

시스템 프로그래밍

실습 과제 (10) (tshlab2)

01분반

TinyShell 구현 2주차
(trace 08 ~ trace 21)

<목차>

1. 소개	3
- 실습 명	
- 목표	
- shell 이란?	
2. 구현 사항	
(1) 소스 구성	4
- 소스 구성에 대한 간단한 설명	
(2) trace별 구현사항(해결 사항과 해결 방법, 관련 지식, flowchart)	5
- trace08	
- trace09	
- trace10	
- trace11, trace12	
- trace13, trace14	
- trace15	
- trace16	
- trace17	
- trace18	
- trace19	
- trace20	
- trace21	
(3) 주요 함수 설명	50
- main	
- eval	
- builtin_cmd	
- 그 외의 함수	
- tsh.c 파일 동작 설명과 flowchart	
3. 고찰 및 실행 결과 분석	60

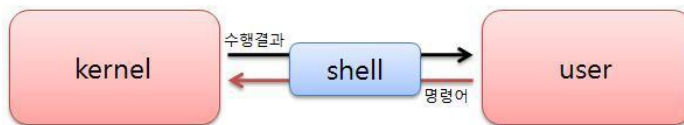
1. 소개

- 실습 명 : Tiny Shell의 구현

- 목표 : 프로세스 및 시그널 제어를 학습하고, 작업관리를 지원하는 간단한 Unix 셸 프로그램의 구현

- Shell 이란?

shell은 사용자와 유닉스의 핵심인 커널을 연결시켜주는 중간적인 역할을 한다.



shell은 사용자와 커널 사이의 인터페이스(interface) 역할을 하며, 사용자가 입력한 명령어를 읽고 해석하는 프로그램이다. (command processer) 해석 형 프로그래밍 언어로 자신의 shell 스크립트를 생성 및 실행 등 그 외 여러 가지 기능을 가진다.

2. 구현 사항

2-(1) 소스 구성

- 소스 구성에 대한 간단한 설명

tsh.c : Tiny Shell Lab 메인 프로그램(이 파일을 통해 shell 프로그램을 구현)
eval 함수, sigchld_handler, sigint_handler, sigtstp_handler 등은 구현해야할 코드부분.
메인 함수에서 command를 입력 받아 eval함수를 호출

tshref : 완전히 구현된 Tiny Shell
실제로 Tiny Shell이 어떻게 동작하는지 확인할 수 있다.

mycat.c myenv.c myintp.c myints.c myspin1.c myspin2.c mysplit.c mysplitp.c mystpp.c mystps.c : Tiny Shell을 테스트할 때 이용되는 프로그램

sdriver.c : Tiny Shell 테스트 프로그램(자신이 만든 쉘 프로그램이 제대로 작동하는지 테스트할 때 이용)

runtrace.c : Tiny Shell 테스트할 때 이용되는 sdriver에 의해 호출되는 프로그램

trace00.txt ~ trace24.txt : Tiny Shell 테스트 케이스

fork.c : Tiny Shell 프로그램에서 사용되는 fork() 함수에 대한 wrapper 함수

기타 : config.h, Makefile, tshlab-writeup.pdf(.ps)

joblist, parseline 과 같은 몇몇 함수는 이미 구현되어 있다.

command line은 공백에 의해 구분이 되며, 첫 번째 단어는 내장 명령어(builtin command) 또는 실행 파일 경로의 이름이다. 나머지 단어들은 command line의 인자(argument)들이다. 첫 번째 단어가 내장 명령어일 경우, Tiny shell은 현재 프로세스에서 해당하는 명령어를 즉시 실행한다.

첫 번째 단어가 실행 파일 경로의 이름일 경우, Tiny shell은 fork() 함수를 통해 자식을 생성하고, 자식 프로세스에서 해당하는 프로그램일 실행시킨다. (execve 함수)

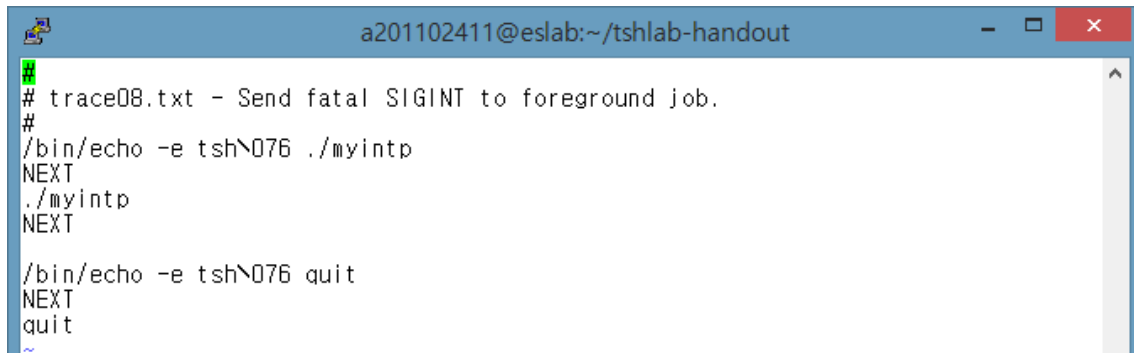
command line 맨 끝에 &를 붙이면 해당하는 작업(job)은 백그라운드(background)에서 실행되며, 그렇지 않은 경우는 포그라운드(foreground)에서 실행된다.

2-(2) trace별 구현사항(해결 사항, 해결 방법, 관련 지식, flowchart)

trace08

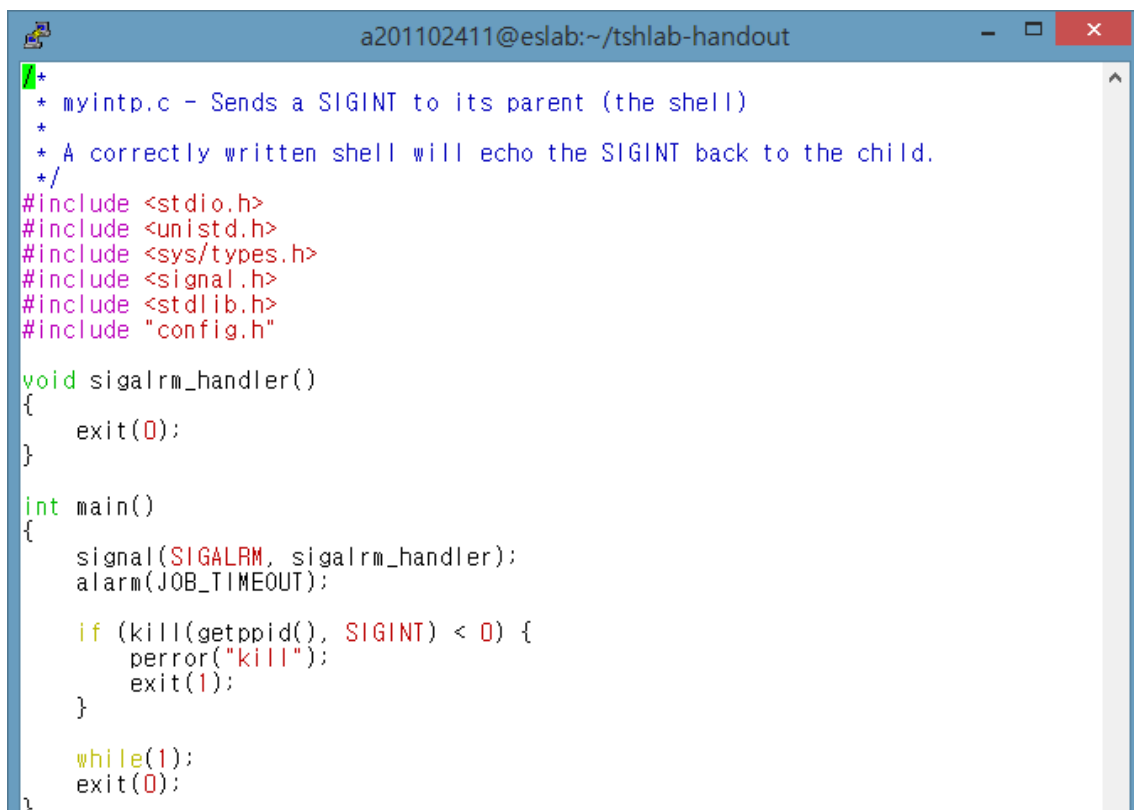
<해결 사항 및 해결 방법>

ctrl-c 가 입력되었을 경우 Signal을 발생시키고, 해당 프로세스를 kill하는 내용을 SIGINT 핸들러에 구현하고, 자식 프로세스가 종료될 경우 부모프로세스에게 넘겨주는 SIGCHLD 시그널을 받아 deletejob을 수행하는 SIGCHLD 핸들러를 구현한다.



```
a201102411@eslab:~/tshlab-handout
#
# 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
~
```

trace08.txt 파일을 열어 확인해 보면, myintp를 호출하는 것을 볼 수 있다.



```
a201102411@eslab:~/tshlab-handout
/*
 * myintp.c - Sends a SIGINT to its parent (the shell)
 *
 * A correctly written shell will echo the SIGINT back to the child.
 */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>
#include "config.h"

void sigalrm_handler()
{
    exit(0);
}

int main()
{
    signal(SIGALRM, sigalrm_handler);
    alarm(JOB_TIMEOUT);

    if (kill(getppid(), SIGINT) < 0) {
        perror("kill");
        exit(1);
    }

    while(1);
    exit(0);
}
```

myintp.c 파일을 열어 확인해 보면, kill 함수를 이용하여 SIGINT 신호를 부모 프로세스에게 보내는 것을 알 수 있다.

```
a201102411@eslab:~/tshlab-handout

/* Install the signal handlers */

/* These are the ones you will need to implement */
Signal(SIGINT, sigint_handler); /* ctrl-c */
Signal(SIGTSTP, sigtstp_handler); /* ctrl-z */
Signal(SIGCHLD, sigchld_handler); /* Terminated or stopped child */
Signal(SIGTTIN, SIG_IGN);
Signal(SIGTTOU, SIG_IGN);

/* This one provides a clean way to kill the shell */
Signal(SIGQUIT, sigquit_handler);
```

메인 함수를 살펴보면 위와 같다.

메인 함수에서 ctrl-c를 입력하면 SIGINT 신호를 보내주고, sigint_handler가 신호를 처리하는 것을 알 수 있다.

또한, 자식 프로세스가 종료되었거나 멈추었을 때, SIGCHLD signal을 보내고, sigchld_handler가 처리하는 것을 알 수 있다.

sigint_handler , sigchld_handler를 구현해야한다.

이를 구현하기 전에 waitpid를 수정해야 한다.

```
a201102411@eslab:~/tshlab-handout

223 /*
224 * waitfg - block until process pid is no longer the foreground process
225 */
226 void waitfg(pid_t pid, int output_fd)
227 {
228     struct job_t *j = getjobpid(jobs, pid);
229     char buf[MAXLINE];
230
231     /* The FG job has already completed and been reaped by the handler*/
232     if(!j)
233         return;
234
235     /* Wait for process pid to no longer be the foreground process.
236      * Note : using pause() instead of sleep() would introduces a race
237      * that could us to miss the signal
238      */
239     while (j->pid == pid && j->state == FG)
240         sleep(1);
241
242     if(verbose) {
243         memset(buf, '\0', MAXLINE);
244         sprintf(buf, "waitfg: Process (%d) no longer the fg process:\n", pid);
245         if(write(output_fd, buf, strlen(buf)) < 0) {
246             fprintf(stderr, "Error writing to file\n");
247             exit(1);
248         }
249     }
250     return;
251 }
252 */
```

시그널을 보낼 때, 치명적인 동시성 버그가 발생 할 수 있다. 다시 말해, 부모의 메인 루틴과 시그널 핸들링 흐름들의 일부 중첩에 대해서, addjob 전에 deletejob이 호출될 가능성이 있다. (경주 race 라고 하는 동기화 에러)

foreground 프로세스가 fork를 수행하여 foreground job을 두 개 이상의 프로세스로 작업할 수 있다. (헬은 시간상의 어느 시점에서도 최대 한 개의 foreground 작업과 0 또는 그 이상의 background job이 존재)

새로운 함수 waitfg를 구현한다. (실습수업자료 참고) 이 함수는 실행되고 있는 foreground job이 종료 될 때 까지 기다린다.

```
a201102411@eslab:~/tshlab-handout
/*
 * sigint_handler - The kernel sends a SIGINT to the shell whenever the
 * user types ctrl-c at the keyboard. Catch it and send it along
 * to the foreground job.
 */
void sigint_handler(int sig)
{
    pid_t pid; /* process ID */
    pid = fgpid(jobs); /* current process PID */
    kill(pid, SIGINT); /* sending signal SIGINT to pid */

    return;
}
```

sigint_handler 는 위와 같이 구현한다.

메인에서 SIGINT(ctrl-c)를 보내면 핸들러로 처리한다. 핸들러에서는 현재 프로세스의 ID를 찾아서, kill 함수를 이용하여 현재 프로세스에 SIGINT 신호를 보내서 프로세스를 종료시킨다.

프로세스가 종료되면 자식 프로세스는 부모 프로세스에게 SIGCHLD 신호를 보낸다. 이를 sigchld_handler가 처리한다.

```
a201102411@eslab:~/tshlab-handout
/*
 * sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
 * a child job terminates (becomes a zombie), or stops because it
 * received a SIGSTOP or SIGTSTP signal. The handler reaps all
 * available zombie children, but doesn't wait for any other
 * currently running children to terminate.
 */
void sigchld_handler(int sig)
{
    int status; /* status variable */
    pid_t pid; /* pid */

    /* waiting child process terminated */
    /* waitpid returns terminated process ID */
    while((pid = waitpid(-1, &status, 0))>0)
    {
        if(WIFSIGNALED(status)!=0) /* child process is terminated by SIGNAL */
        {
            /* print process information */
            /* WTERMSIG returns signal number causing termination */
            fprintf(stdout, "Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(status));
            deletejob(jobs, pid); /* removing job */
        }
    }

    return;
}
```

sigchld_handler는 위와 같이 구현한다.

waitpid 함수를 이용하여 프로세스가 종료하길 기다렸다가, 종료된 프로세스 id 를 반환받는다.

프로세스는 같은 종류의 시그널을 두 번 수신해서, 펜딩 시그널이 존재하게 될 때 더 이상 같은 시그널이 수신이 안 된다. 조건문으로 구현하면 시스템 콜이 중단될 수 있다.

따라서, 조건문이 아닌 반복문으로 구현한다. 그리고 반복문이 waitpid 반환 값이 0보다 클 때 돌아가도록 구현한다.

WIFSIGNALED 함수를 이용하여, 프로세스가 시그널에 의해 종료되었을 때, 종료된 프로세스를 deletejob 함수를 이용하여 job을 삭제한다.

```
a201102411@eslab:~/tshlab-handout
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* store command */
    int bg; /* should the job run in bg or fg? */
    pid_t pid; /* Process id */
    sigset_t mask; /* blocking signal */

    bg = parseline(cmdline, argv); /* parse command line */
    if(!builtin_cmd(argv)) {
        /* Blocking SIGCHLD to avoid a Race */
        if(sigemptyset(&mask) != 0)
            unix_error("sigemptyset error");
        if(sigaddset(&mask, SIGCHLD) != 0)
            unix_error("sigaddset error");
        if(sigprocmask(SIG_BLOCK, &mask, NULL) != 0)
            unix_error("sigprocmask error");

        /* fork() */
        if((pid=fork())<0) { /* forking error */
            unix_error("forking error");
        }
        else if(pid==0) { /* child process */
            if(sigprocmask(SIG_UNBLOCK, &mask, NULL) != 0) /* unblock SIGCHLD */
                unix_error("sigprocmask error");
            if((execve(argv[0], argv, environ)<0))
            {
                printf("%s : Command not found\n",argv[0]);
                exit(0);
            }
        }
        else {
            if(bg) /* add background job to the joblist */
                addjob(jobs, pid, BG, cmdline);
            else /* add foreground job to the joblist */
                addjob(jobs, pid, FG, cmdline);

            if(sigprocmask(SIG_UNBLOCK, &mask, NULL) != 0) /* unblock SIGCHLD */
                unix_error("sigprocmask error");
            /* parent waits for foreground job to terminate */
            if(!bg) { /* foreground */
                waitfg(pid, (int)stdout); /* terminating zombie child */
            }
            else { /* background */
                /* print according to pattern using func pid2jid */
                printf("(%d) (%d) %s", pid2jid(pid), pid, cmdline);
            }
        }
    }
    return;
}
222,1 29%
```

경주를 막기 위해 eval함수를 수정한다.

SIGCHLD 시그널을 fork를 호출하기 전에 블록하고, 그 후 이들을 addjob을 호출한 후에만 블록을 해제하면 자식이 작업 리스트에 추가된 후에 청소되는 것을 보장한다.

자식들이 보무의 blocked 된 집합을 이어받으므로, execve를 호출하기 전에 자식 내의 SIGCHLD 시그널을 unblock 해야 한다.

```
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./myspin2 10
^CJob [1] (2866) terminated by signal 2
eslab_tsh>
```

실행 결과를 확인하면, 위와 같다.

ctrl-c를 입력하면 foreground job이 종료됨을 알 수 있다.

trace08

<관련 지식>

- **int kill (pid_t pid, int sig);**

프로세스는 시그널을 kill 함수를 호출해서 다른 프로세스로 보낸다. (자기 자신을 포함)

if pid > 0, kill 함수는 시그널 번호 sig를 프로세스 pid로 보낸다.

if pid < 0, kill 함수는 시그널 sig를 프로세스 그룹 abs(pid) 내의 모든 프로세스로 보낸다.

- **pid_t waitpid(pid_t pid, int *status, int options);**

부모 프로세스는 waitpid 함수를 호출해서 자식들이 종료되거나 정지되기를 기다림

- 반환 값

if ok, 자식의 pid

if WNOHANG, 0

if error, -1

- 대기 집합의 멤버는 pid 인자에 의해 결정

if pid > 0, 대기 집합은 프로세스 ID가 pid와 동일한 단독 자식 프로세스

if pid = -1, 대기 집합은 부모의 모든 자식 프로세스

- options

WNOHANG : 대기 집합 내의 자식 프로세스 중 아무것도 아직 종료되지 않았다면 즉시 리턴. 기본 동작은 자식이 종료할 때까지 호출한 프로세스를 정지.

WUNTRACED : 대기 집합의 프로세스가 종료되거나 정지될 때까지 호출한 프로세스의 실행을 정지. 리턴을 발생시킨 종료되거나 정지한 자식의 pid를 리턴. 기본 동작은 종료된 자식에 대해서만 리턴.

WNOHANG | WUNTRACED : 대기 집합 모든 자식 프로세스들이 정지하였거나 종료하였다면 리턴 값 0으로 즉시 리턴되거나 자식들 중 한 개의 pid와 동일한 값으로 리턴

- **경주(race)**

같은 저장 장치의 위치에서 읽고 쓰는 동시성 흐름을 프로그래밍 할 때, 다수의 중첩된 제어 인스트럭션 때문에 정확한 답을 얻지 못한다.

즉, 2개 이상의 프로세스가 동일한 자원을 접근하려고 할 때 발생하는 동기화 에러

- **WIFSIGNALED(status)**

status 가 null이 아니면, waitpid 는 리턴되어진 자식에 관한 status 정보를 해독.

WIFSIGNALED는 자식 프로세스가 시그널에 의해 종료 되었을 경우 true를 반환한다.

- **WTERMSIG(status)**

자식 프로세스를 종료하게 한 시그널의 수를 리턴. 이 status는 WIFSIGNALED(status)가 true를 리턴하는 경우에만 정의된다.

- **SIGINT**

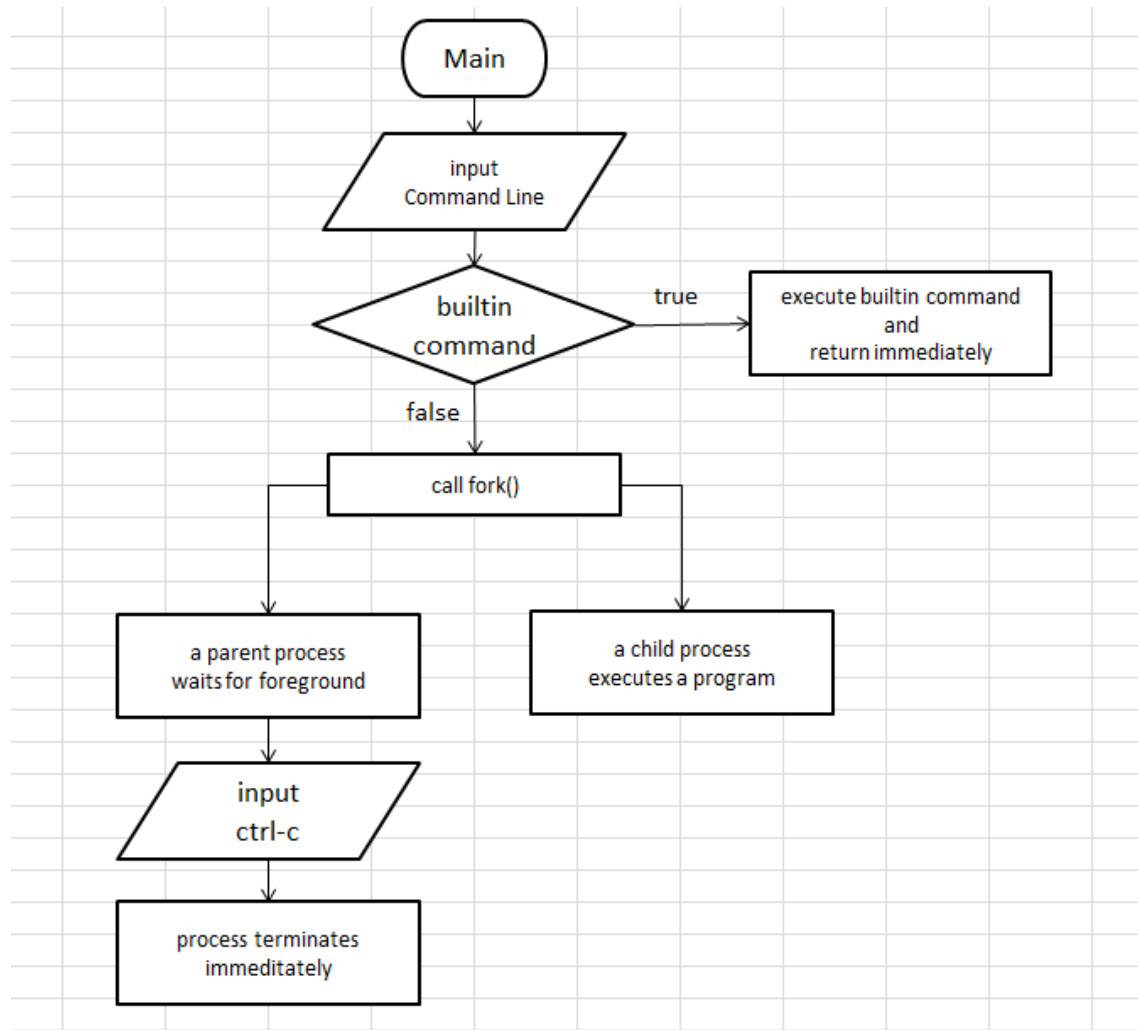
키보드에서 interrupt가 일어날 때, 발생하는 시그널. 기본 동작은 프로세스를 종료하는 것.

- `int sigemptyset(sigset_t *set);`
`set`을 비어있는 집합으로 초기화 해주는 함수
 - 반환 값
 if ok, 0
 if error, -1

- `int sigaddset(sigset_t *set, int signum)`
`signum`을 `set`에 추가해준다.
 - 반환 값
 if ok, 0
 if error, -1

- `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)`
 현재 블록된 시그널의 집합을 변경
 특정 동작은 `how` 값에 따라 달라진다.
`SIG_BLOCK` : `set`에 있는 시그널들을 `blocked`에 추가(`blocked = blocked | set`)
`SIG_UNBLOCK` : `set`에 있는 시그널들을 `blocked`에서 제거한다. (`blocked = blocked & ~set`)
`SIG_SETMASK` : `blocked = set`
 - 반환 값
 if ok, 0
 if error, -1

trace08
<flowchart>



trace08 flowchart

trace09

<해결 사항 및 해결 방법>

ctrl-z 가 입력되었을 경우 Signal을 발생시키고, 해당 프로세스를 stop하는 내용(kill 시스템 콜 이용)을 SIGTSTP 핸들러에 구현하고, 자식 프로세스가 정지될 경우 부모 프로세스에게 넘겨주는 SIGCHLD 시그널을 받아 job 상태를 ST 상태로 전환하는 SIGCHLD 핸들러를 구현한다.

```
a201102411@eslab:~/tshlab-handout
# 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
..
```

trace09.txt를 살펴보면 mytstpp를 호출한다.

```
a201102411@eslab:~/tshlab-handout
/*
 * mytstpp.c - Sends a SIGTSTP to its parent (the shell)
 *
 * A correctly written shell will echo the SIGTSTP back to the child.
 */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>
#include "config.h"

void sigalrm_handler()
{
    exit(0);
}

int main()
{
    signal(SIGALRM, sigalrm_handler);
    alarm(JOB_TIMEOUT);

    if (kill(getppid(), SIGTSTP) < 0) {
        perror("kill");
        exit(1);
    }

    while(1);
    exit(0);
}
```

mytstpp 함수를 살펴보면 kill로 SIGTSTP 시그널을 부모 프로세스에게 보내준다. 따라서 시그널 핸들러를 구현해야한다.

먼저, 문제를 읽어보면, trace08과 비슷하다고 볼 수 있다. 단지 ctrl-z가 입력되면 SIGTSTP 핸들러가 실행되는 점이 차이점이다.

메인 함수에서 ctrl-z를 입력하면, SIGTSTP 시그널을 보낸다. 그리고 이 시그널을 sigtstp_handler에서 처리한다.

핸들러에서 처리하고, 자식 프로세스가 멈출 경우, SIGCHLD 시그널을 보내어 핸들러가 처리한다.

이 문제를 해결하기 위해서는 sigtstp_handler를 이용하여 새롭게 시그널을 정의하고, sigchld_handler에서 프로세스가 정지 상태인 경우를 처리하는 코드를 추가한다.

```
a201102411@eslab:~/tshlab-handout
/*
 * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
 * the user types ctrl-z at the keyboard. Catch it and suspend the
 * foreground job by sending it a SIGTSTP.
 */
void sigtstp_handler(int sig)
{
    pid_t pid;
    pid = fgpjod(jobs); /* get current process pid */
    kill(pid, SIGTSTP); /* sending signal to process pid */
    return;
}
```

sigtstp_handler를 구현하면 위와 같다. (sigint_handler 와 비슷)
현재 프로세스의 id를 받아서, kill 함수를 이용하여 현재 프로세스에 SIGTSTP 시그널을 보낸다. 프로세스가 정지되면, 정지된 프로세스가 부모 프로세스에게 SIGCHLD 신호를 보낸다.
따라서, sigchld_handler를 수정해주어야 한다.

```
a201102411@eslab:~/tshlab-handout
/*
void sigchld_handler(int sig)
{
    int status; /* status variable */
    pid_t pid; /* pid */

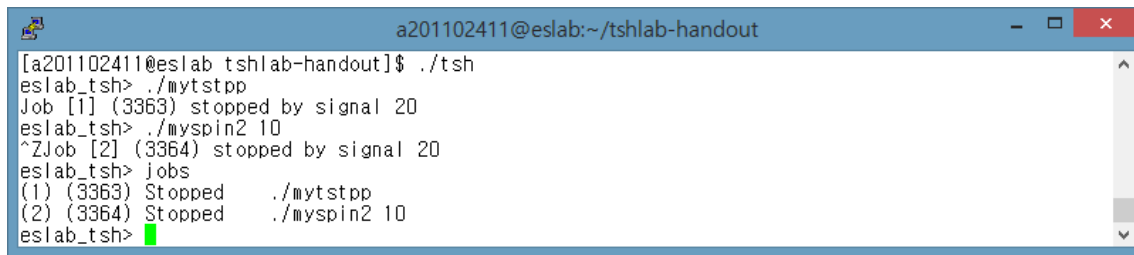
    /* waiting child process terminated */
    /* waitpid returns terminated process ID */
    while((pid = waitpid(-1, &status, WUNTRACED|WNOHANG))>0)
    {
        struct job_t *j = getjobpid(jobs, pid); /*find job having pid in joblist */
        if(WIFSIGNALED(status)) /* child process has terminated by SIGNAL */
        {
            /* print process information */
            /* WTERMSIG returns signal number causing termination */
            fprintf(stdout, "Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(status));
            deletejob(jobs, pid); /* removing job */
        }
        else if(WIFSTOPPED(status)) /* child process has stopped */
        {
            j->state = ST; /* job state is ST(stop) */
            /* WSTOPSIG returns signal number causing stopped */
            fprintf(stdout, "Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(status));
        }
    }
    return;
}
```

sigchld_handler를 수정하면 위와 같다.
정지된 상태의 프로세스를 검사하기 위해 조건문을 추가한다.
WIFSTOPPED를 이용하여 자식 프로세스가 정지된 상태이면, 해당하는 job의 상태를 ST로 바꾸어 준다.
waitpid 옵션으로 WNOHANG|WUNTRACED를 사용한다. 이 옵션은 대기 집합 모든 자식 프로세스들이 정지하였거나 종료하였다면 리턴 값 0으로 즉시 반환한다. 또는 반환되거나 자식들 중 한 개의 pid와 동일한 값으로 반환한다.

그리고 eval함수에서 sigaddset 함수를 이용하여 SIGTSTP를 set에 추가한다.

실행결과를 확인하면 아래와 같다.
ctrl-z를 입력하면 프로세스가 정지됨을 알 수 있다.

```
a201102411@eslab:~/tshlab-handout
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./myspin1 10
^ZJob [1] (29125) stopped by signal 20
eslab_tsh> ./myspin2 20
^ZJob [2] (29126) stopped by signal 20
eslab_tsh>
```



```
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./mytstp
Job [1] (3363) stopped by signal 20
eslab_tsh> ./myspin2 10
^ZJob [2] (3364) stopped by signal 20
eslab_tsh> jobs
(1) (3363) Stopped      ./mytstp
(2) (3364) Stopped      ./myspin2 10
eslab_tsh> █
```

trace09

<관련 지식>

●WIFSTOPPED(status)

리턴을 하게한 자식이 현재 정지된 상태라면 true 반환.

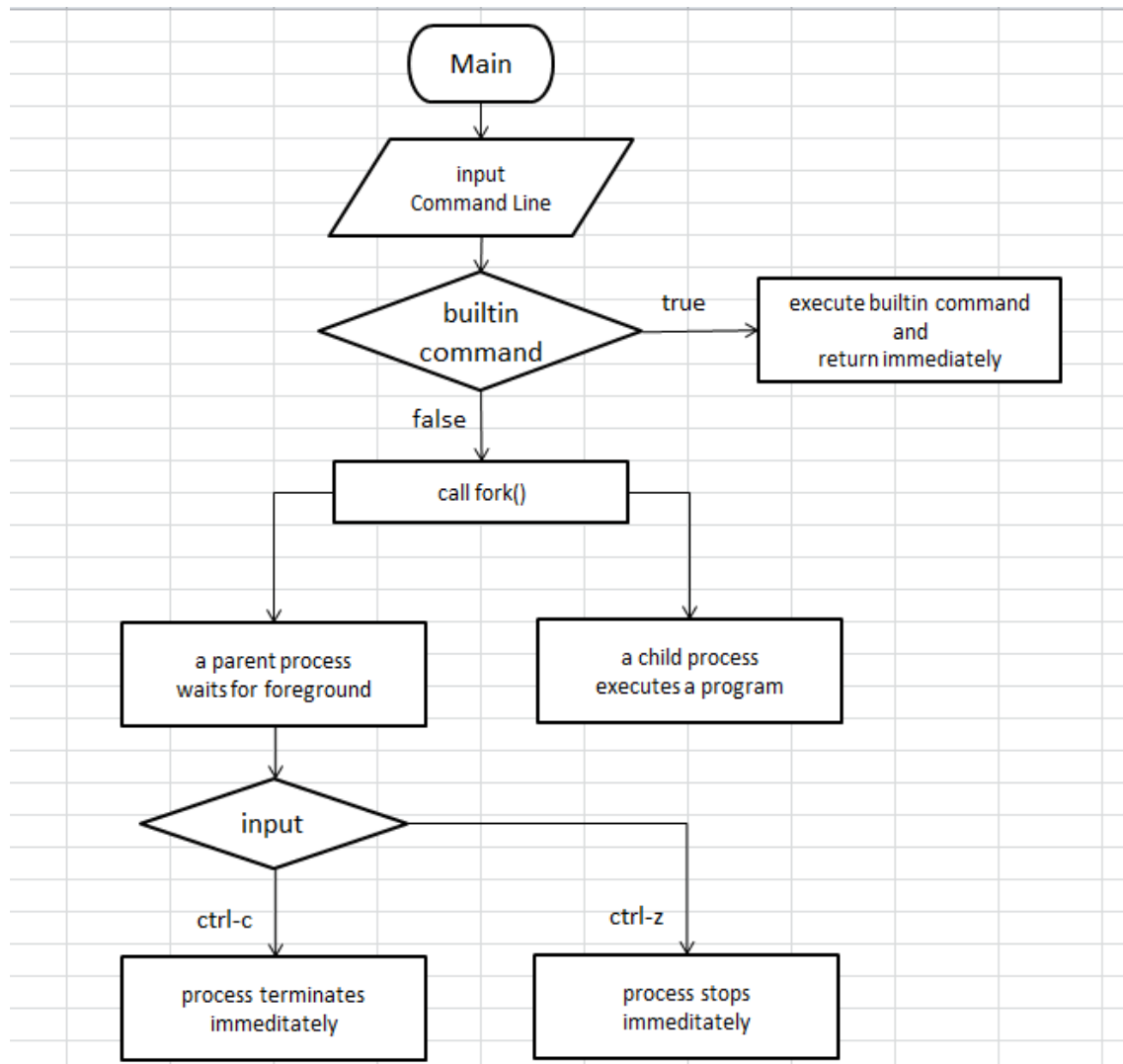
●WSTOPSIG

자식을 정지하게 한 시그널의 수를 리턴. 이 status 는 WIFSTOPPED(status)가 true를 리턴 하는 경우에만 정의

●SIGTSTP

terminal 로부터 stop 시그널이 오는 이벤트.
다음 SIGCONT 시그널이 올 때까지 멈춰있는다.

trace09
<flowchart>



trace09 flowchart

trace10

<해결 사항 및 해결 방법>

백그라운드 job을 종료하는 SIGTERM 내용을 SIGCHLD에서 구현한다.

(SIGTERM 핸들러 필요없음)

SIGCHLD를 처리하는 Handler 함수를 만들고 Signal로 핸들러를 등록한다.

핸들러 내부 기능은 자식 프로세스가 종료한 뒤에 부모 프로세스에게 알려주는 SIGCHLD를 처리하는 내용으로, 기본적으로 자식 프로세스가 사용 중인 메모리 자원을 반환 시켜주는 기능을 처리한다.

```
a201102411@eslab:~/tshlab-handout
# 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
```

trace10은 위와 같다. 대략적으로 살펴보면 myspin1을 5초 동안 background에서 실행하고, kill 함수를 이용하여 SIGTERM을 보낸다는 것을 알 수 있다. 이 시그널은 background job을 종료 시킨다. 그리고 SIGCHLD 시그널이 발생해 sigchld_handler가 실행된다는 것을 알 수 있다. 따라서, sigchld_handler를 적절히 수정해야 한다.

프로세스가 정상적으로 종료되었을 경우, 해당하는 프로세스를 삭제해야 한다.

이러한 해결을 통해서 우리는 좀비를 만들지 않게 된다.

구현해 놓은 sigchld_handler 함수를 다음과 같이 수정한다.

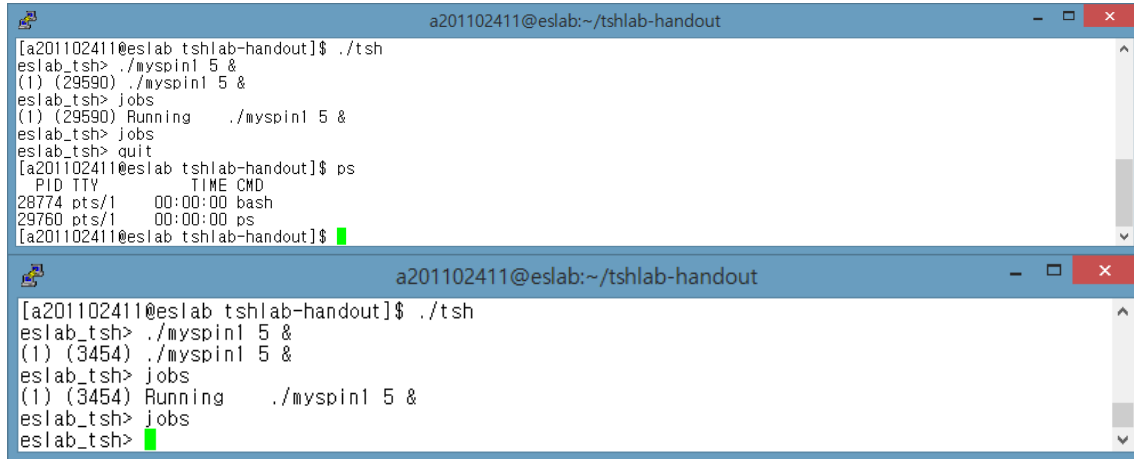
```
a201102411@eslab:~/tshlab-handout
*/
void sigchld_handler(int sig)
{
    int status; /* status variable */
    pid_t pid; /* pid */

    /* waiting child process terminated */
    /* waitpid returns terminated process ID */
    while((pid = waitpid(-1, &status, WUNTRACED|WNOHANG))>0)
    {
        struct job_t *j = getjobpid(jobs, pid); /*find job having pid in joblist */
        if(WIFSIGNALED(status)) /* child process has terminated by SIGNAL */
        {
            /* print process information */
            /* WTERMSIG returns signal number causing termination */
            fprintf(stdout, "Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(status));
            deletejob(jobs, pid); /* removing job */
        }
        else if(WIFSTOPPED(status)) /* child process has stopped */
        {
            j->state = ST; /* job state is ST(stop) */
            /* WSTOPSIG returns signal number causing stopped */
            fprintf(stdout, "Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(status));
        }
        else if(WIFEXITED(status)) /* process normally terminates */
        {
            deletejob(jobs, pid); /* removing job */
        }
    }
    return;
}
```

WIFEXITED를 이용하여 프로세스가 정상적으로 종료 되었는지 확인하고, 해당하는 job을 joblist에서 삭제한다.

프로세스가 시그널에 의해 종료될 경우에도 joblist에서 job을 삭제해야한다. 이는 이미 `trace08`에서 구현하였다.

실행한 결과는 다음과 같다.



```
a201102411@eslab:~/tshlab-handout
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./myspin1 5 &
(1) (29590) ./myspin1 5 &
eslab_tsh> jobs
(1) (29590) Running    ./myspin1 5 &
eslab_tsh> jobs
eslab_tsh> quit
[a201102411@eslab tshlab-handout]$ ps
  PID TTY          TIME CMD
 28774 pts/1    00:00:00 bash
 29760 pts/1    00:00:00 ps
[a201102411@eslab tshlab-handout]$
```



```
a201102411@eslab:~/tshlab-handout
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./myspin1 5 &
(1) (3454) ./myspin1 5 &
eslab_tsh> jobs
(1) (3454) Running    ./myspin1 5 &
eslab_tsh> jobs
eslab_tsh>
```

trace10

<관련 지식>

- SIGTERM

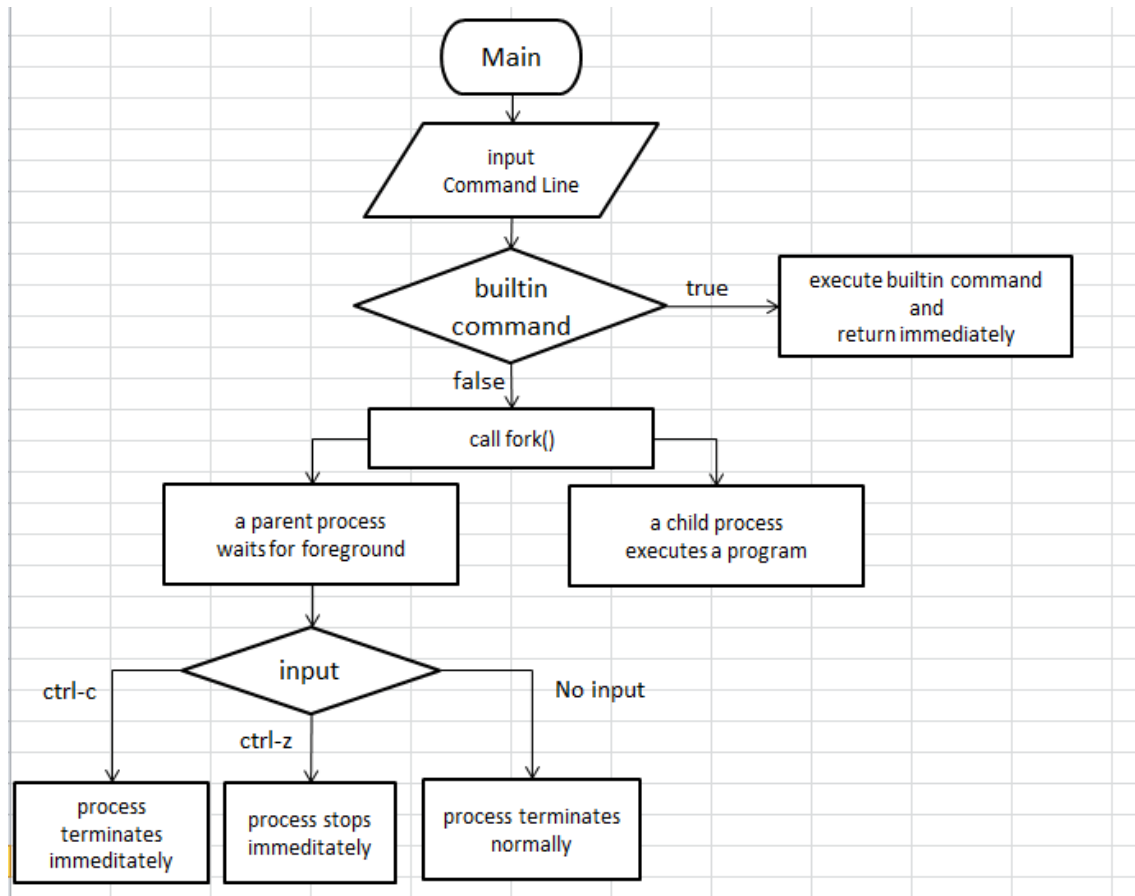
소프트웨어 종료 시그널에 의해 발생하는 이벤트.

기본 동작은 프로세스를 종료시킨다.

- WIFEXITED(status)

자식이 정상적으로 종료되었다면 `exit`으로의 호출 또는 리턴을 통해서 `true`를 리턴.

trace10
<flowchart>



trace10 flowchart

tracel1

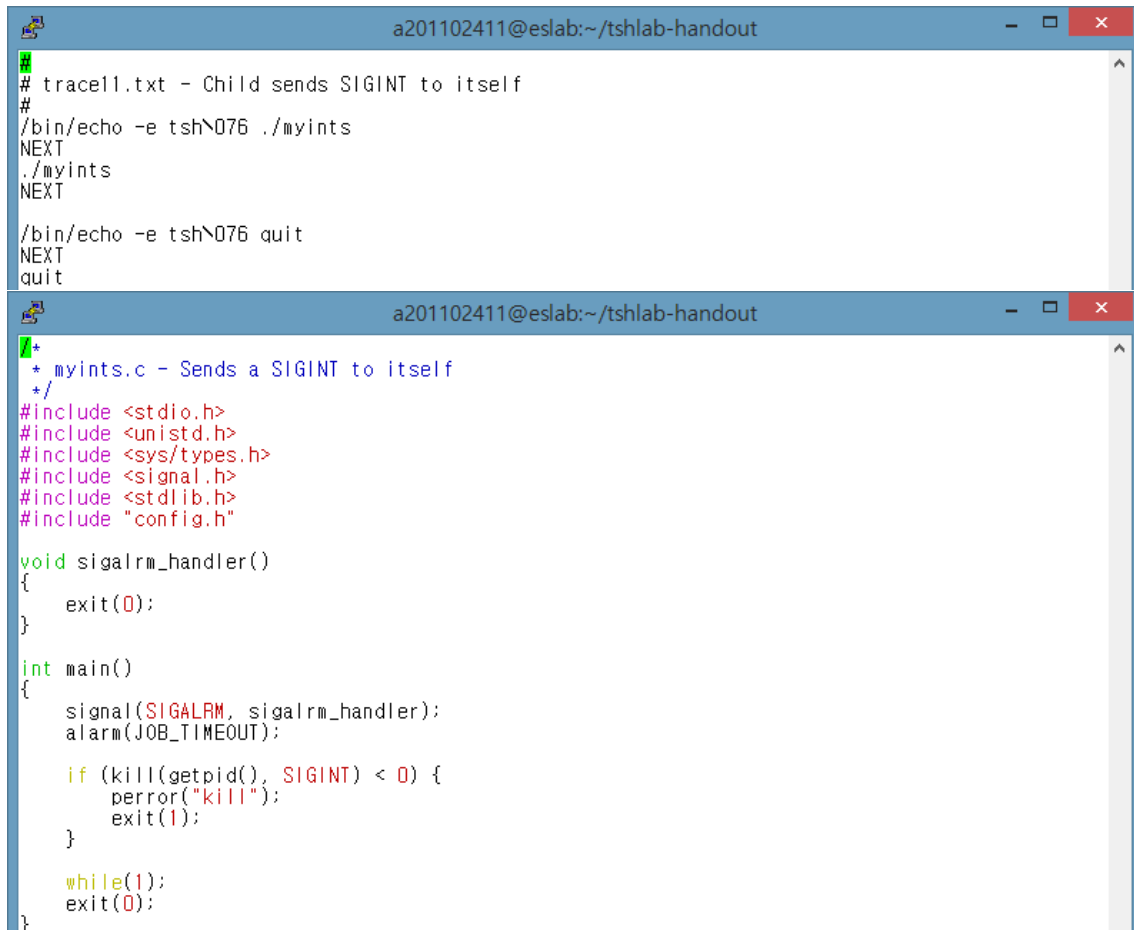
<해결 사항 및 해결 방법>

자식 프로세스가 스스로에게 SIGINT를 보내는 것을 구현한다.

tracel2

<해결 사항 및 해결 방법>

자식 프로세스가 스스로에게 SIGTSTP를 보내는 것을 구현한다.



```
a201102411@eslab:~/tshlab-handout
# tracel1.txt - Child sends SIGINT to itself
#
/bin/echo -e tsh\076 ./myints
NEXT
./myints
NEXT
/bin/echo -e tsh\076 quit
NEXT
quit

a201102411@eslab:~/tshlab-handout
/*
 * myints.c - Sends a SIGINT to itself
 */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>
#include "config.h"

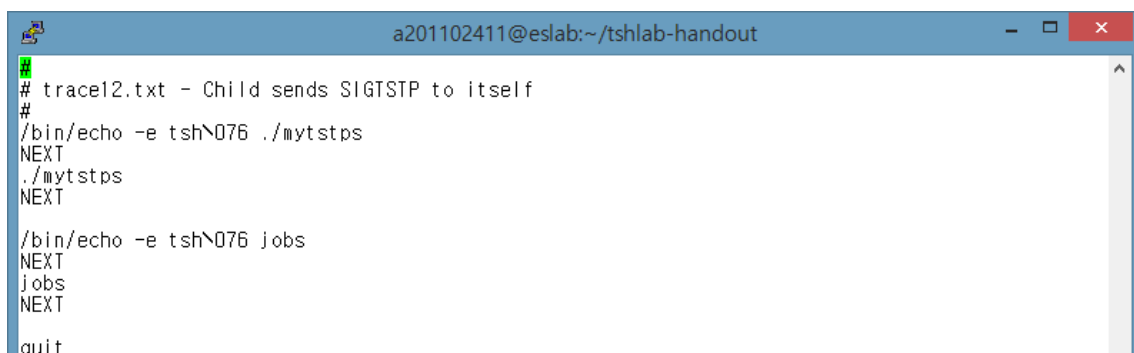
void sigalrm_handler()
{
    exit(0);
}

int main()
{
    signal(SIGALRM, sigalrm_handler);
    alarm(JOB_TIMEOUT);

    if (kill(getpid(), SIGINT) < 0) {
        perror("kill");
        exit(1);
    }

    while(1);
    exit(0);
}
```

tracel1에서 호출하는 myints 코드를 살펴보면, kill 함수를 통하여 SIGINT 신호를 자식 프로세스에게 전해준다.



```
a201102411@eslab:~/tshlab-handout
# tracel2.txt - Child sends SIGTSTP to itself
#
/bin/echo -e tsh\076 ./mytstps
NEXT
./mytstps
NEXT
/bin/echo -e tsh\076 jobs
NEXT
jobs
NEXT
quit
```

```
a201102411@eslab:~/tshlab-handout
/*
 * mytstps.c - Sends a SIGTSTP to itself, terminates when restarted.
 */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

int main()
{
    if (kill(getpid(), SIGTSTP) < 0) {
        perror("kill");
        exit(1);
    }
    exit(0);
}
```

trace12에서 호출하는 mytstp 코드를 살펴보면, kill 함수를 이용하여 SIGTSTP 신호를 자식 프로세스에게 전달한다.

이미 trace8, 9를 구현하면서 위에 사항들이 만족되었다.

pid에 현재의 프로세스 값이 저장되므로 자기 자신 스스로에게 시그널을 보내도록 구현한다.

이미 앞의 trace를 해결하면서 구현이 되었다.

fgpid 함수를 이용하여 현재의 프로세스 id를 반환 받는다.

그리고 kill함수를 이용하여 현재 프로세스에게 시그널 신호를 보낸다.

```
a201102411@eslab:~/tshlab-handout
/*
 * sigint_handler - The kernel sends a SIGINT to the shell whenever the
 * user types ctrl-c at the keyboard. Catch it and send it along
 * to the foreground job.
 */
void sigint_handler(int sig)
{
    pid_t pid; /* process ID */
    pid = fgpid(jobs); /* current process PID */
    kill(pid, SIGINT); /* sending signal SIGINT to pid */

    return;
}

/*
 * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
 * the user types ctrl-z at the keyboard. Catch it and suspend the
 * foreground job by sending it a SIGTSTP.
 */
void sigtstp_handler(int sig)
{
    pid_t pid;
    pid = fgpid(jobs); /* current process pid*/
    kill(pid, SIGTSTP); /* sending signal to process pid */
    return;
}
```

trace11의 실행결과는 다음과 같다.

```
a201102411@eslab:~/tshlab-handout
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./myints
Job [1] (3735) terminated by signal 2
eslab_tsh> quit
[a201102411@eslab tshlab-handout]$
```

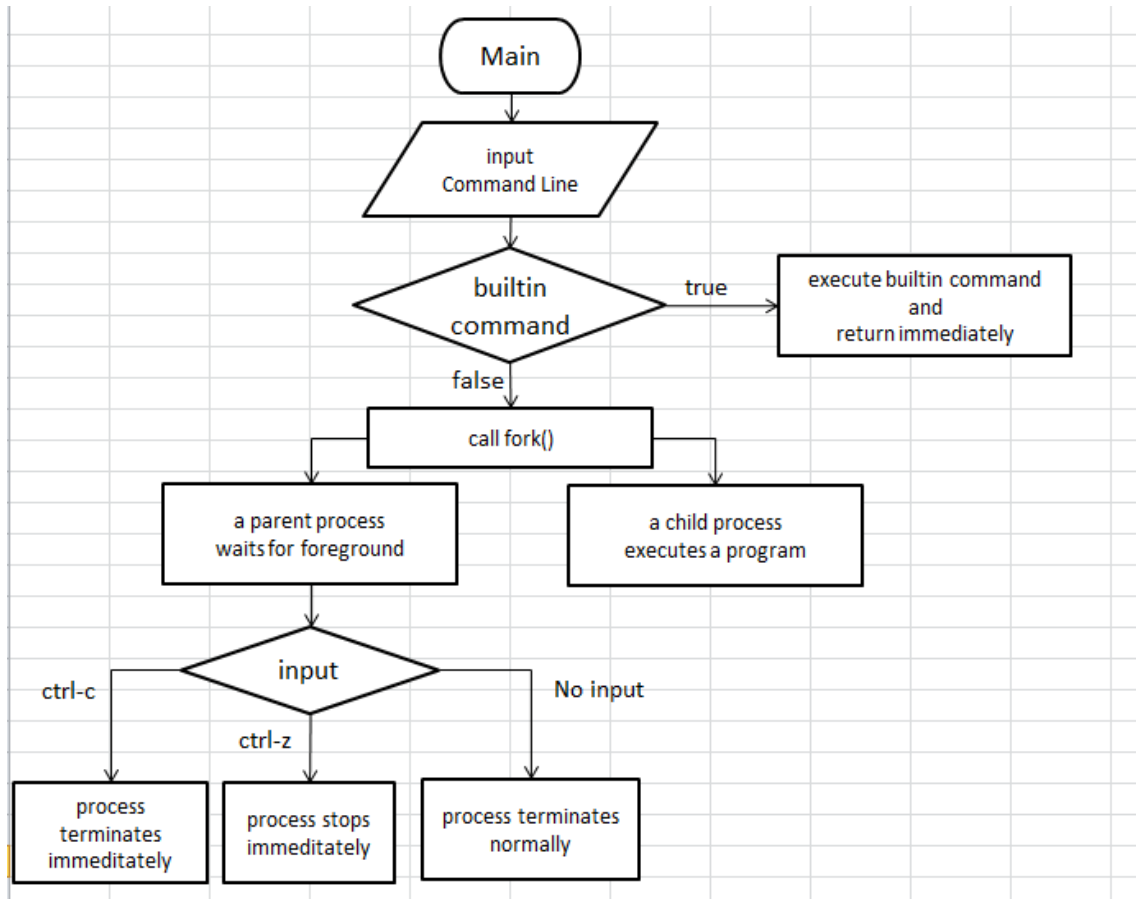
trace12의 실행결과는 다음과 같다.



```
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./mytstps
Job [1] (4109) stopped by signal 20
eslab_tsh> jobs
(1) (4109) Stopped      ./mytstps
eslab_tsh> 
```

trace11
<flowchart>
trace12
<flowchart>

trace10 과 동일한 flowchart.



trace11, 12 flowchart

trace13

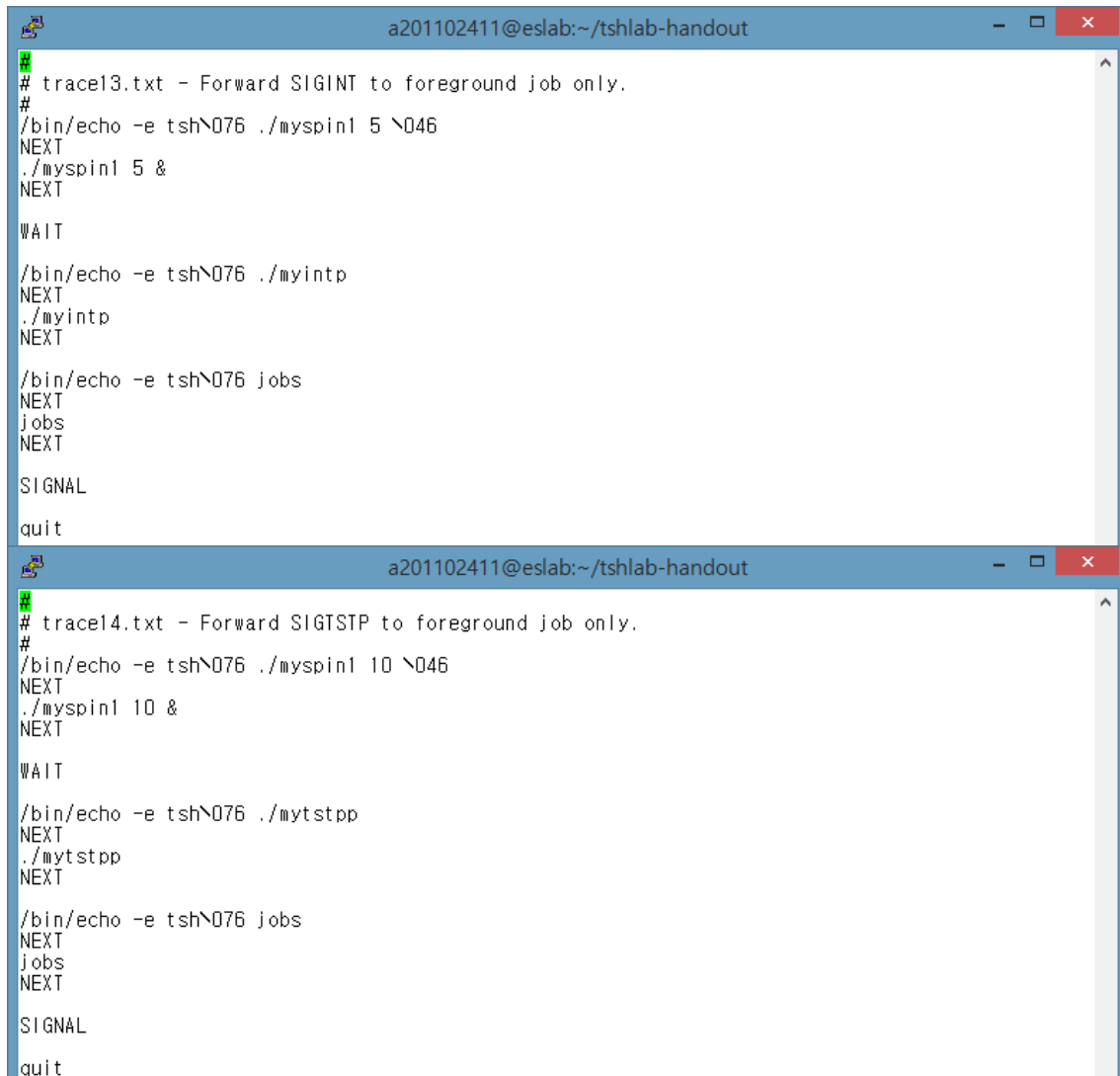
<해결 사항 및 해결 방법>

오직 foreground 에서만 SIGINT가 되도록 SIGINT 핸들러를 수정한다.

trace14

<해결 사항 및 해결 방법>

오직 foreground 에서만 SIGTSTP가 되도록 SIGINT 핸들러를 수정한다.



```
a201102411@eslab:~/tshlab-handout
# trace13.txt - Forward SIGINT to foreground job only.
#
/bin/echo -e tsh\076 ./myspinl 5 \046
NEXT
./myspinl 5 &
NEXT
WAIT
/bin/echo -e tsh\076 ./myintp
NEXT
./myintp
NEXT
/bin/echo -e tsh\076 jobs
NEXT
jobs
NEXT
SIGNAL
quit

a201102411@eslab:~/tshlab-handout
# trace14.txt - Forward SIGTSTP to foreground job only.
#
/bin/echo -e tsh\076 ./myspinl 10 \046
NEXT
./myspinl 10 &
NEXT
WAIT
/bin/echo -e tsh\076 ./mytstpp
NEXT
./mytstpp
NEXT
/bin/echo -e tsh\076 jobs
NEXT
jobs
NEXT
SIGNAL
quit
```

trace13, 14는 오직 foreground 에서만 시그널들이 수행되도록 수정해야한다.

이미 사용한 fgpId를 살펴보면, 현재의 foreground job process id를 반환하는 것을 알 수 있다. 그리고 해당하는 job이 없으면 0을 반환한다. (process id 는 0이 아닌 양수로 나타남)

따라서, 이미 fgpId를 사용해서 foreground 작업 내에서만 수행된다고 볼 수 있다.

foreground job을 리턴

```
a201102411@eslab:~/tshlab-handout
/* fgp_id - Return PID of current foreground job, 0 if no such job */
pid_t fgp_id(struct job_t *jobs) {
    int i;

    for (i = 0; i < MAXJOBS; i++)
        if (jobs[i].state == FG)
            return jobs[i].pid;
    return 0;
}
```

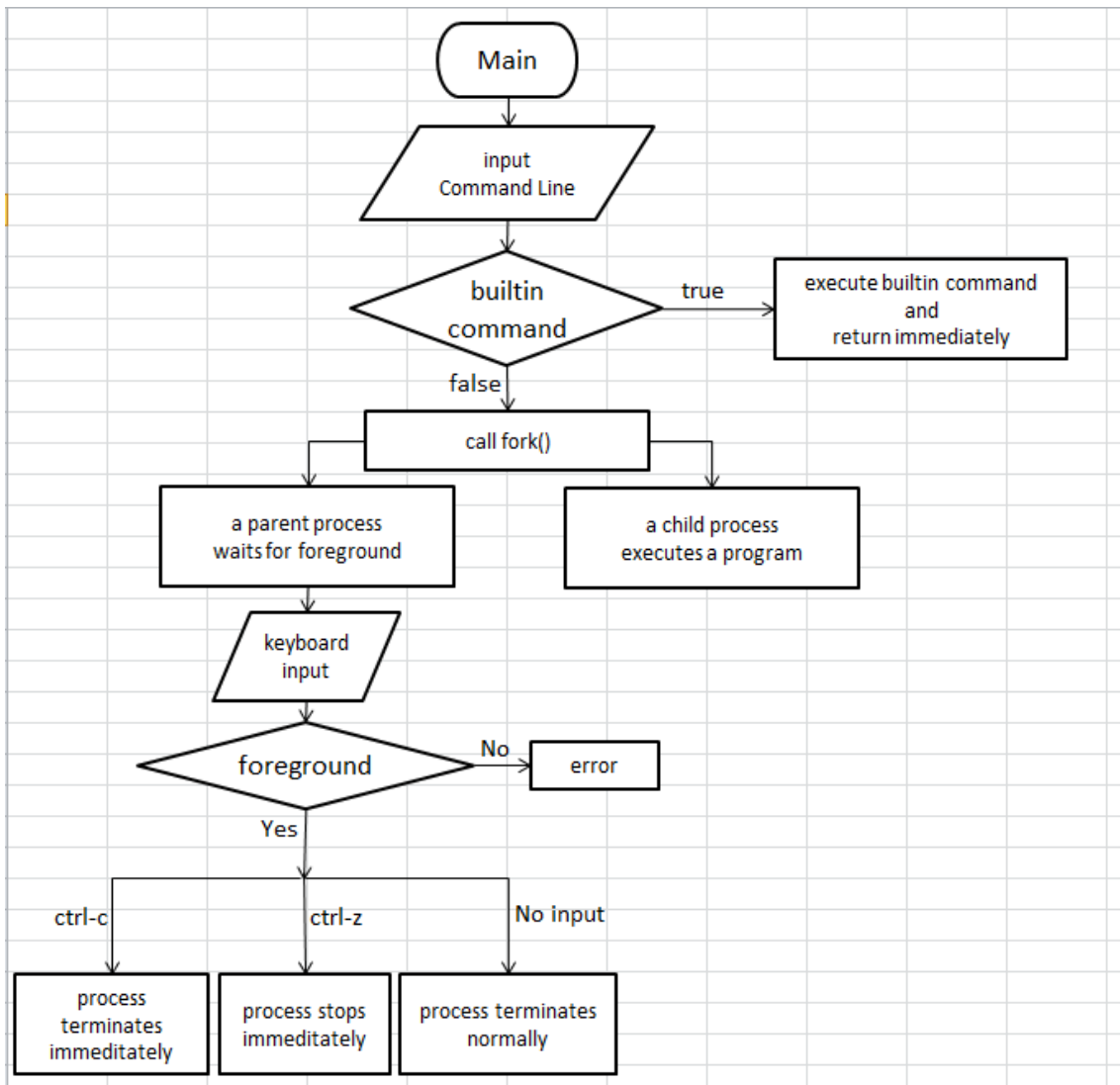
trace13의 실행결과는 다음과 같다.

```
a201102411@eslab:~/tshlab-handout
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./myspin1 5 &
(1) (4179) ./myspin1 5 &
eslab_tsh> ./myintp
Job [2] (4208) terminated by signal 2
eslab_tsh> jobs
(1) (4179) Running    ./myspin1 5 &
eslab_tsh> █
```

trace14의 실행결과는 다음과 같다.

```
a201102411@eslab:~/tshlab-handout
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./myspin1 10 &
(1) (4571) ./myspin1 10 &
eslab_tsh> ./mytstp
Job [2] (4622) stopped by signal 20
eslab_tsh> jobs
(1) (4571) Running    ./myspin1 10 &
(2) (4622) Stopped    ./mytstp
eslab_tsh> █
```

trace13
<flowchart>
trace14
<flowchart>

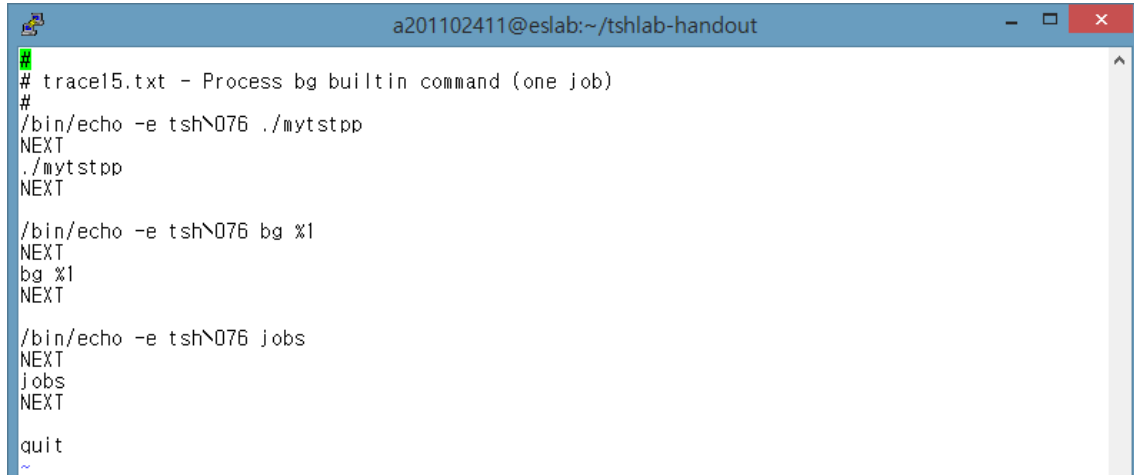


trace13, 14 flowchart

trace15

<해결 사항 및 해결 방법>

내장 명령어 bg를 구현한다. (하나의 job 테스트)



```
a201102411@eslab:~/tshlab-handout
# trace15.txt - Process bg builtin command (one job)
#
/bin/echo -e tsh\076 ./mytstpp
NEXT
./mytstpp
NEXT

/bin/echo -e tsh\076 bg %1
NEXT
bg %1
NEXT

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

quit
~
```

trace15에서 mytstpp 함수를 호출한다. 이 함수는 SIGTSTIP 시그널을 보내는데 사용되었었다. 이것은 job을 정지시킨다. 그 후에 내장 명령어 bg를 이용하여 이 작업을 background에서 다시 동작되도록 한다.

bg 명령어는 bg %job_id , bg pid 형태로 쓰인다.

job id 또는 pid에 해당하는 process에 SIGCONT 시그널을 보내고, 그 job을 background에서 실행시킨다.

먼저, builtin_cmd 함수를 수정한다.

입력받은 command가 bg이일 경우에 실행되는 조건문을 만든다.

bg 다음에 입력되는 명령어는 argv[1][0]에 저장되어있다. 이것이 %이면 job의 id가 입력된 것이다.

argv[1][1]은 %다음에 입력한 job id가 저장되어 있다. 이것을 문자열에서 int형으로 바꾸어 jid에 저장한다.

그리고 이 jid를 가지는 job을 getjobjid 함수를 통해 찾는다.

bg 다음에 문자 없이 숫자를 바로 입력한 경우는 pid를 입력한 경우이다. 이 값을 문자형에서 int형으로 변환하여 pid 변수에 저장한다.

그리고 getjobpid 함수를 이용하여 pid에 해당하는 job을 찾는다.

명령어에 해당하는 job을 찾았으면, kill 함수를 이용하여 해당하는 job의 프로세스로 SIGCONT 신호를 보낸다. background에서 다시 실행되어야 하므로 job의 state를 BG로 바꾼다.

```
a201102411@eslab:~/tshlab-handout
int builtin_cmd(char **argv)
{
    pid_t jid; /* job id */
    pid_t pid; /* process id */
    struct job_t *job; /* job structure variable for temporary save */

    if (!strcmp(argv[0], "quit")) { /* quit command */
        exit(0);
    }
    else if (!strcmp(argv[0], "jobs")) { /* jobs command */
        listjobs(jobs, STDOUT_FILENO);
        return 1;
    }
    else if (!strcmp(argv[0], "bg")) /* bg command */
    {
        if (argv[1][0] == '%') /* bg %(num) command */
        {
            jid = atoi(&argv[1][1]); /* changing char to int */
            job = getjobjid(jobs, jid); /* store jobs of jid */
        }
        else if (isdigit(argv[1][0])) /* bg (num) command */
        {
            pid = atoi(argv[1]); /* changing char to int */
            job = getjobpid(jobs, pid); /* store jobs of pid */
        }
        kill(job->pid, SIGCONT); /* sending SIGCONT */
        job->state = BG; /* changing ST state to BG state */
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
        return 1;
    }
    else {
        return 0; /* Not a builtin command */
    }
}
```

trace15의 실행결과는 다음과 같다.

```
a201102411@eslab:~/tshlab-handout
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./myspin1 10
^ZJob [1] (15728) stopped by signal 20
eslab_tsh> jobs
(1) (15728) Stopped      ./myspin1 10
eslab_tsh> bg %1
[1] (15728) ./myspin1 10
eslab_tsh> jobs
(1) (15728) Running     ./myspin1 10
eslab_tsh> █

a201102411@eslab:~/tshlab-handout
.[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./mytstpp
Job [1] (8296) stopped by signal 20
eslab_tsh> bg %1
[1] (8296) ./mytstpp
eslab_tsh> jobs
(1) (8296) Running     ./mytstpp
eslab_tsh> █
```

trace15

<관련 지식>

- c언어 isdigit 함수

숫자인지를 묻는 함수.

인자로 전달되는 문자가 숫자에 해당하는 문자라면 0이 아닌 값을 반환, 숫자라면 0을 반환한다.

- int atoi(const char *str);

str에 변화하고 싶은 문자열을 넣어주면 문자열을 정수로 변환한다.

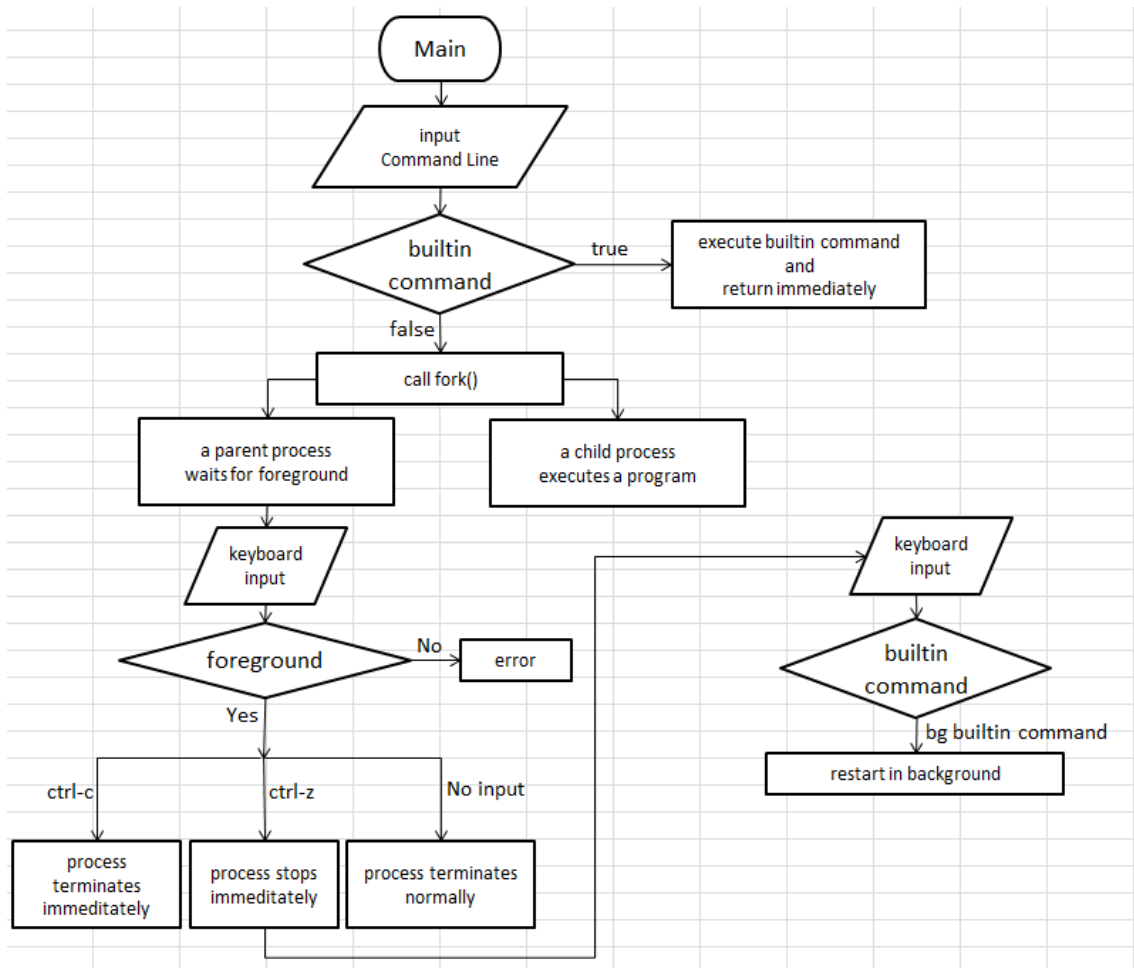
숫자로 된 문자열을 정수로 변환할 때 주로 사용

-반환 값

if ok, 변환에 성공한 정수 값

if fail, 0

trace15
<flowchart>



trace15 flowchart

trace16

<해결 사항 및 해결 방법>

내장 명령어 bg를 구현한다. (두 개의 job 테스트)

```
a201102411@eslab:~/tshlab-handout
# trace16.txt - Process bg builtin command (two jobs)
#
/bin/echo -e tsh\076 ./myspin1 10 \046
NEXT
./myspin1 10 &
NEXT
WAIT

/bin/echo -e tsh\076 ./mytstpp
NEXT
./mytstpp
NEXT

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

/bin/echo -e tsh\076 bg %2
NEXT
bg %2
NEXT

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

SIGNAL
quit
```

trace15와 동일하다. trace16은 두 개의 job을 실행시킨다.
다수의 프로세스에 대해 수행해보면 정상적으로 동작함을 알 수 있다.

실행결과는 다음과 같다.

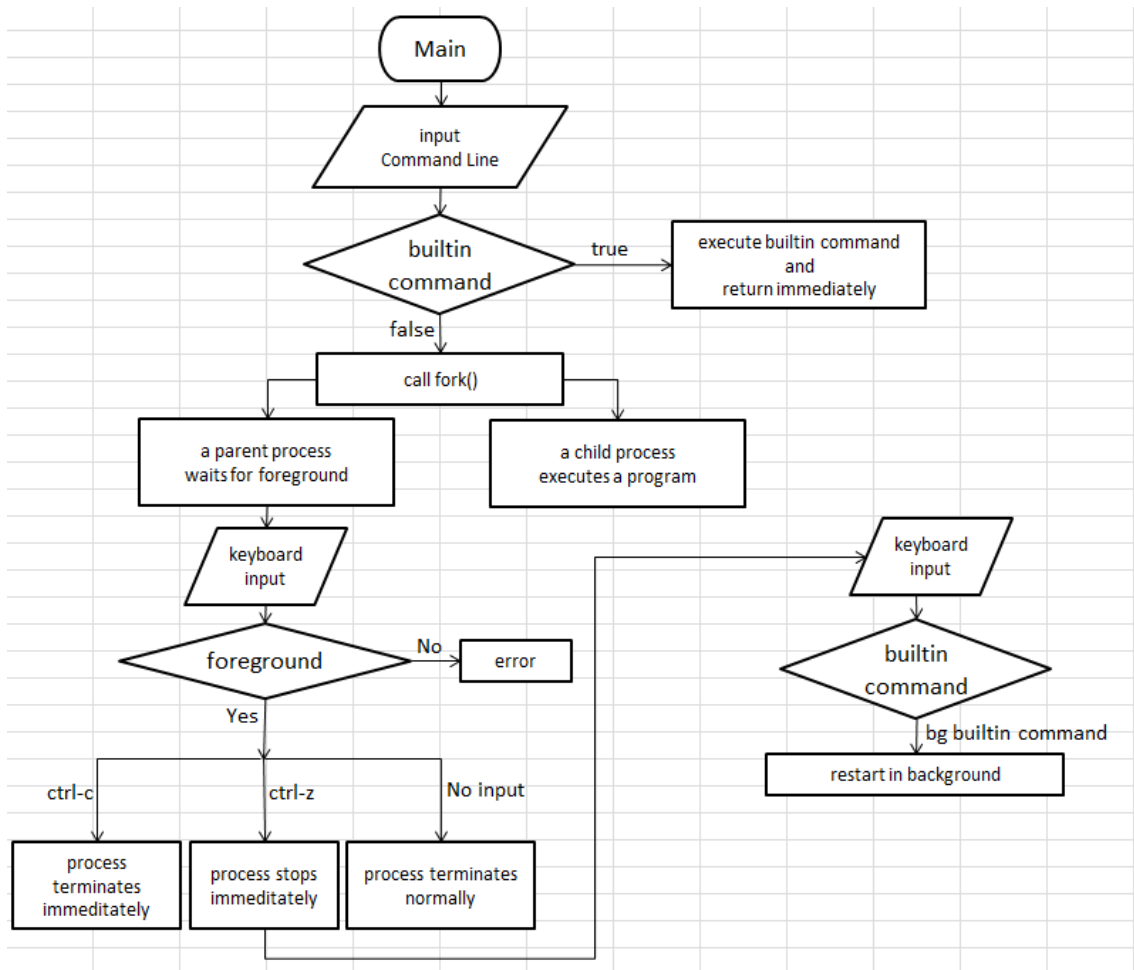
```
a201102411@eslab:~/tshlab-handout
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./myspin1 20
^ZJob [1] (23968) stopped by signal 20
eslab_tsh> ./myspin2 20 &
(2) (23975) ./myspin2 20 &
eslab_tsh> jobs
(1) (23968) Stopped      ./myspin1 20
(2) (23975) Running     ./myspin2 20 &
eslab_tsh> bg %1
[1] (23968) ./myspin1 20
eslab_tsh> jobs
(1) (23968) Running     ./myspin1 20
(2) (23975) Running     ./myspin2 20 &
eslab_tsh> jobs
(2) (23975) Running     ./myspin2 20 &
eslab_tsh> jobs
eslab_tsh> quit
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh>
```



```
a201102411@eslab:~/tshlab-handout
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./myspin1 10 &
(1) (8852) ./myspin1 10 &
eslab_tsh> ./mytstp
Job [2] (8872) stopped by signal 20
eslab_tsh> jobs
(1) (8852) Running    ./myspin1 10 &
(2) (8872) Stopped    ./mytstp
eslab_tsh> bg %2
[2] (8872) ./mytstp
eslab_tsh> jobs
(1) (8852) Running    ./myspin1 10 &
(2) (8872) Running    ./mytstp
eslab_tsh> █
```

trace16
<flowchart>

trace15 와 동일한 flowchart

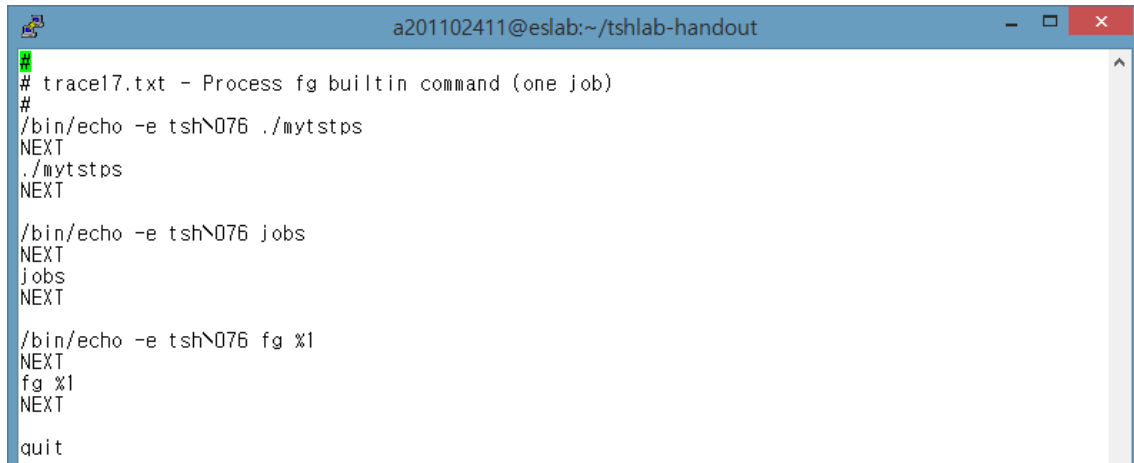


trace16 flowchart

trace 17

<해결 사항 및 해결 방법>

내장 명령어 fg를 구현한다. (하나의 job 테스트)



```
a201102411@eslab:~/tshlab-handout
# trace17.txt - Process fg builtin command (one job)
#
/bin/echo -e tsh\076 ./mytstps
NEXT
./mytstps
NEXT

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

/bin/echo -e tsh\076 fg %1
NEXT
fg %1
NEXT

quit
```

trace17에서는 mytstps를 호출한다. 정지된 작업을 fg 명령어를 이용하여 foreground 형태로 실행시킨다.

bg 명령어를 구현하였던 것과 비슷하게 구현한다.

먼저 builtin_cmd 함수에서 fg가 입력되는 경우 수행하는 부분을 만들고, 매개변수가 job id를 가리키는지 process id를 가리키는지에 따라 다르게 구현한다.

그리고 job의 상태를 FG로 바꿔준다.

```

a201102411@eslab:~/tshlab-handout
*/
int builtin_cmd(char **argv)
{
    pid_t jid; /* job id */
    pid_t pid; /* pocess id */
    struct job_t *job; /* job structure variable for temporary save */

    if (!strcmp(argv[0], "quit")) { /* quit command */
        exit(0);
    }
    else if (!strcmp(argv[0], "jobs")) { /* jobs command */
        listjobs(jobs, STDOUT_FILENO);
        return 1;
    }
    else if (!strcmp(argv[0], "bg")) /* bg command */
    {
        if(argv[1][0] == '%') /* bg %jid command */
        {
            jid = atoi(&argv[1][1]); /* changing char to int */
            job = getjobjid(jobs, jid); /* store jobs of jid */
        }
        else if(isdigit(argv[1][0])) /* bg pid command */
        {
            pid = atoi(argv[1]); /* changing char to int */
            job = getjobpid(jobs, pid); /* store jobs of pid */
        }
        kill(-(job->pid), SIGCONT); /* seding SIGCONT to process group */
        job->state = BG; /* changing ST state to BG state */
        printf("%d (%d) %s", job->jid, job->pid, job->cmdline);
        return 1;
    }
    else if (!strcmp(argv[0], "fg")) /* fg command */
    {
        if(argv[1][0] == '%') /* fg %jid command */
        {
            jid = atoi(&argv[1][1]);
            job = getjobjid(jobs, jid);
        }
        else if(isdigit(argv[1][0])) /* fg pid command*/
        {
            pid = atoi(argv[1]);
            job = getjobpid(jobs, pid);
        }
        kill(-(job->pid), SIGCONT); /* sending SIGCONT to process group */
        job->state = FG; /* change ST state to FG */
        waitfg(job->pid, 0); /* parent waits for foreground to terminate */
        return 1;
    }
    else {
        return 0; /* Not a builtin command */
    }
}

```

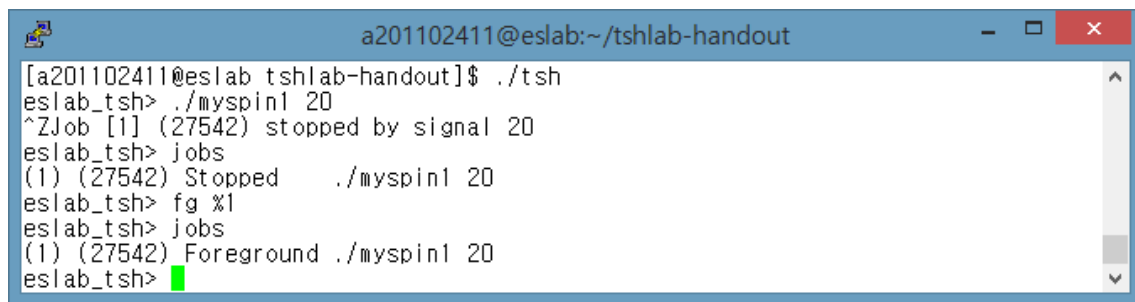
bg 내장 명령어와의 중요한 차이는 fg 명령어는 foreground에서 실행된다는 점이다. 이를 구현하기 위해서 waitfg 함수를 이용해, parent process가 foreground job이 종료될 때 까지 기다리도록 한다. 즉, foreground에서 job이 수행되는 동안, shell에서 prompt 창이 안보이고 커서의 깜빡임이 없어진다. (job이 수행 다 될 때까지)

실행 결과는 다음과 같다.

```

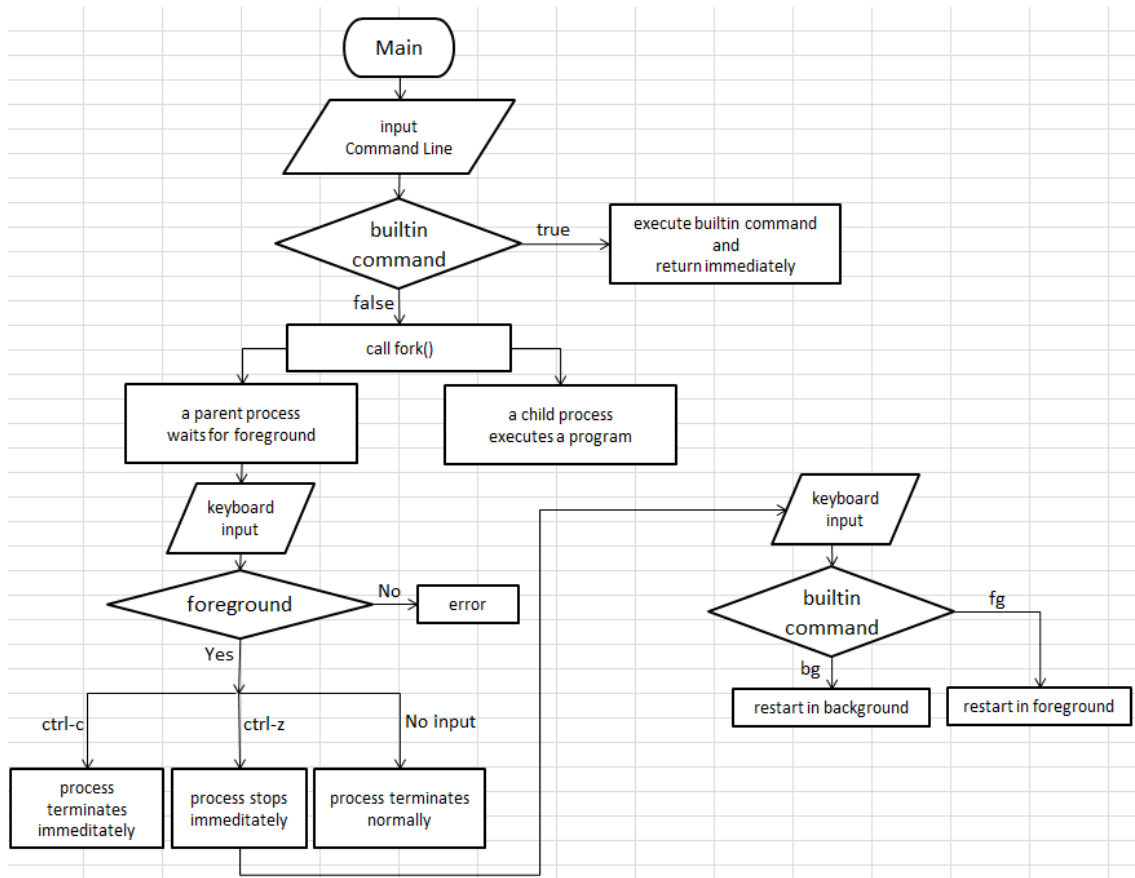
a201102411@eslab:~/tshlab-handout
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./mytstps
Job [1] (9480) stopped by signal 20
eslab_tsh> jobs
(1) (9480) Stopped      ./mytstps
eslab_tsh> fg %1
eslab_tsh>

```



```
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./myspin1 20
^ZJob [1] (27542) stopped by signal 20
eslab_tsh> jobs
(1) (27542) Stopped    ./myspin1 20
eslab_tsh> fg %1
eslab_tsh> jobs
(1) (27542) Foreground ./myspin1 20
eslab_tsh> 
```

trace17
<flowchart>



trace17 flowchart

trace18

<해결 사항 및 해결 방법>

내장 명령어 fg를 구현한다. (두 개의 job 테스트)

```
a201102411@eslab:~/tshlab-handout
#
# trace18.txt - Process fg builtin command (two jobs)
#
/bin/echo -e tsh\076 ./myspin1 10 \046
NEXT
./myspin1 10 &
WAIT
NEXT

/bin/echo -e tsh\076 ./mytstps
NEXT
./mytstps
NEXT

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

/bin/echo -e tsh\076 fg %2
NEXT
fg %2
NEXT

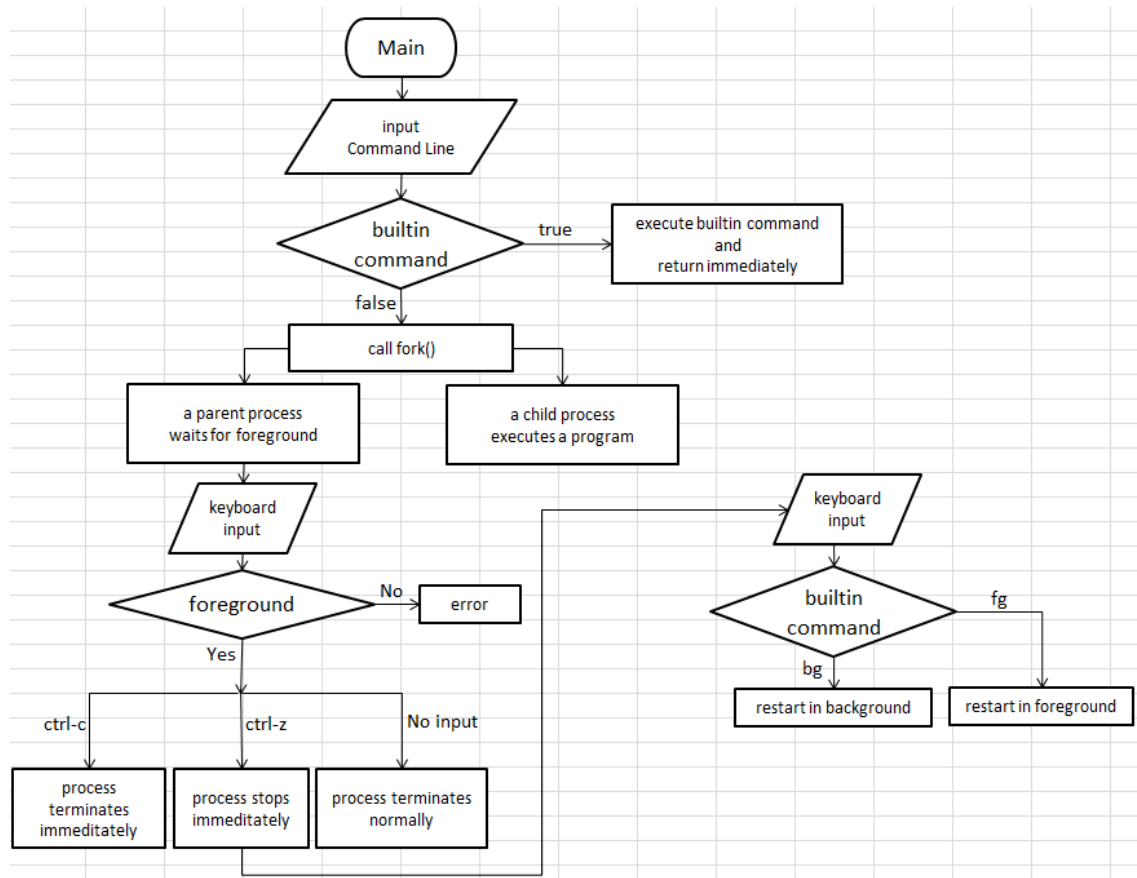
SIGNAL # restart myspin
quit
```

구현한 것은 trace 17과 동일하다. trace18에서는 두 개의 job에 대해 테스트한다. 다수의 프로세스에 대해 제대로 동작하는지 살펴본다.

실행 결과 화면은 다음과 같다.

```
a201102411@eslab:~/tshlab-handout
[a201102411@eslab tshlab-handout]$ ./tsh
eslab_tsh> ./myspin1 10 &
(1) (23635) ./myspin1 10 &
eslab_tsh> ./mytstps
Job [2] (23636) stopped by signal 20
eslab_tsh> jobs
(1) (23635) Running      ./myspin1 10 &
(2) (23636) Stopped     ./mytstps
eslab_tsh> fg %2
eslab_tsh> █
```

trace18
<flowchart>

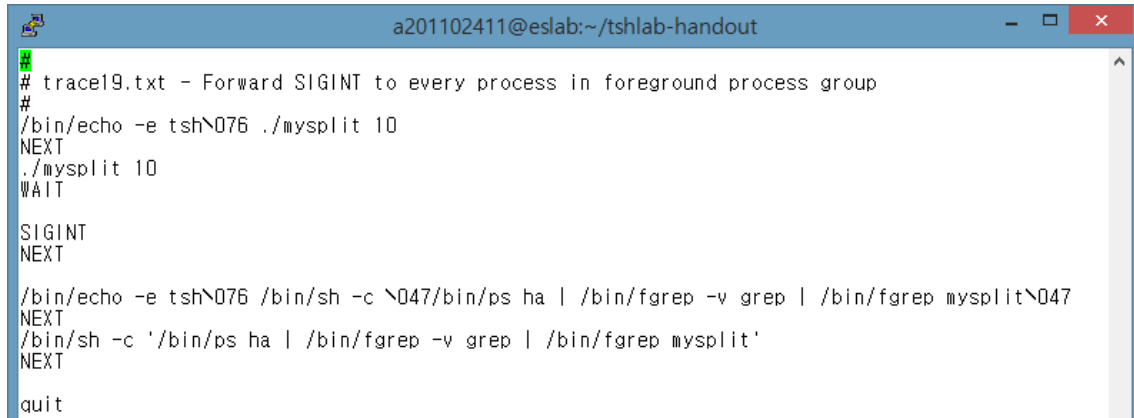


trace18 flowchart

trace19

<해결 사항 및 해결 방법>

모든 프로세스를 foreground 프로세스 그룹으로 설정하여 그룹에 SIGINT signal을 전달하는 것을 구현한다.



```
# trace19.txt - Forward SIGINT to every process in foreground process group
#
/bin/echo -e tsh\076 ./mysplit 10
NEXT
./mysplit 10
WAIT

SIGINT
NEXT

/bin/echo -e tsh\076 /bin/sh -c \047/bin/ps ha | /bin/fgrep -v grep | /bin/fgrep mysplit\047
NEXT
/bin/sh -c '/bin/ps ha | /bin/fgrep -v grep | /bin/fgrep mysplit'
NEXT
quit
```

trace19.txt를 살펴보면 mysplit 함수를 호출한다.

mysplit 함수의 주석을 살펴보면 process group에 시그널을 보내는 것을 테스트하는 함수라는 것을 알 수 있다.

따라서, 프로세스 그룹을 만들고, 이 프로세스 그룹에 시그널을 보내지도록 구현한다.

먼저, 프로세스 그룹 id를 설정해야 한다.

setpgid를 이용해서 foreground의 현재 process id와 동일한 process group id를 가지는 프로세스를 만들고, 프로세스를 추가한다.

```
a201102411@eslab:~/tshlab-handout
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* store command */
    int bg; /* should the job run in bg or fg? */
    pid_t pid; /* Process id */
    sigset_t mask; /* blocking signal */
    bg = parseline(cmdline, argv); /* parse command line */

    if(!builtin_cmd(argv)) {

        /* Blocking SIGCHLD to avoid a Race */
        if(sigemptyset(&mask)<0)
            unix_error("sigemptyset error");
        if(sigaddset(&mask, SIGCHLD))
            unix_error("sigaddset error");
        if(sigaddset(&mask, SIGINT))
            unix_error("sigaddset error");
        if(sigaddset(&mask, SIGTSTP))
            unix_error("sigaddset error");
        if(sigprocmask(SIG_BLOCK, &mask, NULL))
            unix_error("sigprocmask error");
        /* fork() */
        if((pid=fork())<0) { /* forking error */
            unix_error("forking error");
        }
        else if(pid==0) { /* child process */
            if(sigprocmask(SIG_UNBLOCK, &mask, NULL) < 0) /* unblock SIGCHLD */
                unix_error("sigprocmask error");
            if(setpgid(0,0) < 0)
                unix_error("setpgid error");
            if((execve(argv[0], argv, environ)<0))
            {
                printf("%s : Command not found\n",argv[0]);
                exit(0);
            }
        }
        else {
            if(bg) /* add background job to the joblist */
                addjob(jobs, pid, BG, cmdline);
            else /* add foreground job to the joblist */
                addjob(jobs, pid, FG, cmdline);

            if(sigprocmask(SIG_UNBLOCK, &mask, NULL) < 0) /* unblock SIGCHLD */
                unix_error("sigprocmask error");
            /* parent waits for foreground job to terminate */
            if(!bg) { /* foreground */
                waitfg(pid, (int)stdout); /* terminating zombie child */
            }
            else { /* background */
                /* print according to pattern using func pid2jid */
                printf("(%d) (%d) %s", pid2jid(pid), pid, cmdline);
            }
        }
    }
}
```

198,3-24 26%

kill 함수를 통해 시그널을 보낼 때, pid 앞에 마이너스(-)를 붙여 해당 pid의 process group의 process 모두에게 시그널을 보내도록 수정한다.

```
a201102411@eslab:~/tshlab-handout
/*
 * sigint_handler - The kernel sends a SIGINT to the shell whenever the
 * user types ctrl-c at the keyboard. Catch it and send it along
 * to the foreground job.
 */
void sigint_handler(int sig)
{
    pid_t pid; /* process ID */
    pid = fgpid(jobs); /* current process PID */
    kill(-pid, SIGINT); /* sending signal SIGINT to process group */

    return;
}
```


trace19

<관련 지식>

●int setpgid(pid_t pid, pid_t pgid);

프로세스 그룹을 변경할 수 있는 함수

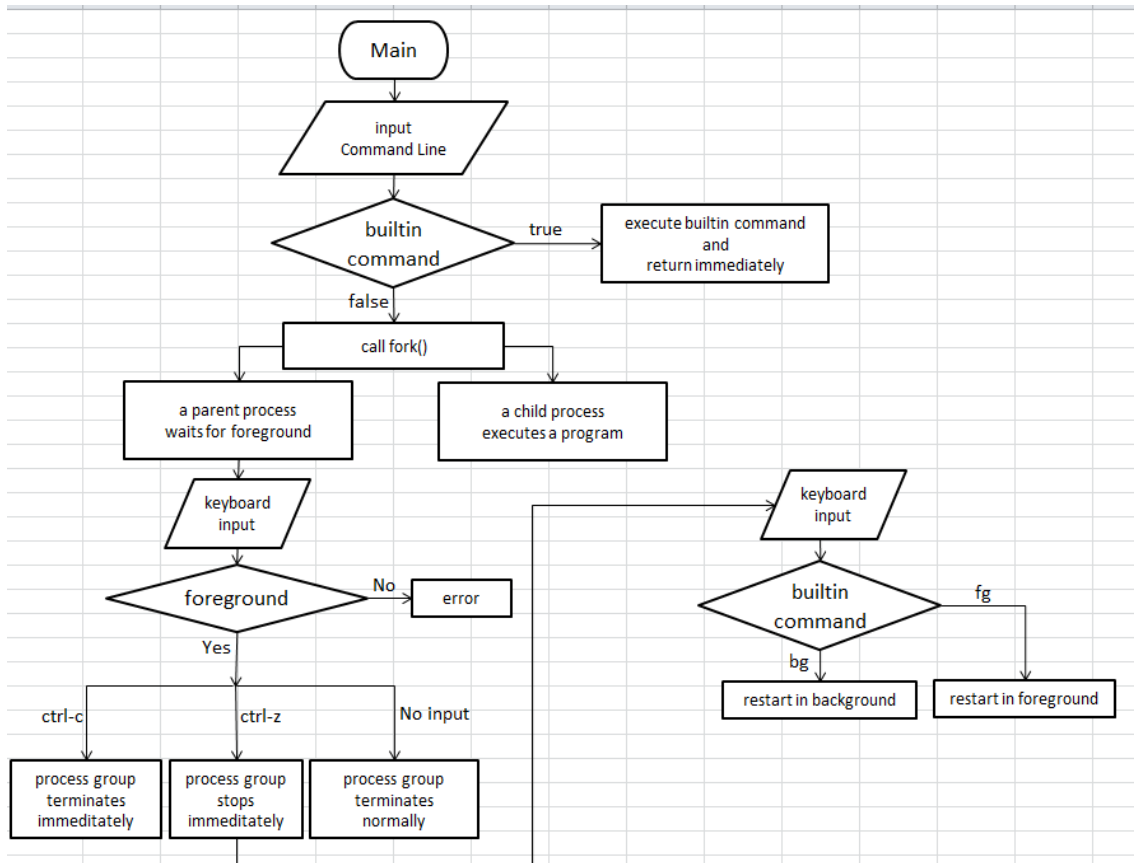
프로세스 그룹 pid를 pgid로 변경하는 함수.

pid가 0일 경우 현재 프로세스 pid가 사용됨.

pgid가 0일 경우 pid로 명시된 프로세스의 pid가 프로세스 그룹 id로 사용

setpgid(0, 0) 인 경우, 프로세스 그룹 id가 호출하는 프로세스 pid로 하는 새 프로세스 그룹 만들고, 프로세스 pid를 새 그룹에 추가한다.

trace19
<flowchart>



trace19 flowchart

trace20

<해결 사항 및 해결 방법>

모든 프로세스를 foreground 프로세스 그룹으로 설정하여 그룹에 SIGTSTP signal을 전달하는 것을 구현한다.

```
a201102411@eslab:~/tshlab-handout
# trace20.txt - Forward SIGTSTP to every process in foreground process group
#
/bin/echo -e tsh\076 ./mysplit 10
NEXT
./mysplit 10
WAIT

SIGTSTP
NEXT

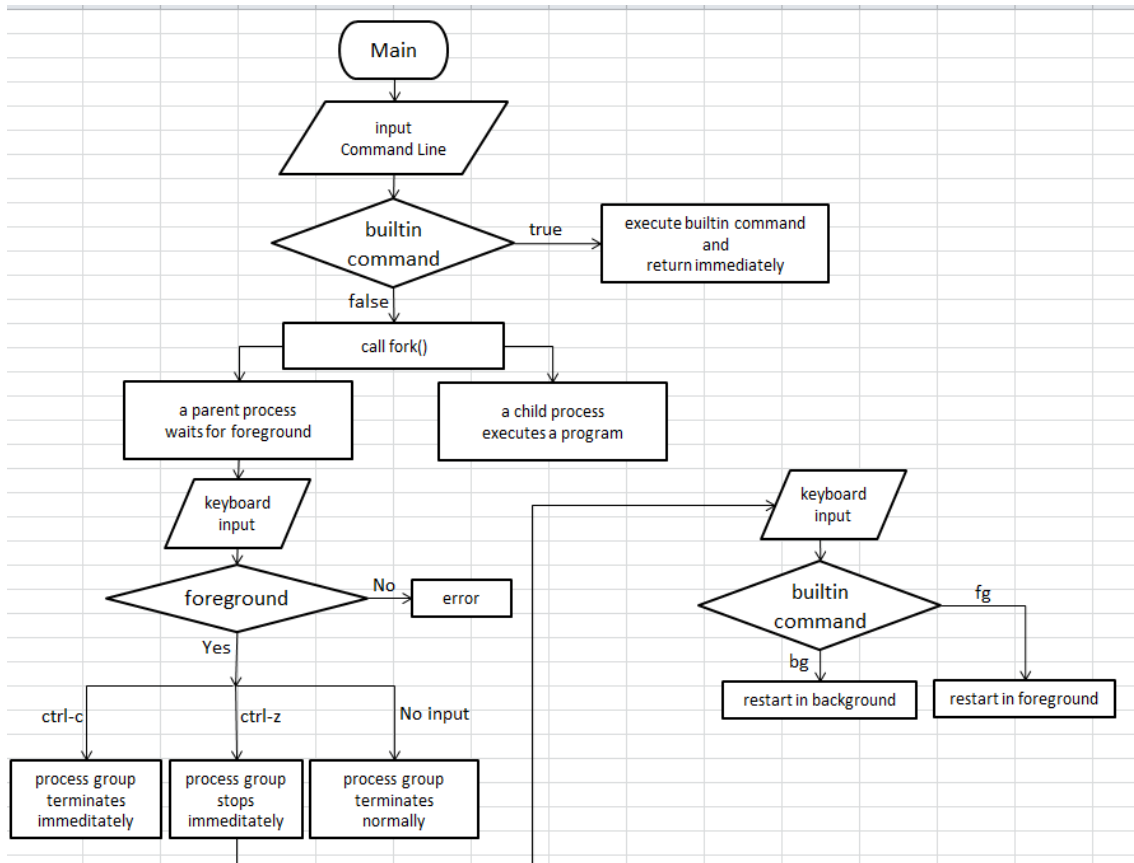
/bin/echo -e tsh\076 /bin/sh -c \047/bin/ps ha | /bin/fgrep -v grep | /bin/fgrep mysplit | /usr
/bin/expand | /usr/bin/colrm 1 15 | /usr/bin/colrm 2 11\047
NEXT
/bin/sh -c '/bin/ps ha | /bin/fgrep -v grep | /bin/fgrep mysplit | /usr/bin/expand | /usr/bin/c
olrm 1 15 | /usr/bin/colrm 2 11'
NEXT
quit
```

trace19와 동일하다. 이미 eval 함수에서 프로세스 그룹을 설정하였다.

sigtstp_handler도 kill함수를 이용하여 시그널을 보낼 때, 프로세스 그룹의 모든 프로세스에게 시그널을 보내기 위해서 pid 앞에 마이너스 부호를 붙인다.

```
a201102411@eslab:~/tshlab-handout
/*
 * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
 * the user types ctrl-z at the keyboard. Catch it and suspend the
 * foreground job by sending it a SIGTSTP.
 */
void sigtstp_handler(int sig)
{
    pid_t pid;
    pid = fgpid(jobs); /* current process pid*/
    kill(-pid, SIGTSTP); /* sending signal to process group */
    return;
}
```

trace20
<flowchart>

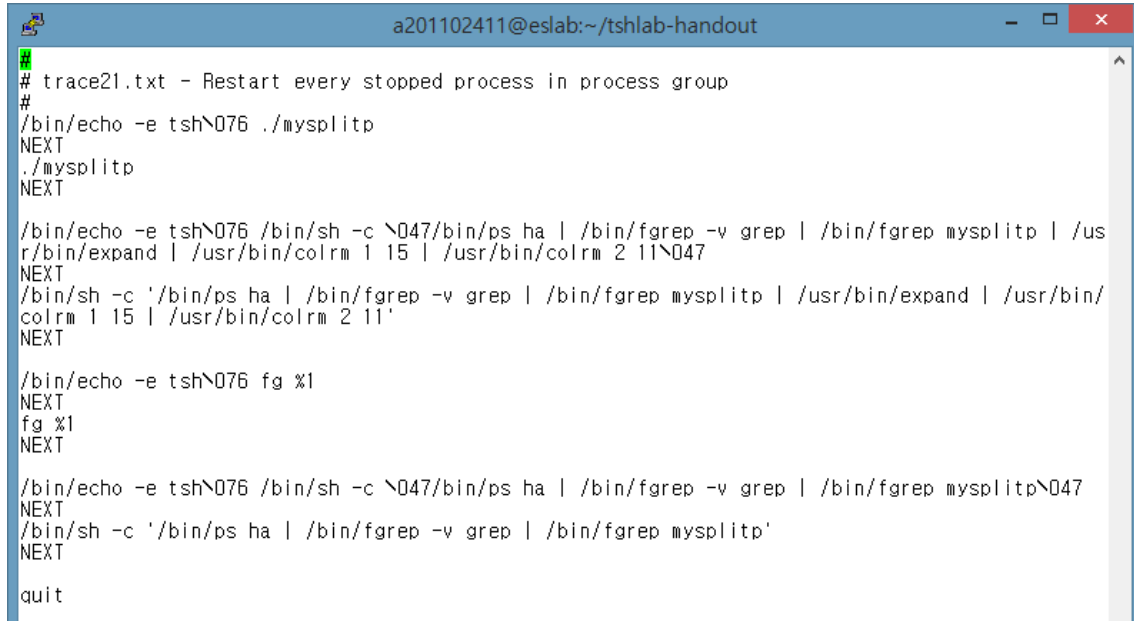


trace20 flowchart

trace21

<해결 사항 및 해결 방법>

같은 프로세스 그룹으로 묶인 정지된 job을 재 시작시킨다.



```
# trace21.txt - Restart every stopped process in process group
#
/bin/echo -e tsh\076 ./mysplitp
NEXT
./mysplitp
NEXT

/bin/echo -e tsh\076 /bin/sh -c \047/bin/ps ha | /bin/fgrep -v grep | /bin/fgrep mysplitp | /usr/bin/expand | /usr/bin/colrm 1 15 | /usr/bin/colrm 2 11\047
NEXT
/bin/sh -c '/bin/ps ha | /bin/fgrep -v grep | /bin/fgrep mysplitp | /usr/bin/expand | /usr/bin/colrm 1 15 | /usr/bin/colrm 2 11'
NEXT

/bin/echo -e tsh\076 fg %1
NEXT
fg %1
NEXT

/bin/echo -e tsh\076 /bin/sh -c \047/bin/ps ha | /bin/fgrep -v grep | /bin/fgrep mysplitp\047
NEXT
/bin/sh -c '/bin/ps ha | /bin/fgrep -v grep | /bin/fgrep mysplitp'
NEXT
quit
```

trace19, 20과 비슷하다.

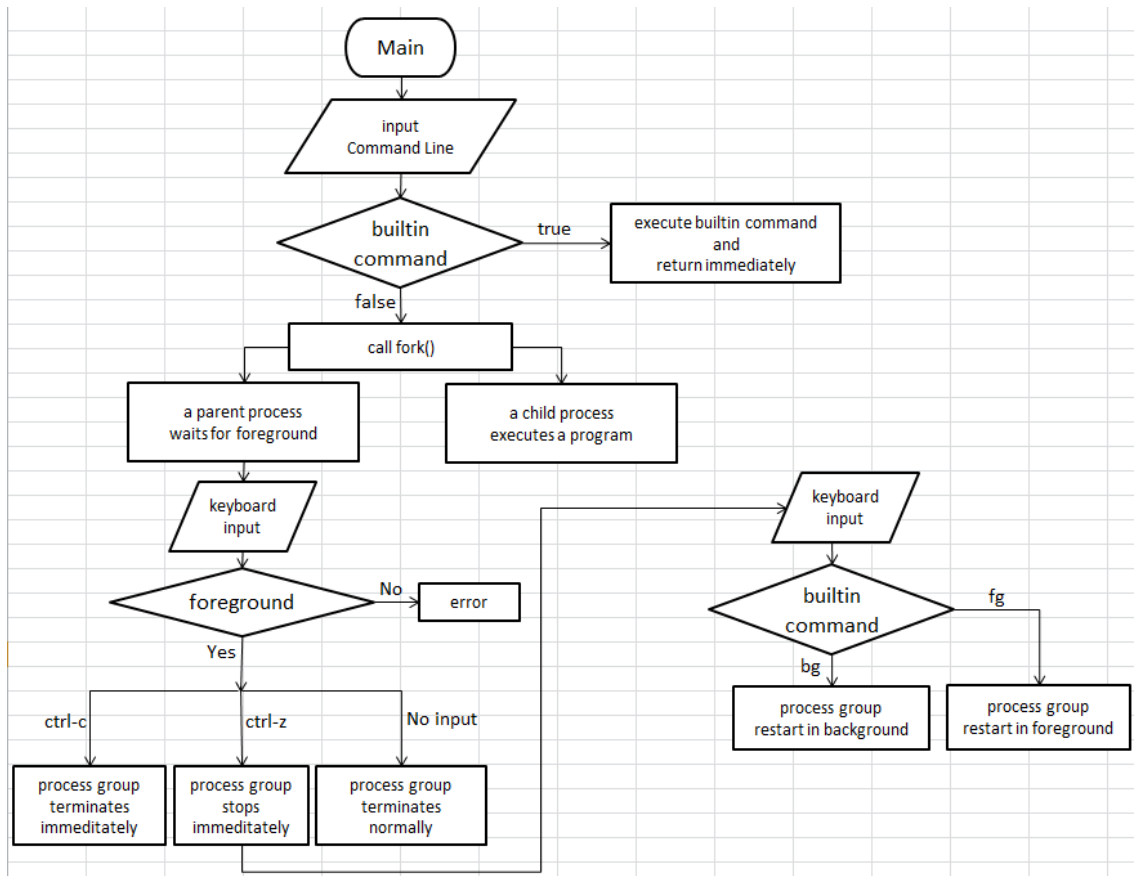
정지된 job을 재시작 시킬 때, 모든 프로세스에 대해 시그널을 보내기 위해서 builtin_cm d함수를 수정한다.

시그널을 보내는 kill 함수에서 job->pid 앞에 마이너스를 붙여, 모든 프로세스 그룹으로 시그널이 보내지도록 구현한다.


```
a201102411@eslab:~/tshlab-handout
*/
int builtin_cmd(char **argv)
{
    pid_t jid; /* job id */
    pid_t pid; /* pocess id */
    struct job_t *job; /* job structure variable for temporary save */

    if (!strcmp(argv[0], "quit")) { /* quit command */
        exit(0);
    }
    else if (!strcmp(argv[0], "jobs")) { /* jobs command */
        listjobs(jobs, STDOUT_FILENO);
        return 1;
    }
    else if (!strcmp(argv[0], "bg")) /* bg command */
    {
        if (argv[1][0] == '%') /* bg %jid command */
        {
            jid = atoi(&argv[1][1]); /* changing char to int */
            job = getjobjid(jobs, jid); /* store jobs of jid */
        }
        else if (isdigit(argv[1][0])) /* bg pid command */
        {
            pid = atoi(argv[1]); /* changing char to int */
            job = getjobpid(jobs, pid); /* store jobs of pid */
        }
        kill(-(job->pid), SIGCONT); /* seding SIGCONT to process group */
        job->state = BG; /* changing ST state to BG state */
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
        return 1;
    }
    else if (!strcmp(argv[0], "fg")) /* fg command */
    {
        if (argv[1][0] == '%') /* fg %jid command */
        {
            jid = atoi(&argv[1][1]);
            job = getjobjid(jobs, jid);
        }
        else if (isdigit(argv[1][0])) /* fg pid command */
        {
            pid = atoi(argv[1]);
            job = getjobpid(jobs, pid);
        }
        kill(-(job->pid), SIGCONT); /* sending SIGCONT to process group */
        job->state = FG; /* change ST state to FG */
        /* not print job information */
        return 1;
    }
    else {
        return 0; /* Not a builtin command */
    }
}
```

trace21
<flowchart>

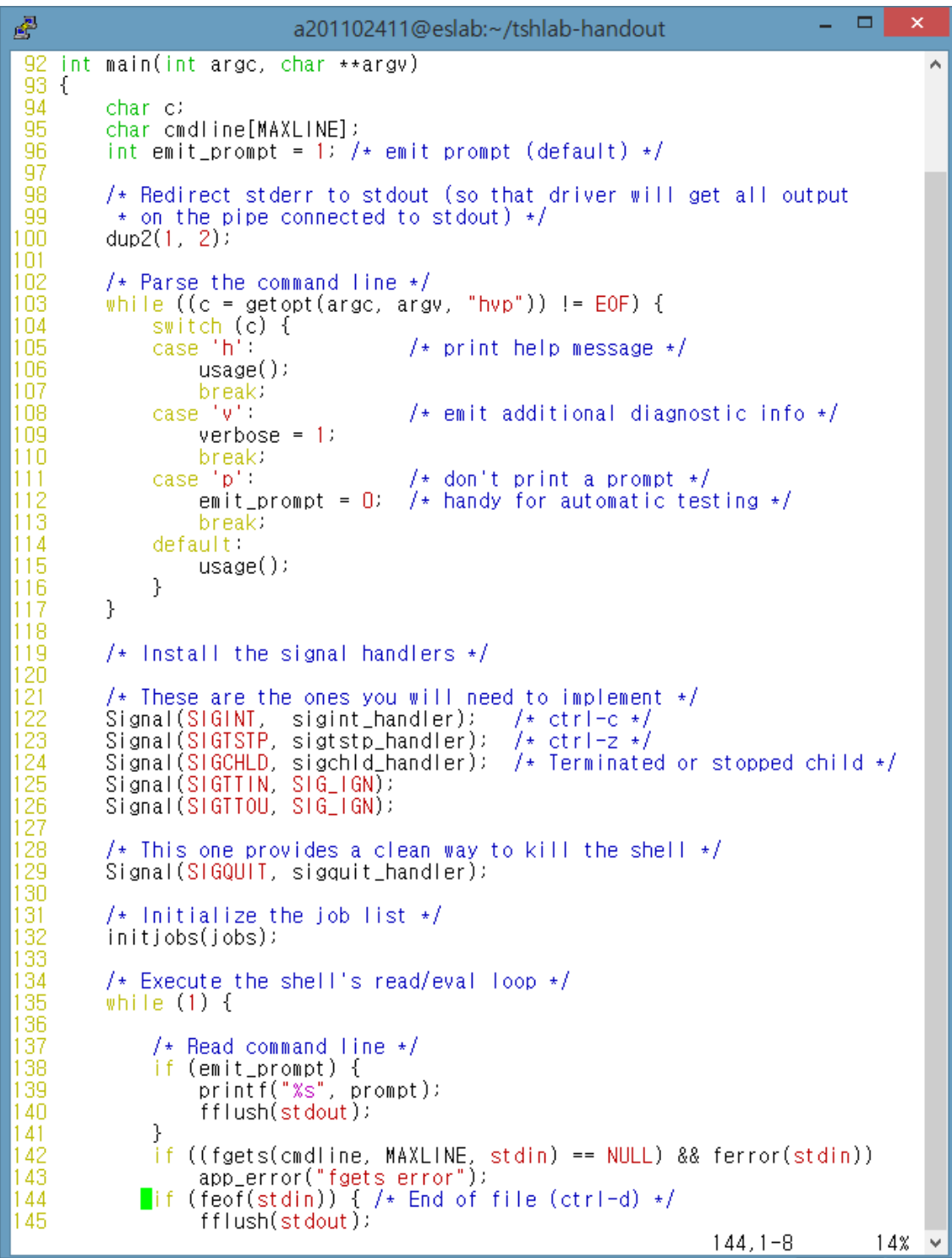


trace21 flowchart

2-(3). 중요 함수 설명

앞의 trace에서 설명한 관련지식 참고.

- int main(int argc, char **argv)



```
92 int main(int argc, char **argv)
93 {
94     char c;
95     char cmdline[MAXLINE];
96     int emit_prompt = 1; /* emit prompt (default) */
97
98     /* Redirect stderr to stdout (so that driver will get all output
99      * on the pipe connected to stdout) */
100     dup2(1, 2);
101
102     /* Parse the command line */
103     while ((c = getopt(argc, argv, "hvp")) != EOF) {
104         switch (c) {
105             case 'h': /* print help message */
106                 usage();
107                 break;
108             case 'v': /* emit additional diagnostic info */
109                 verbose = 1;
110                 break;
111             case 'p': /* don't print a prompt */
112                 emit_prompt = 0; /* handy for automatic testing */
113                 break;
114             default:
115                 usage();
116         }
117     }
118
119     /* Install the signal handlers */
120
121     /* These are the ones you will need to implement */
122     Signal(SIGINT, sigint_handler); /* ctrl-c */
123     Signal(SIGSTP, sigstp_handler); /* ctrl-z */
124     Signal(SIGCHLD, sigchld_handler); /* Terminated or stopped child */
125     Signal(SIGTTIN, SIG_IGN);
126     Signal(SIGTTOU, SIG_IGN);
127
128     /* This one provides a clean way to kill the shell */
129     Signal(SIGQUIT, sigquit_handler);
130
131     /* Initialize the job list */
132     initjobs(jobs);
133
134     /* Execute the shell's read/eval loop */
135     while (1) {
136
137         /* Read command line */
138         if (emit_prompt) {
139             printf("%s", prompt);
140             fflush(stdout);
141         }
142         if ((fgets(cmdline, MAXLINE, stdin) == NULL) && ferror(stdin))
143             app_error("fgets error");
144         if (feof(stdin)) { /* End of file (ctrl-d) */
145             fflush(stdout);
```

```

146         fflush(stderr);
147         exit(0);
148     }
149
150     /* Evaluate the command line */
151     eval(cmdline);
152     fflush(stdout);
153     fflush(stdout);
154 }
155
156 exit(0); /* control never reaches here */
157 }

```

104,1-8 16% ▾

먼저, 메인함수를 살펴본다.

100번 줄에서, dup2 함수가 사용된다. 인자로 1을 가리키는 stdout, 2를 가리키는 stderr를 사용한다. 이 함수는 stdout을 stderr로 복사한다. 모든 출력은 stdout 과 연결된다.

그리고 103번에서 117번의 반복문에서 쉘의 옵션을 처리한다. 여기에서는 h, v, p 각각에 대한 옵션을 처리한다. h는 help 옵션, v는 verbose를 1로 설정하여 명령어 수행 과정을 자세히 보여주고, p는 프롬프트를 출력하지 않는 옵션이다.

122번 줄부터 130번 줄에서는 signal handler를 설치한다. 이때의 signal handler는 랩퍼 함수로 구현된다.

SIGINT, SIGTSTP, SIGCHLD, SIGQUIT 시그널에 따라 각 각의 핸들러가 실행된다.

135번 줄부터의 무한 루프에서 쉘의 명령어를 읽고, 실행한다.

루프 안에서 프롬프트를 출력하고, 명령어 줄을 조건에 맞게 입력하지 않았을 경우, 에러를 표시한다. 그리고 ctrl-d를 입력하면 쉘이 종료되는 조건문이 있다.

150번 줄부터는 입력받은 명령어 줄을 매개변수로 하는 eval 함수를 호출하여 명령어 줄을 계산한다.

main 함수 관련 지식

●int dup2(filedes, int filedes2)

같은 프로세스 내에서 서로 다른 fd가 같은 파일 엔트리를 공유할 수 있도록 한다.

즉, 파일 디스크립터를 복사하는 함수

-반환 값

if ok, 복사된 fd

if fail, -1

-인자

filedes : 복사할 파일 디스크립터

filedes2 : 여기에 입력한 파일 디스크립터로 복사

●파일 디스크립터(File Descriptor)

시스템으로부터 할당 받은 파일, 소켓 등을 대표하는 0이 아닌 정수 값

리눅스 시스템에서 커널은 열린 파일들을 fd를 통해 참조. 파일 별로 번호를 매겨서 그 번호로 파일을 식별한다는 개념.

fd 0번에서 2번까지는 STDIN, STDOUT, STDERR 로 고정되어 있다.

●int getopt(int argc, char *const *argv, const char *shortopts)

프로그램 실행 시 입력 받은 인자를 받아 분석하여 옵션 처리를 가능케 해주는 함수.

- 반환 값

if -1, 더 이상 옵션이 없을 때

if 옵션 문자, 지정되지 않은 옵션일 때

if 그 외, 지정된 옵션일 경우 문자열 반환

- 인자

argc : 인수의 개수

char *const *argv : 인수의 내용

const char *shortopts : 검색하려는 옵션들의 문자열

●verbose

컴퓨터 프로그램 수행 시 특히, 부팅 시 그래픽 화면이 아니라 명령어를 수행하는 과정을 문자로 '자세하게' 보여주는 모드(버보스 모드)

즉, 우리가 사용한 -v옵션.

- void eval(char *cmdline)

eval 함수는 사용자가 입력한 명령어 줄을 계산하는 함수이다.

함수의 주석을 살펴보면, 사용자가 내장 명령어를 입력하면 그것이 즉시 실행되도록 한다. 그런 경우가 아니라면, fork 함수를 통해 자식 프로세스를 생성하고, 자식 프로세스의 con text에서 프로그램을 실행시킨다.

job이 foreground에서 실행된다면, 그것이 종료되거나 반환될 때까지 기다린다.

사용자가 ctrl-c 또는 ctrl-z를 입력하였을 때, background에 있는 자식 프로세스가 커널로부터 SIGINT, SIGTSTP 시그널을 받지 못하도록 하기 위해서, 각 각의 자식프로세스는 자신의 특별한 process group id를 가지고 있어야 한다.

먼저, eval함수는 넘겨받은 명령어 줄을 parseline 함수를 통해 background job인지 아닌지 검사하고, 명령어들을 한 단어씩 나누어서 구성한다.

178번 줄의 조건문에서 builtin_cmd함수를 호출하여 입력받은 명령어가 내장 명령어인지 아닌지를 검사한다. 내장 명령어라면 builtin_cmd함수에서 실행되고, 내장 명령어가 아니면 조건문 안으로 들어간다.

그리고 경주를 피하기 위해 시그널들을 block 시킨다.

시그널들을 block 시킨 후, fork 함수를 이용하여 자식 프로세스를 생성한다. 자식 프로세

스, 부모 프로세스를 조건문으로 나눈 후 해당하는 부분들을 실행한다.

자식프로세스에서는 시그널을 unblock 시킨 후, 현재 프로세스의 pid로 프로세스 그룹 id를 설정한다. 그리고 context 내에서 새로운 프로그램을 실행한다.

부모 프로세스에서는 foreground, background job에 따라 job list 에 jobs을 추가한다. foreground job일 경우 부모 프로세스는 해당 job이 종료 할 때 까지 기다리고, background job일 경우 해당하는 job 내용을 프린트한다.

- int builtin_cmd(char **argv)

```
a201102411@eslab:~/tshlab-handout
263 int builtin_cmd(char **argv)
264 {
265     pid_t jid; /* job id */
266     pid_t pid; /* pocess id */
267     struct job_t *job; /* job structure variable for temporary save */
268
269     if (!strcmp(argv[0], "quit")) { /* quit command */
270         exit(0);
271     }
272     else if (!strcmp(argv[0], "jobs")) { /* jobs command */
273         listjobs(jobs, STDOUT_FILENO);
274         return 1;
275     }
276     else if (!strcmp(argv[0], "bg")) /* bg command */
277     {
278         if(argv[1][0] == '%') /* bg %jid command */
279         {
280             jid = atoi(&argv[1][1]); /* changing char to int */
281             job = getjobjid(jobs, jid); /* store jobs of jid */
282         }
283         else if(isdigit(argv[1][0])) /* bg pid command */
284         {
285             pid = atoi(argv[1]); /* changing char to int */
286             job = getjobpid(jobs, pid); /* store jobs of pid */
287         }
288         kill(-(job->pid), SIGCONT); /* seding SIGCONT to process group */
289         job->state = BG; /* changing ST state to BG state */
290         printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
291         return 1;
292     }
293     else if (!strcmp(argv[0], "fg")) /* fg command */
294     {
295         if(argv[1][0] == '%') /* fg %jid command */
296         {
297             jid = atoi(&argv[1][1]);
298             job = getjobjid(jobs, jid);
299         }
300         else if(isdigit(argv[1][0])) /* fg pid command*/
301         {
302             pid = atoi(argv[1]);
303             job = getjobpid(jobs, pid);
304         }
305         kill(-(job->pid), SIGCONT); /* sending SIGCONT to process group */
306         job->state = FG; /* change ST state to FG */
307         waitfg(job->pid, 0); /* parent waits for foreground to terminate */
308         return 1;
309     }
310     else {
311         return 0; /* Not a builtin command */
312     }
313 }
```

builtin_cmd함수는 사용자가 내장 명령어를 입력하면, 그것을 즉시 실행하고 1을 반환한다. 내장 명령어가 아닐 경우 0을 반환한다.

quit, job, bg, fg 내장 명령어를 구현한다. 해당하는 명령어가 들어오는지 strcmp 함수를 이용하여 비교한다. 그리고, 각각의 조건문을 구현한다.

quit 명령어는 셸을 종료시킨다. 이는 exit을 이용하여 즉시 종료하도록 구현한다.

job 명령어는 job list를 출력한다. listjobs 함수를 이용하여 job의 list를 출력하고 1을 반환한다.

bg 명령어는 정지된 프로세스를 background에서 다시 실행시킨다. bg 명령어 뒤에 % job id를 입력할 수 있고, 바로 pid를 입력할 수도 있다. 정지된 job을 찾아서 background에서 다시 실행시킨다.

fg 명령어는 정지된 job을 foreground에서 다시 실행시키는 명령어이다.

- void sigchld_handler(int sig)

```
a201102411@eslab:~/tshlab-handout
320 * sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
321 * a child job terminates (becomes a zombie), or stops because it
322 * received a SIGSTOP or SIGTSTP signal. The handler reaps all
323 * available zombie children, but doesn't wait for any other
324 * currently running children to terminate.
325 */
326 void sigchld_handler(int sig)
327 {
328     int status; /* status variable */
329     pid_t pid; /* pid */
330
331     /* waiting child process terminated */
332     /* waitpid returns terminated process ID */
333     while((pid = waitpid(-1, &status, WUNTRACED|WNOHANG))>0)
334     {
335         struct job_t *j = getjobpid(jobs, pid); /*find job having pid in joblist */
336         if(WIFSIGNALED(status)) /* child process has terminated by SIGNAL */
337         {
338             /* print process information */
339             /* WTERMSIG returns signal number causing termination */
340             /* WSTOPSIG returns signal number causing stopped */
341             fprintf(stdout, "Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(status));
342             deletejob(jobs, pid); /* removing job */
343         }
344         else if(WIFSTOPPED(status)) /* child process has stopped */
345         {
346             j->state = ST; /* job state is ST(stop) */
347             /* WSTOPSIG returns signal number causing stopped */
348             fprintf(stdout, "Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(status));
349         }
350         else if(WIFEXITED(status)) /* process normally terminates */
351         {
352             deletejob(jobs, pid); /* removing job */
353         }
354     }
355     return;
356 }
```

SIGINT 시그널에 의해 프로세스가 종료되었거나, SIGTSTP 시그널에 의해 프로세스가 정지되었을 경우, SIGCHLD 시그널이 보내지고 sigchld_handler가 실행된다.

sigchld_handler에서는 waitpid 함수의 옵션 WNOHANG|WUNTRACED를 사용하여 모든 자식 프로세스들이 정지하거나 종료하기를 기다린다.

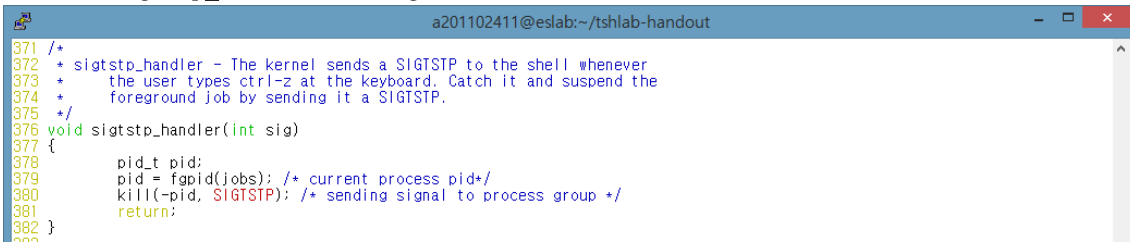
그리고 반복문 안의 조건문에서, 프로세스가 시그널에 의해 종료되었는지, 프로세스가 정지되었는지, 정상적으로 종료되었는지를 조사한다. 각각에 맞춰서 해당하는 내용들을 구현한다.

- void sigint_handler(int sig)

```
a201102411@eslab:~/tshlab-handout
357 /*
358 * sigint_handler - The kernel sends a SIGINT to the shell whenever the
359 * user types ctrl-c at the keyboard. Catch it and send it along
360 * to the foreground job.
361 */
362 void sigint_handler(int sig)
363 {
364     pid_t pid; /* process ID */
365     pid = getpid(jobs); /* current process PID */
366     kill(-pid, SIGINT); /* sending signal SIGINT to process group */
367
368     return;
369 }
```

사용자가 ctrl-c를 입력할 때마다, SIGINT 시그널을 커널에서 보내고, sigint_handler가 실행된다. sigint_handler는 시그널을 잡고, 그것을 foreground job 프로세스 그룹 모두에게 SIGINT 시그널을 보낸다.

- void sigtstp_handler(int sig)



```
a201102411@eslab:~/tshlab-handout
371 /*
372  * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
373  * the user types ctrl-z at the keyboard. Catch it and suspend the
374  * foreground job by sending it a SIGTSTP.
375  */
376 void sigtstp_handler(int sig)
377 {
378     pid_t pid;
379     pid = fgpid(jobs); /* current process pid*/
380     kill(-pid, SIGTSTP); /* sending signal to process group */
381     return;
382 }
```

사용자가 ctrl-z를 입력할 때마다, 커널에서 SIGTSTP 시그널을 보낸다. 그리고 시그널을 sigtstp_handler가 처리한다. 핸들러는 시그널을 잡고, foreground job의 프로세스 그룹내의 모든 프로세스에게 SIGTSTP 시그널을 보낸다.

- 그 외의 함수들

`handler_t *Signal(int signum, handler_t *handler)`

sigaction의 래퍼 함수. portable 시그널 핸들링을 제공한다.

`int parseline(const char *cmdline, char **argv)`

명령어 줄을 분석하고, argv 배열을 만드는 함수이다.

한 단어는 한 개의 인자로 받아들여진다.

사용자가 bg job을 요구하면 true를 반환하고, fg job을 요구하면 false를 반환한다.

`void sigquit_handler(int sig)`

sigquit의 핸들러. 시그널을 보내 셸을 종료한다.

`int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline)`

job을 job list에 추가하는 함수

`int deletejob(struct job_t *jobs, pid_t pid)`

pid의 job을 job list에서 삭제하는 함수

`pid_t fgpjob(struct job_t *jobs)`

현재 foreground job의 process id를 반환한다. 해당하는 job이 없으면 0을 반환

`struct job_t *getjobpid(struct job_t *jobs, pid_t pid)`

pid를 이용하여 job list에서 해당하는 job을 찾는다.

`struct job_t *getjobjid(struct job_t *jobs, int jid)`

jid를 이용하여 job list에서 해당하는 job을 찾는다.

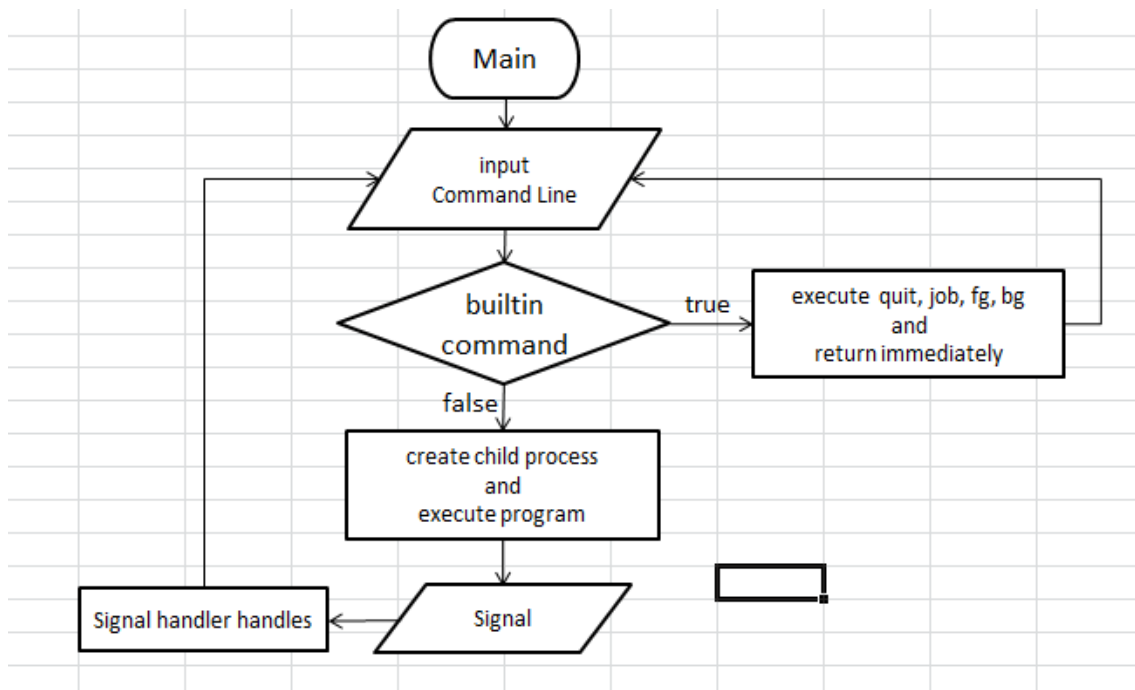
`int pid2jid(pid_t pid)`

process id를 job id로 mapping 한다.

`void listjobs(struct job_t *jobs, int output_fd)`

job list를 출력한다.

- tsh.c 파일 동작 설명과 flowchart(간단히)



tsh.c의 간단한 flowchart

flowchart를 이용하여 tsh.c의 동작을 설명한다.

먼저 메인 함수에서 명령어 줄을 입력받는다. 입력받아진 명령어 줄을 파싱된다. 그리고 어떤 명령어가 입력되었는지 분석한다.

내장 명령어가 아니면 자식 프로세스를 생성하여 프로그램을 실행한다.

내장 명령어이면 quit, job, fg, bg 중 알맞은 동작을 한다.

foreground 작업 중 시그널이 보내어지면 signal handler가 시그널을 처리한다.

이러한 과정을 통해 tsh.c는 반복되어진다.

3. 고찰 및 실행결과 분석

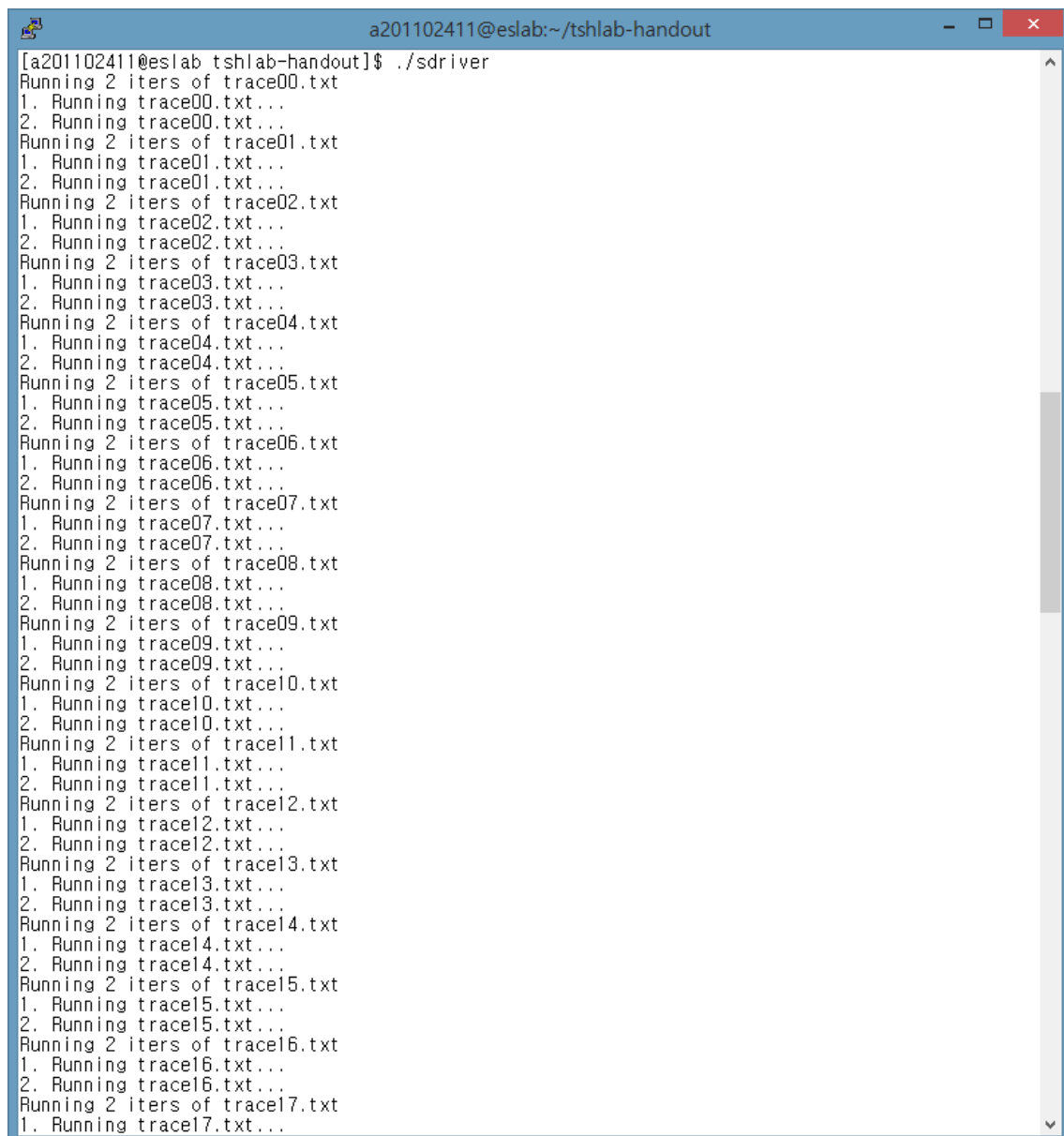
간단한 동작을 수행하는 tiny shell을 구현하였다.

각 trace를 단계 별로 구현하면서, 간단한 unix shell 프로그램이 어떻게 동작하는지 알게 되었다.

최대 한 개의 foreground process가 수행되며, 다수의 background process가 수행된다.

셸에서 프로세스로 signal을 보내면 시그널을 받은 프로세스는 kernel을 통해 signal handler를 호출한다. 그리고 signal handler는 해당하는 시그널을 처리한다.

실행결과를 분석하면 각 trace별로 모두 matched 되었음을 알 수 있다.



```
a201102411@eslab:~/tshlab-handout
[a201102411@eslab tshlab-handout]$ ./sdriver
Running 2 iters of trace00.txt
1. Running trace00.txt...
2. Running trace00.txt...
Running 2 iters of trace01.txt
1. Running trace01.txt...
2. Running trace01.txt...
Running 2 iters of trace02.txt
1. Running trace02.txt...
2. Running trace02.txt...
Running 2 iters of trace03.txt
1. Running trace03.txt...
2. Running trace03.txt...
Running 2 iters of trace04.txt
1. Running trace04.txt...
2. Running trace04.txt...
Running 2 iters of trace05.txt
1. Running trace05.txt...
2. Running trace05.txt...
Running 2 iters of trace06.txt
1. Running trace06.txt...
2. Running trace06.txt...
Running 2 iters of trace07.txt
1. Running trace07.txt...
2. Running trace07.txt...
Running 2 iters of trace08.txt
1. Running trace08.txt...
2. Running trace08.txt...
Running 2 iters of trace09.txt
1. Running trace09.txt...
2. Running trace09.txt...
Running 2 iters of trace10.txt
1. Running trace10.txt...
2. Running trace10.txt...
Running 2 iters of trace11.txt
1. Running trace11.txt...
2. Running trace11.txt...
Running 2 iters of trace12.txt
1. Running trace12.txt...
2. Running trace12.txt...
Running 2 iters of trace13.txt
1. Running trace13.txt...
2. Running trace13.txt...
Running 2 iters of trace14.txt
1. Running trace14.txt...
2. Running trace14.txt...
Running 2 iters of trace15.txt
1. Running trace15.txt...
2. Running trace15.txt...
Running 2 iters of trace16.txt
1. Running trace16.txt...
2. Running trace16.txt...
Running 2 iters of trace17.txt
1. Running trace17.txt...
```

```
a201102411@eslab:~/tshlab-handout
2. Running trace17.txt...
Running 2 iters of trace18.txt
1. Running trace18.txt...
2. Running trace18.txt...
Running 2 iters of trace19.txt
1. Running trace19.txt...
2. Running trace19.txt...
Running 2 iters of trace20.txt
1. Running trace20.txt...
2. Running trace20.txt...
Running 2 iters of trace21.txt
1. Running trace21.txt...
2. Running trace21.txt...
Running 2 iters of trace22.txt
1. Running trace22.txt...
2. Running trace22.txt...
Running 2 iters of trace23.txt
1. Running trace23.txt...
2. Running trace23.txt...
Running 2 iters of trace24.txt
1. Running trace24.txt...
2. Running trace24.txt...

Summary: 25/25 correct traces
[a201102411@eslab tshlab-handout]$
```

```
a201102411@eslab:~/tshlab-handout
Running trace00.txt...
Success: The test and reference outputs for trace00.txt matched!
Running trace01.txt...
Success: The test and reference outputs for trace01.txt matched!
Running trace02.txt...
Success: The test and reference outputs for trace02.txt matched!
Running trace03.txt...
Success: The test and reference outputs for trace03.txt matched!
Running trace04.txt...
Success: The test and reference outputs for trace04.txt matched!
Running trace05.txt...
Success: The test and reference outputs for trace05.txt matched!
Running trace06.txt...
Success: The test and reference outputs for trace06.txt matched!
Running trace07.txt...
Success: The test and reference outputs for trace07.txt matched!
Running trace08.txt...
Success: The test and reference outputs for trace08.txt matched!
Running trace09.txt...
Success: The test and reference outputs for trace09.txt matched!
Running trace10.txt...
Success: The test and reference outputs for trace10.txt matched!
Running trace11.txt...
Success: The test and reference outputs for trace11.txt matched!
Running trace12.txt...
Success: The test and reference outputs for trace12.txt matched!
Running trace13.txt...
Success: The test and reference outputs for trace13.txt matched!
Running trace14.txt...
Success: The test and reference outputs for trace14.txt matched!
Running trace15.txt...
Success: The test and reference outputs for trace15.txt matched!
Running trace16.txt...
Success: The test and reference outputs for trace16.txt matched!
Running trace17.txt...
Success: The test and reference outputs for trace17.txt matched!
Running trace18.txt...
Success: The test and reference outputs for trace18.txt matched!
Running trace19.txt...
Success: The test and reference outputs for trace19.txt matched!
Running trace20.txt...
Success: The test and reference outputs for trace20.txt matched!
Running trace21.txt...
Success: The test and reference outputs for trace21.txt matched!
~
~
~
~
~
~
~
"00_tshlab_201102411_result.txt" 44L, 1937C 44,1 모두
```