**System Programming Project 2**

담당 교수 : 김영재 교수님

이름 : 김종원

학번 : 20211523

1. **개발 목표**

- 이 프로젝트의 목표는 리눅스 시스템 프로그래밍의 기본 개념을 이해하고, 실제로 shell을 구현해보는 것이다.

- 또한, 이 프로젝트는 시스템 프로그래밍으로 프로세스 생성( Fork() ), 실행( void eval(char \*cmdline) ), 시그널 처리( void sigchld\_handler(int sig) ), 파이프라인, 그리고 백그라운드(background) 프로세스 관리(job control) 등의 기능을 직접 구현함으로써 shell에 대해 배워 보는 것이 목표이다.

- 즉, 사용자가 입력한 명령어를 분석하여 실행하고, 파이프(|)와 '&' 기호를 이용해 명령어 연결 및 백그라운드 작업 제어를 지원하는 나만의 쉘을 제작하는 것이 목표이다.

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

기본 쉘 동작 구현(내장 명령어와 외부 명령어 실행)

fork()를 이용해 자식 프로세스를 생성하여 execvp()로 명령어 실행

SIGCHLD 핸들러를 통해 자식 프로세스의 종료 또는 중지를 감지하고 처리

1. Phase 2

파이프라인(Pipeline, ‘|’) 기능 구현

여러 개의 파이프가 포함된 명령어에 대해 각 프로세스 간 표준 입출력 재지정을 통한 데이터 흐름 연결

1. Phase 3

백그라운드 실행(’&’) 프로세스 구현

백그라운드 작업에 대해 job 리스트를 관리하고, jobs, bg, fg, kill 등의 제어 기능을 제공

* 1. **개발 내용**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명

쉘은 사용자가 외부 명령어를 입력하면 eval() 함수 내에서 Fork() 함수를 호출해 자식 프로세스를 생성한다.

생성된 자식은 execvp() 함수를 이용해 입력한 명령어를 실행하므로, 부모와 독립적인 실행 경로가 형성된다.

* + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow

부모 프로세스는 SIGCHLD 핸들러를 등록하여, 자식 프로세스가 종료되거나 중지할 때 자동으로 SIGCHLD 시그널을 받아 처리한다.

SIGCHLD 핸들러에서는 waitpid()를 WNOHANG 및 WUNTRACED 옵션과 함께 호출하여 자식 프로세스의 상태를 확인하고, 자식이 종료되면 job 리스트에서 해당 정보를 삭제하거나 상태를 업데이트하는 방식으로 자원 회수를 수행한다.

* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)

사용자의 명령어 입력에서 ‘|’ 기호가 있는지를 확인한 후, 해당 기호를 기준으로 명령어를 여러 단위로 분리한다.

각 단위의 명령어를 처리하기 위해 eval\_pipe() 함수를 통해 별도의 자식 프로세스를 생성하고, pipe() 시스템 콜을 사용하여 파이프를 생성한다.

생성한 파이프의 쓰기 단은 현재 명령어의 출력으로, 읽기 단은 다음 명령어의 입력으로 설정하기 위해 dup2() 함수를 사용한다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

파이프라인에 포함된 명령어 수가 동적으로 결정되므로, eval\_pipe() 함수 내에서 반복문을 통해 순차적으로 파이프를 생성하고 연결한다.

첫 번째 명령어는 표준 입력 변경 없이 실행되고, 중간의 각 명령어는 앞선 파이프의 읽기 단과 새로운 파이프의 쓰기 단을 연결하며, 마지막 명령어는 추가 파이프 없이 최종 결과를 출력하도록 설정된다.

부모 프로세스는 모든 자식 프로세스의 종료를 wait()로 대기하여 전체 파이프라인의 실행 흐름을 관리한다.

* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명

쉘은 명령어 입력 끝에 '&' 기호가 있으면 해당 명령어를 백그라운드 작업으로 인식한다.

백그라운드 실행 시, fork()를 통해 자식 프로세스를 생성한 후, 부모는 wait() 없이 즉시 사용자에게 프롬프트를 반환하고, 생성된 자식 프로세스의 PID와 명령어, 실행 상태를 job 리스트에 등록한다.

* 1. **개발 방법**
* 모든 함수와 구조체는 myshell.h에 정의해 놓았고, myshell.c에 구체화해 놓았다.
* Phase 1:

myshell.c의 eval() 함수에서 Fork()를 통해 자식 프로세스를 생성하고, execvp()를 통한 명령어 실행 로직을 추가하였다.

자식 프로세스 종료 및 중지 상태를 감지하기 위해 sigchld\_handler() 함수를 추가하여, waitpid()를 호출하고 자식의 상태를 처리하는 로직을 구현하였다.

* Phase 2:

파이프라인 처리를 위해 별도의 eval\_pipe() 함수를 작성하였으며, parseline() 함수를 수정하여 인용부호 처리 등 세부 파싱 로직을 보완하였다.

각 자식 프로세스 생성 후, pipe()와 dup2() 호출로 표준 입출력을 재지정하여 파이프 연결을 구현하였다.

* Phase 3:

백그라운드 작업 처리를 위해 myshell.h에 job 구조체를 정의하고, global 잡 리스트 및 관련 함수(add\_job(), delete\_job(), find\_job\_by\_jid(), list\_jobs() 등)를 추가하였다.

eval() 함수 내에서 '&' 기호를 인식하여 백그라운드 실행 여부를 판단하고, setpgid()를 호출하여 자식 프로세스가 독립적인 프로세스 그룹을 형성하도록 수정하였다.

내장 명령어(jobs, bg, fg, kill)를 추가하여 job 제어 기능을 구현하였다.

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

1. 사용자가 어떤 입력한다.

2. 입력 내용에서 명령어를 외부~내장 명령어인지 구분한다.

3-1. 내장 명령어이면, 입력된 단어에 따라 명령어를 실행하고 flow종료.

3-2. 외부 명령어이면, fork()를 실행하여, 자식 프로세스에서 execvp()를 실행하여, 명령을 실행한다.

4. 자식 프로세스가 종료 또는 정지되면, SIGCHLD 핸들러가 waitpid()를 호출하여, 자식 프로세스의 상태를 확인하고, 자원을 회수한다.

1. **Phase 2 (pipeline)**

1. 사용자가 어떤 입력을 하는데 파이프가 포함됨

2. ‘|’ 기준으로 명령어를 분리한다.

3. 각 명령어별로 자식 프로세스를 생성한다. (위에서 설명한 방식대로 내장,외부 명령어 구분)

4. pipe() 및 dup2()를 통해 표준 입출력에 연결한다.

5. 부모 프로세스가 파이프라인을 전체 실행한 후에 자식 프로세스가 모두 종료할 때까지 대기한다.

1. **Phase 3 (background)**

1. 사용자가 어떤 입력을 하는데 끝에 '&' 포함됨

2. 명령어를 파싱하는데, ‘&’를 만나면 백그라운드 실행 판단한다.

3. fork()를 통해서 자식 프로세스를 생성한다.

4. setpgid()를 호출하여 독립 프로세스 그룹 생성한다

5. 백그라운드 job을 job 리스트에 PID, job 번호, 명령어, 상태와 함께 저장한다.

6. 백그라운드에서 작동하다가 실행이 끝나면, 자동으로 끝나고 job 번호, PID, 출력 내용이 차례차례 나온 후, 종료되었다는 출력이 나온다.