**System Programming Project 2**

담당 교수 : 이영민

이름 : 박지민

학번 : 20231552

1. **개발 목표**

이 프로젝트의 목표는 사용자가 입력한 명령어를 처리할 수 있는 간단한 쉘 프로그램 MyShell을 구현하는 것이다. 직접 사용자 정의 셸 프로그램(myshell)을 구현하는 것이다. 주요 학습 포인트는 프로세스 생성 및 제어, 시그널 처리, 파이프 및 입출력 리디렉션, 백그라운드 작업 관리와 같은 시스템 호출 기반 기능들을 실제 코드로 구현하고 실습해보는 것이다.

1. **개발 범위 및 내용**
   1. **개발 범위**
      1. Phase 1

cd, exit, ls, mkdir, touch, cat 등 기본 명령어를 셸에서 직접 실행할 수 있다.

내장 명령어는 부모 프로세스가 처리하고, 외부 명령어는 fork()와 exec()를 통해 자식 프로세스에서 실행됨을 확인할 수 있었다

* + 1. Phase 2

| 기호를 포함한 명령어가 여러 개 입력되면, 각 명령어가 개별 프로세스로 실행되고 이전 프로세스의 출력이 다음 프로세스의 입력으로 전달된다.

cat file**.**txt | grep hello | sort와 같은 명령을 정상적으로 처리함을 확인하였다.

* + 1. Phase 3

명령어 뒤에 &가 붙으면 백그라운드 프로세스로 실행되며, 이 프로세스들은 jobs 명령어로 상태를 조회하거나, fg, bg, kill 명령어로 제어 가능하다. ctrl+z, ctrl+c 같은 시그널에 따라 포그라운드 프로세스가 중단되거나 종료되는 동작도 정상적으로 작동한다.

* 1. **개발 내용**

**Phase1 (fork & signal)**

* + fork를 통해서 child process를 생성하는 부분에 대해서 설명
  + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow
* 사용자가 입력한 명령어를 eval() 함수에서 파싱한 후, 내장 명령어가 아닌 경우 fork() 시스템 호출을 통해 자식 프로세스를 생성한다. 자식 프로세스에서는 execvp()를 사용하여 새로운 명령어로 실행 흐름을 교체하며, 부모는 자식의 종료 여부를 기다린다. 이러한 구조는 부모 셸과 자식 명령어 실행 환경을 분리하여, 셸이 계속 동작할 수 있도록 한다. 파이프라인을 포함한 경우에도 각 명령어에 대해 순차적으로 fork()를 호출하고, 각각 독립된 자식 프로세스로 실행되도록 구현했다. 자식 프로세스가 정상적으로 종료되거나, 시그널에 의해 종료될 경우 커널은 자동으로 부모 프로세스에 SIGCHLD 시그널을 전송한다. 셸은 sigchld\_handler() 시그널 핸들러를 등록해 두었고, 해당 핸들러에서 waitpid()를 WNOHANG, WUNTRACED, WCONTINUED 플래그와 함께 반복 호출하여 자식 상태를 확인하고 처리한다. 종료된 프로세스는 job 리스트에서 제거되고, WIFSTOPPED 상태인 경우에는 job의 상태를 Stopped로 갱신한다. 포그라운드 프로세스의 경우, eval() 내에서 PID 전역 변수를 통해 해당 프로세스 종료를 직접 감지하고 명시적으로 대기한다. 셸에서 시그널을 안전하게 처리하기 위해 init\_sighandlers() 함수에서 SIGCHLD, SIGINT, SIGTSTP에 대한 핸들러를 등록한다. SIGCHLD 핸들러 (sigchld\_handler)는 자식 프로세스의 종료, 중지, 재개 등의 상태를 감지하여 waitpid()를 통해 정리하고, job\_list를 갱신한다. SIGINT, SIGTSTP 핸들러는 포그라운드 작업에 대해 SIGINT나 SIGTSTP를 전파하여 중단하거나 정지시키도록 구현되어 있다. 포그라운드 프로세스의 종료는 volatile sig\_atomic\_t PID를 통해 감지하며, eval()의 루프는 이 PID 값이 설정될 때까지 sigsuspend()를 통해 대기한다.
* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)
  + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

입력 명령어에 |가 포함되어 있으면, 이를 기준으로 명령어들을 분리한다. (예: ls | grep txt | sort → [ls, grep txt, sort]) 분리된 각 명령어는 새로운 자식 프로세스에서 실행되며, 각 프로세스 간의 입출력은 pipe()로 생성된 파이프를 통해 연결된다. 파이프 개수(n개)에 따라 프로세스도 n개 생성되며, 반복문에서 각 프로세스를 생성하고, 표준 입출력(STDIN/STDOUT)을 dup2()를 통해 리디렉션한다. 마지막 명령을 제외한 모든 명령은 파이프의 write-end로 출력하며, 다음 명령은 pipe의 read-end를 입력으로 받는다. 명령어의 개수는 파이프 수 + 1이며, 파이프가 여러 개일 경우에도 루프를 통해 연결을 자연스럽게 처리한다. 부모 프로세스는 마지막 명령의 pid를 저장하여, 포그라운드일 경우에만 waitpid()를 수행한다.

* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명
* insert\_special\_spaces() 함수를 통해 & 기호 앞뒤에 공백을 추가하여 파싱이 정확하게 이루어지도록 하고, parseline()에서 명령어 끝에 &가 존재하는지 확인한다. 백그라운드 실행으로 판단되면(bg 플래그 ==1) eval()에서 해당 프로세스에 대해 wait()를 호출하지 않고, 대신 addjob()을 통해 job 리스트에 등록한다. 각 job은 Job 구조체를 사용하여 pid, cmdline, state, job\_id 등의 정보를 저장하고, Background 상태로 관리된다. 이후 사용자는 jobs, fg, bg, kill 명령어를 통해 백그라운드 프로세스를 제어할 수 있다. 백그라운드 job이 종료될 경우, sigchld\_handler()에서 이를 감지하여 자동으로 job 리스트에서 제거되며, 중단된 경우에는 Stopped 상태로 남아 사용자가 이후에 fg, bg로 제어할 수 있도록 한다. fg 명령어는 job을 포그라운드로 전환하고 SIGCONT 전달, tcsetpgrp()를 통해 셸 제어권도 전환한다. bg 는 job을 백그라운드 상태로 전환하고 SIGCONT 전달한다. 포그라운드 job의 상태 변화는 SIGCHLD에서 처리되며, 상태가 Stopped가 아니면 job을 deletejob()을 통해 제거한다.
  1. **개발 방법**

Phase 1

1) eval() 함수

셸의 핵심 함수로, 사용자 입력을 받아 명령어를 파싱하고 실행하는 모든 과정을 포함한다. 입력받은 명령어를 복사한 뒤, 파이프라인 여부에 따라 분리하고 각 명령어를 처리한다.

내장 명령어(`cd`, `exit` 등)가 아닌 경우에는 `fork()`를 호출하여 자식 프로세스를 생성한다. 자식 프로세스는 시그널 핸들러를 기본값(SIG\_DFL)으로 설정한 뒤, `execvp()`를 호출하여 해당 명령어를 실행한다.

부모 프로세스는 포그라운드 명령어에 대해 자식 프로세스가 종료될 때까지 기다리며, `fg\_pid`를 통해 추적한다.

2) `builtin\_command()` 함수

내장 명령어들을 실행하는 함수로, 부모 프로세스 내에서 직접 실행되어야 하는 명령(`cd`, `exit`, `jobs`, `fg`, `bg`, `kill`)을 처리한다. 명령어에 따라 현재 작업 디렉토리를 변경하거나 셸을 종료하는 등의 기능을 수행한다.

Phase 3에서 job 제어 명령어도 이 함수에 포함되어 확장된다.

3) 시그널 핸들러들 (`sigchld\_handler`, `sigint\_handler`, `sigtstp\_handler`)

- `sigchld\_handler`: 자식 프로세스 종료 시 호출되며, `waitpid()`로 자식의 상태를 확인하고 job 상태를 업데이트한다. 정상 종료되거나 시그널로 종료된 경우 job을 삭제하고, `WIFSTOPPED` 상태인 경우 job을 `Stopped` 상태로 전환한다.

- `sigint\_handler`: `ctrl+C` 입력 시 포그라운드 job의 프로세스 그룹에 SIGINT를 전송하여 종료시키도록 구현하였다.

- `sigtstp\_handler`: `ctrl+Z` 입력 시 포그라운드 job을 정지시키고 상태를 `Stopped`로 바꾼다.

4) `init\_sighandlers()` 함수

위 시그널 핸들러들을 `Signal()`을 통해 등록하고, 실행 시점에 `job\_list[]` 배열을 초기화한다.

Phase 2

1) `split\_pipeline()` 함수

입력 문자열을 `|` 기호 기준으로 나누어 각각의 명령어를 `cmds[]` 배열에 저장한다. 공백이나 줄바꿈 처리를 통해 명령어를 정확히 분리하며, 최대 파이프 수 제한을 고려한다. 파이프라인의 각 구간을 개별적으로 실행 가능하도록 전처리를 수행한다.

2) `eval()` 함수의 확장

분리된 명령어(`cmds[]`)를 순차적으로 처리하며, 각 단계마다 다음과 같은 작업을 수행한다:

1. `pipe()`를 호출하여 파이프 생성

2. `fork()`를 통해 자식 프로세스 생성

3. 자식에서는 `dup2()`로 `stdin`, `stdout`을 파이프 입출력으로 리디렉션

4. `execvp()`를 통해 명령어 실행

파이프 연결은 다음 명령어의 입력을 현재 명령어의 출력과 연결하도록 구성하며, 불필요한 디스크립터는 자식과 부모에서 모두 닫아준다. 마지막 명령어는 더 이상 파이프가 필요하지 않으므로 표준 출력(STDOUT)을 그대로 사용한다. 포그라운드 명령어의 경우 마지막 프로세스의 PID를 `fg\_pid`로 저장하고, 종료될 때까지 대기한다.

Phase 3

1) Job 구조체 (`myshell.h`)

셸에서 실행되는 각 job을 추적하기 위해 선언된 구조체로,

- pid: 프로세스 ID

- cmdline: 명령어 문자열

- job\_id: 사용자에게 보여지는 고유한 job 번호

- state: 현재 상태(Foreground, Background, Stopped 등)

- last\_idx: 배열 끝 여부 표시

- job\_list[]는 이 구조체의 배열로, 최대 MAXJOBS개의 job을 관리한다.

2) State 열거형

- Invalid, Foreground, Background, Stopped의 네 가지 상태를 정의하며, job 상태 추적과 전환을 명확히 하기 위해 사용된다.

3) Job 관련 함수들

- addjob(): 새로운 job을 job 리스트에 등록하고 고유한 job ID를 부여

- deletejob(): 종료된 job을 `Invalid` 상태로 전환하여 제거

- getjob(), getjob\_by\_id(): PID 또는 job ID를 기반으로 job을 검색

- listjobs(): 현재 활성화된 job들을 상태별로 출력함

4) parseline() 함수의 확장

명령어 끝에 `&` 기호가 존재하는지 판단하여 백그라운드 실행 여부를 결정한다. argv가 NULL인 경우에도 백그라운드 여부 확인 용도로만 사용할 수 있도록 분기 처리하였다.

5) eval() 함수의 Phase 3 확장

백그라운드 명령어(`&` 포함)의 경우, 자식 프로세스를 생성한 후 `addjob()`으로 등록하고 `wait()` 없이 바로 리턴한다. 포그라운드 job은 기존 방식대로 fg\_pid 저장 및 종료 대기 루틴을 수행한다. job 상태는 이후 시그널 핸들러를 통해 업데이트되고, 종료되면 `deletejob()`으로 삭제된다.

6) 내장 명령어 확장 (`fg`, `bg`, `kill`)

- fg: 지정한 job을 `Foreground` 상태로 설정하고 `SIGCONT`를 통해 재개하며, 셸의 터미널 제어권도 해당 프로세스로 넘겨줌

- bg: `Stopped` 상태인 job을 `Background`로 전환하고 `SIGCONT`를 보내 백그라운드 실행 재개

- kill: job ID(`%1` 등)로 특정 job을 찾아 해당 프로세스 그룹에 `SIGKILL`을 전송함

7) `insert\_special\_spaces()` 함수

Eval 함수 호출 전에 명령어 내 `|`와 `&` 주변에 자동으로 공백을 삽입하여 명령어 파싱의 정확성을 높였다.

예: `ls|grep` → `ls | grep`, `sleep 5&` → `sleep 5 &` 형태로 변환

1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**

텍스트, 폰트, 스크린샷, 도표이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

1. **Phase 2 (pipeline)**

텍스트, 폰트, 스크린샷, 도표이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

1. **Phase 3 (background)**

텍스트, 도표, 폰트, 평행이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.