

## Lab02 – Shell

정성태 2018-15515

### eval()

eval()은 command line을 입력 받고 그에 따라 명령을 수행하는 shell의 기본적인 기능을 수행하는 함수이다. 입력된 command에 알맞은 동작을 수행할 수 있도록 구현했다.

```
if (strcmp(cmdline, "\n") == 0) {
    return;                // empty cmdline
}

bg = parseline(cmdline, argv);
command = argv[0];

/* Run built-in commands */
if (builtin_cmd(argv)) {
    return;
}
```

함수 초반에는 if문을 삽입해 command line이 비어 있는 경우 아무런 동작도 하지 않고 return 하도록 했다. 그 다음에는 이미 구현되어 있는 parseline() 함수를 호출하여 command line에 대한 parsing을 수행하고 그 결과를 string array인 argv에 저장한다. argv의 첫 번째 element는 command가 된다.

이제 built-in command 실행을 위한 builtin\_cmd() 함수를 호출한다. command가 quit, jobs, bg, fg 중 하나에 해당하는 경우 그 명령을 실행하며, 그렇지 않은 경우 0이 반환된다.

```
/* Block SIGCHLD before forking */
if (sigemptyset(&sigset) < 0) {                // initialize sigset
    unix_error("sigemptyset() error");
}
if (sigaddset(&sigset, SIGCHLD) < 0) {          // add SIGCHLD to block
    unix_error("sigaddset() error");
}
if (sigprocmask(SIG_BLOCK, &sigset, NULL) < 0) { // block SIGCHLD
    unix_error("sigprocmask() error");
}
```

그 다음에는 sigprocmask()를 사용해 SIGCHLD를 block 처리한다. 이는 child가 job list에 추가되기도 전에 SIGCHLD가 전달되어 deletejob()이 호출되는 것을 막기 위해서이다.

```

else if (chpid != 0) {
    /* Parent */
    if (bg) {
        /* Background */
        printf("[%d] (%d) %s", nextjid, chpid, cmdline);
        if (addjob(jobs, chpid, BG, cmdline) == 0) {
            app_error("addjob() failed");
        }
        if (sigprocmask(SIG_UNBLOCK, &sigset, NULL) < 0) { // unblock SIGCHLD
            unix_error("sigprocmask() error");
        }
    }
    else {
        /* Foreground */
        if (addjob(jobs, chpid, FG, cmdline) == 0) {
            app_error("addjob() failed");
        }
        if (sigprocmask(SIG_UNBLOCK, &sigset, NULL) < 0) { // unblock SIGCHLD
            unix_error("sigprocmask() error");
        }
        waitfg(chpid); // wait until the process terminates
    }
}

```

준비가 완료되면 입력된 프로그램 실행을 위해 fork()를 호출해 프로세스의 흐름을 분기한다. parent process는 입력된 프로그램이 background로 실행되어야 하는지, foreground로 실행되어야 하는지에 따라서 다르게 동작해야 한다. background일 경우 우선 log message를 출력한다. 그런 다음 addjob()을 호출해 job list에 새로운 작업을 추가한 뒤, sigprocmask()로 SIGCHLD에 대한 block 처리를 해제한다. 이제 eval()은 종료되고, shell은 다음 명령을 기다리게 된다.

foreground로 실행해야 하는 경우에도 마찬가지로 addjob()과 sigprocmask()를 호출한다. 다만 이때는 child process가 종료될 때까지 eval()을 종료하지 않고 기다리기 위해 waitfg()를 호출해야 한다.

```

else {
    /* Child */
    if (setpgid(0, 0) < 0) { // create new process group
        unix_error("setpgid() error");
    }
    if (execve(argv[0], argv, environ) < 0) { // run input program
        printf("%s: Command not found\n", argv[0]);
        exit(0);
    }
}

```

child process에서는 우선 setpgid()를 설정해 스스로를 새로운 process group에 속하도록 만든다. 이는 shell에 전달된 SIGINT, SIGTSTP 등 signal이 child process로까지 전달되어버리는 것을 막기 위한 조치이다. 이제 execve()를 호출해 입력된 프로그램을 실행시키면 된다. 실패할 경우 command가 유효하지 않은 것이므로 에러 메시지를 출력해준다.

## builtin\_cmd()

---

```
int builtin_cmd(char **argv)
{
    char* command = argv[0];        // command to run

    if (strcmp(command, "quit") == 0) {
        exit(0);
    }
    if (strcmp(command, "jobs") == 0) {
        listjobs(jobs);
        return 1;
    }
    if (strcmp(command, "bg") == 0 || strcmp(command, "fg") == 0) {
        do_bgfg(argv);
        return 1;
    }

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

builtin\_cmd()는 quit, jobs, bg, fg 등의 built-in command를 실행해주는 함수이다. quit일 경우 exit(0)으로 프로세스를 종료하고, jobs인 경우 listjobs()를 호출해 현재 등록되어 있는 작업들의 목록을 보여준다. bg 또는 fg가 입력된 경우 do\_bgfg()를 호출하고, 해당하는 command가 없는 경우 0을 반환하며 종료한다.

## do\_bgfg()

---

built-in command인 bg와 fg를 실행해주는 함수이다.

```
/* Parse argument */
if (arg == NULL) {        // no argument received
    printf("%s command requires PID or %%jobid argument\n", command);
    return;
}
if (*arg == '%') {        // jid received
    is_jid = 1;
    arg++;
}
while ((ch = arg[l++]) != 0) {
    if (!isdigit(ch)) {    // not number
        printf("%s: argument must be a PID or %%jobid\n", command);
        return;
    }
}
id = atoi(arg);            // PID or JID saved in this variable
```

우선 입력된 argument에 대한 parsing을 수행해 PID 또는 JID를 추출한다. 만약 argument가 비어 있으면 오류 메시지를 출력하며 종료한다. argument가 %로 시작하면 JID임을 의미하므로 그 다음 숫자부터 읽어 들인다. 그리고 나서 argument를 한 글자씩 읽으며 숫자가 맞는지를 검사하고, 숫자가 아닌 문자가 포함되어 있으면 오류 메시지를 출력하며 종료한다. 이때 is\_jid라는 boolean 변수를 선언해 입력된 id가 PID인지 JID인지를 표시하도록 했다.

```

/* Get job object */
if (is_jid) {
    job = getjobjid(jobs, id);
    if (!job) {
        printf("%s: No such job\n", --arg);
        return;
    }
} else {
    job = getjobpid(jobs, id);
    if (!job) {
        printf("%s: No such process\n", arg);
        return;
    }
}
pid = job->pid;           // PID of target job

```

parsing이 완료되면 job list에서 해당하는 job object를 가져온다. 이때 입력된 id가 PID인지 JID인지에 따라 getjobpid(), getjobjid()를 호출한다. 추출에 성공하면 object로부터 pid를 얻을 수 있다.

```

/* Send SIGCONT */
if (kill(-1*pid, SIGCONT) < 0) {
    unix_error("kill error");
}

if (strcmp(command, "fg") == 0) {
    /* Foreground */
    job->state = FG;    // change job state
    waitfg(pid);
}
else {
    /* Background */
    job->state = BG;    // chage job state
    printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
}

```

이제 kill()을 호출해 해당 프로세스의 group에 SIGCONT를 전송한다. command가 bg인지 fg인지에 따라 job state를 적절히 변경해준 뒤, bg이면 log message를 출력하고 fg이면 waitfg()를 호출해 종료될 때까지 대기한다.

## waitfg()

---

```
void waitfg(pid_t pid)
{
    struct job_t* job;           // target job object

    job = getjobpid(jobs, pid);
    if (!job) {                  // job does not exist
        return;
    }

    /* Sleeps until target process terminates */
    while (getpgid(pid) >= 0) {  // return -1 when process does not exist
        sleep(0);               // context switch
        if (job->state == ST) {  // stop waiting when process is stopped
            break;
        }
    }
}
```

waitfg()는 eval()과 do\_bgfg()에서 호출되며, foreground 작업이 종료될 때까지 진행을 멈추는 역할을 한다. 입력된 PID에 해당하는 프로세스가 존재하는 동안 계속해서 while문을 돌며 sleep(0)을 호출한다. 만약 해당 프로세스가 종료되면 while문이 멈추게 된다. 또한 해당 프로세스가 SIGTSTP를 받아 정지되었을 때도 대기를 멈추어야 하므로 while문 내부에 job state를 확인하는 부분을 추가했다.

## sigchld\_handler()

---

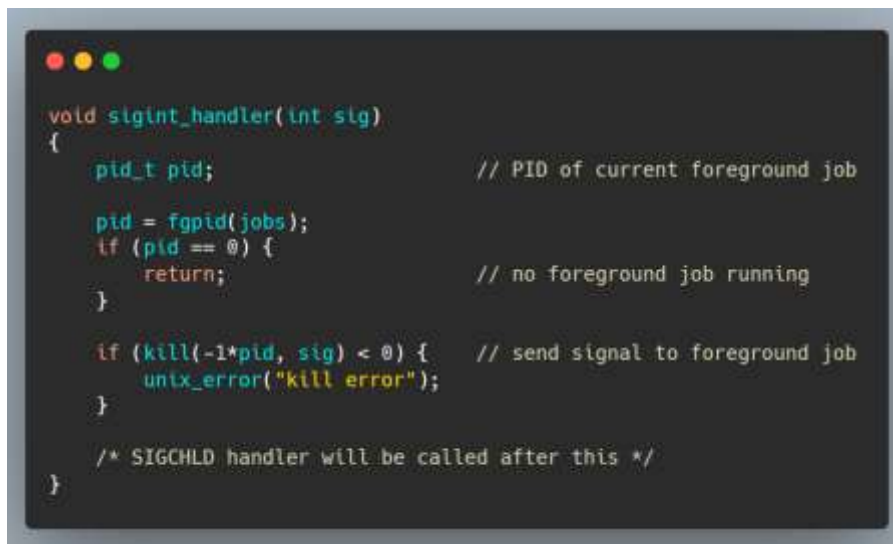
```
while ((pid = waitpid(-1, &status, WNOHANG|WUNTRACED)) > 0) {
    if (WIFSIGNALED(status)) { // if child was terminated by a signal
        jid = pid2jid(pid);
        printf("Job [%d] (%d) terminated by signal %d\n", jid, pid, WTERMSIG(status));
        if (!deletejob(jobs, pid)) { // remove from job list
            app_error("deletejob() failed");
        }
    }
    else if (WIFSTOPPED(status)) { // if child was stopped by a signal
        jid = pid2jid(pid);
        printf("Job [%d] (%d) stopped by signal %d\n", jid, pid, WSTOPSIG(status));
        job = getjobpid(jobs, pid);
        job->state = ST; // change job state
    }
    else {
        if (!deletejob(jobs, pid)) { // remove from job list
            app_error("deletejob() failed");
        }
    }
}
```

실행 중이던 child process가 종료되거나 정지되는 경우 SIGCHLD가 전송된다. 이 handler는 그럴 때 child에 대한 reap을 수행하고 적절한 처리를 하는 역할을 한다.

우선 waitpid()를 반복하며 reap 되지 못하고 남아 있는 모든 child를 reap 처리한다. 이때 WNOHANG과 WUNTRACED 옵션을 주어 더 이상 종료된 child가 남아 있지 않은 경우 대기를 중단하고, SIGTSTP로 인해 정지된 child의 정보까지 수집할 수 있도록 했다. 만약 child가 SIGINT 등 signal로 인해 종료된 경우, WIFSIGNALED(status)가 참이 된다. 이때는 적절한 log message를 출력해주고 job list에서 해당 job을 제거한다. 만약 SIGTSTP 등 signal로 인해 정지되었을 경우 WIFSTOPPED(status)가 참이 되고, 이때는 job list에서 해당 job object를 찾아 state를 ST로 변경해준다. 만약 두 경우에 해당하지 않는다면 별도의 처리 없이 job list에서 해당 job을 제거한다.

## sigint\_handler() & sigtstp\_handler()

---



```
void sigint_handler(int sig)
{
    pid_t pid;                // PID of current foreground job

    pid = fgpid(jobs);
    if (pid == 0) {
        return;                // no foreground job running
    }

    if (kill(-1*pid, sig) < 0) { // send signal to foreground job
        unix_error("kill error");
    }

    /* SIGCHLD handler will be called after this */
}
```

이들 handler는 shell에 Ctrl+C나 Ctrl+Z가 입력되었을 때 수행하는 동작을 정의하는 역할을 한다. shell은 이렇게 SIGINT나 SIGTSTP가 전달된 경우 catch 한 다음 실행되고 있는 foreground 작업에 해당 signal을 전달해야 한다.

signal이 전달되면 handler 내부에서 fgpid()를 호출해 진행 중인 foreground 작업의 PID를 받아온다. 만약 foreground 작업이 없다면 아무 일도 하지 않고 handler를 종료한다. foreground 작업이 존재한다면 같은 group에 속한 모든 프로세스를 대상으로 동일한 signal을 전송한다. 이렇게 하면 child가 또 다시 분기해 child를 생성했을 경우에도 거기까지 signal의 영향이 미치도록 할 수 있다. sigtstp\_handler() 역시 동일한 구조로 구현했다.

## 실행 결과

---

test를 수행한 결과 의도한 대로 shell이 잘 구현되었음을 확인할 수 있었다.

### test 14

```
~/system_programming/lab02_shell .....  
> 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] (7184) ./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  
Job [1] (7184) stopped by signal 20  
tsh> bg %2  
%2: No such job  
tsh> bg %1  
[1] (7184) ./myspin 4 &  
tsh> jobs  
[1] (7184) Running ./myspin 4 &
```

각종 예외 상황에 적합한 에러 메시지를 출력했다.



## test 15

```
~/system_programming/lab02_shell .....  
> make test15  
./sdriver.pl -t trace15.txt -s ./tsh -a "-p"  
#  
# trace15.txt - Putting it all together  
#  
tsh> ./bogus  
./bogus: Command not found  
tsh> ./myspin 10  
Job [1] (7479) terminated by signal 2  
tsh> ./myspin 3 &  
[1] (7484) ./myspin 3 &  
tsh> ./myspin 4 &  
[2] (7486) ./myspin 4 &  
tsh> jobs  
[1] (7484) Running ./myspin 3 &  
[2] (7486) Running ./myspin 4 &  
tsh> fg %1  
Job [1] (7484) stopped by signal 20  
tsh> jobs  
[1] (7484) Stopped ./myspin 3 &  
[2] (7486) Running ./myspin 4 &  
tsh> bg %3  
%3: No such job  
tsh> bg %1  
[1] (7484) ./myspin 3 &  
tsh> jobs  
[1] (7484) Running ./myspin 3 &  
[2] (7486) Running ./myspin 4 &  
tsh> fg %1  
tsh> quit
```

각 기능이 의도한 대로 잘 동작하고 있는 것을 확인할 수 있다.



## test 16

```
~/system_programming/lab02_shell
> make test16
./sdriver.pl -t tracel6.txt -s ./tsh -a "-p"
#
# tracel6.txt - Tests whether the shell can handle SIGTSTP and SIGINT
#               signals that come from other processes instead of the terminal.
#
tsh> ./mystop 2
Job [1] (7609) stopped by signal 20
tsh> jobs
[1] (7609) Stopped ./mystop 2
tsh> ./myint 2
Job [2] (7618) terminated by signal 2
```

## Review

---

### 어려웠던 점

- eval()에서, child의 execve() 호출이 실패한 경우 parent가 addjob()을 호출하면 안 된다고 생각해서 고민을 많이 했다. child의 execve() 실패 여부를 parent가 즉시 알 수 있어야 한다고 생각했던 것이다. 그래서 user-defined signal이나 pipe를 활용한 여러 방법을 고민해보았으나 잘 되지 않았다. 고민을 계속한 결과 잘못된 addjob()이 수행되었더라도 sigchld\_handler()에서 다시 deletejob()을 수행하면 문제가 없으며, 따라서 parent가 실패 여부를 즉시 알 필요는 없다는 결론을 내렸다.
- shell을 통해 전달되지 않고 child가 자체적으로 kill()을 이용해 생성한 signal을 parent에서 포착하는 것이 어려웠다. 검색을 해보며 waitpid()에 WUNTRACED 옵션을 전달하면 종료된 child 뿐 아니라 정지된 child의 정보까지 수집할 수 있다는 것을 알게 되었고, WIFSTOPPED, WIFSIGNALED 등 매크로를 사용해 원하는 기능을 구현할 수 있었다.
- 프로세스의 생성이나 signal 전송 등 system call의 수행 경과를 바로 확인할 수가 없어 디버깅이 쉽지 않았다. 필요한 부분만을 남긴 별도의 작은 테스트 프로그램을 작성해 실행해보거나, 프로그램 곳곳에 log message를 출력하는 부분을 추가해보면서 하나씩 해결할 수 있었다.

### 새롭게 배운 점

- 항상 사용하던 shell 프로그램의 대략적인 작동 방식을 이해하게 되었다.
- 수업 시간에 배운 fork(), execve(), wait() 등 system call과 signal handling이 실제로 어떻게 쓰이는지를 이해하고 연습해볼 수 있었다.
- atoi(), isdigit() 등 몰랐던 C 함수들을 새로 알게 되었다.
- 최종 코드에 포함되어 있지는 않으나 여러 시도를 해보며 pipe의 사용법을 알게 되었다.