Multicore Programming Project 1

담당 교수 : 최재승

이름 : 이동석

학번 : 20191619

1. **개발 목표**

본 프로젝트의 주요 목표는 사용자가 직접 리눅스에서의 쉘과 비슷한 동작을 하는 쉘, "MyShell"을 구현하는 것이다.

- 기본 쉘 기능 제공 : 사용자가 명령어를 입력하고 실행 할 수 있는 기본 쉘 기능을 제공한다. 이를 위해 프로그램은 사용자로부터 입력을 받아 파싱하고, 해당 명령어를 실행하거나, exec()함수를 사용해 외부 프로그램을 호출한다.
- 내장 명령어 구현 : 리눅스 쉘에서 사용되는 다양한 내장 명령어를 지원한다. 여기에는 'cd' , 'history', 'jobs', 'kill', 'bg', 'fg', 'quit', 'exit' 가 포함된다.
- 외부 명령어 구현 : 내장 명령어 외에도 외부 명령어 실행을 지원한다. 이때, exec() 함수를 사용하며 여기에는 'ls', 'echo', 'mkdir', 'rmdir', 'cat', 'touch' 등을 포함된다.
- 백그라운드 작업 처리 : 리눅스 쉘과 동일하게 & 커맨드를 사용해 백그라운드 작업을 처리할 수 있다.
- 파이프 지원 : 리눅스 쉘에서는 | 를 통해 여러 명령어를 관리할 수 있다. 이 프로그램 역시 두 개 이상의 명령어를 파이프를 통해 지원한다.
- 시그널 핸들링 : ctrl+z 나 ctrl+c와 같은 입력을 시그널 핸들링을 통해 제어한다.

이러한 개발 목표를 달성하기 위해, 본 프로젝트에서는 다양한 자료구조를 사용했으며, 강의 자료 및 뼈대코드를 참고했다.

2. 개발 범위 및 내용

A. 개발 범위

1. Phase 1

Phase 1은 Phase2,3의 가장 기본 토대가 될 쉘을 구축한다.

1.1 쉘 프롬프트

쉘은 사용자가 exit하기 전까지 계속 명령어를 입력 받아야 하므로 무한 루프를 통해 쉘의 프롬프트를 구현했다. 이때, 현재 위치한 디렉토리의 정보를 나타내기 위해서 getenv()같은 함수를 사용했다. 이를 통해 이후 cd 명령어로 위치를 변경할 때 프롬프트에 나오는 정보도 같이 변하도록 작성했다.

1.2 내부 명령어

phase1에서 exit를 명령어로 입력하면 프로그램은 종료된다.

다음으로, cd의 경우 getenv()함수를 이용해 'cd' 만 입력되는 경우 홈 디렉토리로 이동할 수 있게 구현했다. 또한, 실제 리눅스 쉘에서는 '~'로 시작하는 경우에도 작동하므로 '~'가 들어올 경우 '~'를 홈 디렉토리 경로로 치환하였다. 이때 내부함수 chdir를 사용해 구현했다.

마지막으로, history의 경우 해당 쉘에서 사용자가 입력했던 명령어를 모두 표시하는 명령어이다. 파일을 생성하여 해당 파일에 log를 저장하였다. 이를 통해 쉘을 exit하고 종료하였다가 다시 실행하여도 history를 기억하도록 구현했다. 또한 history 뒤에 숫자n이 오면 가장 최근에 입력한 명령어 n개를 출력하도록 구현했다. 추가적으로 !!와 !#을 구현했다. !!는 가장 최근에 입력한 명령어를 실행한다. 만약, 어떤 문자열과 함께 !!가 들어오는 경우 !!를 치환하여 명령어가 실행되도록 구현했다. 마찬가지로 !#은 #에 해당하는 로그가 존재한다면 해당 커맨드로 치환된다. 이전 명령어와 입력한 명령어가 동일한 명령어라면 중복으로 처리하고 로그를 업데이트하지 않는다.

1.3 외부 명령어

외부 명령어 역시 구현해야한다. 이때, 외부명령어는 위에서 언급한 내부 명령어를 제외한 실행가능한 모든 명령어이다. 이때, 외부명령어는 fork()함수를 통해 부모프로세스에서 자식프로세스를 생성한 후, 자식프로세스에서 실행되도록 작성했다. 또한, exec()함수 중에서 execvp()함수를 사용하여 PATH환경 변수의 디렉

토리에 존재하는 외부명령어를 구현했다.

1.4 문자열 파싱

echo “abc”와 같은 명령어를 “abc”가 아닌 abc 로 출력하기 위해 적절한 문자열 파싱을 진행했다. 따라서, “ “ 이나 ‘ ‘ 로 감싸진 문자열은 해당 문자를 제거하고, 공백 역시 적절하게 제거하여 변수 argv에 저장하였다.

2. Phase 2

본 페이즈에서는 여러 개의 pipe line이 포함된 명령어를 처리한다. 들어온 문자열에 파이프 문자가 있는 지 확인하고, dup와 dup2 함수를 사용해 해당 명령어를 수행한다. ls | grep filename 이나 cat filename | grep -i “test” | sort – r 과 같은 명령어를 수행할 수 있다.

3. Phase 3

이 프로젝트의 마지막 단계에서는 백그라운드에서 프로세스를 실행하는 기능을 구현했다. 해당 기능은 bg, fg, kill, jobs 명령어로 사용할 수 있다. 따라서 리눅스 쉘의 job control을 지원한다. 사용자는 프로세스를 포어그라운드와 백그라운드로 자유롭게 전환 할 수 있고, 작업의 상태 (Running , Suspended, Kill)을 변경할 수 있다. 명령어의 마지막에 &가 포함되어있다면 쉘은 해당 명령을 백그라운드에서 시작한다. 이때, &와 명령어 사이에 공백이 없어도 동작한다.

추가적으로 시그널 핸들링을 구현했다. 어떤 프로세스가 포어그라운드에서 실행 중일 때 사용자가 ctrl+z 를 입력했다면 해당 프로세스를 중지시킨다. 이때 중지시킨 프로세스는 jobs 명령어를 통해 확인할 수 있다. 또한, ctrl+c 를 입력했다면 해당 프로세스를 종료시킨다.

B. 개발 내용

- **Phase1 (fork & signal)**

쉘 프로그램에서 새로운 프로세스를 생성하기 위해 fork()를 사용한다. 이때, fork()함수는 호출하는 부모 프로세스의 복사본인 자식 프로세스를 생성하며, 이

프로세스는 독립적으로 실행된다. 이때, execvp()함수를 자식 프로세스에서 실행하였다. 해당 함수는 새로운 프로그램으로 현재 프로세스의 메모리를 대체하며 성공적으로 호출될 때 해당 함수 이후의 코드는 실행되지 않고 종료된다. 따라서, 자식 프로세스에서 성공적으로 호출에 성공할 경우 자식프로세스는 종료된다. 이때 fork()의 중요한점은 자식 프로세스와 부모 프로세스간의 실행 우선순위가 없다는 점이다. 따라서, Waitpid를 사용해 부모프로세스에서 자식 프로세스가 종료될 때 까지 대기한다.

하지만 위와 같이 프로그램을 작성할 경우 이후 페이즈에서 자식 프로세스를 관리하기 힘들다. 따라서, 시그널을 사용해 구현한다. 메인함수에 SIGCHLD 시그널 핸들러를 등록하면, 자식 프로세스가 종료될 때 커널은 부모 프로세스에게 SIGCHLD 시그널을 보낸다. 이때 sigaction 함수를 사용해 사용자 정의 핸들러를 설정할 수 있다. 따라서, 자식 프로세스를 생성하고 자식 프로세스가 종료될 때 다음과 같은 flow를 가진다.

1. 외부명령어가 실행되면, 부모 프로세스는 자식프로세스를 생성(fork())
2. 부모 프로세스는 외부 명령어가 완료될 때 까지 기다린다.
3. 자식 프로세스에서 해당 명령어를 실행하고 종료하면, 커널은 부모 프로세스에게 SIGCHLD 시그널 전달
4. 시그널이 전달되는 순간 sigchld_handler() 함수 호출
5. 함수 내부에서 종료된 자식 프로세스의 상태 검사
6. 대기중이던 부모 프로세스가 다음 명령어를 대기

- **Phase2 (pipelining)**

우선, 문자열에 파이프가 있는지 확인한다. 만약 파이프가 있다면 파이프라인 처리를 진행한다.

1. strtok() 함수를 사용해 파이프를 기준으로 명령어를 구분한다.
2. 파이프라인에 포함된 명령어의 개수를 num_commands에 저장한다.
3. 각 명령어에 다음을 수행한다.

첫 번째 명령어부터 마지막 이전 명령어까지, pipe()함수를 사용하여 파이프를 생성하고 출력 파일 디스크립터를 설정한다. 마지막 명령어에서는 표준 출력을

복제하고 출력 파일 디스크립터를 설정한다.

명령어를 처리하기 위한 자식프로세스를 생성한다.

자식 프로세스에서는 dup2()함수를 사용하여 표준 입력 및 출력을 파이프 파일 디스크립터로 리디렉션한다. 해당 명령어를 실행하기 위해 사용자정의함수를 호출한다.

부모 프로세스에서는 wait()함수를 사용하여 자식 프로세스가 완료될 때 까지 대기한다.

4. 각 명령어 실행이 끝나면, 파일 디스크립터를 닫는다.

파이프 개수에 따른 처리는 반복문에서 이루어진다. 각 명령어를 순차적으로 처리하여 이전 명령어의 출력이 다음 명령어의 입력으로 사용되도록 파일 디스크립터를 설정했다. 따라서, 파이프 개수에 관계 없이 파이프를 처리할 수 있다.

- **Phase3 (background process)**

1. 우선, 문자열 파싱을 진행하면서 마지막 부분에 '&'기호가 있는지 확인한다.
2. 다음으로 내장 명령어인지 확인한다. 내장 명령어이면서 백그라운드 실행을 요청 받는 다면 자식프로세스를 생성하여 내장 명령어를 실행하고 작업 목록에 추가한다. 하지만, 백그라운드 실행이 아니라면 부모프로세스에서 직접 실행한다.
3. 내장 명령어가 아닌 경우 파이프라인을 확인하고 처리한다.
4. 실행 요청이 없다면 내장 명령어가 아니라면 부모 프로세스는 외부 명령어가 완료될 때 까지 대기하고, 완료 되면 작업 목록에서 제거한다.
5. 백그라운드 실행 요청이 있다면, 외부 명령어의 경우 작업 목록에 백그라운드 작업을 추가하고, 작업 ID와 프로세스를 출력한다.

이때 작업은 Job 구조체를 사용해 관리하며, 시그널 핸들러 함수를 사용해 백그라운드 작업과 관련된 시그널을 처리한다. (상태 변경, 중단 및 종료 등을 관리) 또한, bg와 fg명령어가 들어오게 되면 해당 프로세스의 PID로 해당 작업을 찾고 상태를 변경한다. (SIGCONT)

C. 개발 방법

우선, 다음은 phase1에서 시그널에 관련되어 사용되는 함수이다.

```
/* 시그널 핸들링 관련 함수 */
void sigint_handler(int sig);
void sigchld_handler(int signum);
void sigtstp_handler(int sig);
```

이를 위해서 메인에서는 다음과 같이 시그널 핸들러를 등록해야 한다.

```
Signal(SIGCHLD, sigchld_handler); // set handler for SIGCHLD
Signal(SIGINT, sigint_handler); // set handler for SIGINT
Signal(SIGTSTP, sigtstp_handler); // set handler for SIGTSTP
```

다음으로는 eval() 함수를 수정한다. eval()함수에서 외부 명령어를 실행하므로 다음 코드를 추가한다.

```
sigset_t mask, prev;
Sigemptyset(&mask);
Sigaddset(&mask, SIGCHLD);
```

```
if ((pid = Fork()) == 0) { // Child process
    setpgid(0, 0);
    if (has_pipe) {
        handle_piped_commands(cmdline);
    } else {
        handle_external_command(argv); // Hand
    }
    exit(0);
```

```
add_job(jobs, pid, FG, cmdline);
fg_pid = pid;
setpgid(pid, pid);

Sigprocmask(SIG_BLOCK, &mask, &prev);
while (fg_pid != 0) {
    Sigsuspend(&prev);
}
Sigprocmask(SIG_SETMASK, &prev, NULL);
```

우선, 외부명령어가 실행되면, 부모 프로세스는 자식프로세스를 생성한다. 이제, 부모 프로세스는 외부 명령어가 완료될 때 까지 기다린다. 이때, Sigpromask로 SIGCHLD를 차단한 후, Sigsuspend를 이용해 대기 한다. 자식 프로세스에서 해당 명령어를 실행하고 종료하면, 커널은 부모 프로세스에게 SIGCHLD 시그널 전달한다. 시그널이 전달되는 순간 sigchld_handler() 함수가 호출되고 함수 내부에서 종료된 자식 프로세스의 상태를 검사한다. 이때, fg_pid를 0으로 바꾸어 대기중이던 루프가 종료되고 Sigchld 차단이 해제된다. 이제, 부모 프로세스가 다음 명령어를 대기한다.

사용자 정의 핸들러 SIGCHLD는 다음과 같이 구현되었다. waitpid()함수를 사용하여 상태가 변경된 모든 자식 프로세스의 PID를 가져온다. 이때, 종료되거나, 시그널로 이내 종료된 자식 프로세스의 경우 fg_pid = 0 으로 설정하는 것을 볼 수 있다.

```
void sigchld_handler(int signum) {
    int status;
    pid_t child_pid;

    while ((child_pid = waitpid(-1, &status,  WNOHANG)) > 0) {
        if (WIFEXITED(status) || WIFSIGNALED(status)) {
            remove_job(jobs, child_pid);
            if (child_pid == fg_pid) {
                fg_pid = 0;
            }
        } else if (WIFSTOPPED(status)) {
            int jid = pid2jid(child_pid);
            jobs[jid-1].state = ST;
        } else if (WIFCONTINUED(status)) {
            int jid = pid2jid(child_pid);
            jobs[jid-1].state = FG;
        }
    }
}
```

다음으로 phase2의 pipe를 구현하기 위해 다음 함수가 사용되었다.

```
/* pipe 관련 함수 */
bool check_for_pipe(char *cmdline);
void handle_piped_commands(char *cmdline);
```

```
bool check_for_pipe(char *cmdline) {
    return strchr(cmdline, '|') != NULL;
}
```

우선 check_for_pipe 함수로 입력받은 명령어에 파이프가 있는지 확인한다. 만약 파이프가 있을 경우 아래 함수를 호출한다.

```
void handle_piped_commands(char *cmdline) {
    char *commands[MAXARGS];
    int num_commands = 0;
    char *token = strtok(cmdline, "|");

    while (token) {
        commands[num_commands++] = token;
        token = strtok(NULL, "|");
    }
    commands[num_commands] = NULL;

    int pipe_fds[2];
    int in_fd = dup(STDIN_FILENO);
    int out_fd;

    for (int i = 0; i < num_commands; i++) {
        if (i < num_commands - 1) {
            if (pipe(pipe_fds) < 0 ) {
                perror("pipe error");
                exit(1);
            }
            out_fd = pipe_fds[1];
        } else {
            out_fd = dup(STDOUT_FILENO);
        }

        char buf[MAX_LINE_LENGTH];
        strcpy(buf, commands[i]);
        char *argv[MAXARGS];
        parseline(buf, argv);

        pid_t pid = Fork();
        if (pid == 0) {
            dup2(in_fd, STDIN_FILENO);
            dup2(out_fd, STDOUT_FILENO);
            handle_external_command(argv);
        } else {
            wait(NULL);
        }

        if (in_fd != STDIN_FILENO) close(in_fd);
        if (out_fd != STDOUT_FILENO) close(out_fd);

        in_fd = pipe_fds[0];
    }
}
```

위 함수는 다음과 같이 진행된다. 우선, strtok() 함수를 사용해 파이프를 기준으로 명령어를 구분한다. 파이프라인에 포함된 명령어의 개수를 num_commands

에 저장한다. 명령어의 개수만큼 반복문을 실행한다. 이렇게 작성하면 파이프의 개수와 상관없이 명령어를 수행할 수 있다. 이제 반복문에서 각 명령어에 다음을 수행한다. 우선, 첫 번째 명령어부터 마지막 이전 명령어까지, pipe()함수를 사용하여 파이프를 생성하고 out_fd, 출력 파일 디스크립터를 설정한다. 마지막 명령어에서는 표준 출력을 복제하고 출력 파일 디스크립터를 설정한다. 이제, 명령어를 처리하기 위해 parseline()으로 명령어와 인수를 분리한다. 이후, 자식프로세스를 생성한다. 자식 프로세스에서는 dup2()함수를 사용하여 표준 입력 및 출력을 파이프 파일 디스크립터로 리디렉션한다. (in_fd와 out_fd를 각각 표준입력과 표준 출력에 연결) 해당 명령어를 실행하기 위해 사용자정의함수를 호출한다. 부모 프로세스에서는 wait()함수를 사용하여 자식 프로세스가 완료될 때 까지 대기한다. 각 명령어 실행이 끝나면, 사용된 파일 디스크립터가 표준 입력이나 출력이 아닌 경우 close 함수를 사용해 닫는다. 다음 명령어의 입력을 위해서 in_fd를 pipe_fds[0]으로 설정한다.

마지막으로 phase3를 구현하기 위해서 다음이 추가되었다.

```c
/* phase3 관련 함수 및 변수 */
typedef struct {
    pid_t pid;
    int jid;
    int state;
    char cmdline[MAX_LINE_LENGTH];
} Job;

/* Job states */
#define UNDEF 0 /* undefined */
#define FG 1    /* running in foreground */
#define BG 2    /* running in background */
#define ST 3    /* stopped */

Job jobs[MAXJOBS];
int nextjid = 1;
pid_t fg_pid = 0;
```

우선, Job 구조체를 추가한다. Job은 쉘 프로그램에서 실행되는 모든 프로세스를 관리한다. 해당 구조체에는 프로세스 id, job의 인덱스, 상태, 명령어를 변수로 가진다. 이후, 해당 구조체 배열을 전역변수로 추가하고, 현재 포어그라운드에서 작업중인 pid를 저장할 변수 fg_pid와 다음 jid를 나타낼 nextjid를 변수로 선언한다.

다음으로 이러한 Job을 관리할 함수를 추가한다.

```c
void add_job(Job *jobs, pid_t pid, int state, char *cmdline);
void remove_job(Job *jobs, pid_t pid);
int pid2jid(pid_t pid);
pid_t jid2pid(int jid);
void list_jobs(Job *jobs);
void do_bgfg(char **argv);
```

add_job() : 백그라운드, 포어그라운드 상관없이 프로세스가 실행될 경우 호출되어 job구조체에 해당 프로세스의 정보를 저장한다.

remove_job() : 쉘 프로그램 내에서 실행중이던 프로세스가 종료되거나 kill과 같은 시그널에 의해 종료될 경우 remove_job()을 sigchld_handler()함수에서 호출하

여 구조체에서 제거한다.

pid2jid() : 프로세스의 pid로 해당 프로세스의 정보가 저장되어 있는 job구조체의 jid 정보를 반환한다.

jid2pid() : job 구조체의 jid를 통해 동일한 jid를 가진 프로세스의 pid를 반환한다.

list_jobs() : 사용자가 jobs 명령어를 입력할 경우 현재 쉘에서 실행중이거나, 중지된 프로세스 목록을 보여준다.

do_bgfg() : 사용자가 bg나 fg 명령어를 입력할 때 해당 번호에 해당하는 프로세스를 백그라운드로 전환하거나 포어그라운드로 전환하는 함수이다.

백그라운드의 요청은 parseline()함수에서 리턴값으로 반환된다. 이때 기본 뼈대 코드는 공백으로 구분되지 않은 &를 확인하지 못하므로 다음 코드를 추가했다.

```
if (argc > 0 && argv[argc-1][strlen(argv[argc-1])-1] == '&') {
    argv[argc-1][strlen(argv[argc-1])-1] = '\0';
    bg = 1;
}
```

다음으로, 백그라운드 요청이 확인 되면 내부명령어와 외부명령어인 경우와 외부명령어인 경우로 나뉜다. 우선, 내부 명령어 이면서 백그라운드로 실행해야 할 경우,

```
if (is_builtin) {
    if (bg) {
        // Execute the builtin command in the background
        if ((pid = Fork()) == 0) {
            handle_builtin_command(argv);
            exit(0);
        } else {
            setpgid(pid, pid);
            add_job(jobs, pid, BG, cmdline);
            printf("[%d] %d %s\n", pid2jid(pid), pid, cmdline);
        }
```

fork()로 자식 프로세스를 생성하고, 내부 명령어를 수행할 함수를 호출한다. 이때, 내부 명령어는 execvp()와 다르게 자식 프로세스가 종료되지 않으므로 exit()함수를 추가한다. 또한, add_job으로 해당 프로세스를 추가하고 출력형식에 맞게 프린트한다.

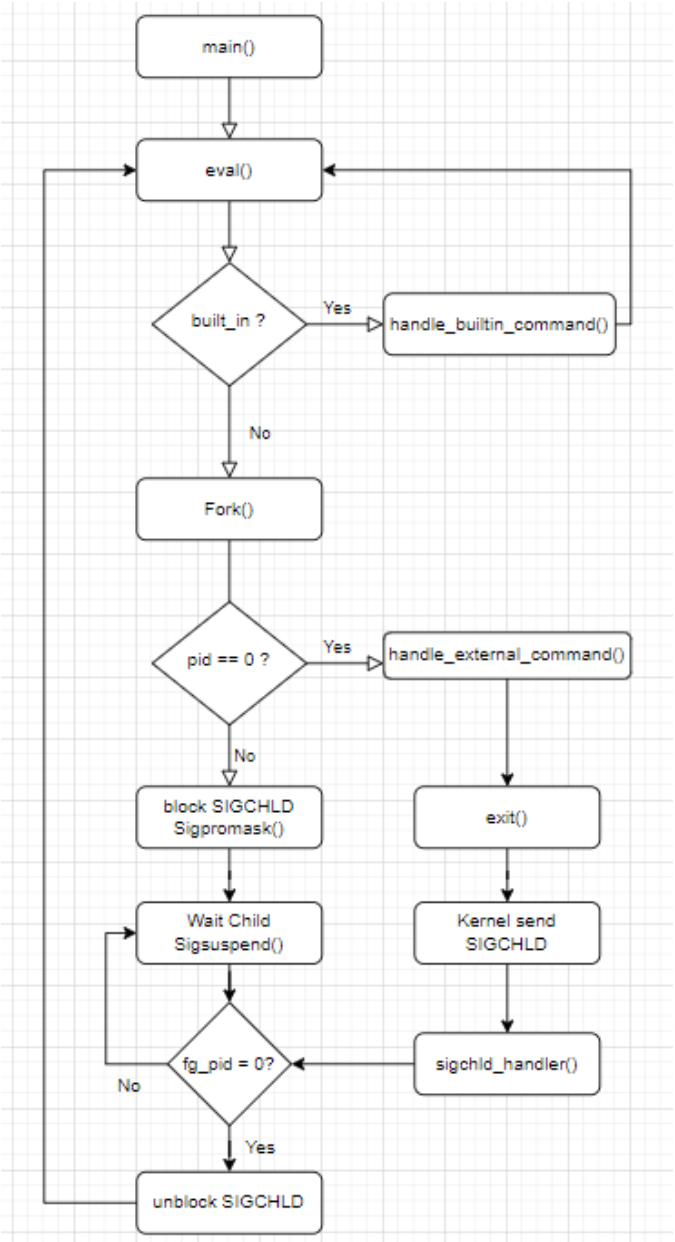
만약 외부 명령어라면 bg와 관계 없이 fork()를 해당 명령어를 수행할 함수를 호출한다. 이후, bg의 요청이면서 내부 명령어가 아니기 때문에 마찬가지로 add_job으로 해당 프로세스를 추가하고 출력형식에 맞게 프린트한다.

```
else {
  if (!is_builtin) {
      // Background external process
      setpgid(pid, pid);
      add_job(jobs, pid, BG, cmdline);
      printf("[%d] %d %s\n", pid2jid(pid), pid, cmdline);
  }
```

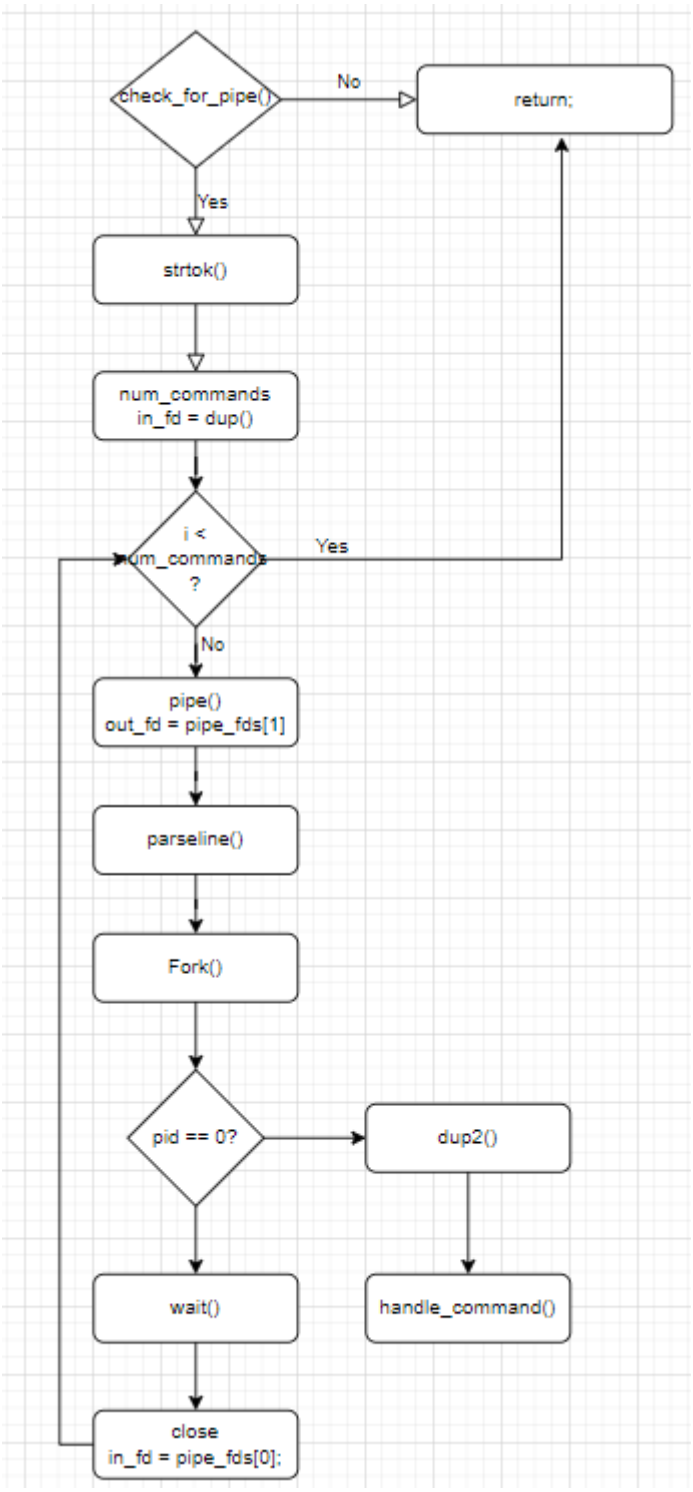
3. 구현 결과

A. Flow Chart

1. Phase 1 (fork)



2. Phase 2 (pipeline)



3. Phase 3 (background)