름은 다음과 같다.

도표, 스케치, 그림, 라인이(가) 표시된 사진

자동 생성된 설명

그림 1 SiSH의 상태도

**3-1. init**

* SiSH에서 사용해야 할 환경 변수들을 로드한다.
  + **Hostname**
  + **Username**
* 해당 단계에서는 사용자 및 시스템 정보를 수집한다.

**3-2. Ready**

* 기본 SiSH 프롬프트 출력 (username, hostname, current working directory 출력)
* stdin으로부터 문자열 명령어 입력
* exit case 처리

**3-3. instruction parser & argument parser & file\_prefixer**

* **instruction parser**를 이용하여, 입력 명령어 파싱
  + **parser state**를 통해, 구분자 간 명령어 별 처리 반복하여 수행
    - Ex) echo “hello” | grep hello
    - echo “hello” 처리 🡪 grep hello 처리 (파이프)
  + 구분자 처리에 따른 예외 경우 제거
    - 문자열 내에서 사용된 구분자 문자에 대한 예외 처리
      * Ex) echo “he**|**1o world” | grep “world”
* argument parser를 이용하여, 단일 명령어를 인자 배열로 변환
  + 단일 명령어 내, 문자열 인자를 처리할 수 있도록 예외 처리
    - Ex) echo “hello world”
    - args[0] == echo, args[1] == “hello world”

**3-4. bin\_path\_finder & file\_prefixer**

* execv로 프로그램 실행 시, 첫번째 인자가 절대/상대경로로 지정되지 않은 경우
  + PATH 환경 변수에 등록된 경로 중 첫번째 인자의 파일을 검색하여 절대 혹은 상대 경로로 명시
* execvp를 사용하는 경우, 프로그램의 경로 검색은 execvp 내에서 구현
  + 해당 프로젝트에서는 PATH 변수에서 프로그램 경로 검색을 구현할 수 있도록 execvp를 사용하지 않고, execv를 사용하면서 bin\_path\_finder 구현

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

그림 2 bin\_path\_finder의 구현

* 첫 번째 인자 이후, 파일을 명시하는 인자에 대해서 file\_prefixer를 통해 상대 경로로 명시된 파일을 절대 경로로 변환
  + Ex) /home/hochacha 경로에서, cat ./temp   
    🡪 cat /home/hochacha/temp으로 변환

**3-5. fork & execv, builtin command**

* 프로그램 호출 시, fork와 execv를 통해 하위 프로세스 생성 및 실행
  + Fork 실행 시, 부모 프로세스의 동일한 복사본으로서 자식 프로세스 생성
    - 자식 프로세스는 동일한 프로그램 및 상태를 복사
    - 그러나, 자식 프로세스는 부모와 다른 주소 공간, 자신의 레지스터, 자신의 PC값을 가지게 됨
    - 부모와 자식 프로세스는 CPU 스케쥴러를 통해 CPU에 의해 처리될 수 있는 독립적인 개체로 존재하게 됨
* 부모 프로세스의 흐름
  + 부모 프로세스는 자식 프로세스와 독자적인 흐름을 가지고 실행될 수 있다.
  + 만약 자식 프로세스의 종료를 대기해야 하는 경우 wait() 함수를 통해 자식 프로세스로부터 전달되는 SIGTERM 신호를 전달받을 때까지 waiting 상태로 대기 가능
* 자식 프로세스의 흐름
  + Execv를 호출하여 새로운 프로그램 실행 가능
    - Execv를 호출하는 경우, 인자로 명시된 실행 파일의 코드와 정적 데이터를 읽어들여 현재 실행 중인 프로세스의 Code Segment와 Data Segment 부분을 덮어 쓴다.
    - 또한, Heap과 Stack 영역 및 프로그램 다른 주소 공간들로 새로운 프로그램 실행을 위해 다시 초기화된다.[[2]](#footnote-2)
* builtin command 호출 시, 명령어 처리 함수 호출

**3-6. pipe & redirect**

파일 디스크럽터는 운영체제에서 파일이나 입력/출력 스트림을 식별하는 정수이다. stdin, stdout, stderr와 같이 입/출력 스트림은 파일로 관리가 되며 이를 파일 디스크럽터를 통해 관리할 수 있다.

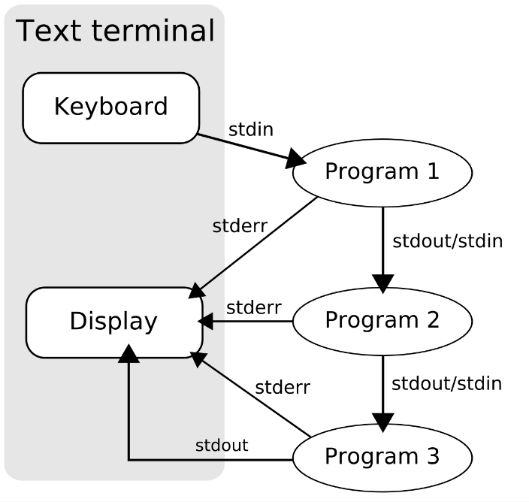
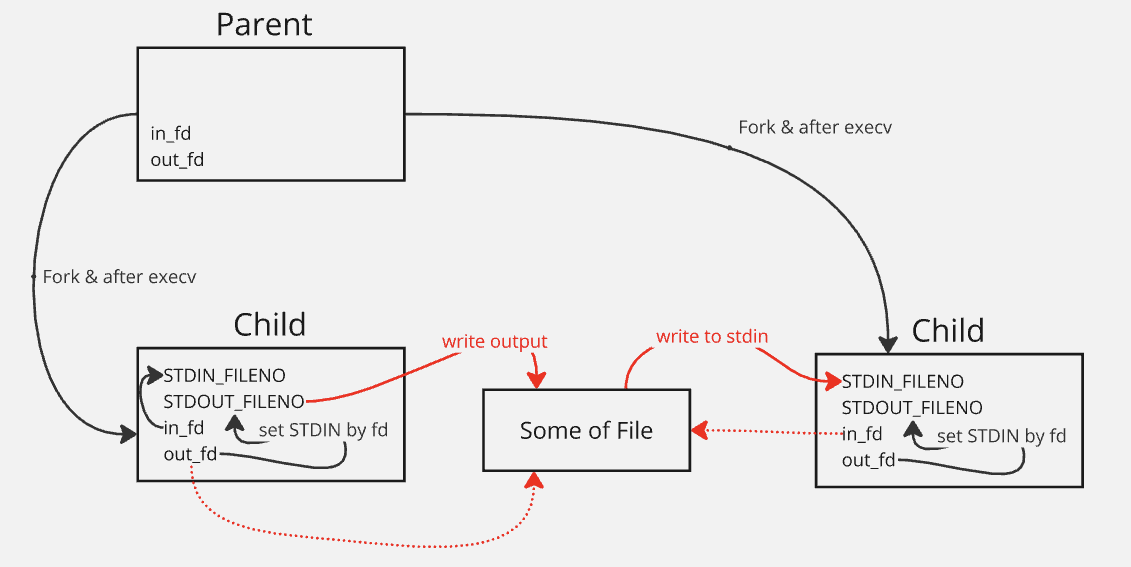


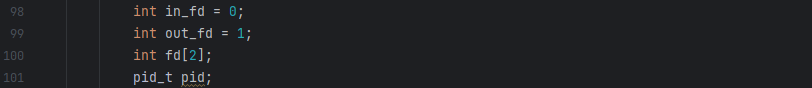
그림 3 UNIX 시스템에서의 파이프라인 동작 방식[[3]](#footnote-3)

pipe() 함수를 통해 fd를 파이프로 생성하면, fd[0]는 파이프로부터 읽는 쪽의 파일 디스크럽터가 되며 fd[1]은 파이프로부터 쓰는 쪽의 파일 디스크럽터가 된다.



Fork된 자식은 부모의 파일 디스크럽터를 그대로 전달받고, in\_fd과 out\_fd를 통해 STDIN\_FILENO와 STDOUT\_FILENO를 지정한다. 이에 따라서, 새로운 프로세스가 입력을 받아들이고 파일을 쓸 때, in\_fd와 out\_fd에 지정된 파일로부터 데이터를 읽거나 해당 파일에 쓰게 된다.

이를 구현한 코드 관점에서 보면 다음과 같다.



* 파이프에 사용할 파일 디스크럽터 배열 생성
  + int fd[2] : 읽기 끝과 쓰기 끝을 가리키는 파일 디스크럽터
  + in\_df: 입력 파일 디스크럽터
  + out\_df: 출력 파일 디스크럽터

텍스트, 폰트, 스크린샷이(가) 표시된 사진

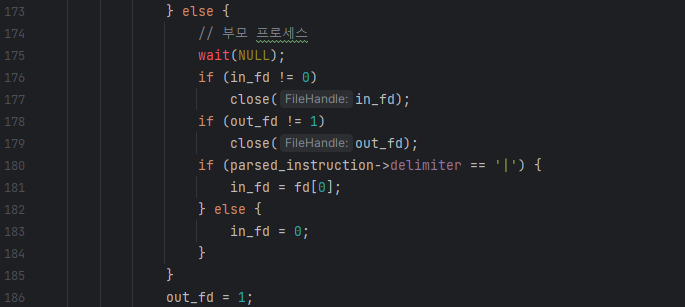
자동 생성된 설명

* 파이프 파일 디스크럽터 할당
  + out\_fd = fd[1] : 파일의 쓰기 끝을 출력파일 디스크럽터로 할당

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* fork 이후, 자식 프로세스의 입출력 파일 디스크립터 할당
  + if\_fd != 0 – 파이프 이전 프로세스로부터 전달받은 파일을 입력으로 설정
  + out\_fd != 0 – 현재 프로세스의 출력을 파일에 작성



* fork 이후, 부모 프로세스의 입출력 파일 디스크립터 관리
  + 부모 프로세스에서 열려있는 파일 디스크럽터 할당 해제
  + 현재 자식 프로세스로부터 받은 출력 (파일)을 파이프해야 하는 경우
    - fd[0] 값을 in\_fd에 할당하여, 이후에 처리될 프로세스에서 입력에 사용

**4. Limitation**

해당 프로그램을 구현하면서, 다양한 입력 패턴에 따른 예외 경우를 제거하기 위해 노력하였으나 해결하지 못한 부분이 존재한다.

1. 파이프 및 리다이렉션 이외의 다양한 Syntax

* >> 혹은 & 과 같은 출력 변환

1. 동적인 프롬프트 구현

* $HOME에 따라 현재 실행 경로 출력의 변화 (~/repo 와 /home/hochacha/repo의 차이)

1. 실제 파이프 구현 방식과 다른 형태의 구현 방식

* Stackoverflow 논의에 따르면, 각 명령어를 동시에 실행시킨 후 Block I/O를 통해 입력 대기를 수행하도록 함[[4]](#footnote-4)
* SiSH 구현에서는 하나의 자식 프로세스가 출력을 파일 디스크립터를 통해 데이터를 작성, 해당 자식 프로세스 종료 후에 다음 자식 프로세스가 입력으로 받도록 구현하였다.

1. Vim 에디터 호출 시 터미널 출력 오류 발생
2. Builtin command의 충분하지 못한 구현

* bash에는 history, export 등의 다양한 builtin command가 존재하지만, SiSH에서 구현하지 못함

**5. Files**

**Annotations: / (directory), \* (executable file), ‘otherwise’ (file)**

2024-os-hw1/ (project root)

* build/
  + Makefile
  + SiSH\*
* docs/
  + os-report1.docx
* src/
  + tools/
    - builtin\_commands.c / .h
    - path\_finder.c / .h
    - stack.c / .h
    - string\_tools.c / .h
  + argument\_parser.c / .h
  + instruction\_parser.c / .h
  + shell.c / .h
  + main.c
* CMakeLists.txt
* Makefile

Conclusion

SiSH를 직접 구현함에 따라서 리눅스의 프로세스 생성 시스템 콜인 fork와 exec, wait를 사용하여 Shell의 동작을 모방하였다. 이 과정에서 fork와 exec로 분리된 프로그램 실행 절차에 따라서, exec를 통해 프로그램을 실행하기 전에 입/출력 파일 설정과 같은 프로세스의 사전 설정을 수행할 수 있었다. 이번 과제를 수행하면서, 아쉬운 점으로 프로세스에게 전달할 수 있는 시그널을 사용하여 프로세스 제어를 시도하지 못한 것이 아쉬웠으며, 이에 따라서 실제 Bash 처럼 실행 중인 프로세스를 ^C 입력을 통해 강제로 종료하거나, 다양한 기능을 구현하지 못한 것이 아쉽다.

Shell이 사용자로부터 문자열 입력을 받음에 따라서 입력될 수 있는 경우들을 제한하고 예외적인 경우를 처리하기 위해 시간을 많이 들였다. 또한 파이프를 구현하기 위하여 Instruction\_parser를 고안하였을 때, 예외 처리를 구현하기 위해 시간이 많이 들었다.

또한, 해당 과제를 수행하면서 AI를 통한 코딩을 사용하였다. 코드 자체를 프롬프트를 통해 완전히 생성한 경우, 함수 본문 위에 /\* Ai generated code \*/라고 명시하였다.

1. Free days remaining: 5 [↑](#footnote-ref-1)
2. Arpaci-Dusseau, Remzi H., and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*., 원유집 역, Arpaci-Dusseau Books, 2018, pp 39-49. [↑](#footnote-ref-2)
3. https://upload.wikimedia.org/wikipedia/commons/thumb/f/f6/Pipeline.svg/420px-Pipeline.svg.png [↑](#footnote-ref-3)
4. Stackoverflow, “how does a pipe work in linux”, accessed on 30th Sep., 2024, https://stackoverflow.com/questions/1072125/how-does-a-pipe-work-in-linux [↑](#footnote-ref-4)