시스템 프로그래밍 Shell lab보고서

2017-12751 컴퓨터공학부 이동학

이번 lab에서는 tsh.c 파일에 있는 7개의 함수 eval, builtin_cmd, do_bgfg, waitfg, sigchld_handler, sigint_handler, sigtstp_handler를 완성하여 process control과 signaling의 개념을 이해하고 간단한 unix shell을 구현하였습니다.

1. eval

```c
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    pid_t pid;
    sigset_t mask;
    // check BG
    int bg = parseline(cmdline, argv);

    // check valid builtin_cmd
    if (!builtin_cmd(argv)) {
        // blocking
        check(sigemptyset(&mask), "sigemptyset() failed");
        check(sigaddset(&mask, SIGCHLD), "sigaddset() failed");
        check(sigprocmask(SIG_BLOCK, &mask, NULL), "sigprocmask() failed");
        // forking
        pid = check(fork(), "fork() failed");
        // child process
        if (pid == 0) {
            check(sigprocmask(SIG_UNBLOCK, &mask, NULL), "sigprocmask() failed");
            check(setpgid(0, 0), "setpgid() failed");
            // check command
            // execve return -1 only error occured
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found\n", argv[0]);
                exit(1);
            }
        }
        // parent
        else {
            if (!bg) { addjob(jobs, pid, FG, cmdline); }
            else { addjob(jobs, pid, BG, cmdline); }
            check(sigprocmask(SIG_UNBLOCK, &mask, NULL), "sigprocmask() failed");

            if (!bg) {
                // wait foreground job finish
                waitfg(pid);
            }
            else {
                // print background job information
                printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
            }
        }
    }
}
```

eval 함수는 우선 parseline 함수를 통해 valid한 built-in command가 입력되었는지, background인지 foreground인지를 판단합니다. built-in command가 valid하다면 sigprocmask 함수를 이용해 SIGCHLD 시그널을 block한 후 프로세스를 fork합니다. 만약 child 프로세스라면 SIGCHLD 시그널을 다시 unblock한 후 unique한 pgid를 부여하고 argv[0]의 프로그램을 실행합니다. 만약 parent 프로세스라면 addjob 함수를 이용해 job list에 child를 추가하고 SIGCHLD 시그널을 다시 unblock한 후 foreground라면 기다리고 background라면 정보를 출력하게 구현했습니다.

builtin_cmd

```
int builtin_cmd(char **argv)
{
    if (strcmp(argv[0], "quit") == 0) { exit(0); }
    else if (strcmp("jobs", argv[0]) == 0) {
        listjobs(jobs);
        return 1;
    }
    else if (strcmp("bg", argv[0]) == 0 || strcmp("fg", argv[0]) == 0) {
        do_bgfg(argv);
        return 1;
    }

    // not built-in command
    return 0;
}
```

builtin_cmd 함수는 주어진 명령어가 built-in commands 중 무엇인지 인식하고 알맞은 동작을 하도록 구현했습니다.

do_bgfg

```
void do_bgfg(char **argv)
{
    struct job_t *job;
    char *p = argv[1];
    int jid;
    pid_t pid;

    // invalid id
    if (p == NULL) {
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }
    // jid
    if (p[0] == '%') {
        // get jid
        jid = atoi(&p[1]);
        // get job
        job = getjobjid(jobs, jid);
        if (job == NULL) {
            printf("%s: No such job\n", p);
            return;
        }
        else {
            //get pid
            pid = job->pid;
        }
    }
    // pid
    else if (isdigit(p[0])) {
        // get pid
        pid = atoi(p);
        // get job
        job = getjobpid(jobs, pid);
        if (job == NULL){
            printf("(%d): No such process\n", pid);
            return;
        }
    }
```

```
    // invalid argument
    else {
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }
    // resume paused process
    check(kill (-pid, SIGCONT), "send SIGCONT failed");

    if (strcmp("fg", argv[0]) == 0) {
        //wait foreground job finish
        job->state = FG;
        waitfg(job->pid);
    }
    else {
        //print background job information
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
        job->state = BG;
    }
}
```

do_bgfg 함수는 bg, fg 명령어가 입력되었을 때 먼저 jid인지 pid인지 확인합니다. 그 후 정지된 프로세스를 계속 실행시키고 fg라면 대기, bg라면 정보를 출력하도록 구현했습니다.

waitfg

```
void waitfg(pid_t pid)
{
    struct job_t *job = getjobpid(jobs, pid);

    // check valid pid
    if (pid == 0) { return; }

    if (job != NULL) {
        // sleep (busy loop)
        while (pid == fgpid(jobs)){
        }
    }
}
```

waitfg 함수는 foreground job이 끝날 때까지 busy loop을 이용해 기다리도록 구현했습니다.

sigchld_handler

```
void sigchld_handler(int sig)
{
    int status;
    pid_t pid;

    // check how process ended
    while ((pid = waitpid(fgpid(jobs), &status, WNOHANG|WUNTRACED)) > 0) {
        if (WIFSTOPPED(status)) {
            // stopped by signal
            getjobpid(jobs, pid)->state = ST;
            printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(status));
        }
        else if (WIFSIGNALED(status)) {
            // terminated by signal
            printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(status));
            deletejob(jobs, pid);
        }
        else if (WIFEXITED(status)) {
            // normal exit
            deletejob(jobs, pid);
        }
    }
}
```

sigchld_handler 함수는 SIGCHLD 신호를 받았을 때 waitpid 함수를 이용해 상태정보를 받아 어떻게 종료되었는지를 WIFSTOPPED, WIFSIGNALED, WIFEXITED를 통해 확인하고 알맞은 정보를 출력하도록 구현했습니다..

sigint_handler

```
void sigint_handler(int sig)
{
    pid_t pid = fgpid(jobs);
    //check valid pid
    if (pid != 0) { check(kill(-pid, sig), "send SIGINT failed"); }
}
```

sigint_handler 함수는 SIGINT(ctrl-c) 신호를 받았을 때 해당하는 프로세스를 종료하도록 구현했습니다.

sigtstp_handler

```
void sigtstp_handler(int sig)
{
    pid_t pid = fgpid(jobs);
    //check valid pid
    if (pid != 0) { check(kill(-pid, sig), "send SIGTSTP failed"); }
}
```

sigtstp_handler 함수는 SIGTSTP(ctrl-z) 신호를 받았을 때 해당하는 프로세스를 정지하도록 구현했습니다.

check

```
int check(int ret, char *msg)
{
    if(ret != -1) { return ret; }
    printf("%s: %s\n", msg, strerror(errno));
    exit(1);
}
```

check 함수는 system call에 대해 return value를 check하기 위해 만든 함수입니다. system call이 제대로 동작하지 않을 경우 -1을 return 하므로 어떤 system call에서 어떤 문제가 발생했는지 출력하도록 구현했습니다.

후기: hand out의 hints 섹션에 도움되는 정보들이 많이 제공되었고 reference output을 참고할 수 있어서 저번 과제인 linker lab에 비해 수월했던 것 같습니다. 또한 직접 shell을 구현하는 과정에서 그냥 수업을 들었을 때보다 더 많은 지식을 얻을 수 있었습니다.