시스템 프로그래밍

Shell Lab2

2016.11.22

황슬아

seula.hwang@cnu.ac.kr

개요

1. 실습명

- ✓ Shell Lab

2. 목표

- ✓ 작업 관리를 지원하는 간단한 Unix Shell 프로그램 구현과 이를 통한 프로세스의 제어와 시그널링(Signalling)의 개념 이해

3. 과제 진행

- ✓ 수업시간에 배운 유닉스 지식을 활용하여 구현한다.

4. 구현사항

- ✓ 기본적인 유닉스 쉘의 구현
- ✓ 작업 관리(foreground / background)기능 및 쉘 명령어 구현

Shell Lab

1. 프로세스의 관리와 시그널의 제어에 대해 이해하고, 작업의 제어를 지원하는 Unix Shell program을 작성하는 것을 목표로 한다.

2. Shell Lab은 trace00 부터 trace21 까지의 Trace를 모두 수행할 수 있도록 구현되어야 한다.
   1) 테스트 프로그램 'sdriver' 를 통해 테스트할 수 있다.

3. 총 2주에 걸쳐 진행되며, 각 주 별 수행 사항은 아래와 같다.
   1) Shell Lab 1주차 : trace00 ~ 07
   2) Shell Lab 2주차 : trace08 ~ 21

Shell Lab - 준비

1. Shell Lab 파일 복사 및 압축해제
   1) cp /home/ubuntu/shlab/shlab-handout.tar.gz ~
   2) tar xvzf ~/shlab-handout.tar.gz

2. Shell Lab 파일 구성

파 일	설 명
Makefile	쉘 프로그램의 컴파일 및 테스트
README	도움말
tsh.c	Shell 프로그램의 소스코드
tshref	Shell binary의 레퍼런스
sdriver	Shell에 각 trace 들을 실행하는 프로그램
trace{00-21}.txt	Shell 드라이버를 제어하는 22개의 trace
myspin.c, mysplit.c, mystop.c myint.c	trace 파일들에서 불려지는 C로 작성된 프로그램 (README 참조)

Shell Lab - 준비

1. tsh.c 파일 내부 구성

함 수	설 명
eval	명령을 파싱 하거나 해석하는 메인 루틴
builtin_cmd	quit, fg, bg와 jobs 같은 built-in 명령어를 해석
waitfg	Foreground 작업이 완료될 때 까지 대기
sigchld_handler	SIGCHILD 시그널 핸들러
sigint_handler	SIGINT(ctrl-c) 시그널 핸들러
sigtstp_handler	SIGTSTP(ctrl-z) 시그널 핸들러

Shell Lab – Trace (2주차)

1. sdriver를 이용하여 Trace{08-21}를 테스트 할 수 있다.
   1) 각 trace의 자세한 내용은 해당 파일을 열어서 확인할 수 있다.

Trace	설 명
trace08	SIGINT 발생 시, foreground 작업 종료
trace09	SIGTSTP 발생 시, foreground 작업 종료
trace10	Background 작업 정상 종료 처리
trace11	자식 프로세스가 스스로에게 SIGINT 전송
trace12	자식 프로세스가 스스로에게 SIGTSTP 전송
trace13	foreground 작업에만 SIGINT 전송
trace14	foreground 작업에만 SIGTSTP 전송

Shell Lab – Trace (2주차)

1. sdriver를 이용하여 Trace{08-21}를 테스트 할 수 있다.
   1) 각 trace의 자세한 내용은 해당 파일을 열어서 확인할 수 있다.

Trace	설 명
trace15	Built-in 명령어 'bg' 구현 (한 개의 작업에 대해)
trace16	Built-in 명령어 'bg' 구현 (두 개의 작업에 대해)
trace17	Built-in 명령어 'fg' 구현 (한 개의 작업에 대해)
trace18	Built-in 명령어 'fg' 구현 (두 개의 작업에 대해)
trace19	Foreground 그룹에서 SIGINT 처리
trace20	Foreground 그룹에서 SIGTSTP 처리
trace21	정지한 모든 프로세스를 재 시작

Shell Lab – Trace 검사

1. sdriver 사용 방법

1) ./sdriver -t <trace number> -s ./<shell name>

- ex) tsh 쉘에서 trace00 검사
- ./sdriver –t 00 –s ./tsh

2) ./sdriver.pl -h를 통해 사용 방법과 사용 가능한 옵션을 확인 할 수 있다.

옵 션	설 명
-h	도움말 출력
-V	자세한 동작 과정에 대한 출력
-t <trace number>	Trace 번호
-s <shell>	테스트 할 쉘 프로그램

Shell Lab – Trace 검사

2. 레퍼런스 코드 확인

1) 완성되어있는 레퍼런스 쉘을 통해 정상 동작하는 쉘의 모습을 테스트하고 살펴볼 수 있다.

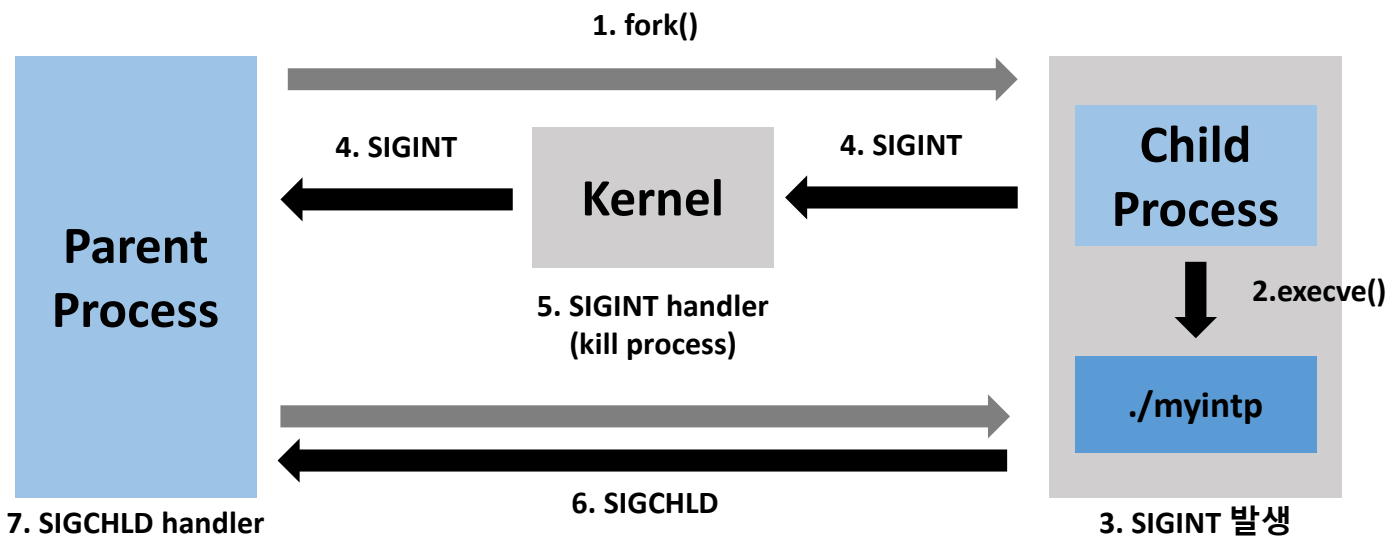
- ./sderver –t XX –s ./tshref

Signal

1. 어떤 Event가 발생했음을 알리기 위해 Process에게 전달되는 소프트웨어 인터럽트.
2. Signal을 발생 시키는 Event의 종류는 아래의 4가지 종류가 있다.
   1) Hardware Exception (나누기 0 등)
   2) Software condition (alarm 시간, expire 등)
   3) 단말기에서 발생하는 사용자 입력 (^c, ^z 등)
   4) kill 등과 같은 시스템 콜
3. Event에 의해서 Signal이 생성 되면 곧 Process에게 전달 된다.
4. Process에게 Signal이 전달되면
   1) 기본 설정 실행(ignore, terminate, terminate+core)
   2) Signal Handler에 의한 Catch 후 로직 수행
   3) 무시
5. Signal이 생성 되었으나 아직 전달 되지 않은 Signal은 Pending이라 함.
6. Process는 signal mask를 사용해 특정 Signal을 Block/Unblock 시킬 수 있음.
7. Process가 특정 Signal을 Block 시켜도 이 Signal은 생성되지만 전달 되지 않을 뿐 Pending 됨.
8. Block된 Signal은 Process가 그 Signal을 Unblock 할 때까지 혹은 해당 Signal에 대한 처리를 ignore로 변경 할 때까지 Pending됨.
9. 어떤 Process에 여러 개의 Signal이 생성되어 전달 되는 경우 순서는 보장할 수 없다.

Signal의 흐름

1. SIGINT 처리 과정 (trace08)
   1) fork( )와 execve( ) 통해 프로그램 실행
   2) SIGINT 발생(ctrl+c)
   3) kernel을 통해 자식 프로세스에서 부모 프로세스로 SIGINT 전달
   4) SIGINT 핸들러를 통해 SIGINT 처리(자식 프로세스 Kill)
   5) 자식 프로세스가 종료되면 SIGCHLD 시그널 발생
   6) SIGCHLD 핸들러를 통해 자식 프로세스 최종 종료 처리



Signal의 번호

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL
5) SIGTRAP	6) SIGABRT	7) SIGBUS	8) SIGFPE
9) SIGKILL	10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM	16) SIGSTKFLT
17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU
25) SIGXFSZ	26) SIGVTALRM	27) SIGPROF	28) SIGWINCH
29) SIGIO	30) SIGPWR	31) SIGSYS	34) SIGRTMIN
35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3	38) SIGRTMIN+4
39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12
47) SIGRTMIN+13	48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14
51) SIGRTMAX-13	52) SIGRTMAX-12	53) SIGRTMAX-11	54) SIGRTMAX-10
55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7	58) SIGRTMAX-6
59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX		

Signal blocking과 unblocking

1. Application은 sigprocmask 함수를 이용하여 선택된 signal들을 blocking 하거나 unblocking 할 수 있다.

2. Signal Block
   1) 시그널을 block 하지 않으면 자식 프로세스가 부모 프로세스가 addjob을 수행하기도 전에, 시그널에 의해 종료되어 버릴 수 있다.
   2) 그렇게 되면 존재하지도 않는 job을 list에 추가하는 사태가 발생한다.
   3) 또한 시그널 핸들러의 처리과정에서 존재하지도 않는 job을 종료하고, list에서 제거하는 작업을 해야 한다.
   4) 따라서 이러한 부분을 signal block을 통해 시그널이 발생해도 처리하지 않도록 막아야 한다.

```
void handler(int sig){
        pid_t pid;
        while ((pid == waitpid(-1, NULL, 0)) > 0) // Reap a zombie child
        deletejob(pid); // Delete the child from the job list
        if(errno != ECHILD)
                unix_error("waitpid error");
}
```

Signal blocking과 unblocking

```c
int main(int argc, char **argv){
        int pid;
        sigset_t mask;

        Signal(SIGCHLD, handler); // job list를 초기화 한다
        initjobs();

        while(1){
                sigemptyset(&mask);
                sigaddset(&mask, SIGCHLD);

                /*
                * Race Condition Problem 제거
                * SIGCHLD를 블럭해서 handler에 의해서 addjob 이전에
                * deletejob 함수가 실행되지 않도록 한다.
                */
                sigprocmask(SIG_BLOCK, &mask, NULL); // 블럭 SIGCHLD

                // 자식 프로세스
                if((pid == fork()) == 0){
                        /*
                        * 자식 프로세스는 부모 프로세스의 blocked set을 상속 받기 때문에
                        * 반드시 unblock SIGCHLD를 해주어야 한다.
                        */
                        sigprocmask(SIG_UNBLOCK, &mask, NULL); // Unblock SIGCHLD
                        execve("/bin/date", argv, NULL);
                }

                // 부모 프로세스
                addjob(pid); // 작업리스트에 자식을 추가

                /*
                * addjob 이후에는 race 문제가 없으므로
                * unblock을 해준다
                */
                sigprocmask(SIG_UNBLOCK, &mask, NULL); // Unblock SIGCHLD
        }
        exit(0);
}
```

sigprocmask

1. Signal handler를 이용하여 만들 때, signal을 catch하거나 block하기 위해 sigprocmask를 사용한다.

2. int sigprocmask(int how, const sigset_t *set, sigset_t *oldest)
   1) 이 함수는 signal.h에 선언되어 있음
   2) 반환 값 : 성공 시 0, 실패 시 -1
   3) int how에 들어가는 옵션
      - SIG_BLOCK : set contains additional signals to block
      - SIG_UNBLOCK : set contains signals to unblock
      - SIG_SETMASK : set contains the new signal mask
      - NULL인 경우 무시

3. Signal mask가 set 되었다면 catch 하기를 원하는 handler를 등록한다.

sigemptyset & sigaddset

1. int sigemptyset(sigset_t *set)
   1) 이 함수는 인자로 주어진 시그널 set에 포함되어 있는 모든 시그널을 비운다.
   2) 반환 값 : 성공 시 0, 실패 시 -1

2. int sigaddset(sigset_t *set, int signum)
   1) 이 함수는 set으로부터 시그널 번호가 signum인 시그널을 추가한다.
   2) 반환 값 : 성공 시 0, 실패 시 -1

Race Condition

1. Race condition이란
   1) 두 개 이상의 프로세스가 경쟁적으로 동일한 자원에 접근하려고 하는 상태를 의미한다.
   2) foreground 프로세스가 실행되는 중에 fork가 발생하면, 한 순간 하나의 foreground 프로세스만 진행 되어야 하는데, 두 개 이상의 foreground 프로세스가 동작할 가능성이 발생한다.
   3) 이를 방지하기 위해 아래 코드를 작성해 준다.

```
/*
 * 부모 프로세스
 *
 * Parent adds the job, and then unblocks signals so that
 * the signals handlers can run again
 *
 */
addjob(jobs, pid, (bg == 1 ? BG : FG), cmdline);
sigprocmask(SIG_UNBLOCK, &mask, NULL);

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

Race Condition

Race Condition

```
/*
 * waitfg - Block until process pid is no longer the foreground process
 */

void waitfg(pid_t pid, int output_fd){
        struct job_t *j = getjobpid(jobs, pid);
        char buf[MAXLINE];

        // The FG job has already completed and been reaped by the hand
        if(!j)
                return;

        /*
        * Wait for process pid to longer be the foreground process.
        * Note : using pause() instead of sleep() would introduce a race
        * that could cause us to miss the signal
        */
        while(j->pid == pid && j->state == FG)
                sleep(1);

        if(verbose){
                memset(buf, '\0', MAXLINE);
                sprintf(buf, "waitfg: Process (%d) no longer the fg process:q\n", pid);
                if(write(output_fd, buf, strlen(buf)) < 0){
                        fprintf(stderr, "Error writing to file\n");
                        exit(1);
                }
        }
        retrun;
}
```

실행되고 있는 foreground 작업이 종료될 때가지 wait

-V 옵션을 위한 부분

Process Group

1. 사용자도 그룹을 가지듯이 프로세스도 자신이 속한 그룹을 가진다.
2. 프로세스 그룹(Process group)이란 동일한 터미널로부터 signal들을 받을 수 있는 하나 이상의 프로세스들의 집합이다.
3. getpgrp() 함수를 통해 현재 프로세스 그룹ID를 받아올 수 있으며 setpgrp()함수로 그룹ID를 설정할 수 있다.

4. int setpgid(pid_t pid, pid_t pgid)
   1) signal.h에 정의 되어있음
   2) 반환 값 : 성공 시 0, 실패 시 -1
   3) setpgid( )를 이용하여 pid로 설정된 프로세스의 프로세스 그룹의 ID를 pgid로 설정한다.
5. pid_t getpgid(pid_t pid)
   1) 반환 값 : 성공 시 0, 실패 시 -1
   2) getpgid( )는 해당 프로세스의 pid를 인자로 받아서 pid에 해당하는 pgid를 반환한다.

Shell Lab - trace08

1. 구현 기능 : SIGINT 발생 시, Foreground 작업 종료

   1) SIGINT(키보드의 'ctrl+c' 입력)가 입력 되면 foreground 작업을 kill 한다.
      - 이전 슬라이드의 시그널 흐름을 이해하는 것이 중요.

   2) trace08.txt를 참조하여 동작을 이해한다.
      - ./myintp 라는 프로그램을 foreground 형태로 실행시킨다.
        - 해당 프로그램은 SIGINT를 발생시키는 프로그램이다.
      - SIGINT 발생 결과 프로세스가 종료되고 아래와 같은 형태로 정보를 출력한다.
        - 'Job [X] (XXXXX) terminated by signal 2'
      - 'quit'명령을 통해 쉘을 종료한다.

```
#
# trace08.txt - Send fatal SIGINT to foreground job.
#
/bin/echo -e tsh\076 ./myintp
NEXT
./myintp
NEXT

/bin/echo -e tsh\076 quit
NEXT
quit
```

Shell Lab - trace08

1. 구현 방법

1) SIGINT가 발생하면, main() 함수에 등록되어있는 sigint_handler() 함수가 자동으로 호출된다.

```
/* These are the ones you will need to implement */
Signal(SIGINT,  sigint_handler);   /* ctrl-c */
```

2) sigint_handler(int sig) 함수 내부에 foreground job을 종료 시키도록 코드를 구현

```
196 void sigchld_handler(int sig)
197 {
198     return;
199 }
```

Shell Lab - trace08

1. 결과 확인

1) trace08를 sdriver를 수행하여 결과를 확인한다.

- 명령어 : ./sdriver -t 08 -s ./tsh -V
- 이때 tsh 수행 결과가 tshref와 다르다면,

```
Running trace08.txt...
Oops: test and reference outputs for trace08.txt differed.
```

- 같다면,

```
Running trace08.txt...
Success: The test and reference outputs for trace08.txt matched!
```

2. HINT

1) 다음 System Call을 이용하여 다른 프로세스로 시그널을 전달한다.

int kill(pid_t pid, int sig);
Return 성공시 0, 에러시 -1

- pid가 0보다 크면, kill 함수는 시그널(sig)를 해당 pid를 가지는 프로세스로 전달
- pid가 0보다 작으면, pid에 해당하는 그룹에 속하는 모든 프로세스에 시그널을 전달
- 참조 : http://i1004me2.blog.me/140190173943

Shell Lab - trace09

1. 구현 기능 : SIGTSTP 발생 시 Foreground 작업 정지
   1) SIGTSTP(키보드의 'ctrl+z' 입력)이 되면 foreground 작업을 stop 한다.
      - 이전 슬라이드의 시그널 흐름을 이해하는 것이 중요.

   2) trace09.txt를 참조하여 동작을 이해한다.
      - ./mytstpp 라는 프로그램을 foreground 형태로 실행시킨다.
        - 해당 프로그램은 SIGTSTP를 발생시키는 프로그램이다.
      - SIGTSTP 발생 결과 프로세스가 정지되고 아래와 같은 형태로 정보를 출력한다.
        - 'Job [X] (XXXXX) stopped by signal 20'
      - 'jobs' 명령을 통해 아래와 같이 정보를 출력한다.
        - '(X) (XXXXX) Stopped ./mytstpp'

```
#
# trace09.txt - Send SIGTSTP to foreground job.
#
/bin/echo -e tsh\076 ./mytstpp
NEXT
./mytstpp
NEXT

/bin/echo -e tsh\076 jobs
NEXT
jobs
NEXT

quit
```

Shell Lab - trace09

1. 구현 방법

1) SIGTSTP 시그널을 받아서 joblist 에서 해당 job의 상태를 stop으로 변경 해주어야 한다.
2) trace08의 구현 방법과 마찬가지로,
3) sigint_handler() 함수의 내용을 구현해야 한다.
4) sigint_handler() 함수 내부에서 joblist를 체크해 해당 job의 상태를 stop으로 변경하고 정지 시켜야 한다.

2. 결과 확인

1) trace09를 sdriver를 수행하여 결과를 확인한다.
   - 명령어 : ./sdriver -t 09 -s ./tsh –V

3. HINT

1) trace09도 trace08번과 동일하게 kill( )함수를 사용한다.

Shell Lab - trace10

1. 구현 기능 : Background 작업 정상 종료 처리

   1) Background 작업이 정상 종료되면 SIGCHLD가 발생하는데, 이 시그널을 받고 Background 작업을 종료 처리해준다.
      - 이때, 핸들러에서는 종료된 자식 프로세스가 사용했던 자원을 반환 시켜주어야 한다.

   2) trace10.txt를 참조하여 동작을 이해한다.
      - ./myspin1 이라는 프로그램을 Background 형태로 5초 동안 실행시킨다.
      - Background 작업이 종료되기 전, /bin/kill 을 통해 SIGTERM을 발생시킨다.
        - SIGTERM은 정상 종료 시그널
      - 작업이 종료되면, 아래와 같이 출력 후 joblist에서 제거된다.
        - 'Job [X] (XXXXX) terminated by signal 15'

```
#
# trace10.txt - Send fatal SIGTERM (15) to a background job.
#
/bin/echo -e tsh\076 ./myspin1 5 \046
NEXT
./myspin1 5 &
NEXT

WAIT

/bin/echo -e tsh\076 /bin/kill myspin1
NEXT
/bin/kill myspin1
NEXT

/bin/echo -e tsh\076 quit
NEXT
quit
```

Shell Lab - trace10

1. 구현 방법

1) sigchld_handler() 함수의 내부를 구현

2) 자식 프로세스가 정상 종료되고, joblist 에 그것을 반영해야 한다.

2. 결과 확인

1) trace10을 sdriver를 수행하여 결과를 확인한다.

- 명령어 : ./sdriver –t 10 -s ./tsh –V

3. HINT

1) waitpid( )함수를 통해, 부모 프로세스는 자식 프로세스가 종료되기까지 기다리고, 자식이 종료되면 처리해준다.

2) 자식 프로세스의 종료 상태

- WIFEXITED(status) : 정상적으로 자식 프로세스가 종료되었을 경우 0이 아닌 값 리턴
- WIFSIGNALED(status) : 자식 프로세스가 어떤 signal에 의해 종료된 경우 TRUE 리턴
- WIFSTOPPED(status) : 자식 프로세스가 정지된 상태라면 TRUE 리턴
- 위 함수를 이용하여 joblist의 job 정보를 업데이트 해야 한다. 또한 무슨 작업에 의해 종료되었느냐에 따라 적절한 메시지를 출력해야 한다.

Shell Lab - trace11~12

trace11

1. 구현 기능 : 자식 프로세스 스스로에게 SIGINT 전송되고 처리
   1) 자신의 pid(자식 프로세스의 pid)로 SIGINT를 전달하였을 때, 정상적으로 SIGINT에 대한 처리가 되도록 구현
   2) trace08를 제대로 구현했다면 자동으로 통과된다.
      - 자동으로 통과되는 이유를 생각해보고 보고서에 작성.

trace12

1. 구현 기능 : 자식 프로세스 스스로에게 SIGTSTP 전송되고 처리
   1) 자신의 pid(자식 프로세스의 pid)로 SIGTSTP를 전달하였을 때, 정상적으로 SIGTSTP에 대한 처리가 되도록 구현
   2) trace08를 제대로 구현했다면 자동으로 통과된다.
      - 자동으로 통과되는 이유를 생각해보고 보고서에 작성.

Shell Lab – trace13

1. 구현 기능: Foreground 작업에만 SIGINT 전송되고 처리
2. 구현 방법
   1) sigint_handler 함수 내부에 해당 기능을 하는 코드 추가하여 오직 foreground 작업에 대해서만 SIGINT가 처리되도록 핸들러를 수정
3. 시험 방법
   1) ./sdriver –s ./tsh –t 13 -V
   2) ./myspin1을 background 형태로 100초 동안 실행
   3) ./myintp는 foreground 형태로 실행
      - 해당 프로그램은 SIGINT를 발생시키는 프로그램이다.

```
eslab_tsh> ./myspin1 100 &
(1) (13020) ./myspin1 100 &
eslab_tsh> ./myintp
Job [2] (13137) terminated by signal 2
eslab_tsh> jobs
(1) (13020) Running ./myspin1 100 &
```

   4) myintp는 SIGINT에 의해 종료되었다. 하지만 myspin1는 background 작업이므로 종료되지 않고, joblist에 남은 것을 확인할 수 있다.
4. 힌트
   1) pid_t fgpid(struct job_t *jobs) 함수를 사용

Shell Lab – trace14

1. 구현 기능: Foreground 작업에만 SIGTSTP 전송되고 처리
2. 구현 방법
   1) sigtstp_handler 함수 내부에 해당 기능을 하는 코드 추가하여 오직 foreground 작업에 대해서만 SIGTSTP가 처리되도록 핸들러를 수정
3. 시험 방법
   1) ./sdriver –s ./tsh –t 14 -V
   2) ./myspin1을 background 형태로 100초 동안 실행
   3) ./mytstpp는 foreground 형태로 실행
      - 해당 프로그램은 SIGTSTP를 발생시키는 프로그램이다.

```
eslab_tsh> ./myspin1 100 &
(1) (29194) ./myspin1 100 &
eslab_tsh> ./mytstpp
Job [2] (29324) stopped by signal 20
eslab_tsh> jobs
(1) (29194) Running ./myspin1 100 &
(2) (29324) Stopped ./mytstpp
```

   4) myintp는 SIGTSTP에 의해 정지되었다. 하지만 myspin1는 background 작업이므로 정지되지 않고, 수행중인 것을 확인할 수 있다.
4. 힌트
   1) trace13과 동일

Shell Lab – trace15

1. 구현 기능: 한 개의 작업에 대해 Built-in 명령어 'bg' 구현
2. 구현 방법
   1) builtin_cmd 함수 내부에 구현
   2) 정지 상태인 프로세스를 검색하고, 해당 프로세스를 Running 상태로 변경한다.
   3) 단, 이때 재 실행되는 프로세스는 background 형태로 실행되어야 한다.
3. 시험 방법
   1) ./sdriver –s ./tsh –t 15 -V
   2) ./mytstpp를 foreground 형태로 실행하여 자기 자신(mytstpp)을 멈춤
   3) bg %1로 job ID 1번(mytstpp)을 background 형태로 재 실행

4. 힌트
   1) builtin_cmd 함수 내부에서 jid와 pid를 비교
      - bg, fg 명령 시, 매개변수로 jid를 전달한다.
      - 이 jid에 맞는 pid를 재 실행 시켜야 한다.
   2) SIGCONT를 이용하여 중지된 작업을 다시 시작
      - 시그널 SIGCONT를 이용하여 작업을 다시 시작 시킨다.

Shell Lab – trace16

1. 구현 기능: 두 개의 작업에 대해 Built-in 명령어 'bg' 구현
2. 구현 방법
   1) trace15와 동일
3. 시험 방법
   1) ./sdriver –s ./tsh –t 16 -V
   2) ./myspin을 background 형태로 10초간 실행
   3) ./mytstpp를 foreground 형태로 실행하여 자기 자신(mytstpp)을 멈춤
   4) bg %2로 job ID 2번(mytstpp)을 background 형태로 재 실행

```
tsh> ./myspin1 10 &
(1) (11699) ./myspin1 10 &
tsh> ./mytstpp
Job [2] (11701) stopped by signal 20
tsh> jobs
(1) (11699) Running ./myspin1 10 &
(2) (11701) Stopped ./mytstpp
tsh> bg %2
[2] (11701) ./mytstpp
tsh> jobs
(1) (11699) Running ./myspin1 10 &
(2) (11701) Running ./mytstpp
```

4. 힌트
   1) trace15와 동일

Shell Lab - trace17

1. 구현 기능 : Built-in 명령어 'fg' 구현 (한 개의 작업에 대해)
   1) 정지 상태 또는 background 형태인 프로세스를 검색하고, 해당 프로세스를 Running 상태로 변경한다.
   2) 단, 이때 재 실행되는 프로세스는 foreground 형태로 실행되어야 한다.
2. 구현 방법 : bulitin_cmd 함수에 구현
   - builtin_cmd 함수 내부에서 jid와 pid를 비교
   - SIGCONT를 이용하여 중지된 작업을 다시 시작
3. 시험 방법 :

```
tsh> ./mytstps
Job [1] (9390) stopped by signal 20
tsh> jobs
(1) (9390) Stopped ./mytstps
tsh> fg %1
```

   - 해당 그림은,
     - mytstps를 실행시켜 프로세스가 정지되도록 한다.
     - jobs 명령어를 통해 정지 됨을 확인한다.
     - fg 명령어를 통해 정지된 프로세스를 foreground 형태로 재 실행시킨다.

1) ./sdriver –s ./tsh –t 19 -V

Shell Lab - trace18

1. 구현 기능 : Built-in 명령어 'fg' 구현 (두 개의 작업에 대해)
   1) trace17번과 동일하게 동작시킨다. 단, 다수(두 개 이상)의 프로세스를 대상으로 수행한다.
      - 이때, foreground는 하나밖에 동작하지 않으므로 남은 하나는 해당 프로그램이 종료되기 전까지 이전 동작을 그대로 수행한다.
2. 구현 방법 : Built-in 명령어 'fg' 구현 (두 개의 작업에 대해)
   1) trace17번과 동일하게 동작시킨다. 단, 다수(두 개 이상)의 프로세스를 대상으로 수행한다.
      - 이때, foreground는 하나밖에 동작하지 않으므로 남은 하나는 해당 프로그램이 종료되기 전까지 이전 동작을 그대로 수행한다.
3. 시험 방법

- 해당 그림은,
  - myspin1을 background 형태로 실행시킨다.
  - mytstps를 실행시켜 프로세스가 정지되도록 한다.
  - jobs 명령어를 통해 정지 됨을 확인한다.
  - fg 명령어를 통해 정지된 두 번째 프로세스를 foreground 형태로 재 실행시킨다.

1) ./sdriver –s ./tsh –t 19 -V

Shell Lab – trace17~18

1. HINT

1) bulitin_cmd 함수에 구현

- builtin_cmd 함수 내부에서 jid와 pid를 비교
  - bg, fg 명령 시, 매개변수로 jid를 전달한다.
  - 이 jid에 맞는 pid를 재 실행 시켜야 한다.
- SIGCONT를 이용하여 중지된 작업을 다시 시작
  - 시그널 SIGCONT를 이용하여 작업을 다시 시작 시킨다.

Shell Lab - trace19

1. 구현 기능 : 모든 프로세스를 foreground 프로세스 그룹으로 설정하여 그룹에 SIGINT를 전달

2. 구현 방법 : trace08과 동일. 단, 그룹 pid로 변경

3. 시험 방법 :

```
eslab_tsh> ./mysplit 10
^CJob [1] (13402) terminated by signal 2
eslab_tsh>
```

1) ./sdriver –s ./tsh –t 19 -V

4. HINT

1) trace08~14에 사용된 시스템 콜을 사용
2) setpgid( )함수를 사용하여 프로세스에 프로세스 그룹 아이디를 셋팅
3) kill 시스템 콜을 이용하여 pid가 속한 그룹 전체에 시그널을 보내는 것을 이용
   - kill( ) 시스템 콜의 매개변수에 대하여 자세히 알아보길 추천

Shell Lab – trace20

1. 구현 기능 : 모든 프로세스를 foreground 프로세스 그룹으로 설정하여 그룹에 SIGSTP를 전달

2. 구현 방법 : trace09과 동일. 단, 그룹 pid로 변경

3. 시험 방법 :

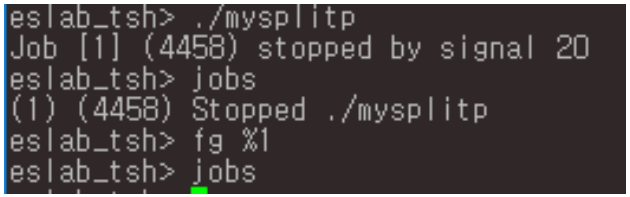
1) ./sdriver –s ./tsh –t 20 -V

Shell Lab – trace21

1. 구현 기능 : 정지된 프로세스 그룹을 모두 재 시작

2. 구현 방법 : kill 함수에서 SIGCONT 시그널을 보내도록 구현하며, 이때 모든 프로세스 그룹에 시그널을 보내도록 구현.

3. 시험 방법 :



1) ./sdriver –s ./tsh –t 21 -V

부록 – 예외 처리

1. 시스템 콜 호출 시, 예외가 발생하면 unix_error 함수(tsh.c에 구현되어있음)를 이용하여 오류 번호에 따른 오류 메시지를 출력할 수 있도록 한다.
   1) ex) fork( )로 프로세스 생성에 실패한 경우의 예외처리

```
/* Create a child process */
if ((pid = fork()) < 0)
    unix_error("fork error");
```

2. 쉘 프로그램에서 예외가 발생할 때, app_error 함수(tsh.c에 구현되어 있음)를 이용하여 단순한 에러 메시지를 출력할 수 있도록 한다.
   1) ex) tsh.c의 main 함수에 구현되어 있는 쉘 커멘드라인에 대한 예외처리

```
if ((fgets(cmdline, MAXLINE, stdin) == NULL) && ferror(stdin))
        app_error("fgets error");
```

주의 사항

1. Shell Lab을 수행하다 보면, 부모 프로세스가 자식 프로세스보다 먼저 종료되어 자식 프로세스가 좀비가 되는 경우가 있다.

   1) 쉘 종료 후, 'ps' 명령어를 입력

```
[b000000000@eslab shlab-handout]$ ps
  PID TTY          TIME CMD
 7714 pts/1    00:00:00 bash
 7805 pts/1    00:00:00 ps
```

   2) 위와 같이 tsh가 좀비가 되어 남아있는 것을 볼 수 있다. 따라서 해당 좀비 프로세스를 제거해야 한다.

   - kill -9 <PID>
   - ex) kill -9 29895

   3) 좀비 프로세스를 많이 만들면 감점! (쉘 테스트 후 실시간 체크 바람)

과제

1. Shell Lab
   1) trace08 ~ trace21에 대한 코드 작성
2. Shell Lab 보고서
   1) 각 trace 별, tshref를 수행한 결과와 본인이 구현한 tsh와의 동작 일치를 증명
      - sdriver 수행 결과와 tsh에서의 정상작동 모습(-v 옵션 사용)
   2) 각 trace 별, 플로우 차트
      - 간단한 수행과정을 플로우차트로 나타내면 됨.
   3) 각 trace 별, 해결 방법에 대한 설명
3. 제출
   1) shlab-handout 디렉토리를 통째로 압축
      - 파일명: [sys00]shell2_학번.tar.gz
   2) 결과 보고서를 작성
      - 파일명: [sys00]shell2_학번.pdf
   3) I.과 II. 두개를 하나로 압축
      - 파일명:[sys00]shell2_학번_이름.zip

제출 사항

1. 사이버캠퍼스에 제출(추가적으로 보고서를 출력하여 서면 제출)
   1) 자세한 양식은 앞장 슬라이드 참고.
   2) 파일 제목: [sys00]shell2_학번_이름.zip
   3) 반드시 위의 양식을 지켜야 함. (위반 시 감점)

2. 제출일자
   1) 사이버 캠퍼스: 2016년 11월 29일 화요일 08시 59분 59초까지
   2) 서면 제출 : 2016년 11월 29일 실습시간까지