# Shell Lab

20230024 문요준

## [ eval ]

<code>

```
void eval(char *cmdline)
{
    char *cmd_args[MAXARGS];  // 명령어 인자 저장 배열
    int  is_background;       // 백그라운드 실행 여부 플래그 변수

    sigset_t signal_mask;     // 시그널 마스크 저장 변수
    pid_t    process_id;      // 자식 프로세스 ID 저장 변수

    // 명령어 파싱 및 백그라운드 여부 확인
    is_background = parseline(cmdline, cmd_args);

    // 명령어 비어 있으면 종료
    if (cmd_args[0] == NULL) return;

    // 빌트인 명령어가 아니면 실행
    if (!builtin_cmd(cmd_args))
    {
        // 시그널 마스크 설정 (SIGCHLD 차단)
        sigemptyset(&signal_mask);                  // 시그널 마스크 초기화
        sigaddset(&signal_mask, SIGCHLD);           // 시그널 마스크에 SIGCHLD 추가
        sigprocmask(SIG_BLOCK, &signal_mask, NULL);  // 부모 프로세스가 SIGCHLD 차단 (자식 프로세스 생성 및 실행 동안 부모 프로세스 방해 금지 목적)


        // 프로세스 생성 실패
        if ((process_id = fork()) < 0)
        {
            return;
        }

        // 프로세스 생성
        else if (process_id == 0)
        {
            // 자식 프로세스
            sigprocmask(SIG_UNBLOCK, &signal_mask, NULL);  // 시그널 차단 해제
            setpgid(0,0);                                  // 자식 프로세스를 독립된 프로세스 그룹으로 설정 -> 백그라운드/포그라운드 작업 구분

            // 명령어 실행
            if (execve(cmd_args[0], cmd_args, environ) < 0)     // 프로그램 경로, 인자 배열, 환경 변수
            {
                printf("%s: Command not found\n", cmd_args[0]); // 에러메시지 출력
                exit(0);
            }
        }
```

```
        // 부모 프로세스
        else
        {
            sigprocmask(SIG_UNBLOCK, &signal_mask, NULL); // 시그널 차단 해제

            // 포그라운드 실행
            if (!is_background)
            {
                addjob(jobs, process_id, FG, cmdline);    // 작업 추가
                waitfg(process_id);                       // 자식 프로세스가 종료될 때까지 대기
            }
            // 백그라운드 실행
            else
            {
                addjob(jobs, process_id, BG, cmdline);    // 작업 추가
                printf("[%d] (%d) %s\n", pid2jid(process_id), process_id, cmdline);
            }
        }
        return;
    }
}
```

eval 함수는 사용자가 입력한 명령어를 파싱하고 명령어가 빌트인인지 여부를 판단하여 처리하고 실행하는 함수이다. 사용할 변수를 선언하고 parseline 함수를 이용해 cmdline 를 파싱, cmd_args 배열에 명령어와 인자를 저장한다. builtin_cmd 함수를 호출해 명령어가 빌트인 명령어가 아닌 경우 다음 로직을 수행한다. sigemptyset, sigaddset, sigprocmask 함수를 호출해 부모 프로세스가 SIGCHLD 시그널을 차단하도록 해서 자식

프로세스가 생성되기 전에 SIGCHLD 핸들러가 호출되는 것을 방지한다. 그리고 fork() 시스템 호출을 사용해 새로운 자식 프로세스를 생성(실패하면 함수 종료), 자식 프로세스(process_id = 0)와 부모 프로세스(process_id != 0)으로 로직을 나누어 실행한다. 자식 프로세스의 경우, 시그널 차단을 해제하고 포그라운드/백그라운드 작업 제어를 용이하게 만들기 위해 setpgid(0, 0)를 호출해 프로세스를 독립된 프로세스 그룹으로 설정한다. 그리고 execve 시스템 호출을 사용해 명령어를 실행, 실패하면 "Command not found" 메시지를 출력하고 함수를 종료한다. 부모 프로세스의 경우, 마찬가지로 SIGCHLD 시그널을 차단 해제해 자식 프로세스가 종료되었을 때 핸들러가 호출될 수 있도록 설정하고, is_background 플래그 변수에 따라 포그라운드, 백그라운드로 로직을 나눈다. 포그라운드 작업의 경우, addjob 함수를 호출해 작업을 추가한 후 waitfg 함수를 호출해 자식 프로세스가 종료될 때까지 대기한다. 백그라운드 작업의 경우 마찬가지로 작업을 추가한 후, 해당 작업 정보를 출력한다.

**[ builtin_cmd ]**

<code>

```
int builtin_cmd(char **argv)
{
    char *command  = argv[0];  // 첫 번째 인자에서 명령어 추출
    int is_builtin = 1;        // 내장 명령어 여부 플래그

    // 내장 명령어 확인
    if (!strcmp(command, "quit"))
    {
        exit(0);               // 프로그램 종료
    }
    else if (!strcmp(command, "jobs"))
    {
        listjobs(jobs);        // 현재 작업 목록 출력
    }
    else if (!strcmp(command, "bg") || !strcmp(command, "fg"))
    {
        do_bgfg(argv);    // 백그라운드 또는 포그라운드 작업 처리
    }
    else
    {
        is_builtin = 0;        // 내장 명령어가 아님
    }

    // 내장 명령어인 경우 1 반환
    if (is_builtin)
    {
        return 1;
    }

    // 내장 명령어가 아닌 경우 0 반환
    return 0;
}
```

builtin_cmd 함수는 사용자가 입력한 명령어가 빌트인 명령어인지 확인하고 처리하는 함수이다. cmd 변수에 argv (명령어 및 인자 배열)의 첫 번째 요소를 저장한다. 그리고 내장 명령어인지 여부를 저장하는 플래그 변수를 선언한다. 이후, strcmp 함수를 호출해 cmd 변수에 저장된 명령어가 빌트인 명령어인지 확인한다. cmd가 "quit"이면 exit(0)을 호출해

프로그램을 종료한다. "jobs"이면 listjobs 함수를 호출해 현재 작업 목록을 출력하고 "bg" 또는 "fg"의 경우 do_bgfg 함수를 호출해 처리한다. 이 문자열 중 하나가 아니라면 빌트인 명령어가 아니므로 bin_flag를 0으로 설정하고, 결과를 반환한다.

**[ do_bgfg ]**

<code>

```
void do_bgfg(char **argv)
{
    struct job_t *job = NULL; // 작업의 상태, ID 등을 저장하는 구조체 포인터
    pid_t process_id;         // 프로세스 ID
    int job_id;               // 작업 ID

    char *command = argv[0];  // 명령어 (fg or bg)
    char *target  = argv[1];  // 대상 (PID or JID)

    // 인자가 제공되지 않은 경우 에러 출력
    if (target == NULL)
    {
        printf("%s command requires PID or %%jobid argument\n", command);
        return;
    }

    // PID 또는 JID 확인
    if (target[0] != '%')
    {
        if (!isdigit(target[0]))
        {
            // 입력이 %로 시작하지 않고 숫자가 아닐 경우 에러 메시지 출력
            printf("%s: argument must be a PID or %%jobid\n", command);
            return;
        }
    }

    // JID로 처리
    if (target[0] == '%')
    {
        job_id = atoi(&target[1]); // '%' 뒤의 숫자를 정수로 변환

        // 유효하지 않은 JID
        if (job_id == 0)
        {
            printf("%s: argument must be a PID or %%jobid\n", command);
            return;
        }

        // 작업 배열 jobs에서 JID에 해당하는 작업 구조체 가져오기
        job = getjobjid(jobs, job_id);

        // 해당 JID 작업이 없을 경우
        if (job == NULL)
        {
            printf("%s: No such job\n", target);
            return;
        }
    }

    // PID로 처리
    else if (isdigit(target[0]))
    {
        process_id = atoi(target); // 문자열을 정수(PID)로 변환

        // 유효하지 않은 PID
        if (process_id == 0)
        {
            printf("%s: argument must be a PID or %%jobid\n", command);
            return;
        }

        // 작업 배열 jobs에서 PID에 해당하는 작업 구조체 가져오기
        job = getjobpid(jobs, process_id);

        // 해당 PID 작업이 없을 경우
        if (job == NULL)
        {
            printf("(%s): No such process\n", target);
            return;
        }
    }
    // 명령어에 따라 동작 수행
    if (!strcmp(command, "fg"))
    {
        // 작업을 포그라운드로 전환
        job->state = FG;                                    // 상태를 FG로 설정
        kill(-job->pid, SIGCONT);                           // SIGCONT 시그널 전송 -> 작업 재개
        waitfg(job->pid);                                   // 포그라운드 작업이 완료될 때까지 대기
    }
    else if (!strcmp(command, "bg")) {
        // 작업을 백그라운드로 전환
        job->state = BG;                                    // 상태를 BG로 설정
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline); // 작업 정보 출력
        kill(-job->pid, SIGCONT);                           // SIGCONT 시그널 전송 -> 작업 재개
    }

    return;
}
```

do_bgfg 함수는 사용자로부터 입력받은 bg, fg 명령어을 처리해 특정 작업을 백그라운드 또는 포그라운드로 전환하는 작업을 수행한다. 먼저 작업 상태와 정보를 저장하는 작업 구조체 포인터, PID와 JID를 저장하는 변수들을 선언한다. 그리고 입력이 제대로 되었는지 확인하기 위해 target 변수를 검사한다. PID 또는 JID가 저장되어있는 target 변수에 제대로 입력이 들어왔는지, 입력이 들어왔다면 '%'로 시작하는지 여부로 JID/PID를 결정하고 atoi 함수를 호출해 job_id, process_id 변수에 각각 JID, PID를 저장한다. 그리고 getjobjid 함수를 호출해 작업 배열(jobs)에 해당 작업이 존재하는지 여부를 확인하고 존재하지 않는다면 경고문을 출력하고 함수를 반환한다. 사용자 입력값을 모두 검사했다면 명령어에 따라 동작을 수행한다. 명령어가 "fg"라면, 구조체 내의 변수를 수정해 작업 상태를 FG로 설정, 작업 그룹에 SIGCONT 신호를 전송해 작업을 재개하고 waitfg 함수를 호출해 방금 전환한 포그라운드 작업이 종료될 때까지 부모 프로세스를 대기시킨다. 만약 명령어가 "bg"라면, 작업 상태를 BG로 설정하고 작업 정보를 출력한다. 그리고 마찬가지로 작업 그룹에 SIGCONT 신호를 전송해 작업을 재개한다.

**[ waitfg ]**

<code>

```
void waitfg(pid_t pid)
{
    // 유효한 PID인지 확인
    if (pid == 0)
    {
        // PID가 유효하지 않으면 종료
        return;
    }

    // 현재 포그라운드 프로세스가 종료될 때까지 대기
    while (fgpid(jobs) == pid)
    {
        // 1초 동안 대기하며 반복
        sleep(1);
    }
    return;
}
```

waitfg 함수는 실행 중인 포그라운드 작업이 완료될 때까지 부모 프로세스를 대기시키는 함수이다. 포그라운드 작업의 PID를 인자로 입력받고, 해당 인자가 유효하지 않으면 함수를 종료한다. 만약 인자로 받은 PID가 유효하다면, fgpid(jobs)를 호출해 얻은 현재 포그라운드 작업의 PID와 인자 PID가 같다면 그 동안 sleep(1)을 호출해 부모 프로세스를 정지시키고 1초에 한 번씩 포그라운드 작업이 종료되었는지(작업이 종료되면 인자 PID와 fgpid(jobs)의 반환값이 달라짐) 주기적으로 확인한다.

**[ sigchld_handler ]**

<code>

```c
void sigchld_handler(int sig)
{
    pid_t child_pid;  // 자식 프로세스의 PID
    int child_status; // 자식 프로세스의 종료 상태 저장 변수

    // 종료되거나 중단된 모든 자식 프로세스를 처리
    while ((child_pid = waitpid(-1, &child_status, WNOHANG | WUNTRACED)) > 0) // waitpid 양수 반환: 상태 처리가 필요한 자식 프로세스 존재
    {
        // 자식 프로세스가 중단된 경우
        if (WIFSTOPPED(child_status))
        {
            struct job_t *job = getjobpid(jobs, child_pid); // PID를 기준으로 작업 목록(jobs)에서 해당 작업 검색

            // 작업이 존재하면
            if (job != NULL)
            {
                job->state = ST; // 상태를 ST(Stopped)로 변경
            }
            // 중단된 작업의 JID, PID, 시그널 번호(WSTOPSIG)를 출력
            printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(child_pid), child_pid, WSTOPSIG(child_status));
        }

        // 자식 프로세스가 종료된 경우
        else if (WIFSIGNALED(child_status) || WIFEXITED(child_status))
        {
            // 자식 프로세스가 정상 종료되었으면
            if (WIFSIGNALED(child_status))
            {
                // 종료된 작업 정보 출력
                printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(child_pid), child_pid, WTERMSIG(child_status));
            }
            deletejob(jobs, child_pid); // 작업 목록에서 제거
        }
    }
    return;
}
```

sigchld_handler 함수는 SIGCHLD 시그널을 처리하는 핸들러로, 자식 프로세스가 종료되거나(SIGINT) 중단되었을 때(SIGSTOP) 작업을 수행한다. 먼저 자식 프로세스의 PID와 종료 상태를 저장할 변수를 선언한다. 그리고 반복문 내에서 자식 프로세스를 처리한다. waitpid 함수는 호출해 모든 자식 프로세스를 대상으로(인자: -1) 검사를 하면서 종료되었거나 중단된 자식 프로세스가 존재하면 그 자식 프로세스의 PID를 반환한다. 만약 대기 중인 자식 프로세스가 없으면 0을 반환해 while문은 종료된다. WIFSTOPPED를 사용해 자식 프로세스가 중단되었음을 확인한 경우, getjobpid 함수를 호출해 작업 목록에서 child_pid와 일치하는 작업을 찾고 작업이 존재하면 해당 작업의 구조체 내 변수 값을 수정해 중단 상태로 변경, 중단된 작업의 정보를 출력한다. 만약 WIFSIGNALED 또는 WIFEXITED를 사용해 자식 프로세스가 종료되었음을 확인한다면, 해당 프로세스가 정상적으로 종료되었는지 확인, 종료된 작업의 정보를 출력하고 deletejob 함수를 호출해 작업 목록에서 해당 작업을 제거한다.

**[ sigint_handler ]**

<code>

```
void sigint_handler(int sig)
{
    // 작업 배열(jobs)에서 현재 포그라운드 프로세스의 PID 가져옴
    pid_t foreground_pid = fgpid(jobs);

    // 유효한 PID가 있을 경우(포그라운드 작업이 있는 경우)
    if (foreground_pid > 0)
    {
        // 프로세스(포그라운드) 그룹에 시그널 전달
        kill(-foreground_pid, sig);
    }

    return;
}
```

sigint_handler는 SIGINT 시그널을 처리하기 위한 핸들러로, Ctrl+C 입력이 들어왔을 때 호출된다. 우선 fgpid(jobs)를 호출해 작업 배열(jobs)에서 현재 포그라운드 프로세스의 PID를 가져온다. 만약 PID가 유효하다면(0보다 크다면) kill 시스템 호출을 사용해서 SIGINT 시그널을 포그라운드 프로세스 그룹에 전달하고 함수를 종료한다.

**[ sigtstp_handler ]**

<code>

```
void sigtstp_handler(int sig)
{
    // 작업 배열(jobs)에서 현재 포그라운드 프로세스의 PID 가져옴
    pid_t foreground_pid = fgpid(jobs);

    // 유효한 PID가 있는 경우(포그라운드 작업이 없는 경우)
    if (foreground_pid > 0)
    {
        // 프로세스(포그라운드) 그룹에 시그널 전달
        kill(-foreground_pid, SIGTSTP);
    }

    return;
}
```

sigtstp_handler는 SIGTSTP 시그널을 처리하기 위한 핸들러로, Ctrl+Z 입력이 들어왔을 때 호출된다. 우선 sigint_handler와 마찬가지로 fgpid(jobs)를 호출해 작업 배열(jobs)에서 현재 포그라운드 프로세스의 PID를 가져온다. 만약 PID가 유효하다면(0보다 크다면) kill 시스템 호출을 사용해서 SIGTSTP 시그널을 포그라운드 프로세스 그룹에 전달하고 함수를 종료한다.

[ test ]

```
[yojun313@programming2 6_ShellLab]$ make rtest01
./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
[yojun313@programming2 6_ShellLab]$ make test01
./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
```

```
[yojun313@programming2 6_ShellLab]$ make rtest02
./sdriver.pl -t trace02.txt -s ./tshref -a "-p"
#
# trace02.txt - Process builtin quit command.
#
[yojun313@programming2 6_ShellLab]$ make test02
./sdriver.pl -t trace02.txt -s ./tsh -a "-p"
#
# trace02.txt - Process builtin quit command.
#
```

```
[yojun313@programming2 6_ShellLab]$ make rtest03
./sdriver.pl -t trace03.txt -s ./tshref -a "-p"
#
# trace03.txt - Run a foreground job.
#
tsh> quit
[yojun313@programming2 6_ShellLab]$ make test03
./sdriver.pl -t trace03.txt -s ./tsh -a "-p"
#
# trace03.txt - Run a foreground job.
#
tsh> quit
```

```
[yojun313@programming2 6_ShellLab]$ make rtest04
./sdriver.pl -t trace04.txt -s ./tshref -a "-p"
#
# trace04.txt - Run a background job.
#
tsh> ./myspin 1 &
[1] (20385) ./myspin 1 &
[yojun313@programming2 6_ShellLab]$ make test04
./sdriver.pl -t trace04.txt -s ./tsh -a "-p"
#
# trace04.txt - Run a background job.
#
tsh> ./myspin 1 &
[1] (20407) ./myspin 1 &
```

```
[yojun313@programming2 6_ShellLab]$ make rtest05
./sdriver.pl -t trace05.txt -s ./tshref -a "-p"
#
# trace05.txt - Process jobs builtin command.
#
tsh> ./myspin 2 &
[1] (20576) ./myspin 2 &
tsh> ./myspin 3 &
[2] (20578) ./myspin 3 &
tsh> jobs
[1] (20576) Running ./myspin 2 &
[2] (20578) Running ./myspin 3 &
[yojun313@programming2 6_ShellLab]$ make test05
./sdriver.pl -t trace05.txt -s ./tsh -a "-p"
#
# trace05.txt - Process jobs builtin command.
#
tsh> ./myspin 2 &
[1] (20636) ./myspin 2 &
tsh> ./myspin 3 &
[2] (20639) ./myspin 3 &
tsh> jobs
[1] (20636) Running ./myspin 2 &
[2] (20639) Running ./myspin 3 &
```

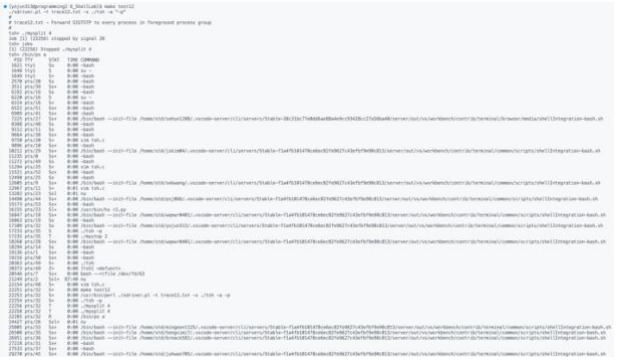
```
[yojun313@programming2 6_ShellLab]$ make rtest06
./sdriver.pl -t trace06.txt -s ./tshref -a "-p"
#
# trace06.txt - Forward SIGINT to foreground job.
#
tsh> ./myspin 4
Job [1] (20732) terminated by signal 2
[yojun313@programming2 6_ShellLab]$ make test06
./sdriver.pl -t trace06.txt -s ./tsh -a "-p"
#
# trace06.txt - Forward SIGINT to foreground job.
#
tsh> ./myspin 4
Job [1] (20768) terminated by signal 2
```

```
[yojun313@programming2 6_ShellLab]$ make rtest07
./sdriver.pl -t trace07.txt -s ./tshref -a "-p"
#
# trace07.txt - Forward SIGINT only to foreground job.
#
tsh> ./myspin 4 &
[1] (20836) ./myspin 4 &
tsh> ./myspin 5
Job [2] (20838) terminated by signal 2
tsh> jobs
[1] (20836) Running ./myspin 4 &
[yojun313@programming2 6_ShellLab]$ make test07
./sdriver.pl -t trace07.txt -s ./tsh -a "-p"
#
# trace07.txt - Forward SIGINT only to foreground job.
#
tsh> ./myspin 4 &
[1] (20889) ./myspin 4 &
tsh> ./myspin 5
Job [2] (20891) terminated by signal 2
tsh> jobs
[1] (20889) Running ./myspin 4 &
```

```
[yojun313@programming2 6_ShellLab]$ make rtest08
./sdriver.pl -t trace08.txt -s ./tshref -a "-p"
#
# trace08.txt - Forward SIGTSTP only to foreground job.
#
tsh> ./myspin 4 &
[1] (20997) ./myspin 4 &
tsh> ./myspin 5
Job [2] (20999) stopped by signal 20
tsh> jobs
[1] (20997) Running ./myspin 4 &
[2] (20999) Stopped ./myspin 5
[yojun313@programming2 6_ShellLab]$ make test08
./sdriver.pl -t trace08.txt -s ./tsh -a "-p"
#
# trace08.txt - Forward SIGTSTP only to foreground job.
#
tsh> ./myspin 4 &
[1] (21062) ./myspin 4 &
tsh> ./myspin 5
Job [2] (21064) stopped by signal 20
tsh> jobs
[1] (21062) Running ./myspin 4 &
[2] (21064) Stopped ./myspin 5
```

```
[yojun313@programming2 6_ShellLab]$ make rtest09
./sdriver.pl -t trace09.txt -s ./tshref -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (21205) ./myspin 4 &
tsh> ./myspin 5
Job [2] (21209) stopped by signal 20
tsh> jobs
[1] (21205) Running ./myspin 4 &
[2] (21209) Stopped ./myspin 5
tsh> bg %2
[2] (21209) ./myspin 5
tsh> jobs
[1] (21205) Running ./myspin 4 &
[2] (21209) Running ./myspin 5
[yojun313@programming2 6_ShellLab]$ make test09
./sdriver.pl -t trace09.txt -s ./tsh -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (21281) ./myspin 4 &
tsh> ./myspin 5
Job [2] (21283) stopped by signal 20
tsh> jobs
[1] (21281) Running ./myspin 4 &
[2] (21283) Stopped ./myspin 5
tsh> bg %2
[2] (21283) ./myspin 5
tsh> jobs
[1] (21281) Running ./myspin 4 &
[2] (21283) Running ./myspin 5
```

```
[yojun313@programming2 6_ShellLab]$ make rtest10
./sdriver.pl -t trace10.txt -s ./tshref -a "-p"
#
# trace10.txt - Process fg builtin command.
#
tsh> ./myspin 4 &
[1] (21405) ./myspin 4 &
tsh> fg %1
Job [1] (21405) stopped by signal 20
tsh> jobs
[1] (21405) Stopped ./myspin 4 &
tsh> fg %1
tsh> jobs
[yojun313@programming2 6_ShellLab]$ make test10
./sdriver.pl -t trace10.txt -s ./tsh -a "-p"
#
# trace10.txt - Process fg builtin command.
#
tsh> ./myspin 4 &
[1] (21458) ./myspin 4 &
tsh> fg %1
Job [1] (21458) stopped by signal 20
tsh> jobs
[1] (21458) Stopped ./myspin 4 &
tsh> fg %1
tsh> jobs
```



```
[yojun313@programming2 6_ShellLab]$ make rtest14
./sdriver.pl -t trace14.txt -s ./tshref -a "-p"
#
# trace14.txt - Simple error handling
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 4 &
[1] (22988) ./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] (22988) stopped by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (22988) ./myspin 4 &
tsh> jobs
[1] (22988) Running ./myspin 4 &
```

```
[yojun313@programming2 6_ShellLab]$ 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] (23115) ./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] (23115) stopped by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (23115) ./myspin 4 &
tsh> jobs
[1] (23115) Running ./myspin 4 &
```

```
[yojun313@programming2 6_ShellLab]$ make rtest15
./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] (23297) terminated by signal 2
tsh> ./myspin 3 &
[1] (23310) ./myspin 3 &
tsh> ./myspin 4 &
[2] (23312) ./myspin 4 &
tsh> jobs
[1] (23310) Running ./myspin 3 &
[2] (23312) Running ./myspin 4 &
tsh> fg %1
Job [1] (23310) stopped by signal 20
tsh> jobs
[1] (23310) Stopped ./myspin 3 &
[2] (23312) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (23310) ./myspin 3 &
tsh> jobs
[1] (23310) Running ./myspin 3 &
[2] (23312) Running ./myspin 4 &
tsh> fg %1
tsh> quit
```

```
[yojun313@programming2 6_ShellLab]$ 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] (23364) terminated by signal 2
tsh> ./myspin 3 &
[1] (23377) ./myspin 3 &
tsh> ./myspin 4 &
[2] (23380) ./myspin 4 &
tsh> jobs
[1] (23377) Running ./myspin 3 &
[2] (23380) Running ./myspin 4 &
tsh> fg %1
Job [1] (23377) stopped by signal 20
tsh> jobs
[1] (23377) Stopped ./myspin 3 &
[2] (23380) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (23377) ./myspin 3 &
tsh> jobs
[1] (23377) Running ./myspin 3 &
[2] (23380) Running ./myspin 4 &
tsh> fg %1
tsh> quit
```