Lab 6

20200437 김채현

본 Lab의 목표는 tiny shell을 만드는 것이다. tsh.c에 있는 함수들을 구현하여 간단한 Unix shell program이 돌아가게 만들면 성공하는 것이다.

---

eval

eval 함수의 역할은 명령어를 입력 받고 이를 처리하는 것이다. main 함수에서 cmdline 변수에 명령어를 받은 것을 parameter로 받아 처리한다.

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

    if (argv[0] == NULL) return; // Check if empty lines

    if (!builtin_cmd(argv)) {
        // block SIGCHILD
        sigemptyset(&mask);
        sigaddset(&mask, SIGCHLD);
        sigprocmask(SIG_BLOCK, &mask, NULL);

        pid = fork();
        if(pid < 0) unix_error("fork error");
        // child
        if (pid == 0) {
            setpgid(0, 0);
            sigprocmask(SIG_UNBLOCK, &mask, NULL);

            int execution = execve(argv[0], argv, environ);
            if (execution < 0) {
                printf("%s: Command not found\n", argv[0]);
                exit(0);
            }
        }
        // parent
        else {
            if(bg) {
                addjob(jobs, pid, BG, cmdline);
                sigprocmask(SIG_UNBLOCK, &mask, NULL);
```

```
                printf("[%d] (%d) %s", pid2jid(pid), (int)pid, cmdline);
            }
            else {
                addjob(jobs, pid, FG, cmdline);
                sigprocmask(SIG_UNBLOCK, &mask, NULL);
                waitfg(pid);
            }
        }
    }
    return;
}
```

가장 먼저 MAXARGS만큼 char pointer를 저장할 수 있는 array인 argv를 선언하는데 이때, MAXARGS는 128로 값이 정의되어 있다. 이후 parseline 함수를 호출하여 parameter로 받는 cmdline의 string을 argv에 저장하고, parseline 함수의 background와 관련된 return 값을 변수 bg에 저장한다. 하지만 만약 argv가 empty line이라면 return하게 해준다.

이후 builtin_cmd 함수를 호출하여 return 값에 따라 builtin instruction 여부를 확인한다. return 값이 1인 경우는 함수 내에서 이미 역할을 수행했다는 뜻이므로 return 값이 0인 경우에만 이후 과정을 수행할 수 있도록 한다. 따라서 return 값이 0인 경우 먼저 fork 함수를 호출해서 새로운 child process를 생성하고 결과값을 pid에 저장한다. 이 때 SIGCHLD 시그널을 block 해주어야 한다. fork의 return값을 저장해둔 pid값이 0보다 작으면 unix_error() 함수를 호출하여 에러가 발생했음을 알려준다. pid값이 0일 때는 child process이고, 0보다 큰 경우에는 parent process이다.

child process에서 setpgid(0,0)를 호출하여 child process를 새 process 그룹에 넣는다. (그렇지 않으면 child process가 shell process와 같은 fg process 그룹에 속하게 되는데 그러면 후에 문제가 발생한다.) 새 signal handle를 통해 shell에서 SIGINT와 같은 것들을 처리해주고, 이후 execve를 통해 입력 받은 프로그램을 실행시키는 과정으로 처리된다. execve 함수를 호출하여 parameter로 argv[0], argv, environ를 넘겨준다. argv[0]에는 파일 이름이, environ은 이미 정의되어 있는 것이다. execve의 return값을 execution 변수에 저장해두고, execve 값이 0보다 작으면 "Command not found"라는 에러 메세지를 띄울 수 있도록 한다.

parent process는 background 실행인 경우와 아닌 경우로 나누어 구현하는데, bg 실행인 경우에는 job list에 BG를 추가해주고 SIGCHLD를 unblock 해준다. 그리고 job 메시지를 출력해준다. bg 실행이 아닌 경우 job list에 FG를 추가해주고 SIGCHLD를 unblock 해준다. 그리고 waitfg 함수를 호출해서 job이 종료될 때까지 기다려준다.

---

builtin_cmd

builtin_cmd 함수의 역할은 builtin command에 따라 역할을 수행하는 것이다.

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

입력으로 "quit"가 들어온 경우, exit를 호출하여 프로그램을 종료한다. 입력으로 "jobs"가 들어온 경우, listjobs 함수를 호출하여 job을 출력하고 1을 return한다. 입력으로 "bg" 혹은 "fg"가 들어온 경우, 뒤에서 설명할 do_bgfg 함수를 호출하여 background는 foreground로, foreground는 background로 process를 바꿔주고 1을 return한다. 이 외인 경우 builtin command가 아니므로 0을 return해준다. 앞에서 설명했듯 0이 return하면 eval 함수로 돌아가게 되는데, 0일 경우에만 eval 함수가 역할을 한다.

---

do_bgfg

do_bgfg 함수의 역할은 앞에서 언급했듯이 background는 foreground로, foreground는 background로 process를 바꿔주는 것이다.

```
void do_bgfg(char **argv)
{
    struct job_t *job;
    int JID;
    pid_t PID;
    int is_digit = 0;

    // if argument is empty
    if(argv[1] == NULL) {
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }

    // if input is JID
    if(argv[1][0] == '%') {

        char *temp = &argv[1][1];
```

```
        while (*temp != '\0') {
            if (isdigit(*temp) != 0) {
                is_digit = 1;
                break;
            }
            temp++;
        }

        if (is_digit == 1) {
            printf("%s: argument must be a PID or %%jobid\n", argv[0]);
            return;
        }

        JID = atoi(&argv[1][1]);
        job = getjobjid(jobs, JID);

        if (job == NULL){
            printf("%s: No such job\n", JID);
            return;
        }
    }
    // if input is a pid
    else {

        char *temp = argv[1];
        while (*temp != '\0') {
            if (isdigit(*temp) != 0) {
                is_digit = 1;
                break;
            }
            temp++;
        }
        if (is_digit == 1) {
            printf("%s: argument must be a PID or %%jobid\n", argv[0]);
            return;
        }

        PID = atoi(argv[1]);
        job = getjobpid(jobs, PID);

        if (job == NULL) {
            printf("(%d): No such process\n", PID);
            return;
        }
    }

    kill(-(job->pid), SIGCONT); //stop
```

```
    if(strcmp("fg", argv[0]) == 0) {
        //wait for fg
        job->state = FG;
        waitfg(job->pid);
    }
    else {
        //print for bg
        job->state = BG;
        printf("[%d] (%d) %s", job->JID, job->pid, job->cmdline);
    }
    return;
}
```

먼저 argv[1]에 저장되어 있는 두 번째 단어가 NULL이면 argument가 비었단 뜻이므로 "command requires PID or jobid argument"라는 에러 메시지를 출력해야 한다.

이후 argv1][0]이 '%'인지 검사한다. '%'인 경우 뒤에 JID가 올 때, 그 뒤에 자연수가 오는지 검사하는 용도로 is_not_digit이라는 변수를 이용한다. 반복문을 돌면서 null에 도달할 때까지 다음 글자를 검사한다. 그 뒤에 오는 게 자연수가 아니면 1을 저장하고 반복문을 빠져나오고, 그렇지 않다면 그대로 0이 저장되도록 한다. is_not_digit이 1일 경우에는 "argument must be a PID or jobid"라는 에러 메시지를 출력하고 return한다. 아닌 경우 atoi 함수를 이용해 sting으로 저장되어 있던 것을 int형 변수 JID에 저장해주고, getjobjid 함수를 이용해 job에 JID로부터 pointer를 저장한다. pointer가 NULL인 경우, 해당하는 job이 없다는 뜻이므로 "No such job"이라는 에러 메시지를 출력하고 return한다.

argv[1][0]이 '%'이 아닌 경우, 뒤에 pid값이 올 때, JID일 때와 마찬가지로 뒤에 자연수가 오는지 알기 위한 용도로 is_not_digit이라는 변수를 이용한다. 똑같이 is_not_digit이 1인 경우 "argument must be a PID or jobid"라는 에러 메시지를 출력하고 return한다. 아닌 경우 atoi 함수를 이용해 sting으로 저장되어 있던 것을 PID에 저장해주고, getjobpid 함수를 이용해 job에 PID로부터 pointer를 저장한다. pointer가 NULL인 경우, 해당하는 process가 없다는 뜻이므로 "No such process"라는 에러 메시지를 출력하고 return한다.

원하는 job을 찾았으므로 kill 함수를 호출하여 job의 process 그룹에 SIGCONT 시그널을 보내서 job이 실행되게 한다. 명령어가 "fg"인 경우 job의 state를 FG로 바꿔주고, waitfg 함수를 이용하여 job이 종료될 때까지 기다린다. 명령어가 "bg"인 경우 job의 state를 BG로 바꿔주고, job state를 출력한다.

---

waitfg

waitfg 함수의 역할은 pid를 parameter로 받아서 job이 foreground job이 아니게 될 때까지 block하는 것이다.

```
void waitfg(pid_t pid)
{
    struct job_t* job = getjobpid(jobs,pid);
    //check if pid is valid
    if(pid == 0) return;

    if(job == NULL) return;
    while(pid == fgpid(jobs)) {
        // sleep
    }
    return;
}
```

getjobpid 함수를 호출하여 해당 job을 찾는다. 해당 job이 있는 경우, job이 current foreground job이면 반복문을 돌도록 한다. 이때, writeup 파일을 참고하여 코드를 작성하였다. process가 끝나면 sigchld_handler 함수를 호출하여 state를 바꾸고 반복문을 빠져나올 수 있도록 한다.

---

sigchld_handler

sigchld_handler 함수의 역할은 child job이 시그널에 의해 terminated된 경우, 혹은 SIGSTOP이나 SIGTSTP 시그널에 의해 stop된 경우 호출되어 child process의 state를 변경하거나 job list에서 삭제하는 것이다.

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

    while ((pid = waitpid(fgpid(jobs), &Child_Status, WNOHANG|WUNTRACED)) > 0) {
        struct job_t *job = getjobpid(jobs, pid);
        if (WIFSIGNALED(Child_Status)) {
            printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(Child_Status));
            deletejob(jobs, pid);
        }
        else if (WIFEXITED(Child_Status)){
            //exited
            deletejob(jobs, pid);
        }
        else if (WIFSTOPPED(Child_Status)){
            //change state if stopped
            job->state = ST;
            printf("Job [%d] (%d) stopped by signal %d\n",pid2jid(pid), pid, WTERMSIG(Child_Status));
        }
```

```
    }
    return;
}
```

while 반복문을 이용하여 끝났던 child process가 reaping 될 수 있도록 한다. waitpid 함수를 이용하여 변수 Child_Status에 저장한다. child process가 모두 끝나면 WNOHANG을 이용하여 바로 return하도록 하고, WUNTRACED를 이용해 stop process의 정보를 받아올 수 있도록 한다. child process가 끝나는 동안 함수가 한 번만 호출될 수도 있기 때문에 한 번만으로 끝나거나 멈춘 모든 child process를 처리해줄 수 있도록 while 반복문을 사용할 수밖에 없다.

Child_Status에 매크로를 이용하여 분석한다. WIFSIGNALED의 경우, child process가 signal에 의해 정상적이지 않게 종료된 것이므로 "Job terminated by signal"이라는 에러 메시지를 출력하고, deletejob 함수를 이용하여 job list에서 child process를 삭제한다. WIFEXITED의 경우, child process가 정상적으로 종료되었으므로 똑같이 deletejob 함수를 이용하여 job list에서 child process를 삭제한다. WIFSTOPPED의 경우, child process가 stop된 것이므로 job의 state를 ST로 바꾸고 "Job stopped by signal"이라는 에러 메시지를 출력한다.

---

sigint_handler

sigint_handler 함수의 역할은 ctrl-c 입력이 들어왔을 경우, 해당 signal을 current foreground job으로 전달해주는 것이다.

```
void sigint_handler(int sig)
{
    pid_t pid = fgpid(jobs);
    if (pid > 0) {
        kill(-pid, sig);
    }
    return;
}
```

먼저 fgpid 함수를 호출하여 current foreground job을 찾아 pid에 저장한다. current foreground job이 있을 때, pid에 양수가 저장되므로, pid 값이 0보다 클 때 kill 함수를 호출하여 job의 process 그룹에 시그널을 전달한다.

---

sigstp_handler

sigstp_handler 함수의 역할은 ctrl-z 입력이 들어왔을 경우, 해당 signal을 current foreground job으로 전달해주는 것이다. 사실상 하는 역할은 sigint_handler 함수와 똑같다. 따라서 코드도 똑같다.

```
void sigint_handler(int sig)
```

```
{
    pid_t pid = fgpid(jobs);
    if (pid > 0) {
        kill(-pid, sig);
    }
    return;
}
```