**System Programming Project 2**

담당 교수 : 이영민

이름 : 강유영

학번 : 20231508

1. **개발 목표**

이 프로젝트의 목표는 리눅스 환경에서 동작하는 나만의 쉘 프로그램인 MyShell을 구현하는 것이다. 사용자의 명령어를 입력받아 해석하고, 내부 명령어나 외부 프로그램을 실행하는 과정을 직접 설계하고 구현한다. 이를 통해 fork(), exec(), wait() 등의 시스템 콜과 시그널 처리, 프로세스 제어 등 운영체제의 핵심 개념을 학습할 수 있다. 프로젝트는 기본 명령어 실행, 파이프라인 처리, 백그라운드 작업 및 job control 등 3단계에 걸쳐 기능을 확장해나가는 구조로 이루어져 있다. 궁극적으로는 실제 리눅스 쉘처럼 작동하는 프로그램을 구현하며 시스템 프로그래밍의 원리를 체험하는 데 그 목적이 있다.

1. **개발 범위 및 내용**
   1. **개발 범위**

**Phase 1**

* 쉘이 사용자로부터 명령어를 입력받고 이를 실행
* fork()를 통해 자식 프로세스를 생성하고 execvp()로 명령 실행
* cd, exit은 부모 프로세스에서 직접 처리

**Phase 2**

* 명령어 내의 | 기호를 파싱하여 여러 개의 프로세스를 파이프로 연결
* 각 명령어마다 새로운 자식 프로세스를 생성하고 입출력을 연속적으로 파이프를 통해 연결
* 여러 개의 pipe도 처리 가능

**Phase 3**

* 명령 끝에 &가 붙은 경우 백그라운드 실행
* jobs 명령으로 백그라운드 실행 중인 job 목록 확인
* fg, bg 명령으로 foreground/background 전환
* kill 명령으로 job 종료
* SIGCHLD, SIGINT, SIGTSTP 시그널을 처리하여 적절한 동작 수행
  1. **개발 내용**

**B-1. Phase1 (fork & signal)**

* + 자식 프로세스 생성

- fork() 시스템 콜을 사용하여 자식 프로세스를 생성한다.

- 자식 프로세스는 execvp()를 호출하여 사용자 명령어를 실행한다.

- 부모 프로세스는 자식 프로세스의 PID를 통해 관리 및 대기(waitpid()) 수행한다.

* + 내장 명령어 처리

- cd, exit과 같은 명령은 자식 프로세스를 생성하지 않고 부모 프로세스에서 직접 처리한다.

- cd는 chdir() 호출, exit은 exit(0)로 쉘을 종료한다.

* + 자식 종료 처리

- 자식 프로세스가 종료되면 부모는 waitpid()를 통해 종료 상태를 수집한다. 이는 좀비 프로세스 방지와 정상적인 리소스 해제를 위해 필수적이다.

* + 시그널 흐름의 기반 마련

Phase 1에서는 명시적인 시그널 핸들러는 구현하지 않지만, SIGCHLD 발생 시 waitpid()를 통해 종료를 처리할 수 있는 구조를 구성한다.

**B-2. Phase2 (pipelining)**

* + 파이프라인 구조 설계

- 명령어 입력에 '|' 문자가 포함되어 있을 경우, 이를 기준으로 명령어를 분할한다.

- 각 명령어를 별도의 argv 배열로 파싱하여, 여러 개의 명령을 리스트 형태로 저장한다.

- 명령어의 개수에 따라 다중 프로세스가 연결된 파이프라인을 구성할 수 있도록 설계한다.

* + 구현 방식

- 각 명령어마다 fork()를 통해 자식 프로세스를 생성하고, 입출력을 pipe()와 dup2()를 이용하여 연결한다.

- 첫 번째 명령은 표준 입력을 그대로 사용하고 출력만 파이프로 연결하고, 중간 명령어는 입력과 출력을 모두 파이프에 연결한다.

- 마지막 명령은 파이프 입력만 연결하고 출력은 표준 출력으로 설정한다.

* + 파이프 개수 처리 방식

- 파이프 개수는 명령어 개수 - 1로 계산되며, 루프를 통해 그만큼의 pipe() 호출을 반복한다.

- 파이프 개수에 제한을 두지 않고, 명령어 개수에 따라 동적으로 처리할 수 있도록 설계한다.

- 각 프로세스에서 불필요한 파이프 디스크립터는 close()를 통해 명시적으로 정리하여 자원 누수를 방지한다.

**B-3. Phase3 (background process)**

* + 백그라운드 실행 처리

- 명령어 끝에 '&' 문자가 있는 경우, 해당 명령어는 백그라운드 작업으로 인식한다.

- parseline() 함수에서 '&' 여부를 검사하고, bg 플래그를 설정하여 처리 여부를 구분한다.

* + Job 등록 및 관리

- 백그라운드로 실행되는 프로세스는 add\_job() 함수를 통해 Job 배열에 등록한다.

- 각 Job은 PID, 명령어 문자열, 실행 상태(실행 중/정지), 종료 여부, 백그라운드 여부 등의 정보를 가진다.

* + 시그널 핸들러를 통한 상태 업데이트

- 자식 프로세스 종료 또는 정지 시 SIGCHLD 핸들러에서 해당 Job의 상태를 업데이트한다.

- SIGINT (Ctrl+C), SIGTSTP (Ctrl+Z) 시 포그라운드 프로세스 그룹에 시그널을 전달한다.

* + Job Control 명령어 지원

- jobs: 실행 중이거나 정지된 백그라운드 작업 목록을 출력한다.

- fg <job>: 백그라운드 Job을 포그라운드로 전환한다.

- bg <job>: 정지된 Job을 백그라운드에서 재개한다.

- kill <job>: Job을 종료한다. (SIGKILL 전송)

* + 프로세스 그룹 관리

- 백그라운드 실행 시 setpgid()를 사용하여 각 자식 프로세스를 별도의 그룹으로 설정한다. 이를 통해 시그널 전달 시 프로세스 그룹 단위로 제어할 수 있도록 한다.

* 1. **개발 방법**

**C-1. Phase 1**

myshell.c

* + eval()  
    → 명령어 입력을 parseline()으로 파싱한 후, fork()를 사용해 자식 프로세스를 생성하고, execvp()를 통해 명령어 실행. 부모는 waitpid()로 자식 종료를 대기한다.
  + builtin\_command()  
    → cd, exit 명령을 부모 프로세스에서 직접 실행하도록 구현하였다.
  + parseline()  
    → 문자열을 공백 단위로 잘라 argv[] 배열로 구성하고, 마지막 인자가 '&'일 경우 백그라운드 플래그를 설정한다.

myshell.h

* + MAXARGS 정의
  + eval(), parseline(), builtin\_command() 함수 선언

**C-2. Phase 2**

myshell.c

* + parse\_pipeline()  
    → '|' 기호를 기준으로 명령어를 분할하여, 각 명령어를 argv 배열로 변환한다.
  + execute\_pipeline()  
    → 각 명령어를 순서대로 pipe()와 dup2()를 이용해 연결하며 실행한다. fork()를 사용하여 명령어별 자식 프로세스를 생성하고 파이프 연결한다.
  + free\_pipeline()  
    → parse\_pipeline()에서 동적으로 생성된 명령어 배열을 해제한다.
  + eval()  
    → 기존의 함수에서 명령어에 |가 포함된 경우 parse\_pipeline()과 execute\_pipeline()을 호출하여 파이프라인 처리 수행하도록 수정하였다.

myshell.h

* + MAXCMDS 정의
  + parse\_pipeline(), execute\_pipeline(), free\_pipeline() 함수 선언

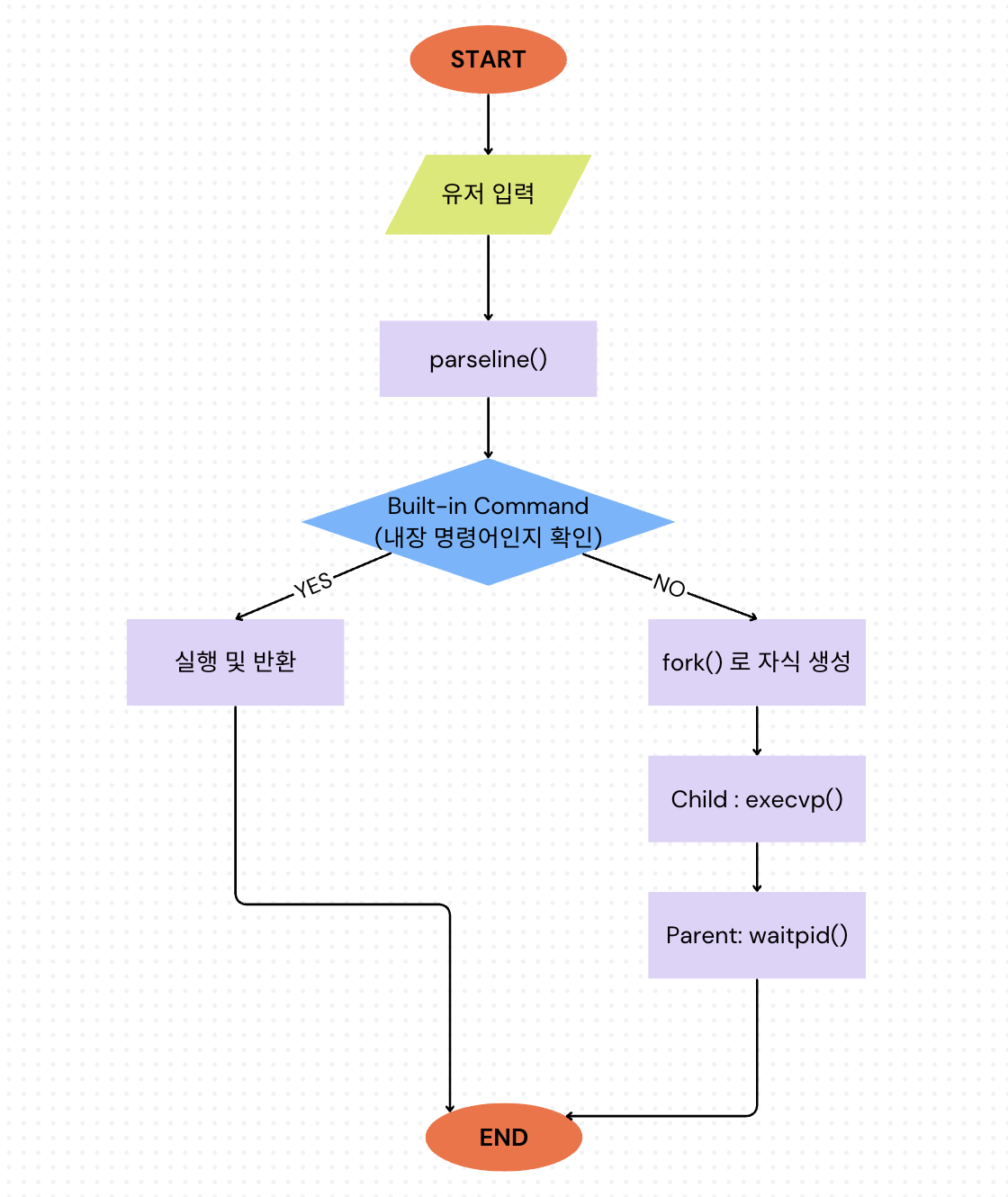
**C-3. Phase 3**

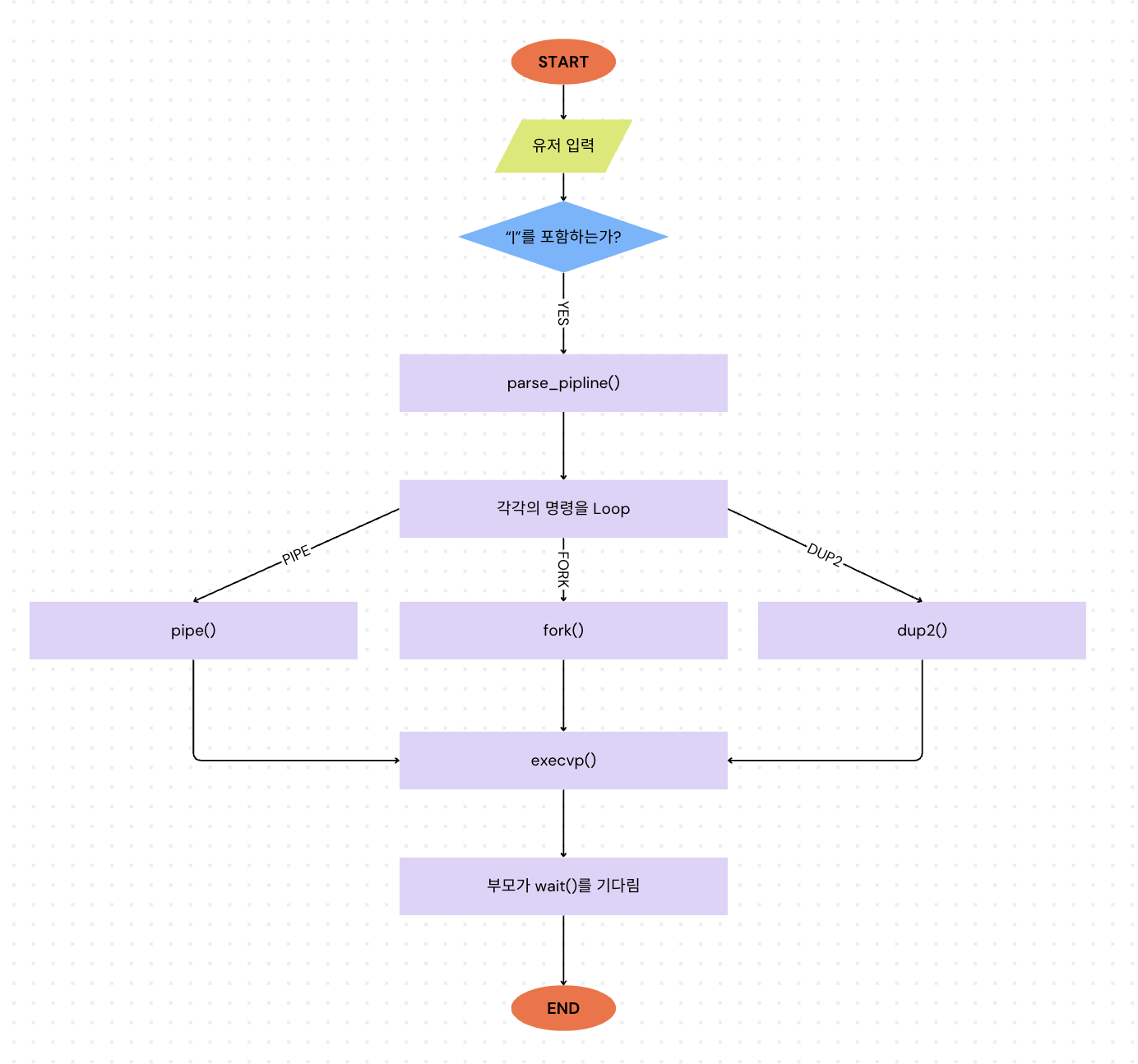
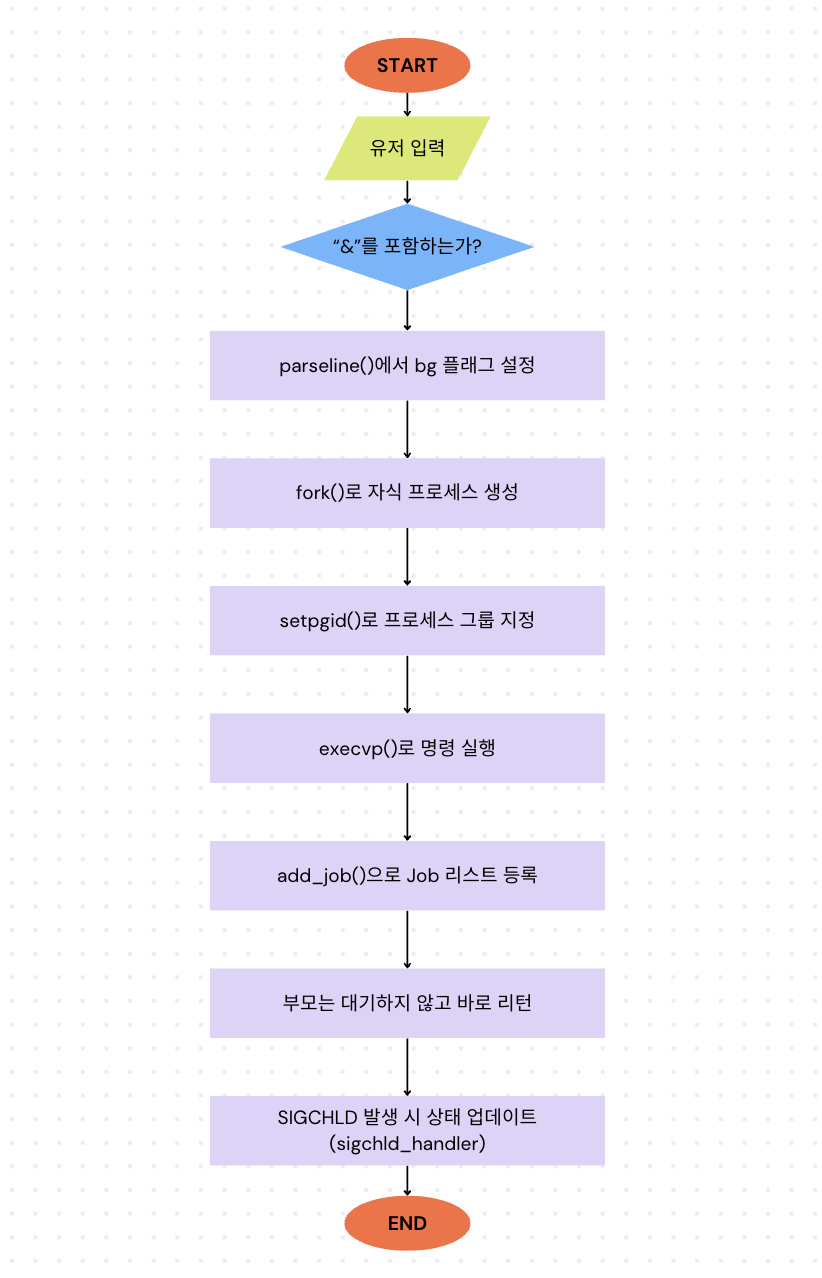
myshell.c

* + Job jobs[MAXJOBS] : Job 리스트 저장 배열이다.
  + int next\_job\_id, fg\_job\_index, last\_plus, last\_minus : Job 상태 관리용 인덱스 전역 변수를 선언하였다.
  + add\_job(), delete\_job(), get\_job\_id(), print\_jobs()  
    → Job 등록, 삭제, ID 조회, 목록 출력의 역할을 하는 함수들이다.
  + sigchld\_handler(), sigint\_handler(), sigtstp\_handler()  
    → 시그널 핸들링 구현하였다. (자식 종료, Ctrl+C, Ctrl+Z 처리)
  + do\_fg(), do\_bg(), do\_kill()  
    → 사용자 명령어 fg, bg, kill 처리 함수를 구현하였다.
  + clean\_quotes()  
    → 명령어 인자에 포함된 쌍따옴표를 제거하는 함수이다.
  + eval()  
    → 명령어가 jobs, fg, bg, kill일 경우 각 함수로 분기 처리하도록 수정하였다. 백그라운드 실행 시 add\_job()으로 등록하고 SIGCHLD 처리로 종료를 감지하고, 포그라운드 Job은 sigsuspend() 방식으로 대기한다.
  + parseline()  
    → &가 인자에 붙어있는 경우까지 고려하여 bg 플래그 정확히 설정하도록 수정하였다.

myshell.h

* + MAXJOBS 정의
  + Job 구조체 정의 ( 백그라운드 Job의 상태 및 식별 정보를 저장하기 위한 구조체)
  + 모든 job control 관련 함수 선언 추가 : add\_job(), delete\_job(), print\_jobs(), sigchld\_handler(), do\_fg()

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

1. **Phase 2 (pipe****line)**
2. **Phase 3 (ba****ckground)**