CSED211 LAB6 report

20210054 정하우

<Lab 시간을 통해 배운점>

이번 랩은 직접 shell을 구현하는 것이다. 스켈레톤 코드가 있는 tsh.c 파일에서 빈칸이 있는 함수들을 코딩해서 tiny shell을 만드는 것이 이번 랩의 목표이다.

shell이란, user와 kernel 을 연결시켜주는 interface 역할을 한다. 사용자가 입력한 commend를 읽고 해석하는 프로그램이라고 볼 수 있다.

tsh.c에 있는 함수들을 구현하고 trace들을 보면서 잘 코딩했는지 확인할 수 있다. 그리고 완전히 구현된 tinyshell인 tshref를 통해 tinyshell이 어떻게 움직이는지 알 수 있다.

1. eval 함수

```
char *argv[MAXARGS];
char buf[MAXLINE];
pid_t pid;
sigset_t mask, prev_mask;
strcpy(buf, cmdline);
int bg;
bg = parseline(buf, argv);

if(argv[0]==NULL)//빈 줄인 경우
    return;
```

Commend를 받을 buf, foreground에서 돌아가는지 background에서 돌아가는지 판단해줄 bg(bg=1➔background, bg=0➔foreground), processID를 담는 pid를 선언해 줬다. Argv에 아무것도 없는 경우(빈줄일 경우) 무시한다.

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

    if((pid = fork()) == 0){ //자식 프로세스인 경우
        sigprocmask(SIG_UNBLOCK, &mask, NULL);
        setpgid(0,0);

        if(execve(argv[0],argv,environ) < 0){
            printf("%s: Command not found\n", argv[0]);
            exit(0);
        }
    }
```

```
        else if(pid < 0)
        {
            unix_error("fork error");
        }

        else{//부모 프로세스인 경우
            addjob(jobs, pid, bg? BG : FG, cmdline);
            sigprocmask(SIG_UNBLOCK, &mask, NULL);

            if(!bg)
                waitfg(pid);
            else
                printf("[%d] (%d) %s",pid2jid(pid) ,pid, cmdline);
        }
    }
    return;
```

Builtin이 아닌 경우에 대해서 처리해준다. 시그널 블록을 하기 위해 sigemptyset함수를 통해 set를 초기화 하고, sigaddset함수를 통해 SIGCHLD 명령어를 넣는다. 그리고 sigprocmask함수를 통해 시그널을 블록한다. 이전에 블록하던 시그널은 mask2에 넣는다.

자식 프로세스일 경우에는 나중에 ctrl-c를 했을 경우에도 모든 프로세스에 명령어가 전달되지 않게 하기 위해 setpid(0, 0)을 해준다.

부모프로세스인 경우는 foreground에서 돌아가는지 background에서 돌아가는지 확인한다. foreground에서 돌아갈 경우 자식프로세스를 기다려 주고 background에서 돌아갈 경

우 출력을 한다.

2. builtin_cmd 함수

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

각 commend 별로 if문으로 처리를 한다. Commend를 실행했으면 1을, 실행하지 못했으면 0을 return한다.

3. do_bfbg 함수

```
void do_bgfg(char **argv)
{
    if(argv[1] == NULL)
    {
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }
```

Bg나 fg다음에 아무것도 나오지 않는 경우는 commend가 더 필요하다고 출력하고 return한다.

```
else
{
    if(!isdigit(argv[1][0]) && argv[1][0]!= '%')
    {
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }
    else
    {
        struct job_t *job;
        if(isdigit(argv[1][0]))//pid
        {
            job = getjobpid(jobs,(pid_t) atoi(argv[1]));
            if(job == NULL)
            {
                printf("(%s): No such process\n",argv[1]);
                return;
            }
        }
        else if(argv[1][0] == '%')//job
        {
            job = getjobjid(jobs, atoi(&argv[1][1]));
            if(job == NULL)
            {
                printf("%s: No such job\n", argv[1]);
                return;
            }
        }
    }
```

또하나 bg나 fg다음에 오는 명령어의 형식이 잘못된 경우도 예외처리 해 주었다. 숫자가 아니거나(job), %로 시작하지 않는 경우는 형식에 맞춘 명령어를 작성하라고 출력하고 return한다. 만약 명령어 형식이 맞아도 list에 없으면 예외처리를 해 주었다.

```
        if(strcmp(argv[0],"fg")==0){
            job -> state = FG;
            kill(-(job -> pid),SIGCONT);
            waitfg(job -> pid);
        }
        else if(strcmp(argv[0],"bg")==0){
            job -> state = BG;
            printf("[%d] (%d) %s",pid2jid(job -> pid) ,job -> pid, job -> cmdline);
            kill(-(job -> pid),SIGCONT);
        }
    }
    return;
}
```

명령어가 형식에 맞고, 존재했으면 kill 함수를 이용해 프로세스를 재식작한다. background 인지 foreground인지 if문을 통해 구별하고 foreground에서 돌아가는 경우 현재 fg가 종료될때까지 wait를 통해 기다린다.

4. waitfg 함수

```
void waitfg(pid_t pid)
{
    while((fgpid(jobs)==pid)&&(getjobpid(jobs,pid)->state ==FG))
        sleep(1);

    return;
}
```

인자로 받은 pid가 foregroup의 ID와 같은지 비교하고, 상태가 FG인지 판단한다. 맞으면 1분간 대기한다. 이 반복문을 통해 프로세스가 foreground가 아닐 때 까지 반복한다.

5. sigchld_handler 함수

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

waitpid함수를 통해 모든 자식 프로세스를 제거한다. 첫 인자로 -1을 받고, 세번째 인자로 WNOHANG|WUNTRACED 를 받아 모든 자식 프로세스가 종료한 것을 판단할 수 있다.

Signal에 의해 프로세스가 종료되었으면 signal과 processID를 출력하고 deletejob를 한다. return되거나 exit된 경우는 그냥 deletejob를 한다. 만약 프로세스를 stop해야하는 경우는 getjobpid를 이용해 job의 상태를 ST로 바꿔주고 정보를 출력해준다.

6. sigint_handler 함수

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

Ctrl-c 를 입력받은 경우 foreground 에 signal를 보내는 함수다. Foreground 가 아닌 경우는 그냥 return해주고, 나머지 경우에 대해서는 foreground 프로세스 전부에게 kill함수를 통해 signal을 전해준다. 이를 위해 -pid(음수)를 인자로 넣는다

7. sigtstp_handler 함수

```
void sigtstp_handler(int sig)
{
    pid_t pid = fgpid(jobs);
    if (pid > 0){
        if(kill(-pid, SIGTSTP)<0)
            unix_error("kill (tstp) error");
    }
        return;
}
```

Ctrl-z를 입력받을 경우 foreground에 signal을 보내는 함수다. kill과정에서 error가 나는 경우도 처리해준다.

<학습내용>

이번 랩을 통해서 shell이 어떻게 구현되는지 전반적인 지식을 얻게 되었다. Waitpid, kill함수가 왜 존재하는지 알게 되었고, 특히 fork 개념을 확실히 이해하고 랩을 하니 프로세스의 전반적인 구조를 잘 알게 된거 같다.