**System Programming Project 2**

담당 교수 : 박성용

이름 : 박민준

학번 : 20212020

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* **(MyShell을 만드는 전체적인 개요에 대해서 작성하면 됨.)**

1. Phase 1은 기본적인 쉘을 생성해 간단한 리눅스 쉘 명령어 (cd, ls, mkdir, rmdir, touch, cat, echo, exit)를 실행할 수 있게 한다. 쉘은 사용자로부터 명령을 입력받아, 파싱 후 자식 프로세스를 생성하여 명령을 실행한다. exit와 cd는 자식 프로세스 생성 없이 직접 처리한다. 일반적인 내장 명령어인 cd, quit, exit를 구현하여 기본적인 파일 시스템 조작 및 쉘 종료 기능을 제공한다. 명령어 뒤에 &를 붙여 백그라운드에서 실행할 수 있도록 지원한다. 이를 통해 사용자는 다른 작업을 실행하는 동안 쉘을 차단하지 않고 여러 명령어를 병렬로 실행할 수 있다. 또한, &&를 사용하여 여러 개의 명령어를 하나의 줄에 연결하여 실행할 수 있도록 한다. 이를 통해 이전 명령어가 성공적으로 실행된 경우에만 다음 명령어를 실행할 수 있다. 마지막으로 사용자가 잘못된 명령어를 입력했을 때 적절한 오류 메시지를 표시하고, 프로그램이 예상치 못한 방식으로 종료되는 것을 방지한다. 또한, 포그라운드 작업이 완료될 때까지 쉘이 차단되지 않도록 안정적인 대기 및 종료 처리를 구현한다.

2. Phase 2는 단순 쉘의 기능을 확장하여 명령어 간 파이프라인을 구현한다. 예를 들어, ls -al | grep filename과 같은 명령을 수행할 수 있어야 한다. 각 명령어는 새로운 프로세스를 생성하고, 부모 프로세스는 파이프라인의 마지막 명령이 완료될 때까지 대기한다. 쉘은 파이프라인을 통해 여러 명령의 출력과 입력을 올바르게 연결하여 결과를 생성해야 한다. 입력된 명령어 사이에 파이프(|)가 있는 경우, 이전 명령의 출력을 다음 명령의 입력으로 연결한다. 이를 4가지 단계로 요약하면 다음과 같다.

1) 명령어를 파이프(|) 기준으로 파싱하여 각각의 명령을 실행한다.

2) 파이프 파일 디스크립터를 사용하여 명령어 간 데이터 전송을 구현한다.

3) 명령어 실행 중 파이프를 통해 데이터가 연속적으로 흘러가도록 관리한다.

4) 마지막 명령에서는 표준 출력으로 결과를 출력한다.

3. Phase 3는 쉘에 작업 제어를 추가하여 사용자가 작업을 background와 foreground사이에서 이동할 수 있게 한다. 명령어 끝에 &가 붙으면 해당 명령을 background에서 실행한다. background 실행된 프로세스는 foreground의 명령 입력과 동시에 실행되며, 사용자는 추가 명령을 계속 입력할 수 있다. 또한 jobs, bg, fg, kill 등의 내장 명령어를 통해 background 작업을 관리할 수 있어야 하고, Ctrl-C, Ctrl-Z와 같은 신호를 적절히 처리해야 한다. jobs는 현재 실행 중이거나 중지된 background 작업의 목록을 출력한다. bg <job>은 중지된 background 작업을 다시 실행한다. fg <job>은 background 작업을 foreground로 가져와 활성화한다. kill <job>은 특정 background 작업을 종료합니다.

신호 처리의 경우, Ctrl-C (SIGINT)는 foreground에서 실행 중인 프로세스를 종료하며 Ctrl-Z (SIGTSTP)는 foreground에서 실행 중인 프로세스를 일시 중지한다. 이러한 신호들을 적절히 처리하여 사용자가 시스템과의 상호작용 중에 발생할 수 있는 중단을 관리할 수 있도록 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

사용자로부터 명령어를 입력받아 처리하는 기본적인 쉘의 기능이 구현된다. 이 단계에서는 간단한 리눅스 쉘 명령어 (cd, ls, mkdir, rmdir, touch, cat, echo, exit)를 포함한 기본적인 내장 명령어(cd, exit 등)와 외부 명령어를 실행한다. 사용자는 쉘 프롬프트에서 기본 명령어를 입력하여 파일 시스템을 탐색하고, 프로그램을 실행할 수 있다. 이를 통해 사용자는 Linux 기반의 명령줄 환경에서 필요한 작업을 수행할 수 있게 된다.

1. Phase 2

쉘에 파이프라인(|) 기능을 추가하여 여러 명령어의 출력과 입력을 연결한다. 이를 통해 명령어 사이에서 데이터를 직접 전달할 수 있다. 사용자는 복잡한 명령어 체인을 구성하여 한 명령어의 출력을 다른 명령어의 입력으로 직접 연결할 수 있다. 예를 들어, ls -l | grep "txt"를 실행하여 모든 텍스트 파일 목록을 필터링할 수 있다. 이러한 기능을 통해 데이터 처리 작업의 효율성을 크게 향상시킬 수 있다.

1. Phase 3

background 실행 지원과 작업 관리 기능을 쉘에 추가한다. 사용자는 명령어 끝에 &를 추가하여 명령을 background 에서 실행할 수 있으며, jobs, fg, bg, kill 명령을 사용하여 background 작업을 관리할 수 있다. 결과적으로 사용자는 여러 프로세스를 동시에 관리하고, background 에서 실행 중인 작업을 모니터링하며, 필요에 따라 foreground나 background 로 전환할 수 있다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술**
* **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명

fork() 시스템 콜을 사용하여 자식 프로세스를 생성한다. fork()는 부모 프로세스의 복사본을 만들며, 이 때 생성된 자식 프로세스는 부모 프로세스가 실행 중이던 코드의 다음 줄부터 실행을 시작한다. 자식 프로세스는 execvp() 함수를 호출하여 새로운 프로그램을 실행함으로써, 기존 프로세스 이미지를 새로운 프로그램으로 대체한다.

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

연결 종료 또는 다른 특정 이벤트 발생 시, 부모 프로세스는 자식 프로세스로부터 시그널을 받아 처리한다. 이는 signal() 함수를 사용하여 처리되며, 예를 들어, SIGCHLD 시그널은 자식 프로세스가 종료될 때 부모에게 전달된다.

signal(SIGCHLD, handler)를 호출하여 자식 프로세스가 종료되었을 때 실행할 함수(handler)를 지정한다. 이 핸들러 함수 내에서 waitpid()를 사용하여 자식 프로세스의 종료 상태를 회수하고, 자원을 정리한다.

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

파이프라인(|) 기능은 여러 명령을 연결하여 이전 명령의 출력을 다음 명령의 입력으로 사용하도록 한다. 이를 위해 pipe() 시스템 콜을 사용하여 데이터 통로 역할을 하는 파이프를 생성한다. 각 명령어 실행 전에 pipe()를 호출하여 파이프를 설정하고, 각 자식 프로세스는 dup2()를 사용하여 표준 입력 또는 출력을 해당 파이프의 적절한 끝에 연결한다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

입력된 명령어에 따라 파이프의 개수를 동적으로 처리한다. 명령어를 파싱하여 파이프 기호(|)를 기준으로 분리한 후, 각 부분을 개별 명령으로 실행한다. 또한, 체인의 각 단계마다 파이프를 생성하고, 각 파이프의 출력을 다음 파이프의 입력으로 연결한다. 마지막 파이프의 출력은 표준 출력으로 연결된다.

* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명
  1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

1. Phase 1의 경우, eval() 함수 내에서 fork() 호출을 추가한다. 명령어를 실행하기 위해 자식 프로세스를 생성하는 코드를 추가한다. 이를 통해 사용자가 입력한 명령어를 실행하기 위해 새로운 자식 프로세스를 생성한다. 시그널 핸들링을 위해 signal() 호출을 추가하여 SIGCHLD 시그널을 적절히 처리할 수 있도록 한다. 자식 프로세스가 종료되면 부모 프로세스가 이를 감지하고 자원을 정리할 수 있도록 핸들러 함수를 구현한다. sigchld\_handler() 함수는 SIGCHLD 시그널을 받았을 때 호출되어, waitpid()를 사용하여 자식 프로세스의 종료 상태를 회수하고 필요한 정리 작업을 수행한다.
2. Phase 2의 경우 또한 eval() 함수를 수정하여 파이프라인을 처리할 수 있도록 한다. 명령어를 | 기호로 분리하고, 각 명령어 사이에 파이프를 생성한다. execute\_command() 함수를 수정하여 입력과 출력을 dup2() 함수를 사용하여 적절한 파이프 파일 디스크립터로 리다이렉션한다.
3. **구현 결과**
   1. **Flow Chart**

* **2.B.개발 내용에 대한 Flow Chart를 작성.**
* **(각각의 방법들에서 추가된 내용(fork, pipeline, background)만 특성이 잘 드러나게 그리면 됨.)**

1. **Phase 1 (fork)**

도표, 스케치, 텍스트, 종이접기이(가) 표시된 사진

자동 생성된 설명

1. **Phase 2 (pipeline)**

도표, 텍스트, 라인, 디자인이(가) 표시된 사진

자동 생성된 설명

1. **Phase 3 (background)**