컴퓨터 SW 시스템 개론 Lab 10, 11

20220923 이윤혁

1. 개요

이번 Lab 10, 11에서는 signal과 exceptional control overflow에 대해 이해하고 shell을 직접 작성한다.

2. 함수 구성

이번 Lab에서는 주어진 빈 함수 7개를 작성한다. 아래는 주어진 빈 함수이다.

```
void eval(char *cmdline);

int builtin_cmd(char **argv);

void do_bgfg(char **argv);

void waitfg(pid_t pid);

void sigchld_handler(int sig);

void sigtstp_handler(int sig);

void sigint_handler(int sig);
```

각 함수의 기능을 설명하며, 함수를 이루는 코드의 부분마다 어떻게 구성되었는지 trace 파일의 내용과 함께 설명할 것이다.

2.1. void eval(char *cmdline)

이 함수는 유저가 shell에 입력한 command line을 실행시키는 함수이다. Eval 함수의 코드를 차례로 설명할 것이다. 아래는 변수 선언이다.

```
char* argv[MAXARGS];
int is_background_job, is_built_in;
int emptyset_result, addset_result, block_result, procmask_result;
int setpgid_result, addjob_result;
int external_command_result;
pid_t pid;
sigset_t mask;
```

아래는 eval함수 코드의 일부분이다.

```
// Parse the command line and check it is a background job call
is_background_job = parseline(cmdline, argv);

// Check if command entered in the shell.
if (argv[0] == NULL)
    return; // No command entered

// Check the command is built-in command. If it so, run the corresponding built-in command
is_built_in = builtin_cmd(argv);
```

parseline 함수를 이용해 argv에 공백을 단위로 command line의 인자들을 담는다. Parseline은 반환값으로 command가 background에서 실행되는 여부를 반환하는데, 이를 is_background_job에 저장한다. 이후에 process를 실행시킬 때, job의 state를 바꿀 때 필요하기 때문이다.

argv[0]가 NULL인지 확인하여 command가 입력되었는지 확인한다. NULL인 경우 command가 입 입력되지 않은 것 이므로 함수를 조기종료한다.

builtin_cmd 함수를 이용해 입력된 command가 built-in command인지 확인한다. Builti-in command라면 built in command를 실행한다. Builtin_cmd는 반환값으로 command가 built-in command인 여부를 반환하는데, 이를 is_built_in에 저장한다. Built-in command라면 이미 command가 실행되었기 때문에 더 이상 eval함수에서 처리할 것이 없기 때문이다.

아래는 eval함수 코드의 다음 부분이다.

```
// Not a built-in command
if (!is_built_in){

    // Blocking
    emptyset_result = sigemptyset(&mask); // Clear
    addset_result = sigaddset(&mask, SIGCHLD); // Add SIGCHLD to mask
    procmask_result = sigprocmask(SIG_BLOCK, &mask, NULL); // Block SIGCHLD

    ...
}
return;
```

if문을 이용해 입력된 command가 built-in command가 아닌 경우 command의 프로세스를 실행시키기 위해 signal blocking을 실행한다. 이때 command는 child process로 돌아가기 때문에, SIGCHLD를 block해야 한다.

SIGCHLD의 Signal blocking은 3개의 일련의 절차로 이루어진다.

1. Signal 초기화
2. Mask에 SIGCHLD 추가
3. Block SIGCHLD

위 절차는 "// Blocking" 주석 아래 세 줄과 동일하다.

아래는 eval 함수 코드의 다음 부분이다.

```
pid = fork();

// This process is child
if (pid == 0) {

    // Unblock
    procmask_result = sigprocmask(SIG_UNBLOCK, &mask, NULL);

    setpgid(0, 0); // Separate process group
    external_command_result = execve(argv[0], argv, environ); // Run external command

    // External command didn't run properly
    if (external_command_result < 0) {
        printf("%s: Command not found₩n", argv[0]); // Print error message
        exit(0);
     }
}

// This process is parent
else if (pid > 0) {

    // The process is running on background
    if (is_background_job)
        addjob_result = addjob(jobs, pid, BG, cmdline); // Add job as background process

    // The process is running on foreground
    else
        addjob_result = addjob(jobs, pid, FG, cmdline); // Add job as foreground process

    procmask_result = sigprocmask(SIG_UNBLOCK, &mask, NULL);
    // The process is running on background
```

```
    if (is_background_job)
        printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline); // Print running state of background

    // The process is running on foreground
    else
        waitfg(pid); // Block until job is not running on foreground
}
```

먼저 fork 함수를 이용해 child process를 생성한다. pid는 childprocess라면 0, parent process라면 0이 아닌 값일 것이기에 pid를 이용해 분기를 생성한다.

Child process (pid == 0)는 먼저 SIGCHLD를 Unblock해야 한다. Child process는 Parent process와 변수의 같은 값을 공유하므로(공유하나 메모리를 공유하는 것은 아니다.) mask를 그대로 이용해 Unblock할 수 있다.

Setpgid(0, 0)를 통해 자신의 process group를 Parent process group으로부터 분리한다. Trace11.txt를 보면 SIGINT를 실행하는데, foreground process group이 분리되어야 shell이 제대로 작동한다.

그후 external command를 execve함수를 이용해 실행한다. 반환 값은 external command result에 저장하는데, 이 변수는 external command가 존재하여 제대로 실행되었는지를 판별할 때 사용된다.

Parent process (pid != 0)는 child process를 background에서 run하는지 foreground에서 run하는지 결정한다. Is_background_job 변수를 이용해 bg, fg 분기점을 만들고 addjob함수를 이용해 job list에 job(child process)을 추가한다. 그 후 SIGCHID를 Unblock해 child process의 signal을 받는다.

만약 bg인 경우 bg에서 job이 돌아가고 있다는 메시지를 출력한다. fg인 경우 child process가 다 돌아갈 때까지 기다려야 하므로 waitfg 함수를 실행해 process의 종료를 기다린다.

2.2. int builtin_cmd(char **argv)

이 함수는 argv[0]의 command가 built-in command일 경우 그 command를 실행한다. built-in command가 들어온 경우 1을, 그렇지 않으면 0을 반환한다.

아래는 builtin_cmd의 코드이다.

```
// Built-in command quit
if (strcmp(argv[0], "quit") == 0) {
    exit(0); // Quit program
}

// Built-in command jobs
```

```
else if (strcmp(argv[0], "jobs") == 0){
	listjobs(jobs); // List jobs
	return 1;
}

// Built-in command bg
else if (strcmp(argv[0], "bg") == 0) {
	do_bgfg(argv); // Run process in background
	return 1;
}

// Built-in command fg
else if (strcmp(argv[0], "fg") == 0) {
	do_bgfg(argv); // Run process in foreground
	return 1;
}

return 0;
```

trace02.txt, trace05.txt, trace09.txt, trace10.txt에서 각각 quit, jobs, fg, bg를 실행한다. built-in command는 이 4개로, built-in command에 대응되는 함수는 다음과 같다.

1. quit: exit()
2. jobs: listjobs()
3. bg: do_bgfg()
4. fg: do_bgfg()

argv[0]와 각 built-in command를 비교해 일치하는 경우 built-in command에 대응되는 함수를 실행하고 1을 반환하며, 일치하는 경우가 없는 경우 0을 반환한다.

2.3. void do_bgfg(char **argv)

이 함수는 background job 또는 foreground job을 실행하는 함수이다. argv의 조건을 이용한 함수 조기종료문들은 trace14.txt로부터 출력 결과를 비교해 구현했다.

아래는 변수 선언이다.

```
struct job_t* job;
int job_id;
pid_t pid;
```

```
int is_job_id = 1, is_pid = 1;

int i;
int kill_result;
```

아래는 do_bgfg() 코드의 일부분이다.

```
// Check if the first-entered argument is fg or bg
if (!(strcmp("fg", argv[0]) == 0 || strcmp("bg", argv[0]) == 0))
    return;
```

argv[0]가 fg 또는 bg인지 검사하여, 아니라면 함수를 조기 종료한다. builtin_cmd()로 bg 또는 fg가 아닌 argv[0]가 들어올 수는 없으나 이 함수가 다른 함수에서도 쓰일 수 있음을 고려해 이 구문을 추가했다.

아래는 do_bgfg() 코드의 다음 부분이다.

```
// Check if the jid or pid is entered in the shell.
if (argv[1] == NULL) {
    printf("%s command requires PID or %%jobid argument₩n", argv[0]); // Print error message
    return;
}
```

argv[1]에 job id 또는 pid가 들어왔는지 확인한다. 그렇지 않은 경우 에러메시지를 출력하고 함수를 조기종료한다.

아래는 do_bgfg() 코드의 다음 부분이다.

```
// Check if the second-entered argument is job id
is_job_id = argv[1][0] == '%';
for(i = 1; argv[1][i] != '₩0' && argv[1][i] != ' '; i++)
    is_job_id = is_job_id && '0' <= argv[1][i] && argv[1][i] <= '9';

// Check if the second-entered argument is pid
for (i = 0; argv[1][i] != '₩0' && argv[1][i] != ' '; i++)
    is_pid = is_pid && '0' <= argv[1][i] && argv[1][i] <= '9';
```

is_job_id는 argv[1]에 job id가 들어왔는지 판별한 결과를 저장하는 변수이다. argv[1]이 job id의 format인

```
%{job_id: int}  (중괄호는 format 자체에 포함하지 않음)
```

에 부합하는지의 여부를 저장한다. for문을 이용해 is_job_id의 결과를 계산했다.

is_pid는 argv[1]에 pid가 들어왔는지 판별한 결과를 저장하는 변수이다. argv[1]이 pid format인

```
{pid: int}  (중괄호는 format 자체에 포함하지 않음)
```

에 부합하는지의 여부를 저장한다. for문을 이용해 is_pid의 결과를 계산했다.

아래는 do_bgfg() 코드의 다음 부분이다.

```
// The second-entered argument is job id
if (is_job_id) {
    job_id = atoi(argv[1] + 1); // Type cast job id from string to int
    job = getjobjid(jobs, job_id); // Get job using job id

    // The corresponding job of entered jid is not valid
    if (job == NULL){
        printf("%s: No such job₩n", argv[1]); // Print error message
        return;
    }

    // The corresponding job of entered jib is valid
    else
        pid = job->pid; // Get pid of job
}
```

if문과 is_job_id를 이용해 argv[1]이 job id인지 판별하고, job id인 경우 argv[1]에서 첫 글자 %를 제거하고 int의 job id를 추출한다. getjobjid() 함수를 이용해 job id로부터 job id에 대응되는 job을 반환받는다. getjobjid는 job id에 해당하는 job이 없는 경우 NULL을 반환하므로, 이를 검사해 job이 없는 경우 에러메시지를 출력하고 함수를 조기 종료한다.

job이 있는 경우 job struct의 pid에 접근해 job의 pid를 pid 변수에 저장한다. pid 변수는 후에 kill 함수에서 필요하기에 저장해야 한다.

아래는 do_bgfg() 코드의 다음 부분이다.

```
// The second-entered argument is pid
else if (is_pid) {
    pid = atoi(argv[1]); // Type cast pid from string to int
    job = getjobpid(jobs, pid); // Get job using pid

    // The corresponding job of entered pid is not valid
    if (job == NULL){
        printf("(%s): No such process₩n", argv[1]); // Print error message
        return;
    }
    // The corresponding job of entered jib is valid
    else
```

```
        job_id = job->jid; // Get job id of job
}
```

else if문과 is_pid를 이용해 argv[1]이 pid인지 판별하고, pid인 경우 argv[1] pid_t의 pid를 추출한다. getjobpid() 함수를 이용해 pid로부터 pid에 대응되는 job을 반환받는다. getjobpid는 pid에 해당하는 job이 없는 경우 NULL을 반환하므로, 이를 검사해 job이 없는 경우 에러메시지를 출력하고 함수를 조기 종료한다.

job이 있는 경우 job struct의 jid에 접근해 job의 job id를 job_id 변수에 저장한다. job_id 변수는 후에 process가 background에서 돌아갈 때 이를 로깅할 메시지에 필요하기에 저장해야 한다.

아래는 do_bgfg() 코드의 다음 부분이다.

```
// The second-entered argument is not jid nor pid
else{
    printf("%s: argument must be a PID or %%jobid₩n", argv[0]); // Print error message
    return;
}
```

is_job_id 그리고 is_pid가 0 인 경우 argv[1]에 잘못된 인자가 들어온 것이기에 에러 메시지를 출력하고 함수를 조기 종료한다.

아래는 do_bgfg() 코드의 다음 부분이다

```
// Kill process
kill_result = kill(-pid, SIGCONT);

// The first-entered argument is foreground process
if (strcmp("fg", argv[0]) == 0) {
    job->state = FG; // Change state of the job to foreground process
    waitfg(pid); // Block until job is not running on foreground
}

// The first-entered argument is background process
else if (strcmp("bg", argv[0]) == 0) {
    job->state = BG; // Change state of the job to background process
    printf("[%d] (%d) %s", job_id, pid, job->cmdline); // Print process message
}

return;
```

kill()함수를 이용해 해당하는 pid의 process에 SIGCONT를 전달한다. 이는 중지된 job(process)의

경우 job을 재실행시키라는 signal이다.

그 후 if문을 이용해 argv[0]가 fg 인지 bg인지 판별한다. fg인 경우 해당하는 job의 state를 미리 선언한 상수 FG로 바꾸고, waitfg()를 실행해 process가 foreground에서 실행이 끝날 때(중단도 포함한다)까지 기다린다. bg인 경우 해당하는 job의 state를 미리 선언한 상수 BG로 바꾸고, process가 돌아가고 있다는 메시지를 출력한다.

모든 절차가 끝나고 do_bgfg() 함수를 정상 종료한다.

2.4. void waitfg(pid_t pid)

이 함수는 foreground에서 돌아가는 process가 끝날 때(중단도 포함한다)까지 기다리는 함수이다.

아래는 함수의 코드이다.

```
struct job_t* job;
job = getjobpid(jobs, pid);

// The corresponding job of entered pid is not valid
if (job == NULL)
    return;

// Wait until job that is running on foreground is not running on foreground
while ((pid == fgpid(jobs)))
    sleep(1);

return;
```

pid와 getjobpid()를 이용해 job을 반환 받는다. job이 NULL이라면 pid에 해당하는 job이 없는 것이므로 함수를 조기종료한다.

그렇지 않은 경우 현재 foreground 에서 돌아가는 process의 pid를 받는 fgpid를 이용해 현재 foreground에서 돌아가는 process의 pid가 입력받은 pid와 같은지 판별한다. 같다면 foreground에서 우리가 종료되길 원하는 process가 돌아가고 있음을 의미하므로 sleep()함수를 통해 1초씩 기다린다. process가 끝나면 sleep()이 실행되는 while loop가 멈춰 함수가 정상종료하도록 코드를 구성했다.

2.5. void sigchld_handler(int sig)

이 함수는 child process가 종료되었을 때(zombie process화, 정지 또는 인터럽트 등) 받는 SIGCHLD를 처리하는 핸들러 함수이다.

아래는 변수 선언이다.

```
struct job_t *job;
int status;
pid_t pid;
int job_id;
int delete_result;
```

아래는 sigchld_handler()의 코드 일부분이다.

```
while ((pid = waitpid(-1, &status, WNOHANG|WUNTRACED)) > 0) {
…
}
```

waitpid()를 실행해 child process를 기다린다. waitpid()의 첫 번째 argument로 -1을 넣어 임의의 child process에 대해 기다리도록 한다. 두 번째 argument로 status 변수의 주소를 넣어 child process의 status를 받는다. 이 status를 WIFSTOPPED, WIFSIGNALED, WIFEXITED 매크로에 넣어 child process가 어떻게 종료되었는지 알 수 있다.

waitpid()는 기다리는 함수인데, child process가 끝나기까지 handler가 정지해 있으면 안 된다. WNOHANG 옵션을 이용해 모든 child process가 실행 중인 경우 waitpid()가 즉시 0을 반환하도록 한다. WUNTRACED도 옵션에 포함하여 child process 중 하나 이상이 종료 상태인 경우 그 중 한 process의 pid를 즉시 반환하도록 한다. 이 두 옵션을 logical or로 묶어 waitpid의 세 번째 argument로 전달하면 waitpid()에 의해 handler가 정지되지 않는다.

아래는 sigchld_handler() 코드의 다음 부분이다.

```
// Process is stopped
if (WIFSTOPPED(status)) {
    // Change the state of corresponding job of pid to stopped
    job = getjobpid(jobs, pid);
    job->state = ST;

    job_id = pid2jid(pid); // Get job id

    printf("Job [%d] (%d) Stopped by signal %d₩n", job_id, pid, WSTOPSIG(status)); // Print log
messsage
}
```

WIFSTOPPED(status)가 1이라면, statue에 해당하는 child process는 stop된 것이다.

getjobpid() 함수를 이용해 job을 반환 받고, 해당 job의 state를 미리 선언한 상수 ST로 바꾼다.

그 후 job_id를 구해 job이 stop되었다는 메시지를 출력한다.

아래는 sigchld_handler() 코드의 다음 부분이다.

```
// Process is terminated (invalid)
else if (WIFSIGNALED(status)) {
	job_id = pid2jid(pid); // Get job id

	printf("Job [%d] (%d) terminated by signal %d₩n", job_id, pid, WTERMSIG(status)); // Print
log messsage

	// Delete the corresponding job of pid
	delete_result = deletejob(jobs, pid);
}
```

WIFSIGNALED(status)가 1이라면 child process가 signal에 의해 terminated된 것이다. job id를 알아내 job이 terminate 되었다는 메시지를 출력한다.

그 후 deletejob() 함수를 이용해 terminate 된 job을 제거한다.

아래는 sigchld_handler() 코드의 다음부분이다.

```
// Process is exited
else if (WIFEXITED(status)) {

	// Delete the corresponding job of pid
	delete_result = deletejob(jobs, pid);
```

WIFEXITED(status)가 1이라면 child process가 exit이나 return에 의해 종료된 것이므로, pid에 해당하는 job을 jobs에서 제거하도록 deletejob()을 실행한다.

2.6. void sigint_handler(int sig) & void sigtstp_handler(int sig)

이 두 함수는 코드의 내용이 동일하여 동시에 묶어서 설명한다. sigint_handler()는 interrupt signal이, sigtstp_handler()는 temporary stop signal을 처리하는 handler이다. trace06.txt trace07.txt trace08.txt trace11.txt, trace12.txt, trace16.txt에서 작동여부를 파악할 수 있다.

아래는 sigint_handler()와 sigtstp_handler() 코드이다.

```
pid_t pid;
int kill_result;

pid = fgpid(jobs); // Get pid foreground job

// No foreground job detected
if (pid == 0)
	return;
```

```
// Kill process
kill_result = kill(-pid, sig);


return;
```

fgpid()를 이용해 foreground에서 돌아가는 process의 pid를 알아낸다. fgpid()는 foreground에서 돌아가는 process가 없는 경우 0을 반환하므로, pid가 0이라면 함수를 조기 종료한다.

kill 함수를 이용해 해당 pid를 가진 foreground process group 전체에 signal을 전달한다. 이 때 foreground process group 전체에 signal을 전달하기 위해 kill의 첫 번째 argument로 pid 대신 -pid를 입력해준다.

3. tshref와 trace 결과 비교

주어진 레퍼런스 파일 tshref와 출력 값을 비교해 구현이 잘 되었는지를 파악할 것이다.

3.1. trace01.txt

```
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
```

3.2. trace02.txt

```
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace02.txt -s ./tsh -a "-p"
#
# trace02.txt - Process builtin quit command.
#
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace02.txt -s ./tshref -a "-p"
#
# trace02.txt - Process builtin quit command.
#
```

3.3. trace03.txt

```
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace03.txt -s ./tsh -a "-p"
#
# trace03.txt - Run a foreground job.
#
tsh> quit
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace03.txt -s ./tshref -a "-p"
#
# trace03.txt - Run a foreground job.
#
tsh> quit
```

3.4. trace04.txt

```
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace04.txt -s ./tsh -a "-p"
#
# trace04.txt - Run a background job.
#
tsh> ./myspin 1 &
[1] (27895) ./myspin 1 &
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace04.txt -s ./tshref -a "-p"
#
# trace04.txt - Run a background job.
#
tsh> ./myspin 1 &
[1] (27907) ./myspin 1 &
```

background job의 pid는 다를 수 밖에 없다.

3.5. **trace05.txt**

```
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace05.txt -s ./tsh -a "-p"
#
# trace05.txt - Process jobs builtin command.
#
tsh> ./myspin 2 &
[1] (28227) ./myspin 2 &
tsh> ./myspin 3 &
[2] (28229) ./myspin 3 &
tsh> jobs
[1] (28227) Running ./myspin 2 &
[2] (28229) Running ./myspin 3 &
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace05.txt -s ./tshref -a "-p"
#
# trace05.txt - Process jobs builtin command.
#
tsh> ./myspin 2 &
[1] (28296) ./myspin 2 &
tsh> ./myspin 3 &
[2] (28303) ./myspin 3 &
tsh> jobs
[1] (28296) Running ./myspin 2 &
[2] (28303) Running ./myspin 3 &
```

3.6. **trace06.txt**

```
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace06.txt -s ./tsh -a "-p"
#
# trace06.txt - Forward SIGINT to foreground job.
#
tsh> ./myspin 4
Job [1] (29860) terminated by signal 2
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace06.txt -s ./tshref -a "-p"
#
# trace06.txt - Forward SIGINT to foreground job.
#
tsh> ./myspin 4
Job [1] (30024) terminated by signal 2
```

3.7. **trace07.txt**

```
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace07.txt -s ./tsh -a "-p"
#
# trace07.txt - Forward SIGINT only to foreground job.
#
tsh> ./myspin 4 &
[1] (30185) ./myspin 4 &
tsh> ./myspin 5
Job [2] (30187) terminated by signal 2
tsh> jobs
[1] (30185) Running ./myspin 4 &
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace07.txt -s ./tshref -a "-p"
#
# trace07.txt - Forward SIGINT only to foreground job.
#
tsh> ./myspin 4 &
[1] (30239) ./myspin 4 &
tsh> ./myspin 5
Job [2] (30244) terminated by signal 2
tsh> jobs
[1] (30239) Running ./myspin 4 &
```

3.8. **trace08.txt**

```
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace08.txt -s ./tsh -a "-p"
#
# trace08.txt - Forward SIGTSTP only to foreground job.
#
tsh> ./myspin 4 &
[1] (30452) ./myspin 4 &
tsh> ./myspin 5
Job [2] (30456) Stopped by signal 20
tsh> jobs
[1] (30452) Running ./myspin 4 &
[2] (30456) Stopped ./myspin 5
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace08.txt -s ./tshref -a "-p"
#
# trace08.txt - Forward SIGTSTP only to foreground job.
#
tsh> ./myspin 4 &
[1] (30527) ./myspin 4 &
tsh> ./myspin 5
Job [2] (30532) stopped by signal 20
tsh> jobs
[1] (30527) Running ./myspin 4 &
[2] (30532) Stopped ./myspin 5
```

3.9. **trace09.txt**

```
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace09.txt -s ./tsh -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (30804) ./myspin 4 &
tsh> ./myspin 5
Job [2] (30807) Stopped by signal 20
tsh> jobs
[1] (30804) Running ./myspin 4 &
[2] (30807) Stopped ./myspin 5
tsh> bg %2
[2] (30807) ./myspin 5
tsh> jobs
[1] (30804) Running ./myspin 4 &
[2] (30807) Running ./myspin 5
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace09.txt -s ./tshref -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (30876) ./myspin 4 &
tsh> ./myspin 5
Job [2] (30880) stopped by signal 20
tsh> jobs
[1] (30876) Running ./myspin 4 &
[2] (30880) Stopped ./myspin 5
tsh> bg %2
[2] (30880) ./myspin 5
tsh> jobs
[1] (30876) Running ./myspin 4 &
[2] (30880) Running ./myspin 5
```

3.10. **trace10.txt**

```
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace10.txt -s ./tsh -a "-p"
#
# trace10.txt - Process fg builtin command.
#
tsh> ./myspin 4 &
[1] (31133) ./myspin 4 &
tsh> fg %1
Job [1] (31133) Stopped by signal 20
tsh> jobs
[1] (31133) Stopped ./myspin 4 &
tsh> fg %1
tsh> jobs
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace10.txt -s ./tshref -a "-p"
#
# trace10.txt - Process fg builtin command.
#
tsh> ./myspin 4 &
[1] (31231) ./myspin 4 &
tsh> fg %1
Job [1] (31231) stopped by signal 20
tsh> jobs
[1] (31231) Stopped ./myspin 4 &
tsh> fg %1
tsh> jobs
```

3.11. trace11.txt

보고서를 쓰는 시점에서 다른 학생들의 process와 출력결과가 섞여 사진으로 결과를 비교하기 힘드나, trace 파일에 나온대로 정상작동함을 확인했다.

3.12. trace12.txt

보고서를 쓰는 시점에서 다른 학생들의 process와 출력결과가 섞여 사진으로 결과를 비교하기 힘드나, trace 파일에 나온대로 정상작동함을 확인했다.

3.13. trace13.txt

보고서를 쓰는 시점에서 다른 학생들의 process와 출력결과가 섞여 사진으로 결과를 비교하기 힘드나, trace 파일에 나온대로 정상작동함을 확인했다.

3.14. trace14.txt

보고서를 쓰는 시점에서 서버가 혼잡하여 기존 trace14.txt로는 ref와 결과가 다르게 나온다. trace 파일의 SLEEP 2구문을 SLEEP4 (Lab Q&A의 해결법을 참조했다.) SLEEP 4로 바꾸면 ref와 결과가 같게 나온다.

```
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace14.txt -s ./tsh -a "-p"
#
# trace14.txt - Simple error handling
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 4 &
[1] (5273) ./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] (5273) Stopped by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (5273) ./myspin 4 &
tsh> jobs
[1] (5273) Running ./myspin 4 &
```

```
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace14.txt -s ./tshref -a "-p"
#
# trace14.txt - Simple error handling
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 4 &
[1] (5373) ./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] (5373) stopped by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (5373) ./myspin 4 &
tsh> jobs
[1] (5373) Running ./myspin 4 &
```

3.15. **trace15.txt**

```
[leeyoonhyuk0@programming2 shlab-handout]$ ./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] (2334) terminated by signal 2
tsh> ./myspin 3 &
[1] (2361) ./myspin 3 &
tsh> ./myspin 4 &
[2] (2367) ./myspin 4 &
tsh> jobs
[1] (2361) Running ./myspin 3 &
[2] (2367) Running ./myspin 4 &
tsh> fg %1
Job [1] (2361) Stopped by signal 20
tsh> jobs
[1] (2361) Stopped ./myspin 3 &
[2] (2367) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (2361) ./myspin 3 &
tsh> jobs
[1] (2361) Running ./myspin 3 &
[2] (2367) Running ./myspin 4 &
tsh> fg %1
tsh> quit
```

```
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace15.txt -s ./tshref -a "-p"
#
# trace15.txt - Putting it all together
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 10
Job [1] (2573) terminated by signal 2
tsh> ./myspin 3 &
[1] (2620) ./myspin 3 &
tsh> ./myspin 4 &
[2] (2627) ./myspin 4 &
tsh> jobs
[1] (2620) Running ./myspin 3 &
[2] (2627) Running ./myspin 4 &
tsh> fg %1
Job [1] (2620) stopped by signal 20
tsh> jobs
[1] (2620) Stopped ./myspin 3 &
[2] (2627) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (2620) ./myspin 3 &
tsh> jobs
[1] (2620) Running ./myspin 3 &
[2] (2627) Running ./myspin 4 &
tsh> fg %1
tsh> quit
```

3.16. **trace16.txt**

```
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace16.txt -s ./tsh -a "-p"
#
# trace16.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] (3699) Stopped by signal 20
tsh> jobs
[1] (3699) Stopped ./mystop 2
tsh> ./myint 2
Job [2] (3728) terminated by signal 2
[leeyoonhyuk0@programming2 shlab-handout]$ ./sdriver.pl -t trace16.txt -s ./tshref -a "-p"
#
# trace16.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] (3773) stopped by signal 20
tsh> jobs
[1] (3773) Stopped ./mystop 2
tsh> ./myint 2
Job [2] (3804) terminated by signal 2
```

모든 trace file에 대해서 정상 작동함을 확인했다.

4. 결론

이번 Lab을 통해 Shell을 직접 구현하면서 Signaling 및 exceptional control flow에 대해서 자세히 이해할 수 있었다. signal을 다루는 함수 및 매크로들은 사용해본 적이 없어서 사용하는데 애를 먹었으나, writeup lab에 작성된 함수들과 교재에 나온 예시를 통하여 사용처를 알 수 있었다.