System Programming Project 2

담당 교수 : 김영재

이름 : 윤준서

학번 : 20211558

1. 개발 목표

1. Phase 1 : shell 생성 및 구현
2. Phase 2 : pipeline 구현
3. Phase 3 : signal handler와 background process 구현

2. 개발 범위 및 내용

A. 개발 범위

1. Phase 1 : shell을 실행하고 명령어(ex. cd, ls, mkdir, rmdir, cat 등)를 실제 리눅스 환경에 적용시킨다.
2. Phase 2 : Phase 1의 shell에 복합 명령어 구조인 pipeline을 적용시킨다.
3. Phase 3 : Phase 2의 shell에 &를 통해 background에서 명령을 실행하는 기능과 background의 명령들을 관리하는 기능을 적용시킨다.

B. 개발 내용

- **Phase1 (fork & signal)**
  - ✓ fork를 통해서 child process를 생성하는 부분에 대해서 설명
    - ◆ stdin에서 명령어를 입력받아 parseline()을 통해 명령어를 파싱 후, argv에 저장한다. 해당 명령어가 리눅스 명령어가 아닌 경우 파일 실행을 위해 fork()를 통해 child process를 생성한다.
  - ✓ connection을 종료할 때 parent process에게 signal을 보내는 signal handling 하는 방법 & flow
    - ◆ Phase 3에서 sigint, sigtstp 등의 signal을 parent process에 전달하여 이를 종료시킨다.
- **Phase2 (pipelining)**
  - ✓ Pipeline('|')을 구현한 부분에 대해서 간략히 설명 (design & implementation)

◆ 문자열을 파싱하는 과정에서 '|'과 '&'이 나올 경우 argv[]에 하나의 명령어처럼 추가한다. 즉 argc를 늘리면서 argv[]에 '|'과 '&'을 따로 저장한다. 명령어에 ""이 포함되는 경우를 고려하여, ' " '이 나올 경우 다음 ' " '이 나올 때까지 같은 문자열로 간주하여 argv[]의 한 element에 모두 저장한다.

✓ Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

◆ parseline()에서 분할한 pipeline을 multiple()에서 실행할 때, stdin을 fd[0]에 저장하고 builtin_command()를 실행한다. 명령어 탐색 중 '|'이 나올 경우 뒤의 명령어를 fd[1]에 저장하여 child process의 stdin으로 간주하고 똑같이 builtin_command()를 실행한다. while loop를 통해 이를 계속 반복하면 argv가 NULL이 나올 때까지 pipeline을 통해 복합 명령어를 실행시킬 수 있다.

- **Phase3 (background process)**

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

◆ 명령어에 '&'이 나올 경우 Phase 1을 통해 background process 자체는 구현이 되어있다. 그러나 이를 관리하기 위해 여러 siganl_handler()들을 구현한다. 각각의 signal을 통해 shell 자체가 아닌 background process만을 중지시키도록 한다. 이를 위해 process들의 정보를 저장하는 jobs[]를 만들어 관리한다. 임의의 process를 중지시키기 위해 해당 process의 pid를 -1로 바꿔야 하는데, 각 process의 pid 정보를 jobs[]에 저장해놓기 때문에 손쉽게 수정할 수 있다.

C. 개발 방법

1. Phase 1 :

builtin_command()에서 cd 명령어 처리는 다음과 같다.

```
if (strcmp(argv[0], "cd") == 0) {
    char path[MAXBUF];
    pid_t pid;
    int channel;

    if (*argv[1] == '~') {
        strcpy(path, getenv("HOME"));
        strcat(path, argv[1] + sizeof(char));
    }
    else strcpy(path, argv[1]);

    channel = chdir(path);
    if (channel != 0) printf("cd: %s: No such file or directory\n", argv[1]);

    return 1;
}
```

이동하고자 하는 디렉토리 위치를 path 문자열에 저장하고 디렉토리 위치를 변경하는 chdir()함수를 사용한다. 홈 디렉토리의 경우 '~'를 입력하면 이동하도록 설계했다.

eval()에서 child process 처리 과정은 다음과 같다.

```
if (!builtin_command(argv)) { //quit -> exit(0), & -> ignore, other -> run
    if (pid = Fork() == 0) {
        if (execvp(argv[0], argv) < 0) {   //ex) /bin/ls ls -al &
            printf("%s: Command not found.\n", argv[0]);
            exit(0);
        }
    }

    /* Parent waits for foreground job to terminate */
    if (!bg) {
        int status;
        if (Waitpid(pid, &status, 0) < 0) unix_error("waitpid error\n");
    }
    else printf("%d %s", pid, cmdline); //when there is backgrount process!
}
```

builtin_command()의 결과를 받아, 일반 명령어의 경우 execvp()를 통해 리눅스 명령어를 받아서 실행하도록 설계했다. 이때 Fork()를 통해 child process에서 실행되도록 하여 shell 명령을 반복할 수 있도록 했다. 생성된 child process는 Waitpid()를 통해 올바르게 reap하여 zombie process가 남지 않도록 했다.

2. Phase 2 :

parseline()에서 문자열을 파싱하는 과정은 다음과 같다.

```
while ((delim = strchr(buf, ' '))) {
    /*****/
    if (*buf == '\"') {
        *buf++ = '\0';
        argv[argc++] = buf;
        while (buf != end && *buf != '\"') buf++;
        if (buf != end) *buf++ = '\0';
    }
    else {
        *delim = '\0';
        while (delim != buf) { // replace '|', '&' to '\0' and append to argv
            if (strncmp("|", buf, 1) == 0) {
                argv[argc++] = "|";
                *buf = '\0';
                buf++;
            }
            else if (strncmp("&", buf, 1) == 0) {
                argv[argc++] = "&";
                *buf = '\0';
                buf++;
            }

            if (delim != buf) { // command after '|', '&'
                argv[argc++] = buf;
                while (delim != buf && '&' != *buf && '|' != *buf) buf++;
            }
        }
    }
    /*****/

    buf = delim + 1;
    while (*buf && (*buf == ' ')) /* Ignore spaces */
        buf++;
}
```

delim을 '₩0'으로 저장한다. *buf를 읽으면서 문자열 파싱을 진행한다. ' " '이 올 경우 다음 ' " ' 사이의 명령어를 한 그룹으로 받아들인다. 그리고 '|' 또는 '&'이 올 경우 argv에 저장하고 해당 부분을 '₩0'으로 바꾼다. 그리고 buf가 delim으로 갈 때까지의 문자열을 파싱한다.

eval()에서 명령어를 실행하던 부분을 새로운 함수 multiple()을 정의하여 따로 구현했다. 복합 명령어(pipeline)를 수행하는 multiple()은 다음과 같다.

```c
void multiple(char **argv, int argc) {
    int fd[2];
    if (pipe(fd)) putchar('error');

    Dup2(0, fd[0]);
    Dup2(1, fd[1]);

    while (argv[argc] != NULL) { /* execute multiple commands */
        int idx = 0, curr = argc;
        char *command[MAXARGS];

        for (idx = 0; argv[curr] != NULL; idx++, curr++) {
            command[idx] = argv[curr];
            if (*argv[curr] == '|' || *argv[curr] == '&') break;
        }
        command[idx] = NULL;

        if (argv[curr] == NULL) {
            if (!builtin_command(command)) {
                pid_t pid;
                if ((pid = Fork()) == 0) {
                    if (execvp(command[0], command) < 0) {
                        printf("%s: Command not found.\n", command[0]);
                        exit(0);
                    }
                }
                int status;
                if (Waitpid(pid, &status, 0) < 0) unix_error("waitpid error\n");
            }
        }
```

stdin을 저장하기 위한 fd[]를 생성한다. Dup2()를 통해 이를 저장한다. 명령어가 마지막 부분일 경우('|'이 더 이상 존재하지 않는 경우) Phase 1과 같은 과정을 진행한다. 명령어가 중간일 경우('|'이 존재하는 경우)는 다음과 같다.

```c
        else {
            if ('|' == *argv[curr++]) {
                if (!builtin_command(command)) {
                    pid_t pid;
                    int fd2[2];
                    if (pipe(fd2)) putchar('error');

                    if ((pid = Fork()) == 0) {
                        close(fd2[0]);
                        Dup2(fd2[1], STDOUT_FILENO);
                        close(fd2[1]);

                        if (execvp(command[0], command) < 0) {
                            printf("%s: Command not found.\n", command[0]);
                            exit(0);
                        }
                    }

                    int status;
                    pid_t wpid = Waitpid(pid, &status, 0);
                    close(fd2[1]);
                    Dup2(fd2[0], STDIN_FILENO);
                    close(fd2[0]);
                }
            }
        }
        argc = curr;
    }

    dup2(fd[0], 0);
    close(fd[0]);
    dup2(fd[1], 1);
    close(fd[1]);
}
```

명령어 탐색 중 '|'이 나올 경우 이전까지 읽어 들인 명령어를 child process를 통해 실행한다. 남은 명령어를 stdout에 출력, parent process는 이를 stdin으로 읽어 들여 다시 fd[]에 저장한다. 이를 명령어를 끝까지 탐색할 때까지 반복한다. 마지막에 dup2()와 close()를 통해 stdin, stdout을 다루는 실행을 중지한다.

3. Phase 3 :

Phase 3에서 추가한 데이터 구조와 함수는 다음과 같다.

```
int job_index;
enum { Run, Stop, Terminate, Foreground };
char *state[4] = { "run", "stop", "terminate", "foreground" };

struct Job {
    pid_t pid;
    char command[MAXBUF];
    int state;
} Job;
struct Job jobs[MAXJOBS];
```

job의 상태와 정보를 저장하기 위해 Job 구조체와 그 배열을 정의했다. 요소로는 pid, 명령어를 저장하는 command[], 그리고 현재 상태를 저장하는 state가 있다. 상태를 다룰 때 strcpy를 사용하지 않기 위해 enum type으로 상태를 재정의했다. run은 process가 bg에서 실행중임을, stop은 process가 중지됨을, terminate는 process가 terminate 되었음을, 그리고 foreground는 process가 fg에서 실행중임을 의미한다.

```
void initJob();
int addJob(pid_t pid, int state, char *command);
void deleteJob(int index);
void printJob();

void sig_int_handler(int signal);
void sig_tstp_handler(int signal);
void sig_chld_handler(int signal);
void sig_quit_handler(int signal);
```

initJob : Job 배열인 jobs[]를 초기화한다.

addJob : jobs[]에 새로 실행한 Job을 추가한다.

deleteJob : jobs[]에 임의의 index의 Job을 삭제(실행 중지)한다.

printJob : jobs[]에 있는 모든(실행중인) Job의 정보를 출력한다.

sig_int_handler : fg에 실행중인 process와 같은 pgid를 가지는 모든 process에 SIGINT를 보낸다.

sig_tstp_handler : fg에 실행중인 process와 같은 pgid를 가지는 모든 process에 SIGTSTP를 보낸다.

sig_chld_handler : 중지된 child process를 waitpid()로 reap한다. child process는 상태가 세 가지로 나뉘는데, WIFEXITED일 경우 해당 Job을 삭제한다. WIFSIGNALED일 경우 signal 종료 메세지를 보내고 Job을 삭제한다. WIFSTOPPED일 경우 process의 state를 stop으로 설정한다.

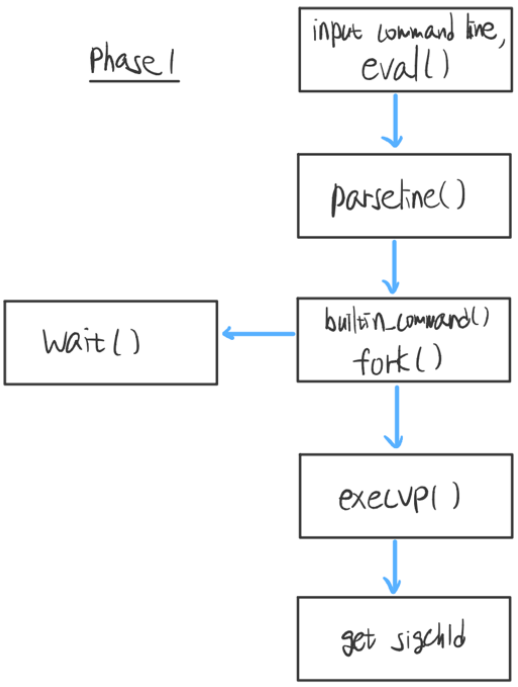
sig_quit_handler : SIGQUIT을 처리한다. 즉 shell 프로그램을 종료시킨다.

signal handler 함수의 경우 다른 signal과 펜딩되는 일을 방지하기 위해 blocked 작업을 거친다. 이를 sigprocmask()를 통해 수행한다.

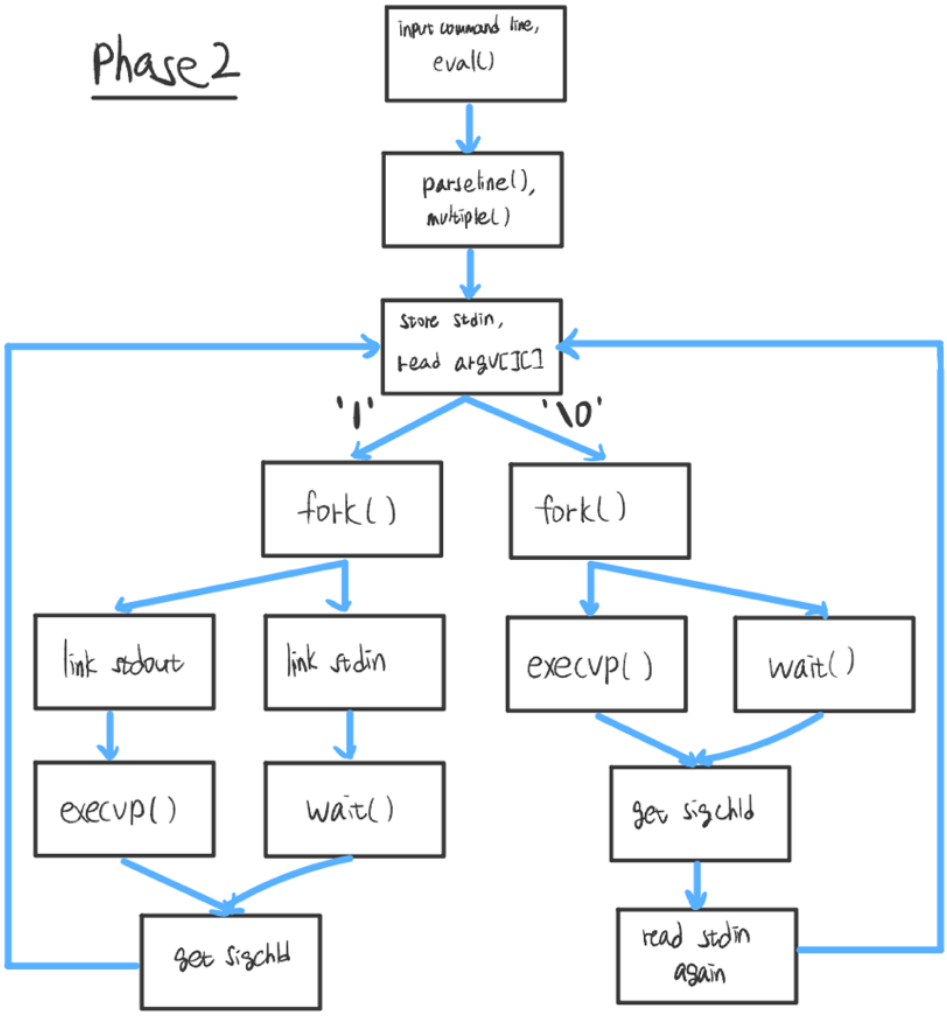
3. 구현 결과

A. Flow Chart

Phase 1 (fork)



Phase 2 (pipeline)



Phase 3 (background)