2021 Fall System Programming

Shell Lab 3

2021. 11. 15

유주원
clearlyhunch@gmail.com

Embedded System Lab.
Dept. of Computer Science & Engineering
Chungnam National University

실습 소개

- 과목 홈페이지
  - 충남대학교 사이버캠퍼스 (http://e-learn.cnu.ac.kr/)
- 연락처
  - 유주원
  - 공대 5호관 533호 임베디드 시스템 연구실
  - clearlyhunch@gmail.com
    - Email 제목은 **'[시스템프로그래밍][01][학번_이름]용건'** 으로 시작하도록 작성

개요

- 실습 명
  - Shell Lab - 3
- 목표
  - 작업 관리를 지원하는 간단한 Unix Shell 프로그램 구현과 이를 통한 프로세스의 제어와 시그널링(Signalling)의 개념 이해
- 과제 진행
  - 수업시간에 배운 유닉스 지식을 활용하여 구현한다.
- 구현사항
  - 기본적인 유닉스 쉘의 구현
  - 작업 관리(foreground / background)기능 및 쉘 명령어 구현

Shell Lab - 3

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

- Shell Lab은 trace08과 trace11의 Trace를 모두 수행할 수 있도록 구현되어야 한다.
  - 테스트 프로그램 'sdriver' 를 통해 테스트할 수 있다.

- 총 3주에 걸쳐 진행되며, 각 주 별 수행 사항은 아래와 같다.
  - Shell Lab 1주차 : trace 00, 01, 02
  - Shell Lab 2주차 : trace 03, 04, 05, (06), 07
  - Shell Lab 3주차 : trace 08, 11

CNU Embedded System Laboratory

Shell Lab – Trace 목록 (1주차)

- sdriver를 이용하여 Trace{00-02}를 테스트 할 수 있다.
  - 각 trace의 자세한 내용은 해당 파일을 열어서 확인할 수 있다.

Trace	설 명
trace00	EOF(End-Of-File)가 입력되면 종료
trace01	Built-in 명령어 'quit' 구현
trace02	Foreground 작업 형태로 프로그램 실행

Shell Lab – Trace 목록 (2주차)

- sdriver를 이용하여 Trace{03-07}를 테스트 할 수 있다.
  - 각 trace의 자세한 내용은 해당 파일을 열어서 확인할 수 있다.

Trace	설 명
trace03	Foreground 작업 형태로 매개변수가 없는 프로그램 실행
trace04	Foreground 작업 형태로 매개변수가 있는 프로그램 실행
trace05	Background 작업 형태로 프로그램 실행
trace06	동시에 foreground 작업 형태와 background 작업 형태로 프로그램을 실행
trace07	Built-in 명령어 'jobs'구현

CNU Embedded System Laboratory

Shell Lab – Trace 목록 (3주차)

- sdriver를 이용하여 Trace{08,12}를 테스트 할 수 있다.
  - 각 trace의 자세한 내용은 해당 파일을 열어서 확인할 수 있다.

Trace	설 명
trace08	SIGINT 발생 시, foreground 작업 종료
trace11	자식 프로세스가 스스로에게 SIGINT 전송

CNU Embedded System Laboratory

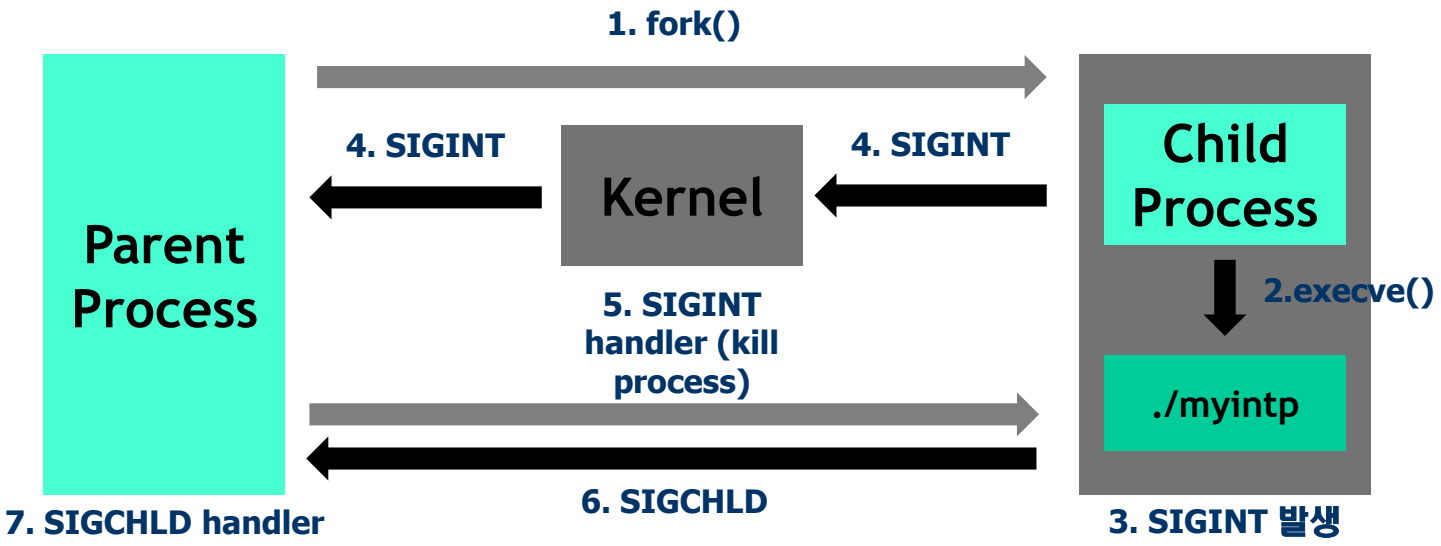
Signal의 개념

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

CNU Embedded System Laboratory

Signal의 처리 과정

- SIGINT 처리 과정 (trace 08)
  - fork( )와 execve( ) 통해 프로그램 실행
  - SIGINT 발생(ctrl+c)
  - kernel을 통해 자식 프로세스에서 부모 프로세스로 SIGINT 전달
  - SIGINT 핸들러를 통해 SIGINT 처리(자식 프로세스 Kill)
  - 자식 프로세스가 종료되면 SIGCHLD 시그널 발생
  - SIGCHLD 핸들러를 통해 자식 프로세스 최종 종료 처리



CNU Embedded System Laboratory

Signal 목록

- 각 Signal에 대한 고유 번호가 있다.
  - 아래 그림을 참조

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		

CNU Embedded System Laboratory

Signal blocking과 unblocking

- Application은 sigprocmask 함수를 이용하여 선택된 signal들을 blocking 하거나 unblocking 할 수 있다.
- Signal Block
  - 시그널을 block 하지 않으면 부모프로세스가 addjob을 수행하기도 전에, 시그널에 의해 자식프로세스가 종료되어 버릴 수 있다.
  - 그렇게 되면 존재하지도 않는 job을 list에 추가하는 사태가 발생한다.
  - 또한 시그널 핸들러의 처리과정에서 존재하지도 않는 job을 종료하고, list에서 제거하는 작업을 해야 한다.
  - 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");
}
```

개념 설명용

CNU Embedded System Laboratory

Signal blocking과 unblocking

```
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);
}
```

개념 설명용

sigprocmask

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

- int sigprocmask(int how, const sigset_t *set, sigset_t *oldest)
  - 이 함수는 signal.h에 선언되어 있음
  - 반환 값 : 성공 시 0, 실패 시 -1
  - 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

- 특정 시그널에 대해 Signal mask가 set 되었다면 해당 시그널을 처리하는 handler를 등록한다.

CNU Embedded System Laboratory

sigemptyset & sigaddset

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

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

CNU Embedded System Laboratory

Race Condition

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

```
addjob(jobs, pid, (bg == 1 ? BG : FG ), cmdline);
sigprocmask(SIG_UNBLOCK, &mask, NULL);
if(!bg){//foreground job
    while(1){
        if( pid != fgpid(jobs))
            break;
        else
            sleep(1);
    }
}
else{//background job
    printf("(%d) (%d) %s",pid2jid(pid),pid, cmdline);//print current job
}//pid2jid is mapping pid to job pid. it's just job's number.
```

```
pid_t fgpid(struct job_t *jobs) {
    int i;

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

CNU Embedded System Laboratory

Shell Lab - trace08

```
#
# 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
```

- 구현 기능 : SIGINT 발생 시, Foreground 작업 종료
  - SIGINT(키보드의 'ctrl+c' 입력)가 입력 되면 foreground 작업을 kill 한다.
    - 이전 슬라이드의 시그널 흐름을 이해하는 것이 중요.
  - 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.
- HINT
  - waitpid( )함수를 통해, 부모 프로세스는 자식 프로세스가 종료되기까지 기다리고, 자식이 종료되면 처리해준다.
  - 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 – trace08

❖ tsh.c 파일 내부 구조

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

CNU Embedded System Laboratory

Shell Lab - trace08

❖ 구현 방법

- 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 */
```

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

```
void sigchld_handler(int sig)
{
    return;
}
```

CNU Embedded System Laboratory

Shell Lab - trace08

- set 관련 지원함수
  - **sigemptyset - 모든 시그널이 비어 있는 집합 생성**
  - **sigfillset - 모든 시그널 번호를 1로 설정**
  - **sigaddset - 특정 시그널 번호를 1로 설정**
  - **sigdelset - 특정 시그널 번호를 0으로 설정**

❖ 구현 방법

- 오른쪽 코드는 관련된 예시 코드입니다. 수정해도 되고 수정해야 할 수도 있습니다.
- 빈칸을 채우는 것 코드는 이론 자료인 sysp-11-signal-st의 여러 페이지 참고하세요!!!!
- 책에도 나와있습니다!!

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    pid_t pid;
    int bg;
    sigset_t mask;

    bg = parseline(cmdline, argv);

    [                                ]시그널 셋 초기화
    [                                ]; //SIGCHLD 시그널을 시그널 셋에 추가
    [                                ]); //SIGINT 시그널을 시그널 셋에 추가
    [                                ]  //SIGTSTP 시그널을 시그널 셋에 추가

    [                                ]; //시그널 블록

    if(!builtin_cmd(argv)) {
        if( (pid=fork()) == 0) {
            [                                ]; //시그널 언블록
            if( execve(argv[0], argv, environ)< 0) {
                printf("%s : Command not found\n", argv);
                exit(0);
            }
        }

        addjob(jobs, pid,(bg ==1?BG: FG), cmdline);
        [                                ]; //시그널 언블록
        if(!bg){
            //부모프로세서가 자식프로세서의 종료를 대기 후 처리
            while(1){
                if( pid != fgpid(jobs))
                    break;
                else
                    sleep(1);
            }
        else {
            //백그라운드 작업 수행시 작업 내용 출력
            printf("(%d) (%d) %s", pid2jid(pid), pid, cmdline);
        }
    }

    return;
}
```

CNU Embedded System Laboratory

Shell Lab - trace08

❖ 구현 방법

- 오른쪽 코드는 관련된 예시 코드입니다. 수정해도 되고 수정해야 할 수도 있습니다.

```
void sigchld_handler(int sig)
{
    int child_status =0;
    pid_t pid;

    while(                                              ){ //자식프로세스 종료를 대기
        if(                                  ){ //자식프로세스 종료를 체크
            printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(child_status));
                                 //자식프로세스를 목록에서 제거
        }
        else {
            deletejob(jobs, pid);
        }
    }
    return;
}


void sigint_handler(int sig)
{
                                  //포어그라운드 프로세스의 pid 저장
     if(pid !=0){
                                 //프로세스에 시그널 전달
     }
     return;
}
```

CNU Embedded System Laboratory

Shell Lab - trace08

결과 확인

- 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!
```

HINT

- 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 지정한 pid를 가지는 프로세스에만 시그널을 전달
  - pid가 0보다 작으면, pid의 절대값 프로세스 그룹에 속하는 모든 프로세스에 시그널을 전달
  - 참조 : http://i1004me2.blog.me/140190173943

Shell Lab - trace11

trace11

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

주의 사항

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

  - 위와 같이 tsh가 좀비가 되어 남아있는 것을 볼 수 있다. 따라서 해당 좀비 프로세스를 제거해야 한다.
    - kill -9 <PID>
    - ex) kill -9 29895
  - 좀비 프로세스를 많이 만들면 감점! (쉘 테스트 후 실시간 체크 바람)

CNU Embedded System Laboratory

주의 사항(중요!!)

- kill system call은 kill 명령어와 다름!!!
  - Kill()은 시그널(SIGINT, SIGTSTP 등)를 프로세스에게 시그널 전달하는 함수.
- Ctrl+C는 eslab_tsh> 에서는 작용하지 않아야 정상.
  - Eslab_tsh> ./myspin1 후 ctrl+c 입력하는 경우
  - ./myintp는 실행 후 ctrl+c까지 자동을 수행해주는 프로세스

CNU Embedded System Laboratory

과제

1. Shell Lab
   I. trace 00 ~ 08, 11 에 대한 코드 작성

2. Shell Lab 보고서
   I. 각 trace 별, tshref를 수행한 결과와 본인이 구현한 tsh와의 동작 일치를 증명
      1) sdriver 수행 결과와 tsh에서의 정상작동 모습(./sdriver -V -t XX -s ./tsh)
   II. 각 trace 별, 플로우 차트
      1) 간단한 수행과정을 플로우차트로 나타내면 됨.
   III. 각 trace 별, 해결 방법에 대한 설명

3. 제출
   I. shlab-handout 디렉토리를 통째로 압축
      1) 파일명: [sys01]HW3_학번.tar.gz
   II. 결과 보고서를 작성
      1) 파일명: [sys01]HW3_학번.pdf
   III. I.과 II. 두개를 하나로 압축
      1) 파일명:[sys01]HW3_학번_이름.zip

CNU Embedded System Laboratory

제출 사항

- 사이버캠퍼스에 제출
  - 자세한 양식은 앞장 슬라이드 참고.
  - 파일 제목: [sys01]HW3_학번_이름.zip

- 제출일자
  - 사이버 캠퍼스: 2021년 11월 22일 월요일 10시 59분 59초까지

CNU Embedded System Laboratory

참고 - Shell Lab - trace09

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

CNU Embedded System Laboratory

참고 - Shell Lab - trace09

- 구현 방법
  - SIGTSTP 시그널을 받아서 joblist 에서 해당 job의 상태를 stop으로 변경 해주어야 한다.
  - trace08의 구현 방법과 마찬가지로,
  - sigchld_handler(), sigtstp_handler() 함수의 내용을 구현해야 한다.
  - sigchld_handler() 함수 내부에서 joblist를 체크해 해당 job의 상태를 stop으로 변경하고 정지시켜야 한다.

- 결과 확인
  - trace09를 sdriver를 수행하여 결과를 확인한다.
    - 명령어 : ./sdriver -t 09 -s ./tsh -V

- HINT
  - trace09도 trace08번과 동일하게 kill( )함수를 사용한다.

CNU Embedded System Laboratory

참고 - Shell Lab - trace10

- 구현 기능 : Background 작업 정상 종료 처리
  - Background 작업이 정상 종료되면 SIGCHLD가 발생하는데, 이 시그널을 받고 Background 작업을 종료 처리해준다.
    - 이때, 핸들러에서는 종료된 자식 프로세스가 사용했던 자원을 반환 시켜주어야 한다.
  - 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
```

참고 - Shell Lab - trace10

구현 방법

- sigchld_handler() 함수의 내부를 구현
- 자식 프로세스가 정상 종료되고, joblist 에 그것을 반영해야 한다.

결과 확인

- trace10을 sdriver를 수행하여 결과를 확인한다.
  - 명령어 : ./sdriver -t 10 -s ./tsh -V

HINT

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

참고 - Shell Lab - trace12

trace12

- 구현 기능 : 자식 프로세스 스스로에게 SIGTSTP 전송되고 처리
  - 자신의 pid(자식 프로세스의 pid)로 SIGTSTP를 전달하였을 때, 정상적으로 SIGTSTP에 대한 처리가 되도록 구현
  - trace09를 제대로 구현했다면 자동으로 통과된다.
    - 자동으로 통과되는 이유를 생각해보고 보고서에 작성.