# CSED211: Shell Lab Report

20230499/KimJaeHwan

## 1. Overview

이번 랩에서는 간단한 내장 명령과 외부 파일을 실행할 수 있는 tsh 쉘을 만드는 것을 목표로 한다. 주어진 tsh 쉘 스켈레톤 코드에서 `eval()`, `builtin_cmd()`, `do_bgfg`, `waitfg`, `sigint_handler()`, `sigtstp_handler()`, `sigchld_handler()` 함수를 구현하는 것을 목표로 하며, 순차적으로 필요한 기능을 구현할 수 있도록 trace 파일이 제공된다.

## 2. tsh.c skeleton 코드 및 trace 파일 분석

tsh.c 스켈레톤 코드는 몇몇 유용한 변수 및 함수가 이미 구현되어 있다.

```
struct job_t jobs[MAXJOBS];
void clearjob(struct job_t *job);
void initjobs(struct job_t *jobs);
int maxjid(struct job_t *jobs);
int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline);
int deletejob(struct job_t *jobs, pid_t pid);
pid_t fgpid(struct job_t *jobs);
struct job_t *getjobpid(struct job_t *jobs, pid_t pid);
struct job_t *getjobjid(struct job_t *jobs, int jid);
int pid2jid(pid_t pid);
void listjobs(struct job_t *jobs)
```

trace 파일은 1번부터 16번까지 다음과 같은 기능을 각각 테스트한다.

```
$ cat trace*.txt | grep '# '
# trace01.txt - Properly terminate on EOF.
# trace02.txt - Process builtin quit command.
# trace03.txt - Run a foreground job.
# trace04.txt - Run a background job.
# trace05.txt - Process jobs builtin command.
# trace06.txt - Forward SIGINT to foreground job.
# trace07.txt - Forward SIGINT only to foreground job.
# trace08.txt - Forward SIGTSTP only to foreground job.
# trace09.txt - Process bg builtin command
# trace10.txt - Process fg builtin command.
# trace11.txt - Forward SIGINT to every process in foreground process group
# trace12.txt - Forward SIGTSTP to every process in foreground process group
# trace13.txt - Restart every stopped process in process group
# trace14.txt - Simple error handling
# trace15.txt - Putting it all together
# trace16.txt - Tests whether the shell can handle SIGTSTP and SIGINT
#     signals that come from other processes instead of the terminal.
```

trace 파일에서 요구하는 각 기능을 따라가면서 구현하였다.

## 3. trace01.txt

EOF, 즉 ctrl + d를 입력받으면 종료해야 한다. 이 기능은 다른 작업 없이 구현되어 있다.

## 4. trace02.txt

quit 내부 명령어를 구현해야 한다. eval(char* cmdline) 함수에서 미리 구현된 parseline(cmdline, argv) 함수를 이용해 명령어를 파싱하고, 파싱된 값을 argv에 저장하여 buildtin_cmd(char** argv) 함수를 호출한다. builtin_cmd(char **argv) 함수 내에서 argv[0] 이 "quit"과 일치하는지 strcmp() 함수를 통해 확인하고, quit 명령어를 받으면 exit(0); 문을 통해 바로 종료하도록 구현하였다.

이 단계에서 입력이 없는 경우, 즉 argv[0] == NULL일 때 예외 처리해 주었다.

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    int bg;
    pid_t pid;

    bg = parseline(cmdline, argv);
    if(argv[0] == NULL) return;
    if(builtin_cmd(argv)) return;
}

int builtin_cmd(char **argv)
{
    /* jobs, fg, bg, quit */
    char* command = argv[0];
    if(!strcmp(command, "quit")) {
        exit(0);
    }
    return 0;     /* not a builtin command */
}
```

## 5. trace03.txt

foreground job을 실행할 수 있도록 구현해야 한다. eval(char* cmdline) 함수에서 강의 노트를[1] 참고하여 fork() 함수를 이용해 구현하였다. fork() 함수를 통해 자식 프로세스 생성을 성공하였을 때 writeup의 hint에 따라 setpgid(0,0); 문을 작성하였고, 이후 execve() 함수를 이용해 자식 프로세스를 원하는 작업으로 변경하였다. 만약 실패한 경우, 명령어를 찾을 수 없다는 문구를 출력하고 종료한다.

1) [CSED211 2024 Fall] Lecture 14 Signals, Prof. Jisung Park, p.9

```
void eval(char *cmdline)
{
    /* ... same as before ... /
    if((pid = fork()) == 0) {
        setpgid(0, 0);
        if (execve(argv[0], argv, environ) <0) {
            printf("%s: Command not found\n", argv[0]);
            exit(0);
        }
    }
}
```

### 6. trace04.txt

background job을 실행할 수 있도록 구현해야 한다. 미리 구현된 parseline() 함수는 마지막 문자가 &인 경우 1을, 아닌 경우 0을 반환한다. 따라서 parseline() 함수의 반환 값을 변수에 저장하여 background job인지 확인할 수 있다. 해당 값을 (int) bg 변수에 저장한 후, 1인 경우 BG 형태로 addjob() 함수를 호출하고, 아닌 경우 FG 형태로 addjob() 함수를 호출하고 하고 알맞은 문구를 출력하도록 하여 구현하였다.

### 7. trace05.txt

jobs 내장 명령어를 구현해야 한다. 실행 중이거나 정지된 job의 목록을 출력하는 함수는 listljobs() 함수의 형태로 미리 구현되어 있으며, 따라서 buildin_cmd() 함수에서 명령어가 jobs인지 확인하고, 맞으면 listjobs() 함수를 호출하도록 구현하면 된다.

```
int builtin_cmd(char **argv)
{
    /* jobs, fg, bg, quit */
    char* command = argv[0];
    if(!strcmp(command, "quit")) { exit(0); }
    if(!strcmp(command, "jobs")) {
        listjobs(jobs);
        return 1;    /* builtin command */
    }
    return 0;     /* not a builtin command */
}
```

### 8. SIGINT - trace06.txt, trace07.txt, trace11.txt

foreground job이 SIGINT 신호를 적절히 처리할 수 있도록 구현해야 한다. SIGINT 신호가 입력된 경우 다음과 같이 알맞은 동작을 수행할 수 있어야 한다.

- foreground job 없이, tsh에서 SIGINT를 받는 경우 무시한다.
- foreground job 실행 중인 경우, 해당 job을 종료한다.
- 만약 foreground job이 자식 프로세스를 가진 경우, 자식 프로세스 또한 종료한다.

처음으로 시그널을 다루는 법을 배우느라 많은 시행착오를 겪었으며 trace06의 요구사항이 올바르게 구

현되었다고 생각될 때까지 헤매었다. 모든 SIGINT trace 들도 올바르게 동작하는 것을 목표로 코드를 구현하였다.

가장 먼저 sigint_handler를 구현하였다. pid_t 자료형의 target 변수를 만들고, 미리 구현된 fgpid(jobs) 함수를 통해 foreground job의 pid를 받는다. 만약 foreground job이 존재하는 경우 해당 target에 신호를 보내도록 구현하였다. 또한 만약을 대비해 함수 맨 처음에 errno 변수의 값을 저장한 후 리턴 전에 복원하며, 핸들러가 다른 시그널의 영향을 받지 않도록 sigprocmask 함수를 이용하여 모든 시그널을 차단하였다.

● sigprocmask(int how, sigset_t this, sigset_t before)는 how에 따라 세 방법으로 동작한다.

- SIG_SETMASK : this에 포함된 신호만 차단하며, 이전 마스크는 before에 저장.
- SIG_BLOCK : this에 포함된 신호를 차단함으로 설정하고, 이전 마스크는 before에 저장.
- SIG_UNBLOCK : this에 포함된 신호의 차단을 해제하며, 이전 마스크는 before에 저장.

```
void sigint_handler(int sig)
{
    sigset_t mask_all, prev_all;
    sigfillset(&mask_all);
    sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
    int saved_errno = errno;
    pid_t target = fgpid(jobs);
    if(target) {
        kill(-target, sig);
    }
    errno = saved_errno;
    sigprocmask(SIG_SETMASK, &prev_all, NULL);
    return;
}
```

foreground job가 SIGINT를 받으면 종료하면서 "Job [jid] (pid) terminated by signal {sig}" 문구를 출력한다. kill() 함수를 통해 target에 원하는 시그널을 전송하도록 하며, 특히 첫 인자에 음수 pid를 넣음으로써 target의 자식 프로세스에도 시그널이 보내지도록 하였다. 또한 처음에는 sigint_handler 내에서 출력까지 처리하도록 작성하였다. 그러나 자식 프로세스는 즉시 종료 하고, 종료되었음을 알려주는 문구는 부모 프로세스에서 처리하는 것이 더 알맞을 것으로 판단하여 해당 로직을 sigchld_handler로 옮기기로 하였다.

sigchld_handler는 자식 프로세스가 종료되거나 정지하면 실행되며 자식 프로세스가 좀비가 되지 않도록 관리하여야 한다. 우선 이 단계에서 완성한 코드는 다음과 같다.

```c
void sigchld_handler(int sig)
{
    sigset_t mask_all, prev_all;
    sigfillset(&mask_all);
    sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
    int saved_errno = errno;
    pid_t pid;
    int status;
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
        if (WIFSIGNALED(status)) {
            printf("Job [%d] (%d) terminated by signal %d\n",
                    pid2jid(pid), pid, WTERMSIG(status));
            deletejob(jobs, pid);
        }
        else if (WIFEXITED(status)) {
            deletejob(jobs, pid);
        }
    }
    sigprocmask(SIG_SETMASK, &prev_all, NULL);
    errno = saved_errno;
    return;
}
```

핵심 로직은 while() 문이다. 먼저 pid, status 변수를 정의하였으며 강의 노트[2]와 writeup의 Hint를 참고하여 while문을 작성해주었다. 또한 위 함수는 signal handler이므로 원래 errno 변수를 저장했다가 복원하는 코드를 추가했으며, 매우 중요한 함수인 deletejob() 함수가 실행되어야 하므로 모든 시그널을 일시적으로 차단하는 코드를 추가하였다. 위 코드에서는 출력을 위해 printf() 함수를 사용하였지만 출력을 제외한 신호 처리가 올바른지 확인한 이후 write() 함수를 이용하도록 코드를 수정하였다. 다만 출력되어야 하는 문구(Job... terminated by signal ..)를 만드는 코드가 매우 길어져 보고서에서는 생략하였다.

2) [CSED211 2024 Fall] Lecture 14 Signals, Prof. Jisung Park, p.50, procmask1.c

eval() 함수에서도 signal을 적절히 처리하기 위해서 sigprocmask() 함수를 통해 신호를 차단하였다.

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    int bg;
    pid_t pid;
    sigset_t mask_all, mask_one, prev_one;
    sigfillset(&mask_all);
    sigemptyset(&mask_one);
    sigaddset(&mask_one, SIGCHLD);
    bg = parseline(cmdline, argv);

    if(argv[0] == NULL) return;
    if(builtin_cmd(argv)) return;
    sigprocmask(SIG_BLOCK, &mask_one, &prev_one);
    if((pid = fork()) ==0) {
       sigprocmask(SIG_SETMASK, &prev_one, NULL);
       /* ... omitted in this report. */
    }
    sigprocmask(SIG_BLOCK, &mask_all, NULL);
    if(!bg) { addjob(jobs, pid, FG, cmdline);}
    else {
        addjob(jobs, pid, BG, cmdline);
        printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
    }
    sigprocmask(SIG_SETMASK, &prev_one, NULL);
    if(!bg) {
        waitfg(pid);
    }
}
```

- 자식 프로세스를 생성(fork())할 때 race condition에 빠지지 않도록 SIGCHLD를 차단하였다.
- 중요한 함수인 addjob() 함수가 방해받지 않도록 addjob() 전에 모든 신호를 차단하였다.
- 본 함수가 종료될 때, 또는 자식 프로세스일 때 signal mask를 복원하였다.

두 signal handler와 eval() 함수를 알맞게 구현하고 나서, SIGINT에 관한 trace 파일이 성공적으로 동작할 수 있었다.

## 9. SIGTSTP - trace08.txt, trace12.txt

foreground job이 SIGTSTP 시그널을 올바르게 받도록 구현하여야 한다. SIGINT를 이미 구현하였으므로, 비슷한 방식으로 SIGTSTP을 처리할 수 있도록 구현하면 된다. 가장 먼저 sigtstp_handler는 sigint_handler와 동일하게 작성하고, sigchld_handler에서 SIGTSTP에 대한 처리를 추가로 작성하였다.

```
void sigchld_handler(int sig)
{
    /* ... Same as before ... */
    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) >0) {
        if (WIFSIGNALED(status)) { /* ... Same as before ... */ }
        else if (WIFSTOPPED(status)) {
            printf("Job [%d] (%d) stopped by signal %d\n",
                    pid2jid(pid), pid, WSTOPSIG(status));
            getjobpid(jobs, pid)->state = ST;
        }
        else if (WIFEXITED(status)) { /* ... Same as before ... */ }
    }
    /* ... Same as before ... */
}
```

SIGINT처럼 현재 프로세스가 종료되었다는 문구를 출력한 이후 해당 프로세스의 상태를 정지 상태로 바꾸도록 구현하였다. SIGINT에서 기본 신호 처리를 작성하였으므로 해당 코드를 약간만 수정하여 SIGTSTP도 처리할 수 있도록 구현하였다.

### 10. fg, bg - trace09.txt, trace10.txt, trace13.txt

내장 명령어로 fg와 bg를 구현해야 한다. 가장 먼저 buildin_cmd() 함수에 입력받은 명령이 fg 또는 bg인지 검사한 후 do_bgfg() 함수를 실행하도록 하였다. do_bgfg() 함수는 다음과 같이 구현하였다.

```
void do_bgfg(char **argv)
{
    pid_t pid; int jid;
    if(argv[1][0] =='%') {
        jid = atoi(argv[1] + 1);
        pid = getjobjid(jobs, jid)->pid;
    }
    else {
        pid = atoi(argv[1]);
        jid = pid2jid(pid);
    }
    if(!strcmp(argv[0], "bg")) {
        getjobpid(jobs, pid)->state = BG;
        kill(-pid, SIGCONT);
        printf("[%d] (%d) %s", jid, pid, getjobpid(jobs, pid)->cmdline);
    }
    else if(!strcmp(argv[0], "fg")) {
        getjobpid(jobs, pid)->state = FG;
        kill(-pid, SIGCONT);
        waitfg(pid);
    }
    return;
}
```

첫 번째 조건문은 입력 형식이 jid인지 pid인지 확인하여 각각 jid와 pid를 받는 코드이며, 두 번째

조건문은 bg 또는 fg 명령을 수행하는 코드이다. bg 명령어인 경우, 지목된 job의 상태를 BG로 바꾸고, SIGCONT 시그널을 해당 그룹 전체에게 보내고 문구를 출력한다. fg 명령어인 경우, 지목된 job의 상태를 FG로 바꾸고 마찬가지로 SIGCONT 시그널을 보낸다. 또한, foreground job일 때 해당 job이 계속해서 실행되어야 하므로 waitfg() 함수를 구현해야 한다. 강의 노트[3]를 참고하여 아래와 같이 구현하였다.

```
void waitfg(pid_t pid)
{
    while(fgpid(jobs) == pid) {
        sleep(1);
    }
    return;
}
```

느리지만 안전하게 동작할 것이다.

** 그러나 나중에 sigsuspend() 함수를 이용하여 waitfg() 함수를 다음과 같이 수정하였다. sigsuspend() 함수가 수행되기 전에 우선 SIGCHLD 시그널을 차단한 후, sigsuspend() 함수에서는 기존 시그널 마스크를 이용해 foreground job이 끝날 때까지 기다리도록 하였다. 기존 함수에 비해서 더욱 뛰어난 반응성을 보여줄 것이다.

```
void waitfg(pid_t pid)
{
    sigset_t mask, prev_mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    sigprocmask(SIG_BLOCK, &mask, &prev_mask);
    while (fgpid(jobs) == pid) {
        sigsuspend(&prev_mask);
    }
    sigprocmask(SIG_SETMASK, &prev_mask, NULL);
}
```

## 11. ERROR handling - trace14.txt

do_bgfg() 함수의 예외를 처리해야 한다. 고려해야 하는 경우는 다음과 같다.

- bg, fg 명령어 이후 두 번째 인자가 없는 경우.
- bg, fg 명령어 이후 두 번째 인자가 (숫자열) 또는 %(숫자열) 이 아닌 경우.
- bg, fg 명령어 이후 두 번째 인자가 jid, pid 형식이 맞으나 해당하는 프로세스가 없는 경우.

각 경우 모두 알맞은 문구를 출력한 이후 실행을 중단한다. 필요한 코드의 길이가 기므로, 본 보고서에서는 생략한다. 주어진 요구사항에 따라 알맞게 구현하였다.

## 12. Finale - trace15.txt, trace16.txt

3) [CSED211 2024 Fall] Lecture 14 Signals, Prof. Jisung Park, p.53

trace15.txt는 여러 테스트케이스를 모아둔 것으로 각 명령어의 예외 처리, 동작 방식 및 출력을 알맞게 처리하였다면 통과할 수 있으며 trace16.txt 또한 함수마다 signal을 잘 처리하였다면 통과할 수 있다.

## 13. Result

`./make rtest` 명령어와 `./make test` 명령어를 통해서 각 trace 파일들에 대한 실행 결과를 비교해본 결과, 각 프로세스의 `pid`를 제외한 나머지 출력이 모두 같음을 확인하였다. 모두 올바르게 구현되었다고 생각된다.

## 14. Reference

[1] Computer Systems: A Programmer's Perspective, 3/E (CS:APP3e), Randal E. Bryant and David R. O'Hallaron, Carnegie Mellon University.

[2] [CSED211 2024 Fall] Lecture 13 Exceptions and Processes, Prof. Jisung Park.

[2] [CSED211 2024 Fall] Lecture 14 Signals, Prof. Jisung Park.

[3] [Writeup] Shell Lab: Writing Your Own Unix Shell, Prof. Jisung Park.