**System Programming Project 2**

담당 교수 : 김영재

이름 : Jumagul Alua

학번 : 20231632

1. **개발 목표**

본 프로젝트의 목표는 Linux 시스템 프로그래밍의 핵심 개념인 프로세스 제어, 프로세스 시그널링, 프로세스 간 통신, 그리고 백그라운드 작업 처리를 학습하고 이를 구현한 프로젝트다. 이를 위해 간단한 Linux 셸인 MyShell을 개발하여 사용자가 명령어를 입력하고 실행하며, 프로세스와 작업을 관리할 수 있도록 설계하였다.

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

먼저, myShell이라는 간단한 쉘 프로그램을 구현하였다. fork와 exec 시스템 호출을 사용하여 새로운 프로세스를 생성하고, 사용자가 입력한 명령어를 실행할 수 있도록 하였다. 기본적으로 제공된 shell.c와 csapp.c 코드를 참고하며, 몇 가지 기능을 개선하고 보완하였다.

이 단계에서는 cd, ls, mkdir, rmdir, touch, cat, echo, exit 등 기본적인 명령어를 처리할 수 있도록 하였다. 특히 cd 명령어의 경우, 쉘 내에서 디렉터리를 변경해야 하므로 자식 프로세스가 아닌 부모 프로세스에서 처리하도록 구현하였다. 또한 기존 코드에는 quit 명령어는 있었지만 exit 명령어는 누락되어 있었기 때문에, exit 명령어를 새롭게 추가하여 쉘 종료 기능을 명확히 하였다.

프로그램은 ./myShell로 컴파일하여 실행할 수 있으며, 실행 후에는 CSE4100-SP-P2> 프롬프트를 통해 지속적으로 명령어를 입력받는다. 결과적으로, ls, mkdir, touch, cat, echo 등의 기본 명령어는 정상적으로 작동하며, cd 명령어는 현재 디렉터리를 이동시키고, exit 명령어를 통해 쉘을 종료할 수 있다.

1. Phase 2

이 단계에서는 파이프라인(|) 기능을 구현하여, 명령어 간에 출력과 입력을 연결할 수 있도록 하였다. 즉, 한 명령어의 출력을 다음 명령어의 입력으로 전달하는 방식이다. 파이프가 포함된 명령어를 인식하고 처리할 수 있도록 하였으며, 단일 파이프뿐만 아니라 여러 개의 파이프가 연결된 복합 명령어도 실행 가능하도록 구현하였다.

예를 들어, ls | grep filename, cat filename | less 같은 간단한 파이프 명령어는 정상적으로 작동하며, cat filename | grep -i abc | sort -r처럼 다단계 파이프 명령어도 문제없이 처리되는 것을 확인하였다. 또한 명령어 인자에 " "(따옴표)가 포함되어 있을 경우에도, 일반적인 인자와 유사하게 작동하도록 구현하여 사용자 입력의 유연성을 높였다.

1. Phase 3

Phase 3에서는 백그라운드 명령어 실행 (&)을 지원하고, 이를 관리하기 위한 job control 기능을 구현하였다. jobs, bg, fg, kill과 같은 내장 명령어를 통해 백그라운드 작업의 목록 확인, 중지된 작업의 재개, 작업의 포그라운드 전환, 특정 작업 종료 등이 가능하도록 하였다.

예를 들어, sleep 100 &, ls | grep filename &와 같은 명령어는 백그라운드에서 정상적으로 실행되며, 쉘은 즉시 다음 명령을 입력받을 수 있게 된다. 포그라운드 작업에 대해 Ctrl+C(SIGINT), Ctrl+Z(SIGTSTP) 시그널을 감지하고 이를 해당 작업에 전달하여, 작업을 중단하거나 종료할 수 있도록 하였다.

* Ctrl+C를 누르면 ^C가 출력되며, 해당 작업은 terminated 상태로 간주되어 job 리스트에서 제거된다.
* 반면, Ctrl+Z를 누르면 해당 작업이 stopped 상태로 전환되어 리스트에 남아 있게 되며, 이후 fg %<job\_id> 명령어로 포그라운드에서 재실행, bg %<job\_id> 명령어로 백그라운드에서 재개할 수 있다.
* 또한 kill %<job\_id> 명령어를 통해 특정 job을 종료할 수도 있다.

이러한 기능들을 통해 사용자는 여러 작업을 동시에 관리할 수 있으며, 실질적인 셸과 유사한 수준의 job control 기능을 경험할 수 있다.

1. 개발 내용

Phase1 (fork )

1. 프로세스 생성

* fork()를 통해 자식 프로세스를 생성하고, 자식 프로세스에서 execvp()를 사용하여 명령어를 실행한다.
* 부모 프로세스는 waitpid()를 통해 자식 프로세스가 종료될 때까지 기다린다.
* Phase2 (pipelining)

1. 파이프라인 설계 및 구현

* 명령어에 |가 포함된 경우, 이를 기준으로 명령어를 좌측과 우측으로 나눈다.
* pipe()와 dup2()를 사용하여 자식 프로세스 간의 입출력 스트림을 연결하였다.

1. 다중 파이프 처리

* 재귀적으로 pipe\_eval() 함수를 호출하여 여러 개의 파이프를 처리할 수 있도록 구현하였다.

예: cmd1 | cmd2 | cmd3는 cmd1의 출력을 cmd2의 입력으로, cmd2의 출력을 cmd3의 입력으로 연결한다.

* Phase3 (background process)

1. 2백그라운드 작업 처리

* 명령어 끝에 &가 포함된 경우 이를 백그라운드 작업으로 인식한다.
* 백그라운드 작업은 부모 프로세스에서 job\_add()를 통해 작업 목록에 추가되며, 부모 프로세스는 해당 작업을 기다리지 않고 다음 명령어를 처리한다.

1. 작업 제어 명령어

* jobs: 현재 실행 중이거나 중지된 백그라운드 작업을 출력한다.
* bg %<job\_id>: 중지된 작업을 백그라운드에서 실행 상태로 전환한다.
* fg %<job\_id>: 백그라운드 작업을 포그라운드로 가져온다.
* kill %<job\_id>: 작업을 종료한다.

1. 시그널 처리

* SIGINT (Ctrl+C)와 SIGTSTP (Ctrl+Z) 시그널을 처리한다.
* SIGINT: 포그라운드 프로세스를 종료한다.
* SIGTSTP: 포그라운드 프로세스를 중지 상태로 전환한다.

이를 위해 sigaction()을 사용하여 시그널 핸들러를 등록하였다.

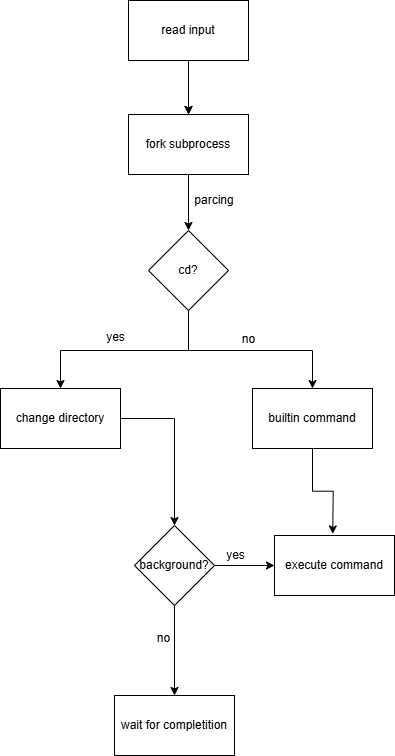
* 1. 개발 방법

eval() 함수에서 명령어를 받고 fork()를 통해 자식 프로세서를 생성하고 exexvp()를 호출하여 명령어 실행한다. 그리고 sigint\_handler() 함수에서 Sigint 시그널을 받아서 포그라운드 작업을 종료하며, sigstp\_handler() 함수에서 sigtstp 시그널을 받아서 포그라운드 작업을 중지하게끔 처리해주었다.

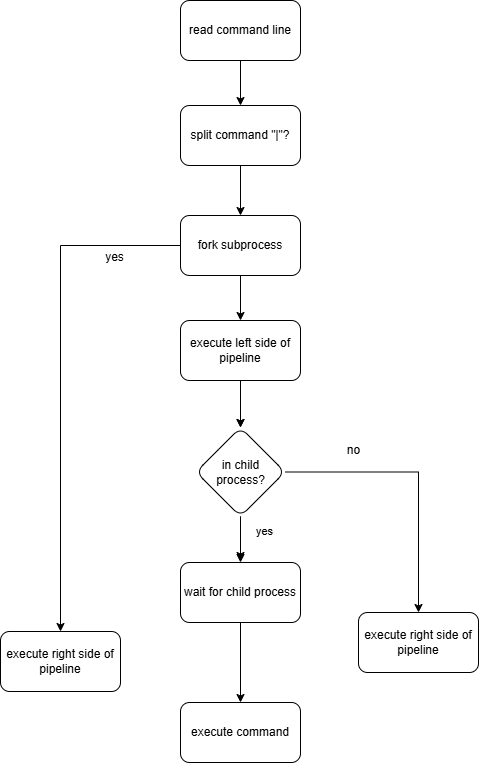
Pipe\_eval()에서 먼저 spit\_pipe()를 사용하여 명령어를 | 기준으로 오른쪽과 왼쪽으로 나누고, 파이프라인 연결을 해준다. 그리고 pipe()와 dup2()와 같은 기존 함수를 불러서 스트림 연결을 한다. 그리고 재귀 호출로 다중 파이픞 처리를 해주었다.

작업을 추가하거나 제거하고 이를 jobs 전역 리스트로 권리하게끔 job\_add() 이랑 job\_remove() 함수를 추가했다. 그리고 bulitin\_command() 함수에서는 ‘jobs, bg, fg, kill’와 같은 명령어를 처리하게끔 추가해주었다. 작업 리스트에서 PID 또는 작업 ID로 작업을 검색이 되도록 job\_get(), job\_getById() 함수를 추가해주었다.

1. 구현 결과 add job to running in background
2. Phase1: Fork



1. Phase2: Pipeline



1. Phase3: Background

