System Programming Project 2

담당 교수 : 김영재

이름 : 김태규

학번 : 20191243

1. 개발 목표

이 프로젝트는 기본적인 Linux Shell을 구현하여 System Level Process Control, Signal Handling, Inter Process Communication 등의 개념을 익히는 것을 목표로 한다. 프로젝트는 3개의 Phase로 구성되어 있다. Phase 1에서는 Shell의 기본적인 'Read-Evaluation' 과정을 구현한다. Phase 2에서는 파이프 라인을 구현한다. 마지막으로 Phase 3에서는 Shell이 background process를 다룰 수 있도록 구현한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Phase 1

   Linux Shell의 기본적인 'Read Evaluation'을 구현하는 것을 목표로 한다. 이는 'ls', 'ps', 'mkdir', 'touch', 'cat', 'echo'와 같은 기본 command부터 'cd'나 'exit'와 같은 built-in command의 정상 작동을 보장할 수 있어야 한다. 해당 MyShell은 이러한 command뿐만 아니라 'execvp' system call으로 실행 가능한 모든 command를 지원하고 있다.

2. Phase 2

   Linux Shell의 Inter Process Communication를 위한 파이프라인 기능을 구현하는 것을 목표로 한다. 파이프 개수에 제한을 두지 않고, 파이프와 명령어 사이의 공백 여부와 관계없이 자유롭게 파이프라인을 기능을 할 수 있도록 해야 한다.

3. Phase 3

   Signal에 대한 개념을 기반으로 프로세스를 background에서 실행할 수 있도록 하는 것을 목표로 한다. 이 Phase에서는 프로세스를 단순히 background에서 실행하는 것뿐만 아니라 Ctrl+Z, Ctrl+C를 입력하고, 'fg', 'bg', 'kill'과 같은 built-in command를 사용하여 프로세스가 Suspended, Terminated, Foreground, Background 상태로 자유롭게 전환될 수 있도록 해야 한다.

B. 개발 내용

- **Phase1 (fork & signal)**

  ✓ fork를 통해서 child process를 생성하는 부분에 대해서 설명

  먼저 'fork'를 호출하기 전에 'eval' 함수를 통해 argument array 'argv'를 만든다. 이 argument가 built-in command인지 확인한 후, built-in command인 경우 'fork'를 호출하지 않고 Shell 내에서 처리하는 별도의 루틴을 처리한 뒤 다시 돌아간다. Built-in command가 아닌 경우에는 'fork'를 호출하여 새로운 프로세스를 생성한다. 이 새로운 프로세스에서는 SIGINT와 SIGTSTP 신호를 받지 않도록 signal masking을 한다. 이렇게 함으로써 Ctrl+C와 Ctrl+Z에 대한 child 프로세스가 signal을 직접 처리하는 것이 아니라 parent가 signal handler를 통해 대신 처리하도록 한다. Signal masking이 완료되면 'Execvp' 함수를 사용하여 새로운 프로세스에서 실행하고자 하는 command를 인자로 넘겨준다. 'Execvp' 함수를 사용하는 이유는 우리가 다루는 명령 파일이 특정 디렉토리에만 있는 것이 아니기 때문에 'Execve' 대신에 파일 이름만으로도 파일을 찾을 수 있는 'Execvp'를 사용한다.

  ✓ connection을 종료할 때 parent process에게 signal을 보내는 signal handling 하는 방법 & flow

  Parent 프로세스에서는 먼저 Ctrl+C 처리를 위해 job을 Job Queue에 push한다. SIGINT handler에서는 이 프로세스가 실행되는 동안 Shell에서 Ctrl+C가 눌렸는지를 확인하고, Job Queue에서 Foreground 프로세스를 찾아 해당 프로세스에 signal을 보내 kill한다. 종료된 child 프로세스는 SIGCHLD handler에 의해 reaping된다.

  Ctrl+C가 안 눌린 경우, parent 프로세스는 Sigsuspend 함수를 통해 explicit waiting을 한다. SIGCHLD를 받는 handler를 통해 waitpid 함수를 반복해서 호출하여 종료된 프로세스를 reaping한다.

- **Phase2 (pipelining)**

  ✓ Pipeline('|')을 구현한 부분에 대해서 간략히 설명 (design & implementation)

  먼저 cmdline을 Fgets로 받자마자 바로 ' | ' 파이프가 존재하는지 확인한다.

파이프가 있다면 eval 함수에서 file descriptor(int fd[2]) 배열에 파이프라인을 형성한다. eval 함수에서는 cmdline buffering을 ' | '을 만나기 전까지 진행한다. 이렇게 buffering한 command만 처리하고 파이프 이후의 command는 다음 eval 함수를 재귀적으로 호출하여 처리한다. 이에 맞춰서 cmdline의 시작 주소값도 변경한다.

fork로 생성된 child 프로세스는 fd[1]에 그 결과를 출력하고 그 결과는 파이프라인을 통해 fd[0]으로 이어진다.

✓ Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

예시: ls | grep 'my' | tail -2

(1) ls

- fork를 통해 child 프로세스를 새로 만들고 Dup2 함수를 통해 eval 함수의 parameter인 passed_fd로 해당 command의 결과를 입력으로 받을 파일을 대체한다. eval 함수를 처음 호출할 때는 passed_fd에 0(stdin)을 넣어준다. 이때 last_flag는 0이다.

- Dup2를 재호출하여 fd[1]로 출력 파일을 대체

- ls 시행 결과는 fd[1]이 가리키는 파일로 출력된다.

- 이 결과 data는 fd[0]를 향하고, fd[0]를 passed_fd 값으로 다음 eval 함수를 재호출한다.

(2) grep 'my'

- fork를 통해 child 프로세스를 새로 만들고 Dup2를 통해 eval 함수를 호출할 때 받은 입력 변수 passed_fd가 가리키는 파일로 입력 파일을 대체한다. 이때 last_flag는 0이다.

- grep의 입력 data로 ls 명령 시행 결과가 들어온다.

- 다시 Dup2를 재호출해 fd[1]으로 출력 파일을 대체

- grep 명령 시행 결과는 fd[1]이 가리키는 파일로 출력

- 마찬가지로 결과 data는 파이프라인을 통해 fd[0]를 향하고, fd[0]를 passed_fd 값으로 주어 eval 함수 재호출

(3) tail -2

- 마지막 명령이 되면 last_flag가 1이 되고, 이때 새로 fork한 child 프로세스는 Dup2를 한 번만 호출하여 eval 함수를 통해 받은 passed_fd 값으로 입력 파일을 대체한다.

- 이에 따라 tail의 입력 data로 grep 명령 시행 결과 data를 받게 된다.

- 마지막 명령이기 때문에 Dup2를 다시 호출하지 않고, stdout(1)으로 command 결과 data를 출력한다.

- **Phase3 (background process)**

✓ Background ('&') process를 구현한 부분에 대해서 간략히 설명

fork로 프로세스를 새로 생성한 뒤, Phase 1과 2처럼 프로세스 종료를 기다리지 않으면 된다. cmdline에 & 기호가 있는지 eval 함수에서 확인하고 이를 bg라는 flag에 기록한다. Background command라면 프로세스를 wait하는 루틴을 수행하지 않고, 아니라면 이전에 했던 것과 동일하게 child process에 대해 wait을 해주면 된다.

- Ctrl+Z (SIGTSTP signal)

Foreground 프로세스 수행 도중 Ctrl+Z를 누르면 SIGTSTP signal을 Shell에게 보내고 sigtstp handler를 통해 Job Queue에서 state가 foreground인 job에 대해서 pid를 조회한 후 해당 프로세스에게 SIGSTOP signal을 보낸다. 그 후 job의 state를 Stopped로 전환한다.

- Ctrl+C (SIGINT signal)

Ctrl+C도 마찬가지로 foreground 프로세스 수행 도중 Ctrl+C를 누르면 SIGINT signal을 Shell에게 보내고 sigint handler를 통해 Job Queue에서 state가 foreground인 job에 대해서 pid를 조회한 후 해당 프로세스에게 SIGKILL signal을 보낸다. 그 후 해당 job을 Job Queue에서 dequeue한다.

- fg, bg, kill command

먼저 '%job_id' 형식으로 job id를 입력 받고 해당 job을 Job Queue에서 검색한다. fg command의 경우 해당 프로세스를 Sigsuspend를 통해 프로세스가

종료되기를 기다리도록 하고 job의 state를 Foreground로 변경해준다. bg의 경우 SIGCHLD signal을 block하여 child 프로세스가 종료되는 것을 기다리지 않도록 하고 job의 state를 Background로 변경해준다. kill command는 SIGKILL signal을 종료하고자 하는 프로세스에 전달하고 해당 프로세스를 Job Queue에서 dequeue한다.

C. 개발 방법

1. Job 구조체, Job Queue 및 관련 함수

```
typedef struct {
    int idx;
    pid_t pid;
    int state;      // 0: Invalid, 1: Foreground, 2: Background, 3: Stopped
    char cmdline[MAXLINE];
    int is_last;
} Job;

Job job_queue[MAXJOB];
int queue_last;
int queue_size;
```

queue_last: 마지막 valid element의 index

queue_size: queue의 valid element 개수

(1) init_queue

```
void init_queue(void) {
    for (int i = 1; i < MAXJOB; i++) {
        job_queue[i].idx = 0;
        job_queue[i].pid = 0;
        job_queue[i].state = 0;
        job_queue[i].cmdline[0] = '\0';
        job_queue[i].is_last = 0;
    }
}
```

job_queue를 초기화하는 작업을 한다. job_queue의 1번 index부터 채운다.

(2) Enqueue

```c
void Enqueue(pid_t pid, int state, char *cmdline) {
    int i;

    if (!queue_size) {
        i = 1;
        queue_last = i;
    }
    else {
        if ((queue_last + 1) >= MAXJOB)
            unix_error("Enqueue error");

        job_queue[queue_last].is_last = 0;
        queue_last++;
        i = queue_last;
    }

    job_queue[i].idx = i;
    job_queue[i].pid = pid;
    job_queue[i].state = state;
    job_queue[i].cmdline[0] = cmdline[0];

    int j = 1;
    for (int k = 1; cmdline[k]; j++, k++) {
        if (cmdline[k] == '\n' || cmdline[k] == '&'
        || (cmdline[k - 1] == ' ' && cmdline[k] == ' ')) {
            j--;
            continue;
        }
        job_queue[i].cmdline[j] = cmdline[k];
    }
    job_queue[i].cmdline[j] = '\0';
    job_queue[i].is_last = 1;
    queue_size++;
}
```

queue가 비어 있으면 1번 index부터 job을 추가하고, 비어 있지 않으면 valid 프로세스 중 가장 마지막 index를 가진 프로세스 다음 index에 새로운 job을 enqueue한다.

(3) Dequeue

```c
void Dequeue(pid_t pid) {
    for (int i = 1; i < MAXJOB; i++) {
        if (job_queue[i].pid == pid && job_queue[i].state != 0) {
            strcpy(job_queue[i].cmdline, "");
            job_queue[i].pid = 0;
```

```
            job_queue[i].idx = 0;
            job_queue[i].state = 0;

            if (job_queue[i].is_last)
                queue_last--;

            queue_size--;
            return;
        }
    }
}
```

kill 하고자 하는 프로세스의 pid를 parameter로 받아서 해당 pid와 일치하는 job을 job queue에서 찾고 dequeue한다.

(4) Change_state

```
void Change_state(pid_t pid, int state) {
    for (int i = 1; i < MAXJOB; i++) {
        if (job_queue[i].pid == pid) {
            job_queue[i].state = state;
            return;
        }
    }
    unix_error("Change_state error");
}
```

job의 state를 바꿔주는 함수이다.

(5) Search_job

```
Job *Search_job(pid_t pid) {
    for (int i = 1; i < MAXJOB; i++) {
        if (job_queue[i].pid == pid)
            return &(job_queue[i]);
    }
    return NULL;
}
```

입력 받은 pid에 해당하는 job을 job_queue를 탐색하여 그 job의 주소를 반환한다.

2. Read-Evaluation

(1) main

SIGINT, SIGTSTP, SIGCHLD handler 설정

eval 함수 호출하기 전에 pipe가 있는지 확인한다.

```
int main() {
    char cmdline[MAXLINE];

    init_queue();
    Signal(SIGCHLD, sigchld_handler);
    Signal(SIGINT, sigint_handler);
    Signal(SIGTSTP, sigtstp_handler);

    while (1) {
        read_flag = 0;

        Sio_puts("CSE4100-SP-P2> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin)) exit(0);
        read_flag = 1;

        for (int i = 0; cmdline[i]; i++) {
            if (cmdline[i] == '&' && cmdline[i + 1] != '&' && cmdline[i -
1] != '&') {
                cmdline[i] = '\0';
                strcat(cmdline, " &\n");
                break;
            }
        }

        is_pipe(cmdline);
        eval(cmdline, 0, 0);
    }

    return 0;
}
```

(2) eval

- pipe_flag가 1이면 파이프라인을 만든다.

- is_background는 파이프와 백그라운드가 동시에 있는 상황에서, 각 명령을 모두 background로 실행하기 위해 매 재귀 호출 때마다 호출된다.

- bg가 1일 때만, mask1을 사용한 마스킹을 진행합니다. 이는 race를 방지하기

위함이다.

- Fork 이후, mask2로 child 프로세스를 masking하여 SIGINT, SIGTSTP에 영향을 받지 않도록 한다.

- 파이프라인 상황인 경우, last_flag를 통해 마지막 command인 경우와 중간 command인 경우를 구분하고, 사용하지 않는 file descriptor는 Close 한다.

- 경로에 관계 없이 명령 파일을 실행하기 위해 execvp 함수를 사용한다.

- parent 프로세스로 돌아와서, foreground 경우에는 Enqueue 후 reaping 루틴을 밟는다. 이때 is_foreground 함수를 통해 실행 중인 foreground 프로세스가 남아 있을 때만 while 루프가 작동하도록 한다.

- background 상황인 경우, Enqueue 후 background 표시 출력만 한다.

- 이어서 B. 항목에서 설명한 file descriptor 처리를 한다.

- pipeline 상황에서 마지막 command가 아닌 경우, eval 함수를 재귀 호출한다.

```
void eval(char *cmdline, int pipe_fd, int cnt) {
    char *argv[MAXARGS];
    char buf[MAXLINE];
    int bg, index, fd[2], last_flag, temp_fd;
    pid_t pid;
    sigset_t mask_all, mask1, mask2, prev;

    Sigfillset(&mask_all);
    Sigemptyset(&mask1); Sigemptyset(&mask2);
    Sigaddset(&mask1, SIGCHLD); Sigaddset(&mask1, SIGINT); Sigaddset(&mask1,
SIGTSTP);
    Sigaddset(&mask2, SIGINT); Sigaddset(&mask2, SIGTSTP);

    if (pipe_flag) {
        Pipe(fd);
    }

    last_flag = cmd_buffer(cmdline, buf, &index);
    parseline(buf, argv);
    bg = is_background(cmdline);

    if (!argv[0]) {
        return;
    }
```

```
    if (!builtin_command(argv)) {   // check if argv is the built-in-command
such as cd
        if (bg) {
            Sigprocmask(SIG_BLOCK, &mask1, &prev);
        }

        /* Child Process */
        if ((pid = Fork()) == 0) {
            Sigprocmask(SIG_SETMASK, &prev, NULL);
            Sigprocmask(SIG_BLOCK, &mask2, &prev);

            if (!last_flag) {       // if this cmd is not last cmd
                Close(fd[0]);
                Dup2(pipe_fd, STDIN_FILENO);
                Dup2(fd[1], STDOUT_FILENO);
            }
            else {                  // if this cmd is the last cmd
                Dup2(pipe_fd, STDIN_FILENO);
            }

            Execvp(argv[0], argv);
        }

        /* Parent Process */
        if (!bg) {  /* Foreground Process */
            Sigprocmask(SIG_BLOCK, &mask_all, &prev);
            Enqueue(pid, 1, cmdline);

            PID = 0;
            while (!PID) {
                if (is_foreground()) {
                    Sigsuspend(&prev);
                }
                else {
                    break;
                }
            }
        }

        else {      /* Background Process */
            Job *temp;
            Sigprocmask(SIG_BLOCK, &mask_all, NULL);
            Enqueue(pid, 2, cmdline);
            temp = Search_job(pid);

            if (!cnt) {
                Sio_printBGjob(temp->idx, pid);
            }
```

```
        }

        /* Common for Foreground and Background */
        if (!last_flag) {
            Close(fd[1]);
            Close(pipe_fd);
        }
        else {
            if (!cnt) {
                temp_fd = 1;
            }
            else {
                temp_fd = 0;
            }
            Dup2(STDIN_FILENO, pipe_fd);
            Dup2(STDOUT_FILENO, temp_fd);
        }
        Sigprocmask(SIG_SETMASK, &prev, NULL);

        if (!last_flag)
            eval(cmdline + index + 1, fd[0], cnt + 1);
    }
    return;
}
```

(3) cmd_buffer

' | ' 문자를 만나면 buffering을 멈춘다. 또한 따옴표 안에 있는 띄어쓰기 정보를 잃지 않기 위해 -1이라는 값으로 임시로 대체하고 나중에 parseline 함수에서 다시 띄어쓰기로 복구된다.

```
int cmd_buffer(char *cmdline, char *buf, int *index) {
    int i = *index;
    int j;
    int last_flag = 1;

    for (i = 0; cmdline[i]; i++) {
        if (cmdline[i] == '\'') {
            cmdline[i] = ' ';
            for (j = i + 1; cmdline[j] != '\'' && cmdline[j]; j++) {
                if (cmdline[j] == ' '){
                    cmdline[j] = -1;
                }
            }
            cmdline[j] = ' ';
        }
```

```
        else if (cmdline[i] == '\"') {
            cmdline[i] = ' ';
            for (j = i + 1; cmdline[j] && cmdline[j] != '\"'; j++) {
                if (cmdline[j] == ' ') {
                    cmdline[j] = -1;    // -1 will be returned with ' ' in
parseline function
                }
            }
            cmdline[j] = ' ';
        }

        else if (cmdline[i] == '|') {
            last_flag = 0;
            break;
        }
        buf[i] = cmdline[i];
    }
    buf[i] = '\0';

    *index = i;
    return last_flag;
}
```

(4) built-in command

- cd command

```
int cd_command(char **argv) {
    if (argv[1] == NULL || strcmp(argv[1], "~") == 0)
        return chdir(getenv("HOME")) + 1;
    if (chdir(argv[1]) < 0)
        printf("No such directory\n");

    return 1;
}
```

- jobs command

```
int jobs_command(void) {
    for (int i = 1; i < MAXJOB; i++) {
        if (job_queue[i].pid) {
            printf("[%d] ", i);

            if (job_queue[i].state == 2) {
                printf("Running %s\n", job_queue[i].cmdline);
```

```
                continue;
            }
            else if (job_queue[i].state == 3) {
                printf("Stopped %s\n", job_queue[i].cmdline);
                continue;
            }
        }
    }
    return 1;
}
```

- fg / bg / kill command

B. 항목에서 기술했던 그대로 구현하였다.

```
int fg_command(char **argv) {
    int idx;
    pid_t target_pid;
    sigset_t mask, prev;

    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);
    Sigprocmask(SIG_BLOCK, &mask, &prev);

    if (argv[1] == NULL || argv[1][0] != '%') {
        printf("bash: fg: current: no such job\n");
        return 1;
    }
    argv[1] = 1 + &(argv[1][0]);
    idx = atoi(argv[1]);
    if (!(idx > 0 && idx < MAXJOB)) {
        printf("bash: fg: %d: no such job\n", idx);
        return 1;
    }
    target_pid = job_queue[idx].pid;
    if (job_queue[idx].state == 0) {
        printf("bash: fg: %d: no such job\n", idx);
        return 1;
    }

    Sio_printjob(idx, job_queue[idx].cmdline);
    Kill(target_pid, SIGCONT);
    Change_state(target_pid, 1);

    PID = 0;
    while (!PID) {
        if (is_foreground())
```

```
            Sigsuspend(&prev);
        else
            break;
    }
    Sigprocmask(SIG_SETMASK, &prev, NULL);
    return 1;
}
```

```
int bg_command(char **argv) {
    int idx;
    pid_t target_pid;
    sigset_t mask, prev;

    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);
    Sigprocmask(SIG_BLOCK, &mask, &prev);

    if (!argv[1] || argv[1][0] != '%') {
        printf("bash: bg: current: no such job\n");
        return 1;
    }

    argv[1] = 1 + &(argv[1][0]);
    idx = atoi(argv[1]);
    if (!(idx > 0 && idx < MAXJOB)) {
        printf("bash: bg: %d: no such job\n", idx);
        return 1;
    }
    target_pid = job_queue[idx].pid;
    if (job_queue[idx].state == 0) {
        printf("bash: bg: %d: no such job\n", idx);
        return 1;
    }

    Sio_printjob(idx, job_queue[idx].cmdline);
    Kill(target_pid, SIGCONT);
    Change_state(target_pid, 2);

    Sigprocmask(SIG_SETMASK, &prev, NULL);
    return 1;
}
```

```
int kill_command(char **argv) {
    int idx;
    pid_t target_pid;
    sigset_t mask, prev;

    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);
    Sigprocmask(SIG_BLOCK, &mask, &prev);

    if (argv[1] == NULL || argv[1][0] != '%') {
        printf("kill: usage: kill [-s sigspec | -n signum | -sigspec] pid |
jobspec ... or kill -l [sigspec]\n");
        return 1;
    }

    argv[1] = 1 + &(argv[1][0]);
    idx = atoi(argv[1]);
    if (!(idx > 0 && idx < MAXJOB)) {
        printf("bash: kill: %d: no such job\n", idx);
        return 1;
    }
    target_pid = job_queue[idx].pid;
    if (job_queue[idx].state == 0) {
        printf("bash: kill: %d: no such job\n", idx);
        return 1;
    }

    Kill(target_pid, SIGKILL);
    printf("[%d] Terminated %s\n", idx, job_queue[idx].cmdline);
    Dequeue(target_pid);

    Sigprocmask(SIG_SETMASK, &prev, NULL);
    return 1;
}
```

(5) Signal handler

Signal hanler 함수에서는 async-signal-safety 함수만 사용하였고, 모든 signal을 임시로 block한다.

- sigchld_handler

```
void sigchld_handler(int sig) {
    int olderrno = errno;
    int status;
```

```
    sigset_t mask_all, prev;

    Sigfillset(&mask_all);
    while ((PID = waitpid(-1, &status, WNOHANG)) > 0) {
        Sigprocmask(SIG_BLOCK, &mask_all, &prev);

        if (status == SIGPIPE || WIFEXITED(status))
            Dequeue(PID);

        Sigprocmask(SIG_SETMASK, &prev, NULL);
    }

    if (!((PID == 0) || (PID == -1 && errno == ECHILD)))
        Sio_error("waitpid error");
    errno = olderrno;
}
```

남아 있는 종료된 프로세스를 모두 확인하고자 waitpid를 while문의 조건으로 설정하였다.

WNOHANG은 모든 종료된 프로세스를 받아들이도록 한다.

SIGPIPE 상태이거나 WIFEXTED(status) 체크를 통해 정상적으로 프로세스가 종료됐거나 background pipeline 상황에서 비정상 출력됐을 때만 handler에서 dequeue를 해준다. Ctrl+C나 kill과 같은 상황에서는 해당 부분에서 dequeue한다.

- sigint handler

```
void sigint_handler(int sig) {
    int olderrno = errno;
    sigset_t mask_all, prev;
    Sigfillset(&mask_all);
    Sigprocmask(SIG_BLOCK, &mask_all, &prev);

    if (!read_flag) {
        Sio_puts("\nCSE4100-SP-P2> ");
    }
    else {
        Sio_puts("\n");
    }

    for (int i = 1; i < MAXJOB; i++) {
        if (job_queue[i].state == 1) {
            Kill(job_queue[i].pid, SIGKILL);
            Dequeue(job_queue[i].pid);
```

```
        }
    }
    Sigprocmask(SIG_SETMASK, &prev, NULL);

    errno = olderrno;
}
```

Foreground 프로세스만 SIGINT에 반응하도록 하였다.

- sigtstp handler

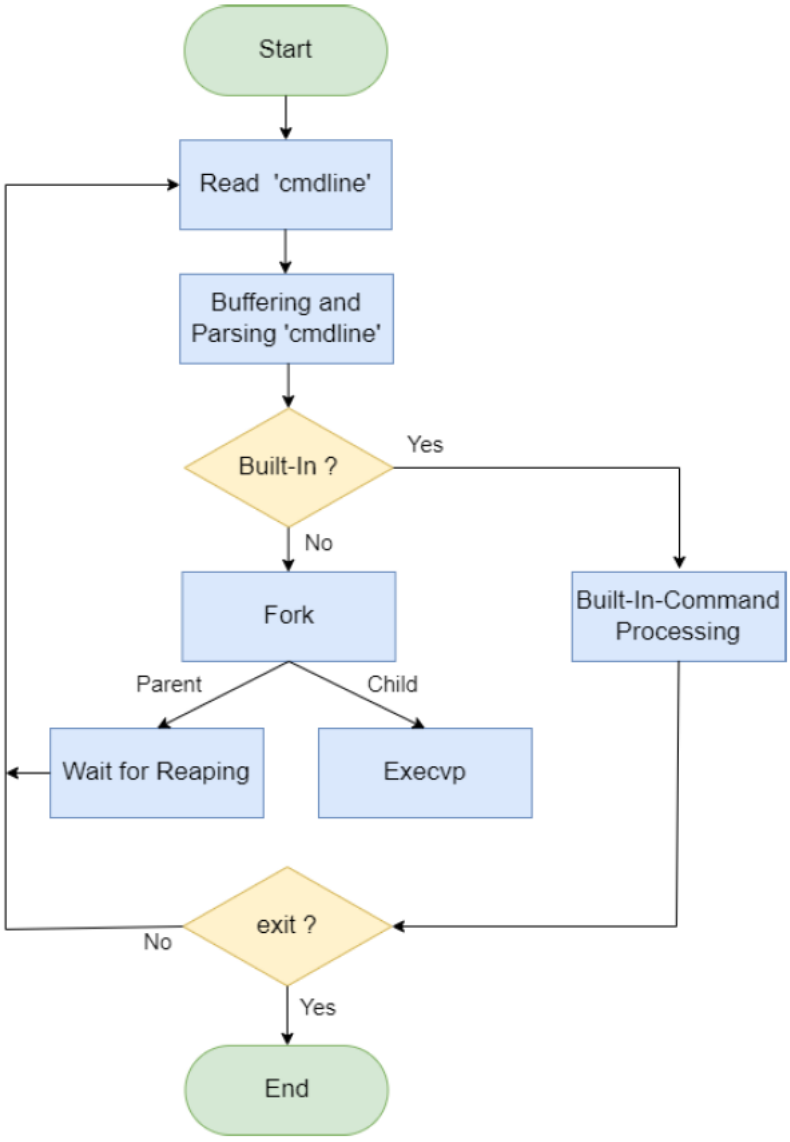
```
void sigtstp_handler(int sig) {
    int olderrno = errno;
    sigset_t mask_all, prev;
    Sigfillset(&mask_all);
    Sigprocmask(SIG_BLOCK, &mask_all, &prev);

    Sio_puts("\n");
    for (int i = 1; i < MAXJOB; i++) {
        if (job_queue[i].state == 1) {
            Kill(job_queue[i].pid, SIGSTOP);
            job_queue[i].state = 3;
        }
    }
    Sigprocmask(SIG_SETMASK, &prev, NULL);

    errno = olderrno;
}
```

마찬가지로 foreground 프로세스만 SIGTSTP에 반응하도록 하였다.

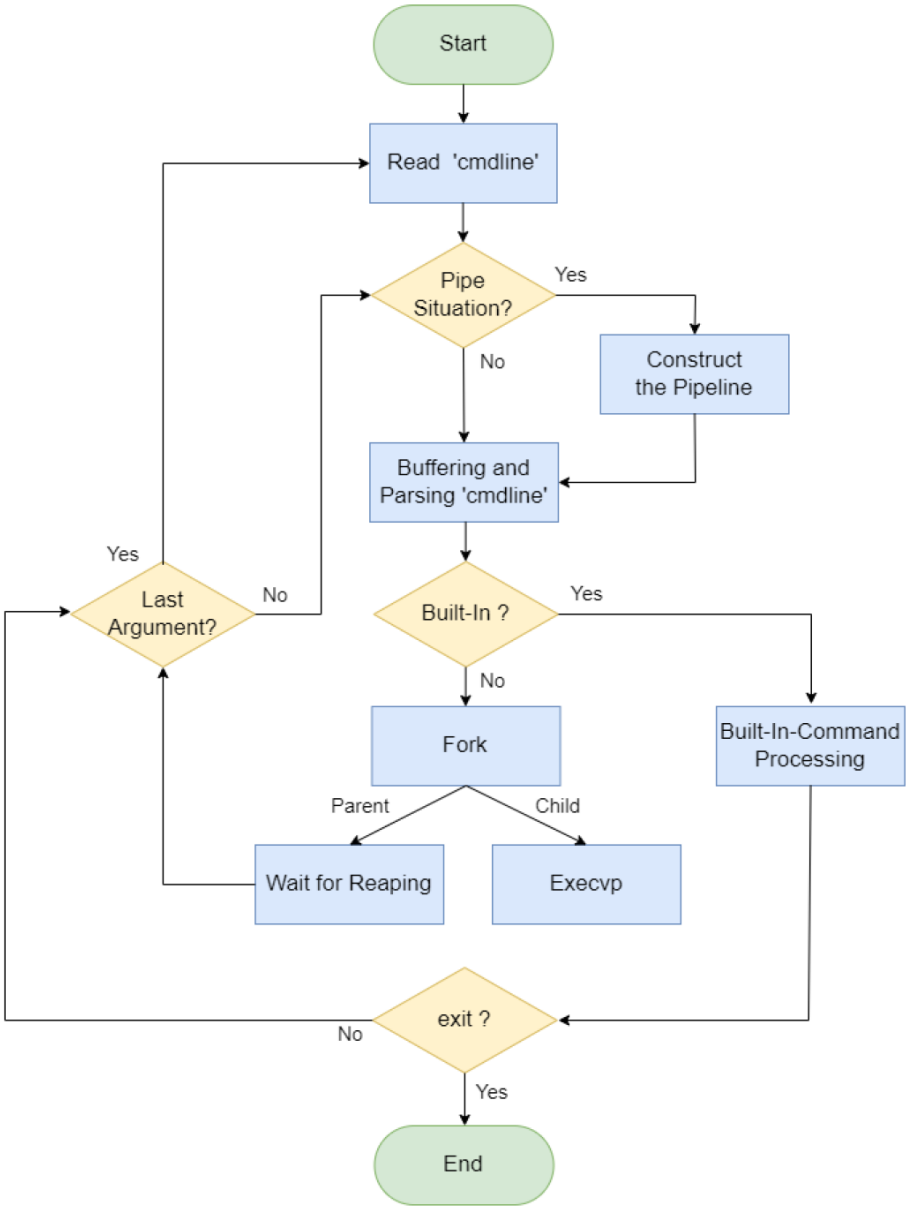
3. 구현 결과

A. Flow Chart

1. Phase 1 (fork)



2. Phase 2 (pipeline)



3. Phase 3 (background)