2019-2학기

컴퓨터 SW시스템 개론

Lab #6

20180085 무은재학부 송수민

Tiny Shell 구현

1. eval

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    int isbg;//Background에서 실행시키는 command인지 판별하기 위한 변수.
    pid_t pid;
    sigset_t mask;

    // parse the line
    isbg = parseline(cmdline, argv);//parseline method를 보면 argv-1이 &이면 1을 return 하므로, background면 1, foreground면 0.
    //eval method 설명에서 언급한 quit, jobs, & (bg or fg) command이면 아래 if로 들어가지 않고 builtin_cmd에서 처리하고 끝냅니다.
    if(!builtin_cmd(argv)) {
        sigemptyset(&mask);
        sigaddset(&mask, SIGCHLD);
        sigprocmask(SIG_BLOCK, &mask, NULL);
        //그 외 명령어를 받기 전에 각 signal에 대해 blocking을 위해 masking을 해줍니다.
        // forking for process
        if((pid = fork()) < 0)
        {
            unix_error("forking error");
        }
        else if(pid == 0)
        {
            sigprocmask(SIG_UNBLOCK, &mask, NULL);
            setpgid(0, 0);
            if(execvp(argv[0], argv) < 0)
            {
                printf("%s: Command not found\n", argv[0]);
                exit(1);
            }
        }
        else
        {
            if(!isbg)
            {
                addjob(jobs, pid, FG, cmdline);
            }
            else
            {
                addjob(jobs, pid, BG, cmdline);
            }
            sigprocmask(SIG_UNBLOCK, &mask, NULL);
            //background를 판별하는 변수인 bg를 통해, bg가 0이면은 foreground job으로 추가하고, 1이면 background job을 추가합니다.
            if (!isbg)
            {
                waitfg(pid);
            }//foreground면 foreground가 끝날때 까지 wait.
            else
            {
                printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
            }
        }
    }
}
```

명령어는 cmdline에 저장되었을 것이고, 이 명령어를 바탕으로 처리를 하는 함수이다. 먼저 background로 실행시켜야 하는지에 대해 판별하기 위해 parseline에서 보면 마지막 명령어가 & 이면 1을 return하도록 한다. 이를 return value를 isbg에 저장한다. 그 다음, quit, jobs, bg or fg라는 명령어가 들어오면 builtin_cmd에서 처리하고 1을 return 하고, 그 외 명령어에 대해서는 0을 return 하기 때문에 그에 맞게 if를 사용하여 control한다.

2. builtin_cmd

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

이 함수는 간단하다. Bg,fg가 오면 do_bgfg를 사용하고 jobs가 오면 helper function인 listjobs를 사용하고, quit이면 단순히 exit()을 호출한다.

3. do_bgfg

```
void do_bgfg(char **argv)
{
    struct job_t *job;
    /*
    struct job_t
    pid_t pid;              /job PID
    int jid;                 job ID [1, 2, ...]
    int state;              /UNDEF, BG, FG, or ST
    char cmdline[MAXLINE]  command line
    */
    char *tmp;
    int jobID;
    pid_t pid;

    tmp = argv[1];

    //ID가 존재 하지 않을 경우.
    if(tmp == NULL)
    {
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }
```

이 부분의 코드는 Parsing 된 ID가 없을 때 그것을 처리해주는 부분이다.

```
//ID가 Job ID일 때.(1,2,3,4....)
if(tmp[0] == '%')
 {
    jobID = atoi(&tmp[1]);
    job = getjobjid(jobs, jobID);
    if(job == NULL)
    {
        printf("%s: No such job\n", tmp);
        return;
    }
    else
    {
        pid = job->pid;
    }
}
//ID가 PID일 때.
else if(isdigit(tmp[0]))
 {
    pid = atoi(tmp);
    job = getjobpid(jobs, pid);
    if(job == NULL)
    {
        printf("(%d): No such process\n", pid);
        return;
    }
}
else
{
    printf("%s: argument must be a PID or %%jobid\n", argv[0]);
    return;
}
kill(-pid, SIGCONT);//PID가 속한 모든 그룹에.
```

Parsing된 ID는 두가지로 나뉘는데 하나는 JobID(1,2,3,4...)이고 하는 Process의 ID인 PID이다. 그것을 구분하는 기준은 %의 유무이고, 그걸 기준으로 적절한 IF로 들어가 해당하는 job을 지정한다. KILL(SIGCONT)는 정지된 프로세스를 계속 실행시키려 할 때 발생시키는 것이다.

```
if(!strcmp("fg", argv[0]))
{
    job->state = FG;
    waitfg(job->pid);
}
else
{
    printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
    job->state = BG;
}
```

Fg와 bg에 따라 fg이면 지정된 job의 state를 FG로 바꾸고 기다린다. BG이면 지정된 job의 state를 BG로 바꾼다.

4. waitfg

```
void waitfg(pid_t pid)
{
    struct job_t* job;
    job = getjobpid(jobs,pid);
    if(pid == 0)
    {
        return;
    }//이 함수는 Foreground를 위한 함수.
    if(job != NULL)
    {
        while(pid==fgpid(jobs)){
        }
    }
    return;
}
```

이 함수는 eval과 do_bgfg에서 사용되며, foreground의 process의 완료를 기다리는 함수이다.

5. handler _1

```
void sigchld_handler(int sig)
{
    int status;
    pid_t pid;
    /*
    WNOHANG : 어떤 자식이 종료되지 않았더라도 함수는 바로 리턴된다.
    WUNTRACED : 종료된 프로세스 뿐만 아니라 멈춘 프로세스로부터도 상태정보를 얻어온다.
    */
    while ((pid = waitpid(fgpid(jobs), &status, WNOHANG|WUNTRACED)) > 0)
    {
        if (WIFSTOPPED(status))
        { //FOR STOP
            getjobpid(jobs, pid)->state = ST;
            int jid = pid2jid(pid);
            printf("Job [%d] (%d) Stopped by signal %d\n", jid, pid, WSTOPSIG(status));
        }
        else if (WIFSIGNALED(status))
        { //FOR TERMINATE
            int jid = pid2jid(pid);
            printf("Job [%d] (%d) terminated by signal %d\n", jid, pid, WTERMSIG(status));
            deletejob(jobs, pid);
        }
        else if (WIFEXITED(status))
        {
            deletejob(jobs, pid);
        }
    }
    return;
}
```

이 함수는 Stop, terminate, exit 3가지 케이스로 나뉜다. Stop은 sigstop 또는 sigtstp를 받았을 때 일어나고, terminate는 SIGCHLD를 받았을 때 나타난다. 그에 맞게 IF를 통해 처리해주고 그에 따른 문구를 출력해준다.

6. handler _2,3

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

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

두 함수는 TERMINATE시키거나 STOP시키는 함수로 PID가 0이 아니면 KILL을 이용하여 수행한다.