2020 시스템 프로그래밍

- Shell Lab -

제출일자	2020.11.22
분 반	01
이 름	박재우
학 번	201902694

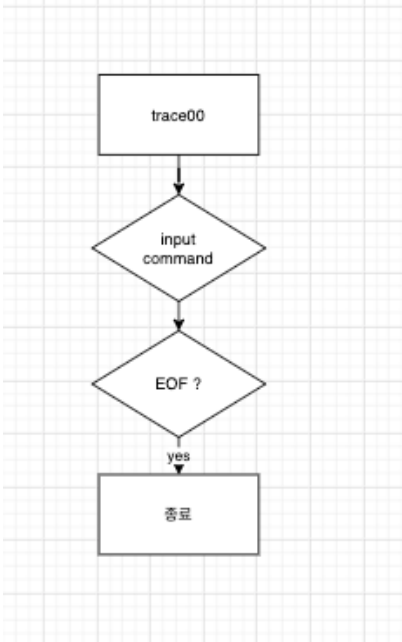
Trace 번호 (00)

```
b201902694@2020sp:~/shlab-handout$ ./sdriver -V -t 00 -s ./tsh
Running trace00.txt...
Success: The test and reference outputs for trace00.txt matched!
Test output:
#
# trace00.txt - Properly terminate on EOF.
#

Reference output:
#
# trace00.txt - Properly terminate on EOF.
#
```

각 trace 별 플로우 차트

https://app.diagrams.net/사이트를 이용하여 플로우 차트를 구현하였습니다.



trace 해결 방법 설명

```
if (feof(stdin)) { /* End of file (ctrl-d) */
    fflush(stdout);
    fflush(stderr);
    exit(0);
}
```

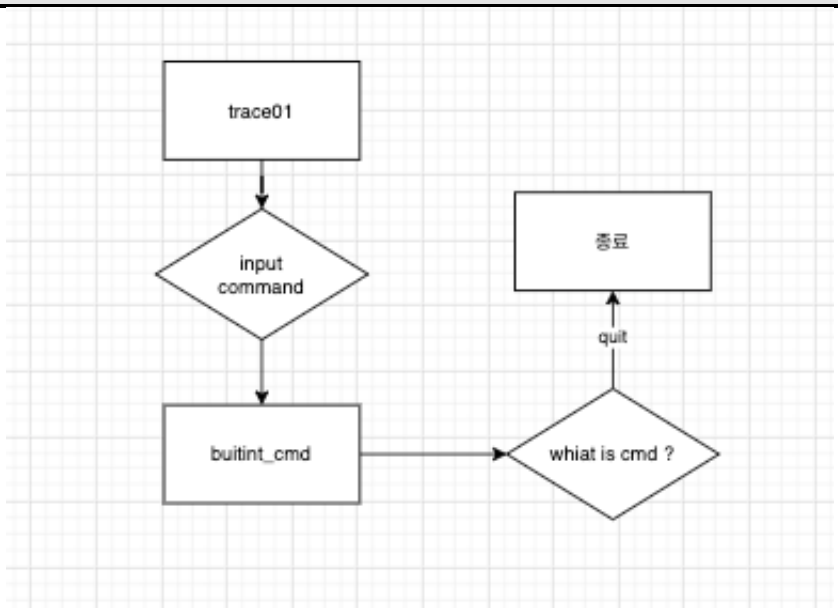
이미 main 에 구현되어있는 기능이였다. Feof 는 파일의 끝을 가리키는 함수이며 stdin, stdout 은 표준 입출력 스트림이다. Stdin 은 키보드를 가리키며 stdout 은 모니터를 가리킨다. Flush()를 통해 버퍼를 비워준다.

Trace 번호 (01)

```
b201902694@2020sp:~/shlab-handout$ ./sdriver -V -t 01 -s ./tsh
Running trace01.txt...
Success: The test and reference outputs for trace01.txt matched!
Test output:
#
# trace01.txt - Process builtin quit command.
#

Reference output:
#
# trace01.txt - Process builtin quit command.
#
```

각 trace 별 플로우 차트



trace 해결 방법 설명

```
if(!builtin_cmd(argv)) {
```

위는 eval함수에 적혀있는 코드이다. 위의 함수를 통해 built_in 함수로 이동한다.

```
int builtin_cmd(char **argv)
{
    if (!strcmp(argv[0], "quit")) {
        exit(0);
    }
```

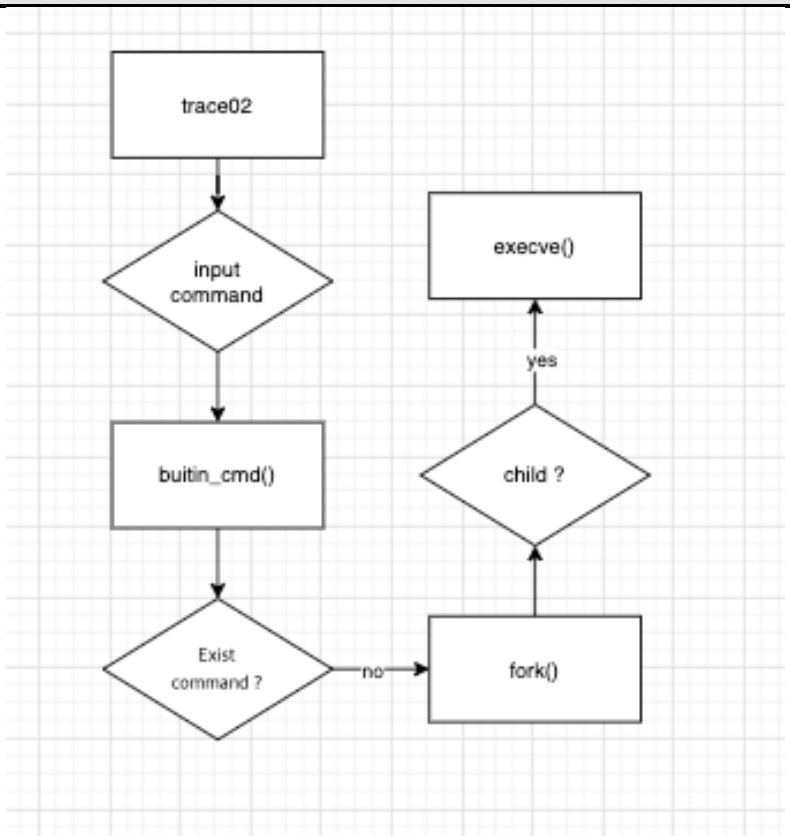
이후 built_in에서 입력받은 명령어가 'quit'일 경우 프로그램을 종료한다.

Trace 번호 (02)

```
b201902694@2020sp:~/shlab-handout$ ./sdriver -V -t 02 -s ./tsh
Running trace02.txt...
Success: The test and reference outputs for trace02.txt matched!
Test output:
#
# trace02.txt - Run a foreground job that prints an environment variable
#
# IMPORTANT: You must pass this trace before attempting any later
# traces.  In order to synchronize with your child jobs, the driver
# relies on your shell properly setting the environment.
OSTYPE=/home/sys01/b201902694/.vscode-server/bin/2af051012b66169dde0c4dfae3f5ef48f787ff69/bin:/home/sys01/b201902694/.vscode-server/bin/2af051012b66169dde0c4dfae3f5ef48f787ff69/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games

Reference output:
#
# trace02.txt - Run a foreground job that prints an environment variable
#
# IMPORTANT: You must pass this trace before attempting any later
# traces.  In order to synchronize with your child jobs, the driver
# relies on your shell properly setting the environment.
OSTYPE=/home/sys01/b201902694/.vscode-server/bin/2af051012b66169dde0c4dfae3f5ef48f787ff69/bin:/home/sys01/b201902694/.vscode-server/bin/2af051012b66169dde0c4dfae3f5ef48f787ff69/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

각 trace 별 플로우 차트



trace 해결 방법 설명

```
if(!builtin_cmd(argv)) {
    if((pid = fork()) == 0) {
        if(execve(argv[0], argv, environ) < 0) {
            printf("%s : Command not found\n\n", argv);
            exit(1);
        }
    }
```

Fork()함수는 child 프로세스를 생성하는 함수이다. 부모는 pid 에 생성한 자식의 값을 갖고 자식은 0 을 갖는다. 자식은 pid == 0 을 만족시켜 execve 함수를 실행한다. 이함수는 새로운 프로그램을 로드하고 실행하는 함수이다. 총 3 가지 인자를 입력받으며, 이후 코드를 실행하는데 오류가 발생하면 이유를 출력하고 프로그램을 종료시킨다.

Trace 번호 (03 ~ 04)

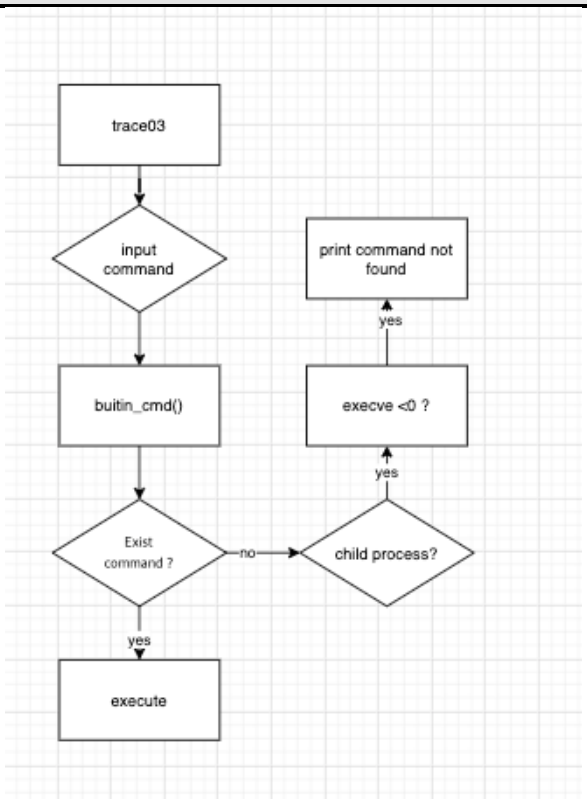
```
b201902694@2020sp:~/shlab-handout$ ./sdriver -V -t 03 -s ./tsh
Running trace03.txt...
Success: The test and reference outputs for trace03.txt matched!
Test output:
#
# trace03.txt - Run a synchronizing foreground job without any arguments.
#

Reference output:
#
# trace03.txt - Run a synchronizing foreground job without any arguments.
#
```

```
b201902694@2020sp:~/shlab-handout$ ./sdriver -V -t 04 -s ./tsh
Running trace04.txt...
Success: The test and reference outputs for trace04.txt matched!
Test output:
#
# trace04.txt - Run a foreground job with arguments.
#
tsh> quit

Reference output:
#
# trace04.txt - Run a foreground job with arguments.
#
tsh> quit
```

각 trace 별 플로우 차트



trace 해결 방법 설명

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

    bg = parseline(cmdline, argv);

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

Trace 03 및 04 번은 각각 Foreground 작업 형태로 매개변수에 따른 프로그램 실행을 보여주도록 해야했다. Eval 함수를 사용하여 받은 명령어 분리 후 들어온 이 명령어가 builtin_cmd 을 통해 실행할 수 있는 명령어인지 판단한다. 실행 할 수 없다면 Command not found 를 출력하도록 했다. 입력 에서 명령어를 받을 때 parseline 으로 끊어서 argv[0]에 저장하고, 매개변수들도 같이 입력으로 들어간다. 따라서 이것을 분리하여 처리를 한다.

Trace 번호 (05 ~ 06)

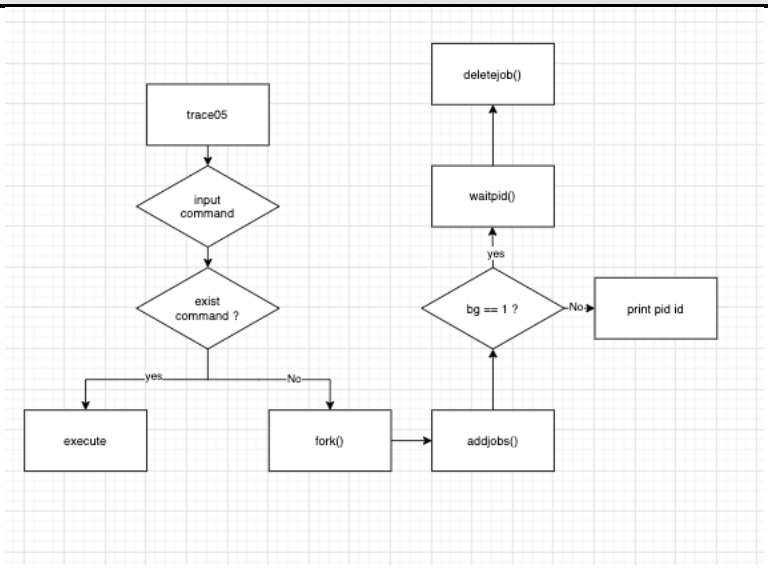
```
b201902694@2020sp:~/shlab-handout$ ./sdriver -V -t 06 -s ./tsh
Running trace06.txt...
Success: The test and reference outputs for trace06.txt matched!
Test output:
#
# trace06.txt - Run a foreground job and a background job.
#
tsh> ./myspin1 &
(1) (15571) ./myspin1 &
tsh> ./myspin2 1

Reference output:
#
# trace06.txt - Run a foreground job and a background job.
#
tsh> ./myspin1 &
(1) (15580) ./myspin1 &
tsh> ./myspin2 1
```

```
b201902694@2020sp:~/shlab-handout$ ./sdriver -V -t 05 -s ./tsh
Running trace05.txt...
Success: The test and reference outputs for trace05.txt matched!
Test output:
#
# trace05.txt - Run a background job.
#
tsh> ./myspin1 &
(1) (15536) ./myspin1 &
tsh> quit

Reference output:
#
# trace05.txt - Run a background job.
#
tsh> ./myspin1 &
(1) (15544) ./myspin1 &
tsh> quit
```

각 trace 별 플로우 차트



trace 해결 방법 설명

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

    bg = parseline(cmdline, argv);

    if(!builtin_cmd(argv)) {
        if((pid = fork()) == 0) {
            if(execve(argv[0], argv, environ) < 0) {
                printf("%s : Command not found\n\n", argv);
                exit(0);
            }
        }
        addjob(jobs, pid, (bg == 1 ? BG : FG), cmdline);
        if(!(bg)) {
            waitpid(pid, NULL, 0);
            deletejob(jobs, pid);
        } else {
            printf("(%d) (%d) %s", pid2jid(pid), (int)pid, cmdline);
        }
        usleep(100000);
    }
    return;
}
```

Trace05 번은 Background 작업 형태로 프로그램을 실행 시키는 것이였고 , Trace06 은 동시에 foreground 작업 형태와 background 작업 형태로 프로그램을 실행하는 것이 이었다. Foreground 는 앞에 보이는 쉘 형태를 유지하며, 사용자가 명령어 입력시 그 명령어를 처리하는 뒤의 background 과정을 처리하도록 해야했다.

```
void clearjob(struct job_t *job);
void initjobs(struct job_t *jobs);
int maxjid(struct job_t *jobs);
int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline);
int deletejob(struct job_t *jobs, pid_t pid);
pid_t fgpid(struct job_t *jobs);
struct job_t *getjobpid(struct job_t *jobs, pid_t pid);
struct job_t *getjobjid(struct job_t *jobs, int jid);
int pid2jid(pid_t pid);
void listjobs(struct job_t *jobs, int output_fd);
```

```
/* Misc manifest constants */
#define MAXLINE    1024   /* max line size */
#define MAXARGS     128   /* max args on a command line */
#define MAXJOBS      16   /* max jobs at any point in time */
#define MAXJID    1<<16   /* max job ID */
```

문제를 해결하기 위해 이용해야하는 다양한 함수들이 존재했다. 또한 다양한 함수들 이외에도 여러 개의 정의된 변수들을 확인 할 수 있었다. 위의 여러 함수와 여러 개의 정의된 변수를 통해 addjob 의 코드를 살펴 보았다.

```c
/* addjob - Add a job to the job list */
int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline)
{
    int i;

    if (pid < 1)
        return 0;

    for (i = 0; i < MAXJOBS; i++) {
        if (jobs[i].pid == 0) {
            jobs[i].pid = pid;
            jobs[i].state = state;
            jobs[i].jid = nextjid++;
            if (nextjid > MAXJOBS)
                nextjid = 1;
            strcpy(jobs[i].cmdline, cmdline);
            if(verbose){
                printf("Added job. [%d] %d %s\n", jobs[i].jid, jobs[i].pid, jobs[i].cmdline);
            }
            return 1;
        }
    }
    printf("Tried to create too many jobs\n");
    return 0;
}
```

Addjob 은 인자를 4 개를 받는데, 첫번째로 Job 의 구조체를 받고, pid 를 받고, state 를 받고, 마지막으로 명령어를 받았다. pid 가 1 보다 작을경우 할당된 값이 없기에 0 을 리턴했다. Pid 를 할당해주고, 상태를 준 다음 jid 를 증가시키도록 되어있었다. 작업 리스트에 작업을 추가하기 위한 함수였다.

```c
/* deletejob - Delete a job whose PID=pid from the job list */
int deletejob(struct job_t *jobs, pid_t pid)
{
    int i;

    if (pid < 1)
        return 0;

    for (i = 0; i < MAXJOBS; i++) {
        if (jobs[i].pid == pid) {
            clearjob(&jobs[i]);
            nextjid = maxjid(jobs)+1;
            return 1;
        }
    }
    return 0;
}
```

Delete job 은 실행되고 있는 작업 리스트에서 작업을 제거하는 함수였다. job 의 구조체와 pid 를 받아서 clearjob 이라는 함수를 통해 작업을 종료시키도록 되어있었다. 그리고 maxjid 함수에서 반환값에 1 을 더해 nextid 를 바꾸도록 되어있었다.

```c
/* clearjob - Clear the entries in a job struct */
void clearjob(struct job_t *job) {
    job->pid = 0;
    job->jid = 0;
    job->state = UNDEF;
    job->cmdline[0] = '\0';
}


/* maxjid - Returns largest allocated job ID */
int maxjid(struct job_t *jobs)
{
    int i, max=0;

    for (i = 0; i < MAXJOBS; i++)
        if (jobs[i].jid > max)
            max = jobs[i].jid;
    return max;
}
```

Clearjob 함수는 job structure 안에 있는 변수들을 모두 초기화 시키는 동작을 하고 있었다. Maxjid 는 job 이 현재 최대 몇 개 실행되고 있는지를 체크하고 반환하는 함수 같았다.

```c
/* pid2jid - Map process ID to job ID */
int pid2jid(pid_t pid)
{
    int i;

    if (pid < 1)
        return 0;
    for (i = 0; i < MAXJOBS; i++)
        if (jobs[i].pid == pid) {
            return jobs[i].jid;
        }
    return 0;
}
```

pid2jid 는 인자로는 pid 만 받는다. 생성된 job 의 pid 와 현재 들어온 pid 가 일치할 경우 job 의 jid 를 리턴한다.

```
    addjob(jobs, pid, (bg == 1 ? BG : FG), cmdline);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);  //시그널 언블록
    if(!(bg)) {
        /*waitpid(pid, NULL, 0);
        deletejob(jobs, pid);*/
        while(1) {
            if(pid != fgpid(jobs))
                break;
            else
             sleep(1);
        }
    } else {
        printf("(%d) (%d) %s", pid2jid(pid), (int)pid, cmdline);
    }

    usleep(100000);
}
return;
```

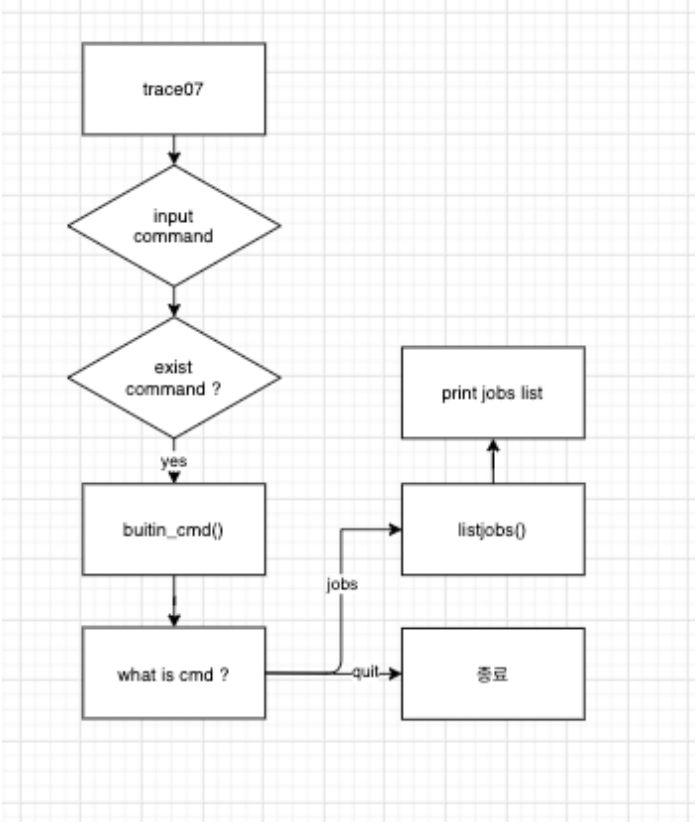
결론적으로 Addjob 함수를 통해 job 을 추가, bg == 1 이면 background 실행, waitpid 함수를 이용하여 부모 프로세서가 자식 프로세서의 종료 대기 후에 처리하고 Deletejob 함수를 통해 job 을 초기화하여 지우도록 하였으며 bg != 1 경우에는 foreground 를 활성화와 화면에 매핑시킨 프로세서 jid 와 작업한 Id, 명령어를 출력하게 만들었다.

Trace 번호 (07)

```
b201902694@2020sp:~/shlab-handout$ ./sdriver -V -t 07 -s ./tsh
Running trace07.txt...
Success: The test and reference outputs for trace07.txt matched!
Test output:
#
# trace07.txt - Use the jobs builtin command.
#
tsh> ./myspin1 10 &
(1) (7375) ./myspin1 10 &
tsh> ./myspin2 10 &
(2) (7377) ./myspin2 10 &
tsh> jobs
(1) (7375) Running    ./myspin1 10 &
(2) (7377) Running    ./myspin2 10 &

Reference output:
#
# trace07.txt - Use the jobs builtin command.
#
tsh> ./myspin1 10 &
(1) (7385) ./myspin1 10 &
tsh> ./myspin2 10 &
(2) (7387) ./myspin2 10 &
tsh> jobs
(1) (7385) Running ./myspin1 10 &
(2) (7387) Running ./myspin2 10 &
```

각 trace 별 플로우 차트



trace 해결 방법 설명

Trace 7 번은 bulit_in 명령어인 job 을 보여주게 하는 기능을 구현하는 것이었다. 이전에 구현했던 것들이 제대로 잘 실행이 잘 되고 있는지 그것들의 목록을 출력하는 기 능을 구현해야 했다. Pdf 에는 builtin_cmd 함수에 명령어를 하라고 나와있었다. 들어온 입력이 'jobs' 일 때를 처리해야 했다. 이를 위해 listjobs 의 작동을 상세히 살펴 보았다.

```c
/* listjobs - Print the job list */
void listjobs(struct job_t *jobs, int output_fd)
{
    int i;
    char buf[MAXLINE];

    for (i = 0; i < MAXJOBS; i++) {
        memset(buf, '\0', MAXLINE);
        if (jobs[i].pid != 0) {
            sprintf(buf, "(%d) (%d) ", jobs[i].jid, jobs[i].pid);
            if(write(output_fd, buf, strlen(buf)) < 0) {
                fprintf(stderr, "Error writing to output file\n");
                exit(1);
            }
            memset(buf, '\0', MAXLINE);
            switch (jobs[i].state) {
                case BG:
                    sprintf(buf, "Running    ");
                    break;
                case FG:
                    sprintf(buf, "Foreground ");
                    break;
                case ST:
                    sprintf(buf, "Stopped    ");
                    break;
                default:
                    sprintf(buf, "listjobs: Internal error: job[%d].state=%d ",
                            i, jobs[i].state);
            }
            if(write(output_fd, buf, strlen(buf)) < 0) {
                fprintf(stderr, "Error writing to output file\n");
                exit(1);
            }
            memset(buf, '\0', MAXLINE);
            sprintf(buf, "%s", jobs[i].cmdline);
            if(write(output_fd, buf, strlen(buf)) < 0) {
                fprintf(stderr, "Error writing to output file\n");
                exit(1);
            }
        }
    }
    if(output_fd != STDOUT_FILENO)
        close(output_fd);
}
```

istjobs 함수는 인자로 job 을 받고, output_fd 를 받는다. Output_fd 는 piazza 질문내용을 통해 파일 디스크립터라는 것을 알게 되어 해결을 했다. 리눅스에서는 일반적으로 0, 1, 2 번을 파일 디스크립터의 특수한 목적으로 사용하게 되는데 0 번은 표준입력(stdin), 1 번은 표준 출력 (stdout), 2 번은 표준 오류(stderr)에 매핑이 되어있다는 사실을 알게 되었다.

Job 의 pid 가 0 이 아닐 때, 즉 제대로 된 pid 값이 할당되어 있을 때 조건문 안에 있는 동작을 실 행하게 된다. write 함수를 이용해 문자열을 쓰는 동작이 정의되어 있었다. Listjobs 안의 함수들은 이전 42seoul 을 할 때 직접 구현한 적이 있어서 익숙했다.

Structure job 의 변수의 state 가 무엇인지에 따라 그에 따른 상태를 출력하고, write 가 잘못되었다면 에러 메세지를 출력하도록 되어있었다. 이전에 생성되었던 buf 의 출력 형식에 어떠한 상태인지를 buf 에 써준다음 보여주게 되는 동작을 하고 job list 를 볼 수 있게 하는 함수였다.

```c
int builtin_cmd(char **argv)
{
    if (!strcmp(argv[0], "quit")) {
        exit(0);
    }
    else if (!strcmp("&", argv[0])){
        return 1;
    }
    else if (!strcmp("jobs", argv[0])) {
        listjobs(jobs,1);
        return 1;
    }
    return 0;
}
```

따라서 argv 에서 가져온 값이 jobs 일경우 listjobs 를 통해 buf 에 write 를 이용하기위해 1 을 넣어준뒤 Return 1 을 해준뒤에 작업을 종료한다.

Trace 번호 (08, 11)

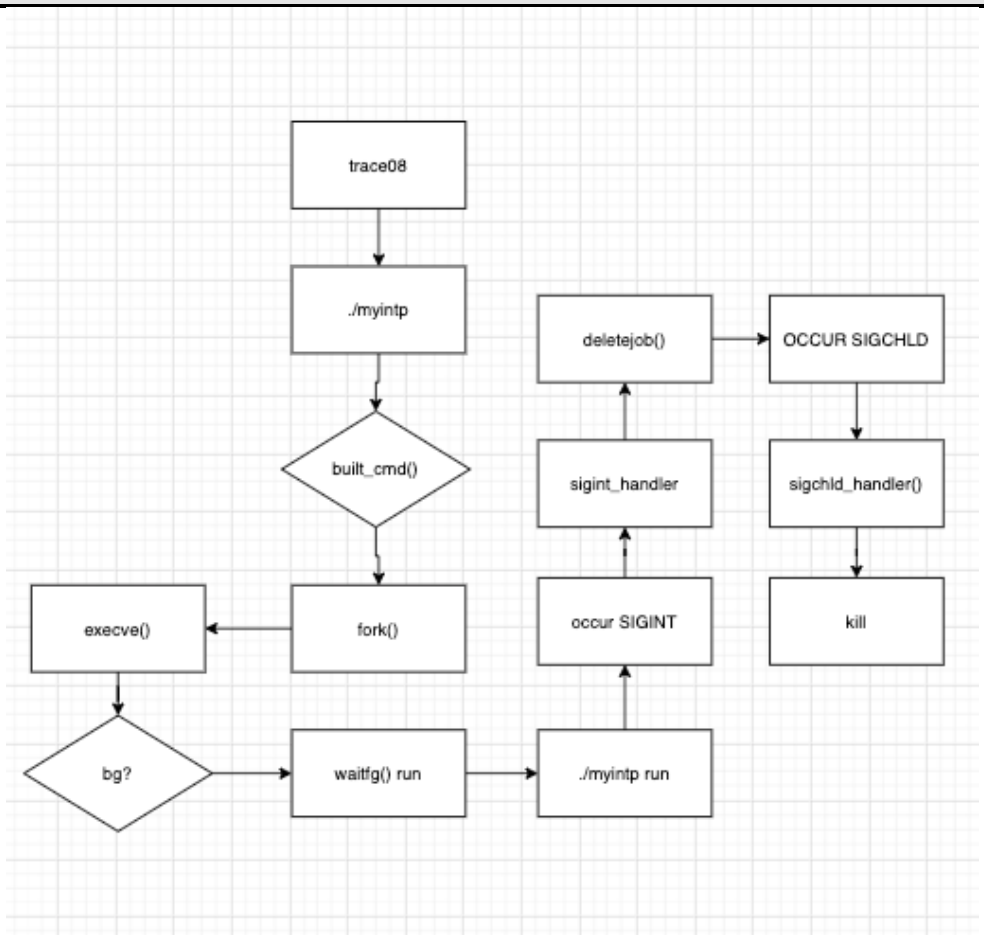
```
b201902694@2020sp:~/shlab-handout$ ./sdriver -V -t 08 -s ./tsh
Running trace08.txt...
Success: The test and reference outputs for trace08.txt matched!
Test output:
#
# trace08.txt - Send fatal SIGINT to foreground job.
#
tsh> ./myintp
Job [1] (18386) terminated by signal 2
tsh> quit

Reference output:
#
# trace08.txt - Send fatal SIGINT to foreground job.
#
tsh> ./myintp
Job [1] (18394) terminated by signal 2
tsh> quit
```

```
b201902694@2020sp:~/shlab-handout$ ./sdriver -V -t 11 -s ./tsh
Running trace11.txt...
Success: The test and reference outputs for trace11.txt matched!
Test output:
#
# trace11.txt - Child sends SIGINT to itself
#
tsh> ./myints
Job [1] (18598) terminated by signal 2
tsh> quit

Reference output:
#
# trace11.txt - Child sends SIGINT to itself
#
tsh> ./myints
Job [1] (18611) terminated by signal 2
tsh> quit
```

각 trace 별 플로우 차트



trace 해결 방법 설명

Trace 08 에서의 처리는 SIGINT 발생시에, fg 작업을 종료하는 것이었고, 11 에서의 작업은 자식 프로세스가 스스로에게 SIGINT 를 전송하는 것이었다. 즉, Trace 08 을 구현하는 과정안에서 11 번이 구현이 된다는 것이다. 이것을 해결하기 위한 방법으로 아래의 방법을 시도해 보았다. 우선, SIGINT 는 fork 와 execve 를 통하여 프로그램을 실행시키고, SIGINT 를 발생시키 게 된다. 그 후에 커널을 이용하여 자식 프로세스에서 부모 프로세스로 SIGINT 를 전달하고, 이 핸들러를 통하여 SIGINT 를 처리하게 된다. 자식 프로세스를 KILL 명령어를 통하여 핸들 러에게 전달하게 되고, 이렇게 전달된 것에 의하여 자식 프로세스가 종료되면 sigcld 시그널 을 발생시키고, 그 후에 sigcld 핸들러를 이용하여 자식 프로세스의 최종 종료 처리를 하게 된다. 위의 방식을 이용하기 위해서 아래와 같은 코드를 작성하였다.

```
sigset_t mask;

bg = parseline(cmdline, argv);

sigemptyset(&mask);           //시그널 셋 초기화
sigaddset(&mask, SIGCHLD);    //SIGCHLD 시그널을 시그널 셋에 추가
sigaddset(&mask, SIGINT);     //SIGINT
sigaddset(&mask, SIGTSTP);      //SIGTSTP
sigprocmask(SIG_BLOCK, &mask, NULL); //시그널 블록

if(!builtin_cmd(argv)) {
    if((pid = fork()) == 0) {
        sigprocmask(SIG_UNBLOCK, &mask, NULL);  //시그널 언블록
        setpgid(0, 0);
        if(execve(argv[0], argv, environ) < 0) {
            printf("%s : Command not found\n\n", argv);
            exit(1);
        }
    }
    addjob(jobs, pid, (bg == 1 ? BG : FG), cmdline);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);  //시그널 언블록
```

위는 eval 에 구현되어있는 코드이다.

시그널 사용을 위해 셋팅을 해주었다. 첫번째로 시그널 셋을 초기화 해주었다. 그 후, 각각의 필요한 시그널들을 셋에 차례로 SIGCHLD, SIGINT, SIGITST 을 추가해 주었다. 세팅후에 시그널 블록을 실행하도록 했다.

mask 에 대해서 블록하도록 하는데 이유는 블록이 지정되어있는 부분에 fork()로 자식을 생성하고 있는 코드인 것을 확인 할 수 있는데. 자식을 생성하는 동안 kill 시그널이 들어와서 만들어지는 동안 프로세서를 죽이면 안되기에, 이것을 보호하기 위해 사용한다. 자식이 다 만들어진 후 언블럭을 해준 후 다시 신호를 받을 수 있도록 한다. 자식이 아닐 경우도 블록이 됨으로 elese 를 통해 언블럭을 실행해주었다.

```c
void sigchld_handler(int sig)
{
    int status;
    pid_t pid;

    while ((pid = waitpid(-1, &status, WNOHANG|WUNTRACED)) > 0) {
        if (WIFSTOPPED(status)){
            getjobpid(jobs, pid)->state = ST;
            int jid = pid2jid(pid);
            printf("Job [%d] (%d) stopped by signal %d\n", jid, (int)pid, WSTOPSIG(status));
        }
        else if (WIFSIGNALED(status)){
            int jid = pid2jid(pid);
            printf("Job [%d] (%d) terminated by signal %d\n", jid, (int)pid, WTERMSIG(status));
            deletejob(jobs, pid);
        }
        else if (WIFEXITED(status)){
            deletejob(jobs, pid);
        }
    }
    return;
}

/*
 * sigint_handler - The kernel sends a SIGINT to the shell whenver 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 = fgpid(jobs);

    if (pid != 0) {
        kill(pid, sig);
    }
    return;
}
```

Sigint_handler 에서는 fg 의 job 을 실행하는 자식 프로세스에게 SIGINT 를 보내서 자식 프로세스가 종료되게 한다. 자식 프로세스에 SIGINT 를 보내기 위해서 fg job 을 수행하는 자식 프로세스의 pid 를 얻은 후 fgpid 함수를 이용하여 자식에게 SIGINT 를 보내도록 작동한다.

sigcld_handler 는 waitpid 함수를 사용하여, 시그널에 의해서 종료된 함수인지를 체크하는 함수이다. Waitpid 함수는 자식 프로세스의 id, status, option 으로 되어있는데, 자식프로세스가 신호로 종료되었다면 true 를 반환하는 WIFSIGNAMLED()를 통해 종료된 함수인지 아닌지 체크하였다. 시그널에 의해서 종료되었다면, 자식 프로세스를 종료하도록 한 신호의 번호를 반환하는 WTERMSIG() 함수를 통하여, 값을 얻었다. 그리고 delete job 을 통해서 작업을 삭제하도록 구현하였다.

Trace 번호 (09, 12)

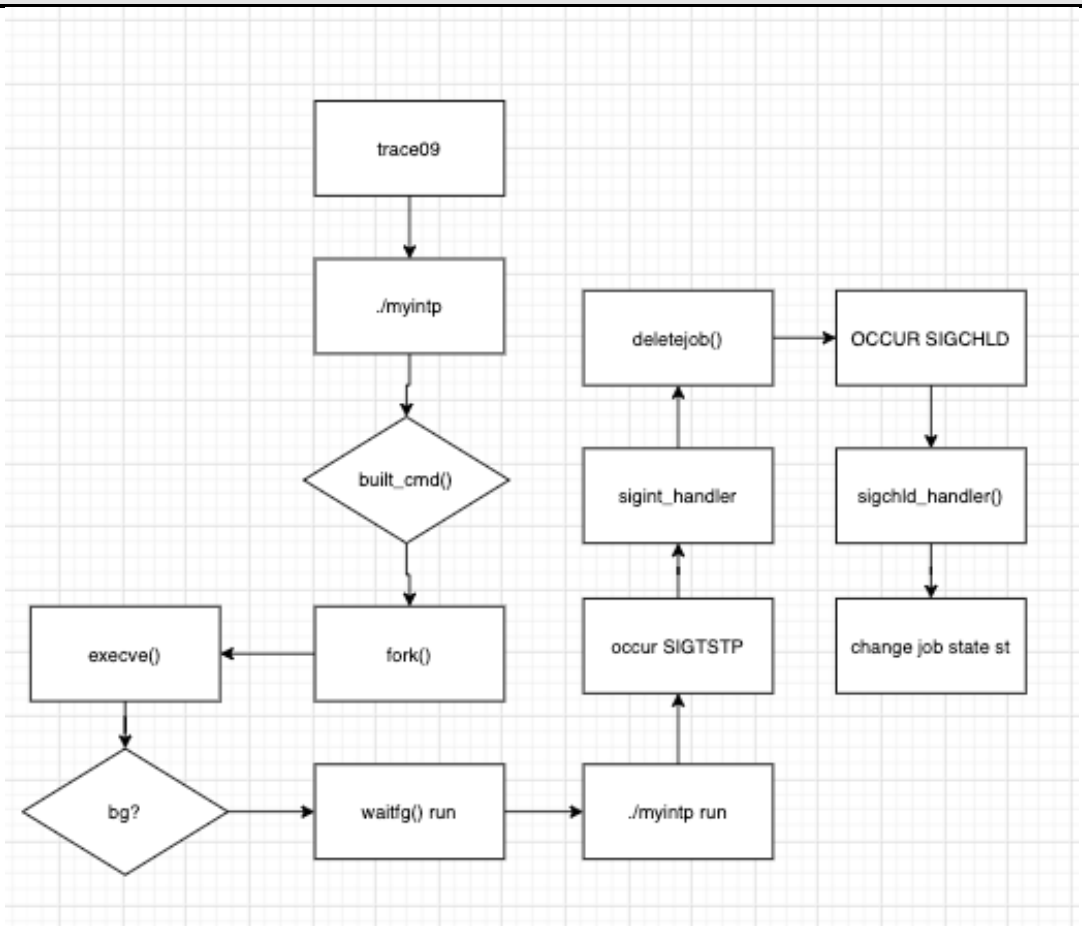
```
b201902694@2020sp:~/shlab-handout$ ./sdriver -V -t 09 -s ./tsh
Running trace09.txt...
Success: The test and reference outputs for trace09.txt matched!
Test output:
#
# trace09.txt - Send SIGTSTP to foreground job.
#
tsh> ./mytstpp
Job [1] (25558) stopped by signal 20
tsh> jobs
(1) (25558) Stopped    ./mytstpp

Reference output:
#
# trace09.txt - Send SIGTSTP to foreground job.
#
tsh> ./mytstpp
Job [1] (25566) stopped by signal 20
tsh> jobs
(1) (25566) Stopped ./mytstpp
```

```
b201902694@2020sp:~/shlab-handout$ ./sdriver -V -t 12 -s ./tsh
Running trace12.txt...
Success: The test and reference outputs for trace12.txt matched!
Test output:
#
# trace12.txt - Child sends SIGTSTP to itself
#
tsh> ./mytstps
Job [1] (25609) stopped by signal 20
tsh> jobs
(1) (25609) Stopped    ./mytstps

Reference output:
#
# trace12.txt - Child sends SIGTSTP to itself
#
tsh> ./mytstps
Job [1] (25617) stopped by signal 20
tsh> jobs
(1) (25617) Stopped ./mytstps
```

각 trace 별 플로우 차트



trace 해결 방법 설명

작동 구조는 trace08 과 거의 동일했다.

Trace 09 는 SIGTSTP 발생 시에 fg 작업을 종료해야 했으며, 12 는 자식 프로세스가 스스로에게 SIGTSTP 를 전송하도록 해야 했다. 자식 프로세스가 mytstp 를 실행하면, 부모 프로세스에게 SIGSTOP signal 을 전송하게 된다. 컨트롤 + z 에 대한 처리를 하면 된다(SIGINT 와 다른 동작). SIGSTOP 을 받은 쉘은 실행되고 있는 fg job 에게 SIGSTOP 신호를 보내며 fg 작업을 정지시킨다.

위의 문제는 정지후에도 자식이 계속 sigchld signal 을 보내는 것 이였다. Sigchld_handler 는 자식이 정지될 경우, 이 자식을 job list 에서 제거하는 것이 아닌, 다른 처리를 해주어야 해서 sigchld_handler 를 수정했다. job 의 pid 를 활용하여, job 의 몇 번째에 있는 job 인 지

알아내어 해당 작업의 FG state 를 ST 상태로 변환해주었다.

```
sigprocmask(SIG_BLOCK, &mask, NULL); //시그널 블록

if(!builtin_cmd(argv)) {
    if((pid = fork()) == 0) {
        sigprocmask(SIG_UNBLOCK, &mask, NULL);  //시그널 언블록
        setpgid(0, 0);
        if(execve(argv[0], argv, environ) < 0) {
            printf("%s : Command not found\n\n", argv);
            exit(1);
        }
    }
    addjob(jobs, pid, (bg == 1 ? BG : FG), cmdline);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);  //시그널 언블록
    if(!(bg)) {
        /*waitpid(pid, NULL, 0);
        deletejob(jobs, pid);*/
        while(1) {
            if(pid != fgpid(jobs))
                break;
            else
             sleep(1);
        }
    } else {
        printf("(%d) (%d) %s", pid2jid(pid), (int)pid, cmdline);
    }
```

우선, 부모가 SIGSTOP 을 addjob 에 바로 들어가면 오류가 나기에 그전에 블록을 시켰다. 부모가 addjob 을 하기 전, 자식이 실행하는 mytstp 프로그램이 먼저 SIGTSTP 시그널을 발생시킨 후 부모의 SIFTSTP 핸들러가 fg 작업에게 SIFTSTP 를 전송, 자식이 정지한다. 이후 부모가 작업을 추가하고 waitfg()를 이용해 foreground 작업을 기다린다. SIGTSTP 에 의해 fg 작업이 정지된 작업으로 바뀌지 않아 waitfg()에서 sleep 시스템 콜이 계속 실행된다.

```
/*
 * 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 = fgpid(jobs);
    //check for valid pid
    if (pid != 0) {
        kill(-pid, sig);
    }
    return;
}
```

위 함수는 부모 프로세스로부터 컨트롤 + z 를 입력받을경우 실행하게 된다. 쉘은 fg job 을 수행할 때 SIGTSTP 신호를 받으면 작업 종료, 자식에게 같은 신호를 보내는데, 이를 통해 자식은 정지, 신호를 SIGCHLD 를 보낸다. 위와 같이 kill 을 사용하여 자식의 pid 를 가져오며, fgpid 를 사용하여 pid 를 얻고 kill 을 통해 자식 프로세스를 종료하게 된다.

```c
void sigchld_handler(int sig)
{
    int status;
    pid_t pid;

    while ((pid = waitpid(-1, &status, WNOHANG|WUNTRACED)) > 0) {
        if (WIFSTOPPED(status)){
            getjobpid(jobs, pid)->state = ST;
            int jid = pid2jid(pid);
            printf("Job [%d] (%d) stopped by signal %d\n", jid, (int)pid, WSTOPSIG(status));
        }
        else if (WIFSIGNALED(status)){
            int jid = pid2jid(pid);
            printf("Job [%d] (%d) terminated by signal %d\n", jid, (int)pid, WTERMSIG(status));
            deletejob(jobs, pid);
        }
        else if (WIFEXITED(status)){
            deletejob(jobs, pid);
        }
    }
    return;
}
```

위의 함수는 이전의 Trace 08 의 코드에 이어서 SIGTSTOP 에 대한 내용을 추가로 작성했다. 자식 프로세스는 정지된 후에도, 부모에게 SIGCHLD 신호를 계속 보낸다. 따라서 종료되지 않은 것으로 인식 되게 된다. 때문에 현재 pid 를 이용하여, getjobpid 을 이용하여 job_t 에 접근한 다음, 그 구조체의 state 를 ST(stop)으로 바꿔준다. 이후, 정지된 자식을 화면에 출력해 준다. SIGINT 와 같은 옵션으로 처리를 하면 오류가 난다. 따라서 자식을 정지하도록 원인 을 제공한 신호의 숫자를 반환해주는 WSTOPSIG status 를 써서 오류를 해결해준다.

Trace 번호 (10)

```
b201902694@2020sp:~/shlab-handout$ ./sdriver -V -t 10 -s ./tsh
Running trace10.txt...
Success: The test and reference outputs for trace10.txt matched!
Test output:
#
# trace10.txt - Send fatal SIGTERM (15) to a background job.
#
tsh> ./myspin1 5 &
(1) (28898) ./myspin1 5 &
tsh> /bin/kill myspin1
kill: failed to parse argument: 'myspin1'
tsh> quit

Reference output:
#
# trace10.txt - Send fatal SIGTERM (15) to a background job.
#
tsh> ./myspin1 5 &
(1) (28910) ./myspin1 5 &
tsh> /bin/kill myspin1
kill: failed to parse argument: 'myspin1'
tsh> quit
```

각 trace 별 플로우 차트

Trace10 은 Background 작업 정상 종료 처리를 하는 것이다, 따라서 이전의 Trace 구현이 제대로 되어있다면 문제없이 Trace10 은 통과 된다. 따라서 따로 플로우 차트를 첨부하진 않는다.

trace 해결 방법 설명

이번 문제는 bg job 종료 시 발생하는 SIGTERM 을 SIFCHLD 로 구현하는 것이다.
SIFCHLD 를 처리하는 핸들러 함수를 만들고, signal 로 핸들러를 등록시킨다. 핸들러의 내부 기능은 자식 프로세스가 종료한 뒤에, 부모 프로세스에게 날려주는 SIGCHLD 를 처리하는 내용으로 기본적으로 자식 프로세스가 사용 중인 메모리 자원을 반환시켜주는 기능을 처리한다.