CSED211 Shell-Lab Report

20210643 김현준

<Lab10>

Lab10에서는 간단하게 shell의 개념에 대해 다루고, Shell-Lab에 대한 설명을 들었다. 먼저, Shell이란 사용자가 운영체제에 접근할 수 있도록 만들어준 인터페이스이다. 사용자를 대신하여 프로그램을 실행하는 Interactive command-line interpreter라고 할 수 있다. Command line은 ASCII text words로 받으며 linux에서 대부분의 application은 shell을 통해서 실행된다. Linux Kernal에는 system call 인터페이스가 있는데, system call은 functional level에서의 인터페이스이기 때문에 이를 유저가 쉽게 명령어 기반으로 사용할 수 있도록 하는 것이 바로 Shell이다.

Shell의 기본적인 기능으로는 먼저 다른 프로그램을 command line으로 실행시켜줄 수 있다는 것이다. 운영체제 안에 있는 파일들과 프로세스들을 관리할 수 있다는 기능이 있다. Shell의 Basic function들로는 job(running and stopped backgrounds jobs를 리스트로 보여줌), bg와 fg(각각 stopped background job과 stopped or running background job을 running background job과 foreground job으로 변경함), ctrl+c(SIGINT), ctrl+z(SIGSTP)이 있다.

Shell-Lab은 간단한 shell 프로그램을 만들어 보는 lab assignment로, Helper function들을 이용하여 7개의 함수(eval, builtin_cmd, do_bgfg, wait_fg, sigchld_handler, sigint_handler, sigstp_handler)를 수정하면 된다. trace에는 command 들이 작성되어 있어, 각 case를 돌리며 올바르게 shell을 구현했는지 확인하면 된다.

<Lab11>

Lab11에서는 Exceptional Control Flow와 Signal Handling에 대해 다루었다. Program flows에서는 jump로 condition, iteration, branch, call and return 등에 따라 control flow가 이동하였는데, Error case들과 Asynchronous signal의 경우에 Exceptional control flow에 따라 kernal을 통해 flow가 예외 처리 관련으로 os로 이동하게 된다.

Exception들은 os가 다루게 되는데, user program에서 exception 상황이 발생하게 되면 os의 exception handle code로 이동하여 exception 상황에 대처하게 된다.

Signal은 process에서 특정 event가 발생하여 몇 가지 type의 이벤트가 시스템에 발생하였을 때 그것을 알리기 위해서 발생하는 메시지다. Signal은 signal ID 값을 가지고 있으며, os나 다른 progress에서 user signal handler에 의해 signal이 보내지게 된다. signal들은 SIGHUP, SIGINT, SIGQUIT, SIGKILL 등이 각 번호에 대응되어 존재한다. 자주 사용되는 Signal은 ID 2의 SIGINT(interruption), ID 9의 SIGKILL(Kill program), ID 11의 SIGSEGV(Segmentation violation), ID 14의 SIGALRM(Timer signal), ID 17의 SIGCHLD(Child stopped or terminated) 등이 있다.

각 signal에 대해 shell-lab에서 각 action을 define할 수 있다. 추가로, lab11시간에는 signal handling에 대한 간단한 예시 코드를 이용하여 실습을 진행하였다. Server에서 Signal Handler를 통해 signal을 확인해 볼 수 있었다.

<Shell-Lab>

Shell-lab은 signal 함수를 이해하고 간단한 shell, tsh.c의 7가지 함수를 구현하여 shell을 구현해보는 lab assignment이다. 먼저, 7가지 함수를 확인하기 전에 main이 어떤 식으로 구현되어 있는지 확인하였다. Main 함수는 shell의 main routine이 들어가 있었고, h, v, p를 확인해서 각 경우 처리를 하고, Signal 함수로 각 sinal의 경우로 호출이 되고, 마지막 부분에 eval을 호출하는 형식으로 구성이 되어 있었다. 따라서 eval 함수를 확인해보았고, cmdline을 받아서 각 command line에 따라 처리를 하는 함수라는 것을 확인하였다. 따라서 eval 함수를 먼저 구현해보려고 하였다.

그런데 eval 함수를 작성하려면 다른 주어진 함수들을 사용해야 할 것 같아, 어떤 함수를 사용해야 할 지 정리가 필요했다. 먼저, parseline함수가 필요하였다. parseline함수는 command lind과 argv 배열을 parsing하는 함수고, user가 bg job을 요청하면 true를, fg job을 요청하면 false를 리턴하는 함수였다. 따라서, 변수들을 선언하고, parsing을 하고 argv 시작이 0일 경우를 따로 처리하는 부분을 먼저 구현하였다.

```
void eval(char *cmdline)
{

    //variables: mask, argv, pid
    sigset_t mask;
    char* argv[MAXARGS];
    pid_t pid;

    //variable that checks bg or fg
    int check_job;

    //parsing and empty argv case. check_job is true if bg case, fals if fg case
    check_job = parseline(cmdline, argv);
    if(argv[0]==NULL){
        return;
    }
```

다음으로 필요한 것은 builtin_cmd 함수였다. 이 함수는 사용자가 built-in command를 입력하면 그 명령을 실행시키도록 하는 함수이므로 필요하였다. If의 조건으로 !builtin_cmd(argv)를 넣어서 valid command인지를 판별하고 실행하는 부분을 만들면 될 것으로 생각하였다.

따라서 bulitin_cmd함수를 구현해보았다. 이 함수는 quit인지 fg인지, bg인지, jobs인지 각 경우에 따라 처리를 해주면 되므로, 간단하게 아래와 같이 구현할 수 있었다. 이때 quit의 경우에는 exit(0)으로 바로 종료를 해주고, bg나 fg의 경우 이를 처리해주는 do_bgfg함수를 호출한 뒤, built-in 명령이므로 1을 리턴해주고, jobs의 경우에는 listjobs함수를 호출해주고 1을 리턴하도록 프로그램을 구현하였다. 마지막으로 built-in command가 아닐 경우 if문을 빠져나와 0을 리턴하도록 하였다.

```c
int builtin_cmd(char **argv)
{

    //quit case
    if(!strcmp(argv[0], "quit")){

        //end the program
        exit(0);

    }
    //bg and fg case
    else if((!strcmp(argv[0], "bg")) || (!strcmp(argv[0], "fg"))){

        //call the bg and fg function and return 1
        do_bgfg(argv);
        return 1;

    }
    //job case
    else if(!strcmp(argv[0], "jobs")){

        //list up every jobs
        listjobs(jobs);
        return 1;

    }

    //if its not a builtin command
    return 0;     /* not a builtin command */
}
```

이렇게 7개의 함수 중에서 첫 번째로 builtin_cmd함수 작성을 완료하였다. 그런데 이때, listjobs는 helper function이고, do_bgfg는 구현하여야 하는 함수였다. 따라서 다음으로 do_bgfg를 구현해보고자 하였다. do_bgfg는 말 그대로 bg와 fg의 경우 처리를 담당하는 함수였다. 먼저 job을 나타는 구조체 변수를 선언하고, temp, jid(job id), pid(process id) 변수들을 선언해주었다. 그리고 argv[1]에 id가 들어있으므로 temp로 해당 값을 받아왔다. 그리고 이제 id가 잘 있는지, 적절한지를 체크하기 위해 두 개의 조건문을 돌려주었다.

```c
void do_bgfg(char **argv)
{

    //variables job(represent jobs), temp(temporary variable), jid(job id), pid(process id)
    struct job_t *job;
    char *temp;
    int jid;
    pid_t pid;

    //get the id
    temp = argv[1];

    //id not exist case
    if(temp==NULL){
        //if no argument, then print error
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }

    //checking if argument form is proper
    if((!isdigit(argv[1][0]))&&argv[1][0]!='%'){
        //if not appropriate argument form, then print error
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }
```

다음으로, jid인 경우와 pid인 경우를 나누어 각각 따져 주었다. atoi함수로 id값을 불러왔고, helper 함수 중 하나인 getjobpid를 이용해 job을 받아왔다. getjobpid함수는 job list를 이용해서 job을 찾아내는 함수이므로, 이 함수를 사용하였다.

```
//job id case
if(temp[0] == '%'){
    //get job id and figure out which job it is
    jid = atoi(&temp[1]);
    job = getjobjid(jobs, jid);
    //no job case
    if(job==NULL){
        //if no job, then print error
        printf("%s: No such job\n", argv[1]);
        return;
    }
}
//process id case
else{
    //get process id and figure out which job it is
    pid = atoi(&argv[1][0]);
    job = getjobpid(jobs, pid);
    //no job case
    if(job == NULL){
        //if no job, then print error
        printf("(%d): No such process\n", pid);
        return;
    }
}
```

마지막으로 fg인 경우와 bg인 경우를 따로 나누어 job 구조체의 state에 FG, BG를 각각 넣어주었다. 그리고 두 경우 모두 SIGCONT signal을 보내기 위해 kill 함수를 사용하여 먼저 signal을 전송해주었다. 이렇게 do_bgfg함수를 모두 구현하였다.

```
    //send the signal "SIGCONT" to process group that has pid of job
    kill(-pid, SIGCONT);

    //fg case
    if(!strcmp("fg", argv[0])){
        //set FG state and wait for job's terminatation
        job->state = FG;
        waitfg(job->pid);
    }
    //bg case
    else{
        //set BG state and print job's status
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
        job->state = BG;
    }

    //return finally
    return;
}
```

다시 처음의 eval함수로 돌아와 구현을 이어가려고 하였는데, waitfg함수 또한 helper function이 아니라 구현해야 하는 함수였다. 간단하게 waitfg 함수를 아래와 같이 구현하였다. do_fgbg와 마찬가지로 job을 받아 주고, pid가 0일 때 리턴을 시켜주고, fpid함수를 호출해서 그 리턴값이 pid와 다를 때까지 루프를 돌려서 대기를 시키도록 구현하였다. 따라서 wait_fg함수는 foreground job의

경우 실행중인 job이 종료된 이후에서야 다음 command를 받는 것이 가능하므로 pid를 현재 fg의 pid와 같은지 fpid함수의 리턴으로 비교하여 대기를 시키는 것이다.

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

    //case of pid is zero
    if(pid==0){
        return;
    }
    //wait until pid and return value of fpid function is different
    if(job!=NULL){
        while(pid==fgpid(jobs)){
            sleep(1);
        }
    }

    //returns
    return;
}
```

이제 다시 처음의 eval로 돌아와 구현을 계속하였다. 먼저, blocking을 위해서 if문 안에서 sigemptyset, sigaddset, sigprocmask의 함수들을 호출해주었다. 다음으로, fork함수를 호출해서 pid에 리턴을 저장하고, child process의 경우를 따로 처리해주었다. Child process인 경우 setpgid(0, 0)을 호출해주어 새로운 process group의 리더가 자신이 되도록 처리하였고, sigprocmask로 unblock하고 process를 실행시키도록 되었다. Command가 없을 경우에는 오류 메시지를 출력하고 exit하여 종료하도록 하였다. Helper function들을 많이 이용하여 이 부분을 구현할 수 있었다.

```
//check if valid builtin command
if(!builtin_cmd(argv)){

    //blocking
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    sigprocmask(SIG_BLOCK, &mask, NULL);

    //Child case with executing fork function
    pid = fork();
    if(pid==0){
        //setting pid and unblcok
        setpgid(0, 0);
        sigprocmask(SIG_UNBLOCK, &mask, NULL);
        //execve the process
        if(execve(argv[0], argv, environ)<0){
            //error printing
            printf("%s: Command not found\n", argv[0]);
            exit(0);
        }
    }
```

다음으로, bg의 경우와 fg의 경우를 따로 나누어 처리해주었다. Check_job이 true일 경우가 bg이고 false일 경우가 fg이다. 따라서 각각의 경우에서 addjob함수로 process의 id를 추가해주고, sigprocmask함수를 호출 후, bg의 경우 상태를 출력해주고, fg의 경우 waiting이 필요하므로 이전에 구현했던 waitfg함수를 호출하여 대기시키는 과정을 만들어주었다. 이러한 방식으로 eval의 구

현이 마무리되었다.

```
        //if background command
        if(check_job){
            //add job with child process' ID
            addjob(jobs, pid, BG, cmdline);
            //SIGCHLD unblocking and printing status
            sigprocmask(SIG_UNBLOCK, &mask, NULL);
            printf("[%d] (%d) %s", pid2jid(pid), (int)pid, cmdline);
        }
        else{
            //add job with child process' ID
            addjob(jobs, pid, FG, cmdline);
            //SIGCHLD unblocking and waiting for foreground next command
            sigprocmask(SIG_UNBLOCK, &mask, NULL);
            waitfg(pid);
        }

    }

    //finally returns
    return;
}
```

이제 7개의 함수 중에서 eval, builtin_cmd, do_bgfg, wait_fg의 4개를 구현하였고, sigchld_handler, sigint_handler, sigstp_handler 함수가 남았다. 각각의 함수는 sigchld, sigint, sigstp의 경우를 처리해주는 handler 함수이다. 먼저, sigchld_handler부터 구현해보았다.

sigchld_handler함수는 커널에서 shell로 SIGCHLD signal을 보내면 child job이 terminates하는지 stop하는지를 처리하고 available zombie children을 reap하지만 다른 child들의 running이 terminate하는 것을 기다리지는 않도록 하여야 했다. (tsh.c의 sigchld_handler앞 설명 내용) 따라서 먼저, 변수들을 선언해주고, while문으로 waitpid함수의 리턴을 pid로 넣어주는 것과 동시에 0보다 클 때 반복을 시켜주었다. 이때 waitpid의 option으로 WNOHANG|WUNTRACE를 설정하여 대기 group의 모든 child process들이 정지 또는 종료되었다면 리턴 0으로 바로 리턴되거나 자식들 중 한 개의 pid와 같은 값을 가지고 리턴하게 되므로, 이 루프와 같이 작성하면 원하는 대로 구현이 이뤄질 것으로 생각하였다.

```
void sigchld_handler(int sig)
{
    //st represents status, pid represents process id, jid is for the job id
    int st;
    pid_t pid;
    int jid;

    //loop until return value of waitpid is over 0
    while((pid=waitpid(fgpid(jobs), &status, WNOHANG|WUNTRACED))>0){
```

그리고, 세 가지 경우가 있을텐데, WIFEXITED와 WIFSTOPPED와 WIFSIGNALED의 경우로 나누어 각각 처리를 해 주었다. 먼저 WIFEXIED의 경우 deletejob을 호출하였고, WIFSTOPPED의 경우 job의 state를 ST로 바꾸어 주고, signal이 몇 번째 signal에 의해 stop된 것인지에 대해 출력을 해주었다. 다음으로 WIFSIGNALED의 경우에는 몇 번째 siganl에 의해서 종료가 되었는지를 출력한 후 job을 delete함수로 삭제해주었다. 결론적으로, 아래와 같이 구현을 완료하였다.

```c
void sigchld_handler(int sig)
{
    //st represents status, pid represents process id, jid is for the job id
    int st;
    pid_t pid;
    int jid;

    //loop until return value of waitpid is over 0
    while((pid=waitpid(fgpid(jobs), &st, WNOHANG|WUNTRACED))>0){
        //case that child terminates for normal
        if(WIFEXITED(st)){
            //deleting job
            deletejob(jobs, pid);
        }
        //case that child stopped by the signal
        else if(WIFSTOPPED(st)){
            //then set the jid and print about that signal
            getjobpid(jobs, pid)->state = ST;
            jid = pid2jid(pid);
            printf("Job [%d] (%d) stopped by signal %d\n", (int)jid, (int)pid, WSTOPSIG(st));

        }
        //case that child terminated by the signal
        else if(WIFSIGNALED(st)){
            //then set the jid and print about that signal and deletes the job
            jid = pid2jid(pid);
            printf("Job [%d] (%d) terminated by signal %d\n", (int)jid, (int)pid, WTERMSIG(st))
            deletejob(jobs, pid);
        }
    }

    //returns
    return;
}
```

다음으로, sigint_handler를 구현해보았다. Sigint는 간단하게 구현할 수 있었는데, fgpid로 현재 foreground의 pid를 받아서 그 값이 0이 아닌 경우 kill함수를 이용해 pid의 process group에 signal을 보내주었다. 현재 foreground가 아닐 경우 pid값이 0일 것이므로, 이 경우에는 그냥 리턴하도록 하였다. 아래와 같이 코드를 구현하였다.

```c
void sigint_handler(int sig)
{
    //get foreground's present pid
    pid_t pid=fgpid(jobs);

    //case for not valid pid
    if(pid==0){
        return;
    }
    //case for valid pid
    else{
        kill(-pid, sig);
    }
    return;
}
```

이제 7개 함수 중에서 6개를 구현하였고, 마지막 sigtstp만 남았다. Sigtstp_handler의 경우에도 간단하게 코드를 작성할 수 있었다. Pid를 받아서 0이 아니면 kill로 signal을 보내주고, 0인 경우 그냥 리턴하도록 구현하면 되었다. 따라서 sigint의 경우와 같은 코드가 나왔다. 아래와 같이 구현 완료하였다.

```c
void sigtstp_handler(int sig)
{
    //get foreground's present pid
    pid_t pid=fgpid(jobs);

    //case for not valid pid
    if(pid==0){
        return;
    }
    //case for valid pid
    else{
        kill(-pid, sig);
    }
    return;
}
```

마지막으로 모든 코드를 구현하였으므로 trace들에 대해 작동하는지 테스트를 해보았다. make test01, make rtest01을 terminal에 입력하고, 01부터 16까지 그 결과가 일치하는지 확인해 보았다. 거의 대부분 일치했지만 14번 case에서 다른 상황이 나타났다.

```
[hyunjunekim@programming2 ~]$ make test14
./sdriver.pl -t trace14.txt -s ./tsh -a "-p"
#
# trace14.txt - Simple error handling
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 4 &
[1] (17449) ./myspin 4 &
tsh> fg
fg command requires PID or %jobid argument
tsh> bg
bg command requires PID or %jobid argument
tsh> fg a
fg: argument must be a PID or %jobid
tsh> bg a
bg: argument must be a PID or %jobid
tsh> fg 9999999
(9999999): No such process
tsh> bg 9999999
(9999999): No such process
tsh> fg %2
%2: No such job
tsh> fg %1
tsh> bg %2
%2: No such job
tsh> bg %1
%1: No such job
tsh> jobs
[hyunjunekim@programming2 ~]$ make rtest14
./sdriver.pl -t trace14.txt -s ./tshref -a "-p"
#
# trace14.txt - Simple error handling
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 4 &
[1] (17530) ./myspin 4 &
tsh> fg
fg command requires PID or %jobid argument
tsh> bg
bg command requires PID or %jobid argument
kill (tstp) error: No such process
tsh> fg a
```

Lab-Q&A 게시판을 참고해서 centos 버전으로 바꾸어 실행해보았다. 그런데 마찬가지로 0~16 중에서 14번 trace만 make test14와 make rtest14의 결과가 다르게 나타났다.

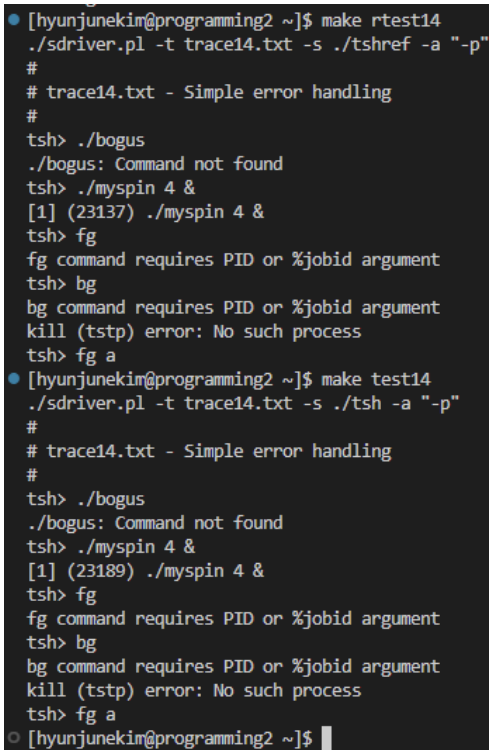
```
#
# trace14.txt - Simple error handling
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 4 &
[1] (22563) ./myspin 4 &
tsh> fg
fg command requires PID or %jobid argument
tsh> bg
bg command requires PID or %jobid argument
tsh> fg a
fg: argument must be a PID or %jobid
tsh> bg a
bg: argument must be a PID or %jobid
tsh> fg 9999999
(9999999): No such process
tsh> bg 9999999
(9999999): No such process
tsh> fg %2
%2: No such job
tsh> fg %1
tsh> bg %2
%2: No such job
tsh> bg %1
%1: No such job
tsh> jobs
[hyunjunekim@programming2 ~]$ make rtest14
./sdriver.pl -t trace14.txt -s ./tshref -a "-p"
#
# trace14.txt - Simple error handling
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 4 &
[1] (22800) ./myspin 4 &
tsh> fg
fg command requires PID or %jobid argument
tsh> bg
bg command requires PID or %jobid argument
kill (tstp) error: No such process
tsh> fg a
```

SIGTSTP에서 error: No such process 부분이 출력되지 않는 것을 확인하였고, 이를 해결하기 위해 sigtstp_handler함수를 수정하였다.

```
void sigtstp_handler(int sig)
{
    //get foreground's present pid
    pid_t pid=fgpid(jobs);

    //case for valid pid
    if(pid!=0){
        if (kill(-pid, sig) < 0){
            unix_error("kill (tstp) error");
        }
    }
    return;
}
```

위와 같이 kill의 리턴이 0보다 작으면 바로 unix_error로 에러 메시지를 보내도록 처리해주었고, 그 결과 rtest14와 test14의 결과가 일치하게 나오는 것을 확인할 수 있었다.



마지막으로, make handin을 터미널에 입력하여 최종 tsh.c 파일을 내보내기 하였다. 이 때 Makefile에서 HANDINDIR 디렉토리를 home/std/hyunjunekim으로 바꾸었다. NOBODY-1-tsh.c를 얻어 이를 파일명 변경 후 제출하였다.