**System Programming Project 2**

담당 교수 : 이영민 교수님

이름 : 김지섭

학번 :20201572

1. **개발 목표**

리눅스 환경에서 작동하는 기본적인 Shell을 C언어를 이용하여 단계별로 구현하였다. Phase 1에서는 기본적인 명령어 실행, Phase 2에서는 파이프라인(|)을 이용한 명령어 연결, Phase 3에서는 백그라운드 실행(&) 및 작업 제어(job control) 기능을 구현하여 점진적으로 완성도를 높이도록 구현하였다.

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

fork()와 execvp()를 사용하여 기본적인 외부 명령어 실행 기능을 구현하였다. waitpid()를 통해 자식 프로세스가 종료될 때까지 부모 프로세스가 대기하도록 하였고 cd, echo와 같은 간단한 내장 명령어 처리 기능도 추가하였다.

1. Phase 2

파이프(|)를 사용하여 여러 명령어를 연결하고, 한 명령어의 출력이 다음 명령어의 입력으로 전달되도록 구현하였다. pipe()을 사용하여 파이프를 생성하고, dup2()를 사용하여 file descriptor를 복제하여 입출력을 redirection 하였다. 이때 파이프로 연결된 명령어들은 parseline() 함수에서 잘라내어 Command 구조체 배열에 저장하고 순차적으로 실행했다.

1. Phase 3

백그라운드 실행(&) 기능을 추가하여 명령어를 백그라운드에서 실행할 수 있도록 하였다. Job 구조체를 정의하여 백그라운드/포그라운드 작업의 상태(실행 중, 정지됨, 종료됨)와 정보를 관리하도록 하였다. 또한 SIGCHLD, SIGINT, SIGTSTP signal handler를 구현하여 자식 프로세스의 상태 변화(종료, 인터럽트, 정지)를 감지하고 처리하도록 하였다. jobs, fg, bg, kill과 같은 작업 제어 내장 명령어 역시 추가 구현하여 사용자가 작업을 관리할 수 있도록 하였다.

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

execute\_command 함수 내에서 fork() 시스템 콜을 호출하여 새로운 자식 프로세스를 생성한다. 이때 부모 프로세스는 fork()의 반환값으로 자식 프로세스의 PID를 받고, 자식 프로세스는 0을 받아 각자 다른 작업을 수행한다. 자식 프로세스는 부모 프로세스와 다른 메모리 공간에서 독립적으로 execvp()를 호출하여 사용자가 입력한 명령어를 실행하며, 부모 프로세스는 자식 프로세스를 waitpid()를 호출하여 종료까지 기다린다.

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

Phase 1의 부모 프로세스는 waitpid()를 호출하여 자식 프로세스가 종료될 때까지 명시적으로 기다립니다. 자식 프로세스가 종료되면 운영체제 커널이 부모 프로세스에게 SIGCHLD 시그널을 보내지만, Phase 1에서는 이 시그널을 처리하는 별도의 핸들러를 등록하지 않았다. 따라서 waitpid() 호출이 block되어 있다가 자식 프로세스가 종료 시 반환되고, 다음 작업 (반복문을 통한 다음 명령 입력 대기)을 진행한다. Phase 3에서는 해당 시그널을 처리하는 핸들러를 등록하여 필요한 처리 (자료구조 상태 최신화 등) 를 하도록 구현하였다.

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

사용자가 파이프(|)를 포함한 명령어를 입력하면, parseline 함수가 |를 구분자로 사용하여 각 명령어를 분리하고, 각 명령어와 그 인자들을 Command 구조체 배열(commands)에 저장한다. execute\_pipeline 함수는 이 commands 배열을 순회하면서 각 명령어에 대해 자식 프로세스를 생성한다. 이때, pipe() 시스템 콜을 사용하여 프로세스 간 통신을 위한 파이프를 생성하고, dup2() 시스템 콜을 사용하여 이전 명령어의 출력(파이프의 읽기 끝)을 현재 명령어의 입력 (stdin)으로, 현재 명령어의 출력(파이프의 쓰기 끝)을 stdout으로 재지정하여 명령 사이의 데이터 흐름을 연결하였다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

execute\_pipeline 함수 내의 반복문은 파이프에 의해 분리된 명령어의 개수(cmd\_count)만큼 실행된다. 각 반복마다 파이프를 생성하고(pipe()), 자식 프로세스를 생성하며(fork()), dup2()를 이용해 입출력을 연결한다. 이 과정을 통해 지정된 MAXCMDS 한도 내에서 여러 개의 파이프라인 명령어를 동적으로 처리할 수 있게 된다. 또한 이때 부모 프로세스는 파이프라인의 마지막 명령어에 해당하는 자식 프로세스의 종료만 waitpid()로 기다리도록 구현하였다.

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

명령어 마지막에 &가 있으면 parseline 함수가 이를 감지하여 bg 플래그를 1로 설정하고 명령어 문자열에서 &를 제거한다. execute\_command 또는 execute\_pipeline 함수에서 자식 프로세스(또는 파이프라인의 마지막 자식 프로세스)를 생성한 후, 부모 프로세스는 bg 플래그가 1이면 sigsuspend()로 자식 프로세스의 종료를 기다리지 않고 즉시 반환하여 다음 명령어를 입력받을 수 있도록 하고, 백그라운드로 실행된 작업의 정보(PID, 명령어, 상태 등)는 add\_job 함수를 통해 Job 구조체에 저장되고 job\_list 배열에서 관리된다.

* 1. **개발 방법**

**- Phase 1**

execute\_command 함수 추가: fork(), execvp(), waitpid()를 사용하여 명령어 실행 및 대기 로직 구현.

parseline 함수 수정: 공백 기준으로 명령어와 인자 분리하여 argv 배열 생성.

execute\_command 내부에 cd, echo 등 내장 명령어 처리 로직 추가.

- **Phase 2**

Command 구조체 정의: 파이프라인 내 각 명령어와 인자 저장용.

commands 전역 배열 및 cmd\_count 변수 추가: 파싱된 명령어 저장 및 개수 관리.

parseline 함수 수정: | 기준으로 명령어 분리 및 commands 배열 채우기.

execute\_pipeline 함수 추가: pipe(), dup2(), fork(), execvp(), waitpid()를 이용한 파이프라인 실행 로직 구현.

eval 함수 수정: 명령어에 | 포함 여부에 따라 execute\_command 또는 execute\_pipeline 호출 분기.

**- Phase 3**

JobStatus enum 및 Job 구조체 정의: 작업 상태(RUNNING, STOPPED, TERMINATED) 및 관련 정보(job\_id, pid, cmd, bg, notified) 관리용.

job\_list 전역 포인터 배열 및 job\_cnt 변수 추가: 현재 실행/정지 중인 작업 목록 관리.

parseline 함수 수정: 명령어 끝의 & 감지 및 bg 플래그 설정.

시그널 핸들러 함수 (sigchld\_handler, sigint\_handler, sigtstp\_handler) 구현: 각 시그널 발생 시 작업 상태 변경 및 정리 로직 수행.

main 함수 내 Signal() 호출 추가: 위 핸들러들을 해당 시그널에 등록.

작업 관리 함수 (add\_job, remove\_job, find\_job\_by\_id, find\_job\_by\_pid, find\_foreground\_job, update\_job\_list, cleanup) 추가: 작업 목록 추가/삭제/검색 및 상태 업데이트, 종료 시 메모리 정리.

execute\_command 내부에 jobs, fg, bg, kill 내장 명령어 처리 로직 추가.

eval, execute\_command, execute\_pipeline, main 등 여러 함수 내부에 sigprocmask, sigsuspend 호출 추가: 시그널을 임시로 블록하거나 특정 시그널을 기다려 race condition 방지 및 안정적인 작업 제어 구현.

1. **구현 결과**
   1. **Flow Chart**
2. **텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

   AI가 생성한 콘텐츠는 부정확할 수 있습니다.Phase 1 (fork)**
3. **텍스트, 스크린샷, 폰트, 그래픽 디자인이(가) 표시된 사진

   AI가 생성한 콘텐츠는 부정확할 수 있습니다.Phase 2 (pipeline)**
4. **텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

   AI가 생성한 콘텐츠는 부정확할 수 있습니다.Phase 3 (background)**