2020 시스템 프로그래밍
- Shell Lab -

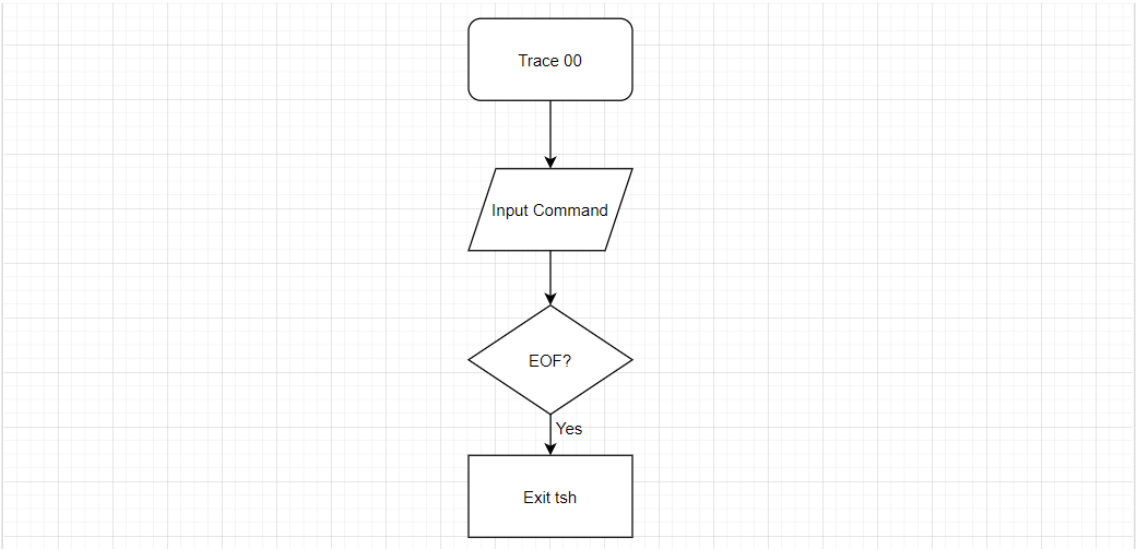
제출일자	
분 반	
이 름	
학 번	

Trace 번호 (00)

```
[redacted]@2020sp:~/shlab-handout$ ./sdriver -V -s ./tsh -t 0
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



trace 해결 방법 설명

trace00은 EOF('Ctrl+D'를 입력했을 때를 의미함)가 입력되면 쉘이 종료되도록 하는 것이 목적이다. 하지만 이 기능은 tsh.c의 main()에 아래와 같이 파일의 끝(EOF)에 도달했으면 1을 반환하는 feof() 함수를 사용하는 조건문으로 이미 구현되어 있기 때문에 따로 코드를 추가하지 않아도 tshref와 동일한 동작을 tsh에서 수행할 수 있었다.

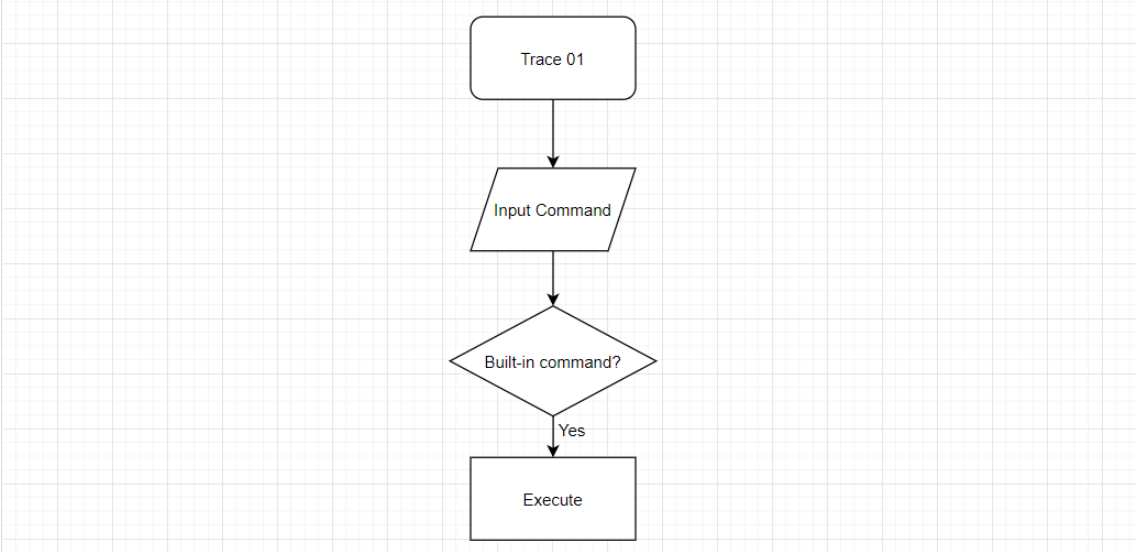
```
146         if (feof(stdin)) { /* End of file (ctrl-d) */
147             fflush(stdout);
148             fflush(stderr);
149             exit(0);
150         }
```

Trace 번호 (01)

```
[redacted]@2020sp:~/shlab-handout$ ./sdriver -V -s ./tsh -t 1
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

trace01은 입력받은 명령어가 built-in 명령어 중 하나인 'quit'이면 쉘을 종료하는 것이 목적이었다. 명령어를 입력받은 다음, eval() 함수를 사용하여 입력받은 명령어를 parseline() 함수를 통해 파싱한 다음, 파싱된 명령어를 builtin_cmd() 함수로 전달하여 해당 명령어가 'quit'이면 쉘을 종료하도록 builtin_cmd() 함수를 작성하였다.

```
172 void eval(char *cmdline)
173 {
174     char *argv[MAXARGS];    // 명령어 저장
175     // parseline을 사용하여 명령어 파싱
176     parseline(cmdline, argv);
177     // 파싱된 명령어를 전달
178     builtin_cmd(argv);
179     return;
180 }
```

```
205 int builtin_cmd(char **argv)
206 {
207     char *cmd = argv[0];
208
209     if(!strcmp(cmd, "quit")){
210         exit(0);
211     }
212
213     return 0;
214 }
```

strcmp() 함수는 c언어의 <string.h> 헤더파일에 정의되어 있으며, 매개변수로 전달받은 두 개의 문자열을 비교하여 동일한 문자열이면 0을 리턴하는 함수이다. 따라서 매개변수로 입력받은 명령어 cmd와 문자열 "quit"을 strcmp() 함수에 전달하여 명령어가 "quit"이어서 0을 반환하면 조건문을 만족시켜 쉘을 종료하도록 strcmp() 앞에 '!'를 붙였다.

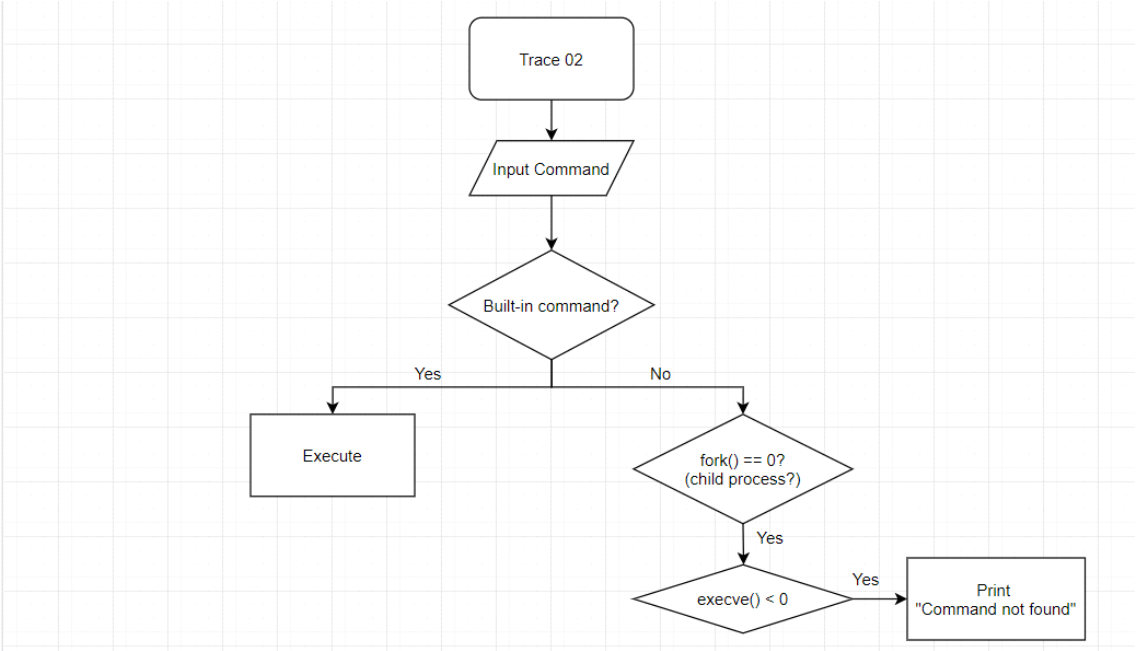
참고로 변수 cmd에 argv[0]을 저장하는데, argv[0]는 입력받은 명령어에서 공백 이전까지의 문자열을 읽은 값을 나타낸다. 예를 들어 명령어 'quit B'가 입력되면 argv[0]은 quit을 나타내며, 좀 더 자세히 들여다보면 argv[0][0]=q, argv[0][1]=u, argv[0][2]=i, argv[0][3]=t 과 같이 명령어가 파싱된다.

Trace 번호 (02)

```
[redacted]@2020sp:~/shlab-handout$ ./sdriver -V -s ./tsh -t 2
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=/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=/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

```
172 void eval(char *cmdline)
173 {
174     char *argv[MAXARGS];
175
176     parseline(cmdline, argv);
177
178     if(!builtin_cmd(argv)){
179         if(fork() == 0){
180             if(execve(argv[0], argv, environ) < 0){
181                 printf("%s : Command not found\n\n", argv);
182                 exit(0);
183             }
184         }
185         usleep(100000);
186     }
187
188     return;
189 }
```

trace02는 입력받은 명령어가 built-in 명령어가 아닐 경우 Foreground 작업 형태로 프로그램이 실행되도록 하는 것이 목적이었다. trace01에서 구현한 builtin_cmd() 함수에서, built-in 명령어가 아닌 명령어가 전달될 경우 0을 리턴하였으므로, 0이 리턴되면 fork()를 호출하여 자식 프로세스를 생성하고, 자식 프로세스가 execve() 함수를 호출하여 명령어에 맞는 프로그램을 수행하도록 eval() 함수를 수정하였다.

execve() 함수는 현재 실행되는 프로세스를 다른 프로그램으로 대체하는 함수로서, 첫 번째 인자로는 실행할 프로그램의 전체 경로를, 두 번째 인자로는 실행할 프로그램의 옵션 정보를, 세 번째 인자로는 환경변수를 받는다. 만약 프로그램이 정상 실행되지 않을 경우 execve() 함수가 음수를 입력한다. 따라서 이 경우 "Command not found"를 출력하고 쉘을

종료하도록 구현하였다.

자식 프로세스가 execve()를 호출하여 프로그램을 로드 및 실행을 완료하면, 부모 프로세스가 자식 프로세스의 종료를 알 수 있도록 해야 하지만, 아직 부모와 foreground에서 실행되는 자식 프로세스 간의 통신을 배우지 않았기 때문에, usleep(100000)을 실행하여 자식 프로세스가 0.1초 안에 종료될 것이라 기대하고 자식 프로세스가 실행되는 동안 부모 프로세스는 0.1초 sleep 상태가 되도록 하였다.

Trace 번호 (03, 04)

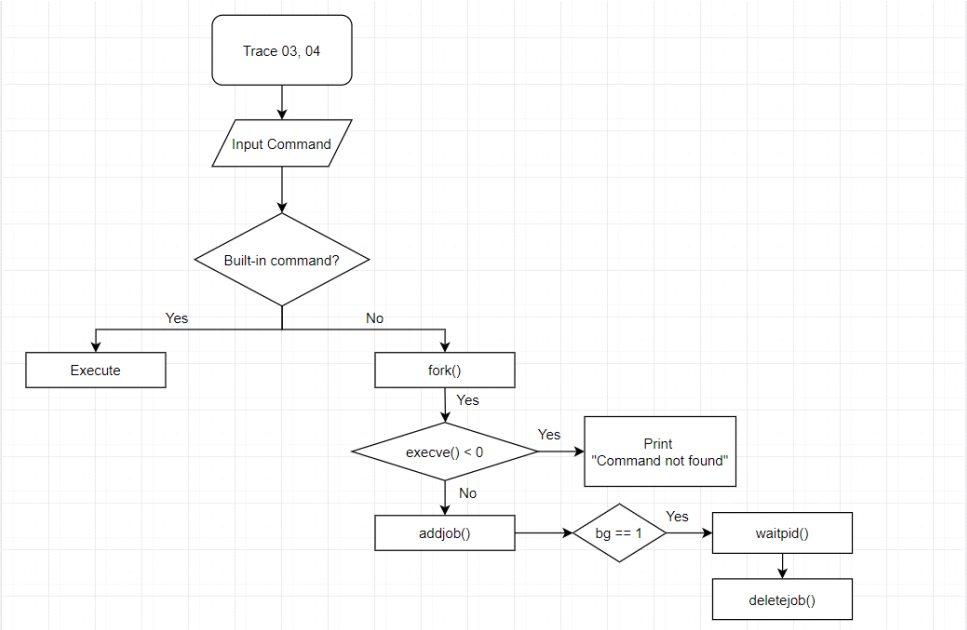
```
[redacted]@2020sp:~/shlab-handout$ ./sdriver -V -s ./tsh -t 3
Running trace03.txt...
Success: The test and reference outputs for trace03.txt matched!
Test output:
#
# trace03.txt - Run a synchronizing foreground job without any arguments.
#

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

```
[redacted]@2020sp:~/shlab-handout$ ./sdriver -V -s ./tsh -t 4
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

trace03은 foreground 작업을 shell과 동기화하여 앞서 trace02에서와는 달리 부모 프로세스가 foreground 작업을 실행한 자식 프로세스의 종료를 기다리도록 하는 것이 목적이었다. trace02에서 execve의 인자로 환경변수와 명령어의 옵션을 모두 받았기 때문에 입력 인자가 있는 foreground 작업을 shell과 동기화하여 실행하는 trace04는 trace03을 해결하면 함께 해결된다.

```
172 void eval(char *cmdline)
173 {
174     char *argv[MAXARGS];
175     pid_t pid;
176     int bg;
177
178     bg = parseline(cmdline, argv);
179
180     if(!builtin_cmd(argv)){
181         if((pid=fork())<0) unix_error("fork error");
182
183         if(pid == 0){
184             if(execve(argv[0], argv, environ) < 0){
185                 printf("%s : Command not found\n\n", argv);
186                 exit(0);
187             }
188         }
189
190         addjob(jobs, pid, (bg == 1? BG : FG), cmdline);
```

우선 trace02번에서는 입력받은 명령어가 built-in 명령어가 아니면 fork() 함수를 호출하기만 하고 반환되는 PID값은 따로 기록해두지 않았는데, trace03에서는 먼저 fork() 함수를 호출한 다음 변수 pid값을 사용하여, fork()가 정상적으로 실행되었는지 에러 체크를 하고, if(pid==0) 조건문으로 자식 프로세스가 실행되도록 하였다. 부모 프로세스에서는, addjob() 함수를 호출하여 자식 프로세스를 Job list에 등록하는데, addjob 함수는 실행 중인 프로세스가 저장되는 전역변수 jobs에, PID와 입력된 명령어를 전달받아 프로세스를 등록한다. 이

때 등록할 프로세스가 background이면(변수 bg==1) 매크로 상수 BG를, foreground이면 FG를 state 인자값으로 받는다. 변수 bg는 line 178의 parseline() 함수의 반환값으로 초기화되는데, 아래의 parseline() 함수의 코드를 살펴보면 입력받은 명령어 끝에 '&'가 있으면 1을 bg값으로 하여 반환하는 것을 볼 수 있다.

```
317     /* should the job run in the background? */
318     if ((bg = (*argv[argc-1] == '&')) != 0)
319         argv[--argc] = NULL;
320
321     return bg;
322
```

eval() 함수의 나머지 코드를 살펴보면, if(!bg) 조건문을 사용하여, 자식 프로세스가 foreground(bg==1) 프로세스인 경우 waitpid() 함수를 통해 자식 프로세스가 종료될 때까지 기다리다가, 종료되면 deletejob() 함수를 호출하여 job list에서 자식 프로세스를 제거하도록 구현하였다.

```
189
190         addjob(jobs, pid, (bg == 1? BG : FG), cmdline);
191
192         if(!bg){
193             waitpid(pid, NULL, 0);
194             deletejob(jobs, pid);
195         }else{
196
197         }
198     }
199
200     return;
201 }
```

Trace 번호 (05, 06)

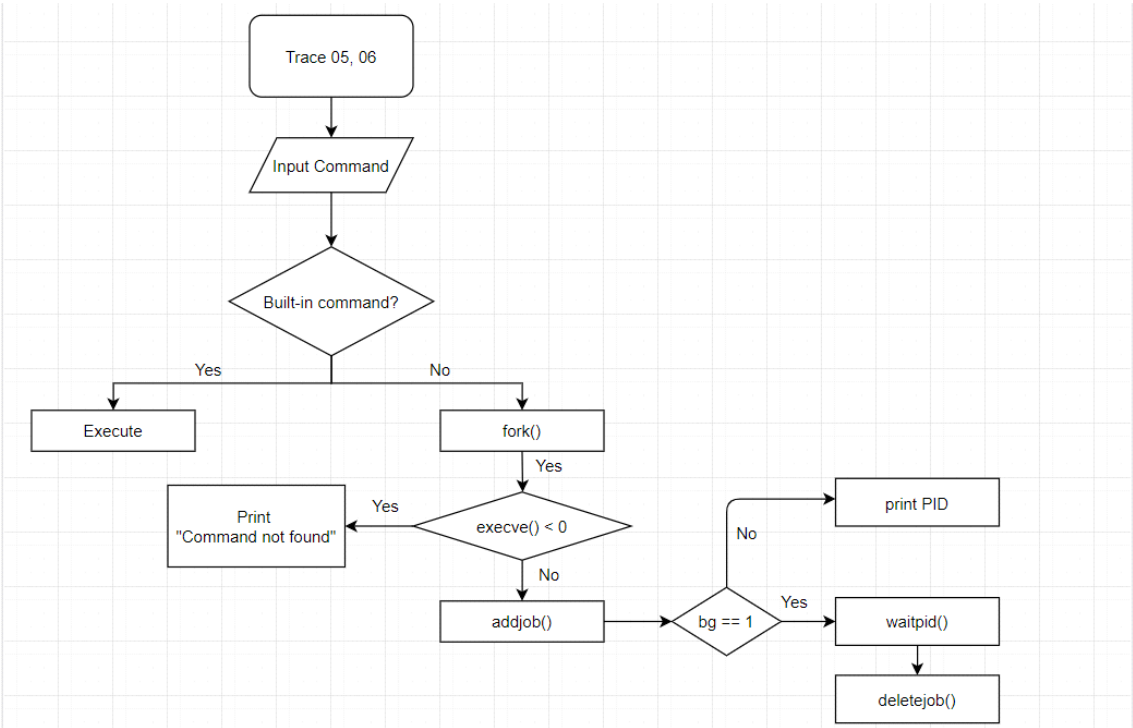
```
[redacted]@2020sp:~/shlab-handout$ ./sdriver -V -s ./tsh -t 5
Running trace05.txt...
Success: The test and reference outputs for trace05.txt matched!
Test output:
#
# trace05.txt - Run a background job.
#
tsh> ./myspin1 &
(1) (13815) ./myspin1 &
tsh> quit

Reference output:
#
# trace05.txt - Run a background job.
#
tsh> ./myspin1 &
(1) (13823) ./myspin1 &
tsh> quit
```

```
[redacted]@2020sp:~/shlab-handout$ ./sdriver -V -s ./tsh -t 6
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) (13869) ./myspin1 &
tsh> ./myspin2 1

Reference output:
#
# trace06.txt - Run a foreground job and a background job.
#
tsh> ./myspin1 &
(1) (13878) ./myspin1 &
tsh> ./myspin2 1
```

각 trace 별 플로우 차트



trace 해결 방법 설명

trace05는 Background 작업 형태로 프로그램을 실행하는 것이 목적이었고, trace06은 동시에 foreground와 background 작업 형태로 프로그램을 실행하는 것이 목적이었다. 따라서 trace 06을 구현하면 trace05도 동시에 구현된다.

```
172 void eval(char *cmdline)
173 {
174     char *argv[MAXARGS];
175     pid_t pid;
176     int bg;
177
178     bg = parseline(cmdline, argv);
179
180     if(!builtin_cmd(argv)){
181         if((pid=fork())<0) unix_error("fork error");
182
183         if(pid == 0){
184             if(execve(argv[0], argv, environ) < 0){
185                 printf("%s : Command not found\n\n", argv);
186                 exit(0);
187             }
188         }
189
190         addjob(jobs, pid, (bg == 1? BG : FG), cmdline);
191
192         if(!bg){
193             // foreground
194             waitpid(pid, NULL, 0);
195             deletejob(jobs, pid);
196         }else{
197             // background
198             printf("(%d) (%d) %s", pid2jid(pid), pid, cmdline);
199         }
200     };
201
202     return;
203 }
```

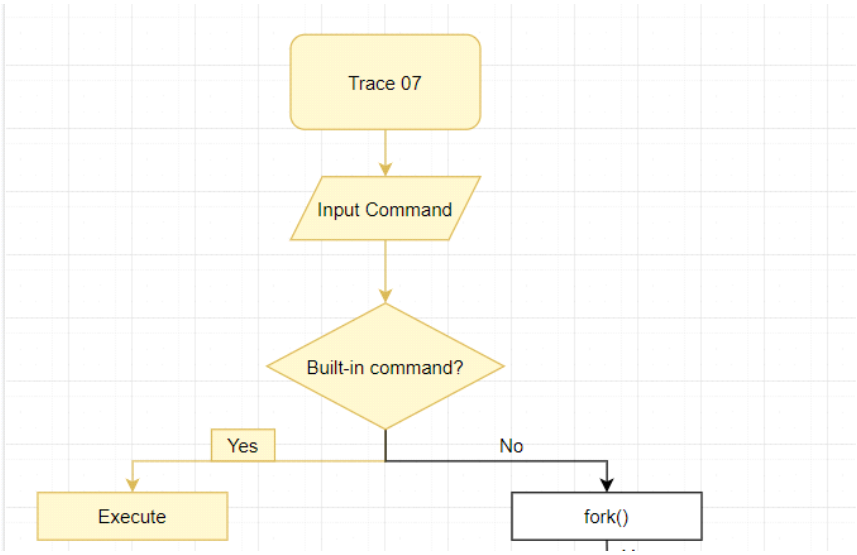
trace06의 경우, 명령어의 끝에 &이 붙은 명령어가 입력될 경우 job list에 추가하고, 쉘 창은 그대로 쓸 수 있는 상태에서 background 작업 수행 시의 작업 내용을 출력하도록 구현해야 한다. 따라서, printf() 함수를 사용하여, 인자로 전달받은 pid를 jobs 내의 작업 ID로 mapping해주는 pid2jid() 함수를 사용하여 얻은 작업 ID, fork() 반환값으로 받은 자식 프로세스의 PID(변수 pid), 입력받은 명령어를 한 문장으로 출력하는 코드를 작성하였다(line 198).

Trace 번호 (07)

```
[redacted]@2020sp:~/shlab-handout$ ./sdriver -V -s ./tsh -t 7
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) (27613) ./myspin1 10 &
tsh> ./myspin2 10 &
(2) (27615) ./myspin2 10 &
tsh> jobs
(1) (27613) Running    ./myspin1 10 &
(2) (27615) Running    ./myspin2 10 &

Reference output:
#
# trace07.txt - Use the jobs builtin command.
#
tsh> ./myspin1 10 &
(1) (27623) ./myspin1 10 &
tsh> ./myspin2 10 &
(2) (27625) ./myspin2 10 &
tsh> jobs
(1) (27623) Running ./myspin1 10 &
(2) (27625) Running ./myspin2 10 &
```

각 trace 별 플로우 차트



trace 해결 방법 설명

trace07은 두 개의 background 작업을 실행하고 built-in 명령어 'jobs'를 명령어로 입력받으면, 현재 실행되고 있는 작업 리스트를 출력하는 것이 목적이었다. 앞서 built-in 명령어 'quit'을 실행하는 trace01과 비슷하게 builtin_cmd() 함수를 수정하여 구현할 수 있다. strcmp() 함수를 사용하여 입력받은 명령어가 'jobs'와 같으면 작업 리스트를 출력하는 코드를 작성하면 되는데, 이때 사용하는 함수는 미리 정의되어 있는 listjobs()이며, 코드는 다음과 같다.

```
450 /* listjobs - Print the job list */
451 void listjobs(struct job_t *jobs, int output_fd)
452 {
453     int i;
454     char buf[MAXLINE];
455 
456     for (i = 0; i < MAXJOBS; i++) {
457         memset(buf, '\0', MAXLINE);
458         if (jobs[i].pid != 0) {
459             sprintf(buf, "(%d) (%d) ", jobs[i].jid, jobs[i].pid);
460             if(write(output_fd, buf, strlen(buf)) < 0) {
461                 fprintf(stderr, "Error writing to output file\n");
462                 exit(1);
463             }
464             memset(buf, '\0', MAXLINE);
```

인자로 job 구조체 jobs와, output_fd를 받는데, output_fd 값은 유닉스 시스템에서 시스템으로부터 할당받은 파일을 대표하는 값인 '파일 디스크립터'이다. 일반적으로 표준입력(STDIN_FILENO)은 0번, 표준출력(STDOUT_FILENO)은 1번, 표준에러(STDERR_FILENO)는 2번으로 값이 할당된다. for문을 살펴보면 jobs에서 i번째 프로세스가 0이 아니면, 해당 프로세스의 작업 ID와 PID를 출력하기 위해 sprint() 함수로 buf에 저장한 다음, write 함수를 사용하여 문자열을 쓰는 것을 알 수 있었다.

```
465             switch (jobs[i].state) {
466                 case BG:
467                     sprintf(buf, "Running    ");
468                     break;
469                 case FG:
470                     sprintf(buf, "Foreground ");
471                     break;
472                 case ST:
473                     sprintf(buf, "Stopped    ");
474                     break;
475                 default:
476                     sprintf(buf, "listjobs: Internal error: job[%d].state=%d ",
477                             i, jobs[i].state);
478             }
479             if(write(output_fd, buf, strlen(buf)) < 0) {
480                 fprintf(stderr, "Error writing to output file\n");
481                 exit(1);
482             }
483             memset(buf, '\0', MAXLINE);
484             sprintf(buf, "%s", jobs[i].cmdline);
485             if(write(output_fd, buf, strlen(buf)) < 0) {
486                 fprintf(stderr, "Error writing to output file\n");
487                 exit(1);
488             }
489         }
490     }
```

그다음, 프로세스의 상태를 확인하여 background 프로세스이면 "Running"을 출력하고, 그 뒤에 입력받은 명령어까지 버퍼에 넣어서 write함수로 출력하는 것을 볼 수 있었다. write가 잘못 실행되었을 경우 에러 메시지를 출력하는 것도 볼 수 있었다.

```
205 int builtin_cmd(char **argv)
206 {
207     char *cmd = argv[0];
208 
209     if(!strcmp(cmd, "quit")){
210         exit(0);
211     }else if(!strcmp(cmd, "jobs")){
212         listjobs(jobs, STDOUT_FILENO);
213         return 1;
214     }
215 
216     return 0;
217 }
```

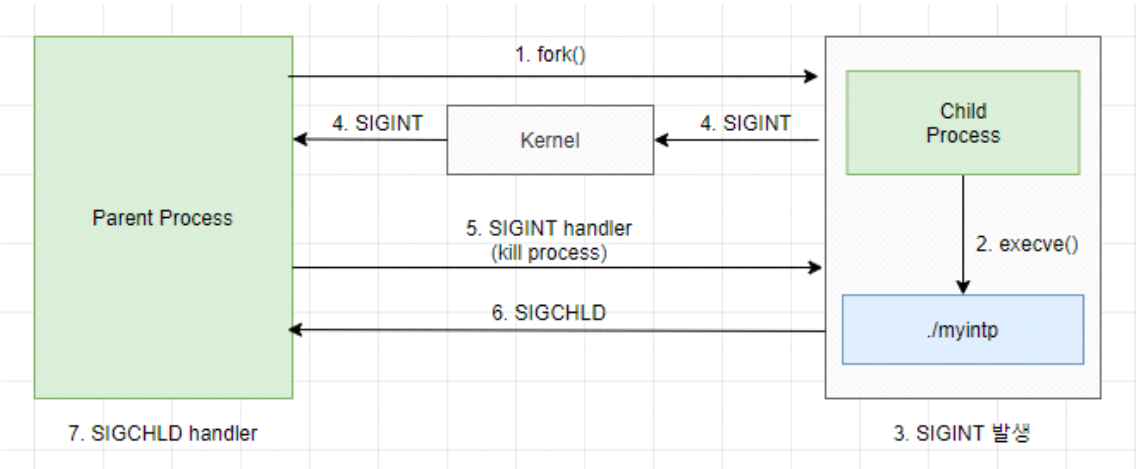
결과적으로, 작업 리스트들을 출력하기 위하여 listjobs 함수를 사용하며, 인자로는 작업 리스트 jobs와, 표준 출력(STDOUT_FILENO)을 전달하는 코드를 작성하였다. 그 다음, 1을 리턴하여 main()의 exit(0) 코드가 실행되지 않고 쉘에서 계속 다음 명령어를 받을 수 있도록 하였다.

Trace 번호 (08)

```
[redacted]@2020sp:~/shlab-handout$ ./sdriver -V -s ./tsh -t 8
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] (17703) terminated by signal 2
tsh> quit

Reference output:
#
# trace08.txt - Send fatal SIGINT to foreground job.
#
tsh> ./myintp
Job [1] (17711) terminated by signal 2
tsh> quit
```

각 trace 별 플로우 차트



trace 해결 방법 설명

trace08은 자식 프로세스가 FG로 실행하는 ./myintp에서 부모 프로세스에게 SIGINT를 전달하면 부모 프로세스에서는 자식 프로세스의 작업을 종료하는 것이 목적이었다.

우선 시그널 사용을 위해 eval() 함수에서 시그널 세팅을 해 주어야 한다.

```
177     sigset_t mask, prev;
178     bg = parseline(cmdline, argv);
179
180     sigemptyset(&mask);         // 시그널 셋 초기화
181     sigaddset(&mask, SIGCHLD);  // 시그널 셋에 SIGCHLD 시그널 추가
182     sigaddset(&mask, SIGINT);   // 시그널 셋에 SIGINT 시그널 추가
183     sigaddset(&mask, SIGTSTP);  // 시그널 셋에 SIGTSTP 시그널 추가
184
185     sigprocmask(SIG_BLOCK, &mask, &prev);   // 시그널 블록
186     if(!builtin_cmd(argv)){
187         if((pid=fork())<0) unix_error("fork error");
188
189         if(pid == 0){
190             sigprocmask(SIG_SETMASK, &prev, NULL);  // 시그널 언블록
```

시그널 셋을 초기화한 다음, SIGCHLD, SIGINT, SIGTSTP 시그널을 사용하기 위해 셋에 추가해 주었다. 그다음, fork()를 실행하기 전에 mask에 대해 블록을 지정하는데, 그 이유는 fork()를 실행하여 자식을 생성하는 동안 kill 시그널을 전달받아 프로세스가 종료될 수 있기 때문에 시그널 셋을 블록하여 이를 막기 위함이다. 또한 부모 프로세스에서는 자식을 fork한 다음 job list에 자식 프로세스를 등록하기 위해 addjob을 실행하는데, 이를 실행하기 전에 자식 프로세스에서 SIGINT를 전달하고, 부모 프로세스가 자식 프로세스를 종료시키면 잘못된 동작이 발생하기 때문에 addjob이 끝나고 난 다음에 부모 프로세스의 시그널 셋을 언블록해주어야 한다(아래 그림의 line 199). 자식 프로세스가 fork되면서 부모 프로세스의 블록된 시그널 셋도 복사하는데, 자식 프로세스는 생성된 다음 시그널을 받을 수 있어야 하므로 190번 줄 코드에서 다시 언블록시켜준다.

```
190             sigprocmask(SIG_SETMASK, &prev, NULL);  // 시그널 언블록
191             if(execve(argv[0], argv, environ) < 0){
192                 printf("%s : Command not found\n\n", argv[0]);
193                 exit(0);
194             }
195         }
196 
197         addjob(jobs, pid, (bg == 1? BG : FG), cmdline);
198         // fork 다음 addjob을 하는 사이의 race condition 제거
199         sigprocmask(SIG_SETMASK, &prev, NULL);  // 시그널 언블록
200 
201         if(!bg){
202             // foreground
203             // fork 후 제어권을 넘겨준 상태에서
204             // 부모 프로세스는 joblist에 있는 foreground job의 pid가
205             // fork한 자식프로세스의 pid와 같은지 계속 확인
206             // deletejob으로 자식 프로세스 job이 joblist에서 제거되면 while문 탈출
207             while(1){
208                 if(pid != fgpid(jobs))
209                     break;
210                 else
211                     sleep(1);
212             }
213         }else{
214             // background
215             printf("(%d) (%d) %s", pid2jid(pid), pid, cmdline);
216         }
217     };
```

이제 자식 프로세스가 실행할 foreground 프로그램을 처리해야 한다. 이전 trace에서는 line 207의 while문 안에서 waitpid() 함수를 사용하여 프로그램의 종료를 기다렸는데, trace08에서는 시그널 핸들러를 사용하므로 시그널 핸들러에서 waitpid()를 사용하여 종료된 자식 프로세스를 거두어들인다. 따라서 eval() 함수의 while문 안에서는, 작업 리스트 jobs에서 foreground 작업이 더 이상 없는지 계속 체크하여, 핸들러에 의해 foreground 작업이 종료되면 break문을 통해 무한루프를 탈출하도록 코드를 작성하였다.

```
18 int main()
19 {
20     signal(SIGALRM, sigalrm_handler);
21     alarm(JOB_TIMEOUT);
22 
23     if (kill(getppid(), SIGINT) < 0) {
24     perror("kill");
25     exit(1);
26     }
27 
28     while(1);
29     exit(0);
30 }
```

자식 프로세스가 실행하는 ./myintp의 코드를 확인하면 23번째 줄에서 부모 프로세스의 PID를 구하여 부모 프로세스에게 kill 함수로 SIGINT를 전달한 다음, 무한루프 상태로 들어가는

것을 볼 수 있다. 커널을 통해 SIGINT를 전달받은 부모 프로세스에서는 SIGINT handler를 통해 자식 프로세스에게 kill 함수로 SIGINT를 다시 전달해줘야 하므로 다음과 같이 sigint_handler() 함수를 작성하였다.

```
278 void sigint_handler(int sig)
279 {
280     pid_t pid = fgpid(jobs);
281     if( pid != 0 )
282         kill(pid, SIGINT);
283     return;
284 }
```

fgpid(jobs)를 실행하여 현재 작업 리스트에서 foreground로 실행되는 작업 프로세스 PID가 있으면(반환값이 0이 아니면) 해당 프로세스에 SIGINT를 전달하도록 구현하였다.
이렇게 하면 자식 프로세스가 다시 SIGINT를 받고 종료되며, 종료되면서 SIGCHLD를 발생시킨다. 그러면 부모 프로세스에서는 SIGCHLD handler를 통해 자식 프로세스의 종료 처리를 해야 한다.

```
252 void sigchld_handler(int sig)
253 {
254     int child_status = 0;
255     pid_t pid;
256     while((pid = waitpid(-1, &child_status, WNOHANG|WUNTRACED)) > 0){   // 자식 프로세스 종료를 대기
257         if(WIFSIGNALED(child_status)){  // 자식프로세스를 종료시킨 시그널이 있는지 체크
258             printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(child_status));
259             deletejob(jobs, pid);
260         }else if(WIFSTOPPED(child_status)){
261             // pid에 해당하는 자식 프로세스의 작업 정보를 jobs로부터 포인터로 얻어서
262             // state에 접근하여 상태를 ST로 변경
263             getjobpid(jobs, pid)->state = ST;
264 
265             printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(child_status));
266         }else{
267             deletejob(jobs, pid);
268         }
269     }
270     return;
271 }
```

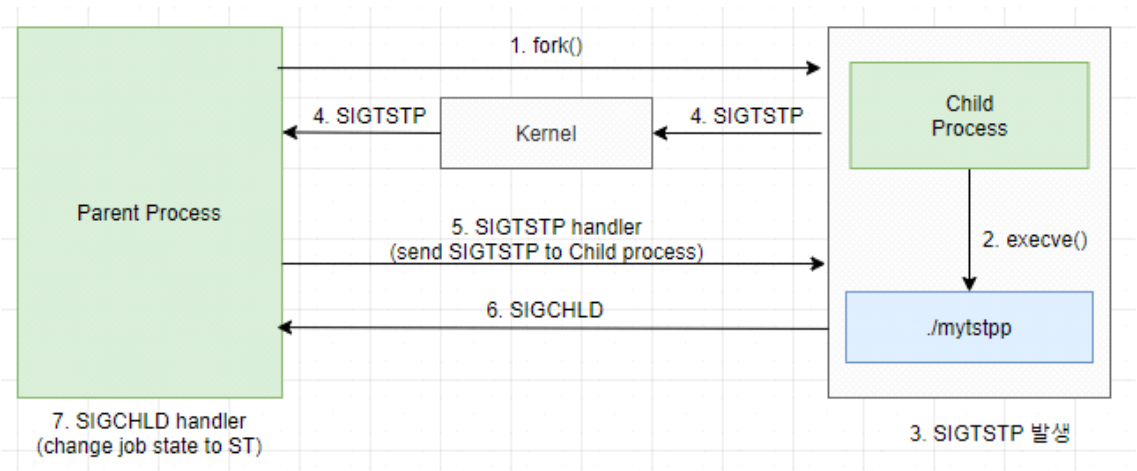
SIGCHLD handler는, waitpid() 함수를 통해 종료되거나 중단된(WUNTRACED) 임의의 프로세스를 기다리고, 종료되거나 중단된 작업이 있으면 해당 pid를 받는다. 그다음, WIFSIGNALED() 함수를 통해 자식 프로세스를 종료시킨 시그널이 있는지 체크한 다음, 있을 경우 종료된 작업의 Job ID, pid, 시그널 정보를 출력하고 deletejob을 통해 작업 리스트에서 자식 프로세스를 제거하도록 구현하였다.

Trace 번호 (09)

```
[redacted]2020sp:~/shlab-handout$ ./sdriver -V -s ./tsh -t 9
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] (6503) stopped by signal 20
tsh> jobs
(1) (6503) Stopped    ./mytstpp

Reference output:
#
# trace09.txt - Send SIGTSTP to foreground job.
#
tsh> ./mytstpp
Job [1] (6511) stopped by signal 20
tsh> jobs
(1) (6511) Stopped ./mytstpp
```

각 trace 별 플로우 차트



trace 해결 방법 설명

trace09는 trace08과 비슷하게, SIGTSTP을 자식 프로세스가 부모 프로세스에게 전달했을 때, 부모 프로세스에서 자식 프로세스의 foreground 작업의 상태를 ST(stop)으로 변경해주는 것이 목적이었다. 아래와 같이 자식 프로세스가 ./mytstpp을 실행하면 부모 프로세스에게 SIGTSTP을 전달하고 무한루프에 들어간다.

```
18 int main()
19 {
20     signal(SIGALRM, sigalrm_handler);
21     alarm(JOB_TIMEOUT);
22
23     if (kill(getppid(), SIGTSTP) < 0) {
24     perror("kill");
25     exit(1);
26     }
27
28     while(1);
29     exit(0);
30 }
```

SIGTSTP을 전달받은 부모 프로세스에서, 자식 프로세스에게 다시 SIGTSTP을 kill 함수를 사용하여 전달하는 것은 trace08과 비슷하게 아래와 같이 sigtstp_handler() 함수에 구현할 수 있었다.

```
291 void sigtstp_handler(int sig)
292 {
293     pid_t pid = fgpid(jobs);
294     if( pid != 0)
295         kill(pid, SIGTSTP);
296     return;
297 }
```

SIGTSTP을 전달받은 자식 프로세스는 SIGTSTP handler가 없으므로 SIGCHLD를 부모 프로세스에게 전달하게 되므로, 부모 프로세스의 SIGCHLD handler에서, SIGTSTP에 의해 SIGCHLD를 보낸 자식 프로세스의 PID를 받아서(중단된 상태임을 체크하기 위해 WIFSTOPPED 함수 사용), 해당 작업의 상태를 ST(stop)로 변경해주는 코드를 sigchld_handler() 함수에 구현하였다. 작업 상태를 변경하기 위해, pid에 해당하는 자식 프로세스의 작업 정보를 getjobpid() 함수를 통해 작업 리스트 jobs로부터 포인터로 얻어서, 해당 포인터를 통해 작업의 멤버 변수 state에 접근하여 상태값을 ST로 변경하였다. 그다음, 해당 프로세스가 어떤 시그널에 의해 중단되었는지 출력하는 출력문을 작성하였다.

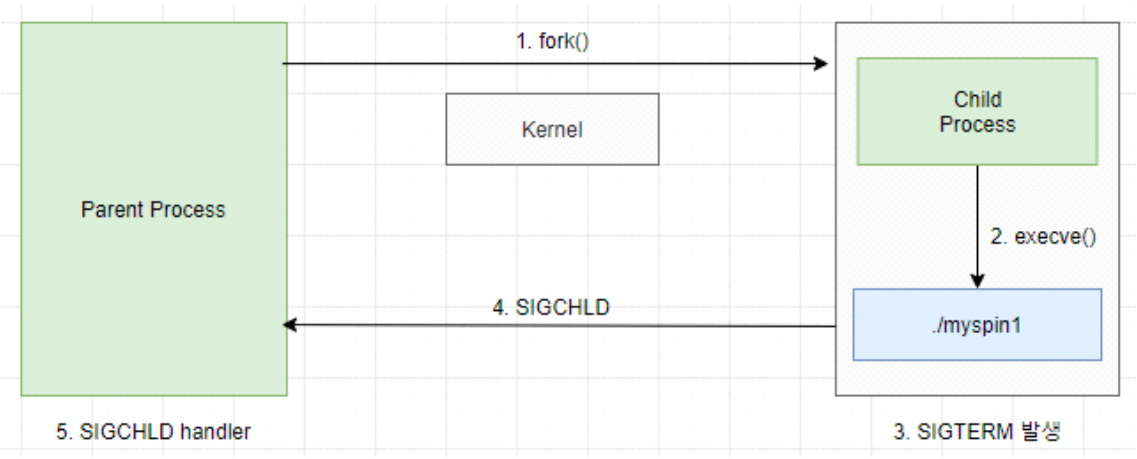
```
256     while((pid = waitpid(-1, &child_status, WNOHANG|WUNTRACED)) > 0){   // 자식 프로세스 종료를 대기
257         if(WIFSIGNALED(child_status)){  // 자식프로세스를 종료시킨 시그널이 있는지 체크
258             printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(child_status));
259             deletejob(jobs, pid);
260         }else if(WIFSTOPPED(child_status)){
261             // pid에 해당하는 자식 프로세스의 작업 정보를 jobs로부터 포인터로 얻어서
262             // state에 접근하여 상태를 ST로 변경
263             getjobpid(jobs, pid)->state = ST;
264
265             printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(child_status));
266         }else{
```

Trace 번호 (10)

```
[redacted]@2020sp:~/shlab-handout$ ./sdriver -V -s ./tsh -t 10
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) (12752) ./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) (12762) ./myspin1 5 &
tsh> /bin/kill myspin1
kill: failed to parse argument: 'myspin1'
tsh> quit
```

각 trace 별 플로우 차트



trace 해결 방법 설명

trace10은 background 작업이 정상 종료될 때 발생하는 SIGCHLD를 받고 background 작업을 종료 처리하는 것이 목적이었다. trace10.txt를 확인해 보면, ./myspin1 이라는 프로그램을 background에서 5초 동안 실행시키고, background 작업이 종료되기 전에 /bin/kill을 통해 SIGTERM(정상 종료 시그널)을 발생시킨다. 작업이 종료되면, SIGTERM(15) 시그널을 받고 작업이 종료되었음을 출력한 다음 작업 리스트에서 제거해야 한다.

즉, background job이 정상 종료될 때 발생하는 SIGTERM에 의해 부모 프로세스에게 SIGCHLD 가 전달되면, 부모 프로세스는 SIGCHLD 핸들러에서 종료된 자식이 정상적으로 종료된 프로세스인지 WIFEXITED() 함수를 사용하여 확인한 후, 정상적으로 종료된 자식이면 deletejob()을 통해 작업 리스트에서 제거해주면 된다. 그런데 이전 trace에서 이미 waitpid()로 받은 종료된 자식 프로세스를 종료 상태에 따라 WIFSIGNALED(어떠한 시그널을 받고 종료된 경우), WIFSTOPPED(정지된 경우) 함수를 사용하여 조건문으로 처리를 구분해 놓았기 때문에, 정상 종료된 경우에 해당하는 작업은 나머지 종류들의 else에 해당하므로 그냥 else 문에서 deletejob()을 수행하는 것으로 충분하다.

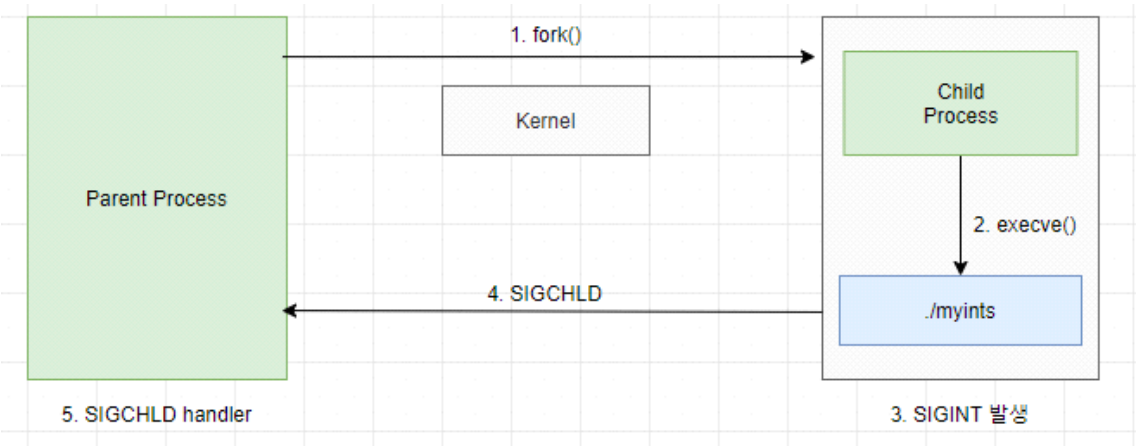
```
252 void sigchld_handler(int sig)
253 {
254     int child_status = 0;
255     pid_t pid;
256     while((pid = waitpid(-1, &child_status, WNOHANG|WUNTRACED)) > 0){   // 자식 프로세스 종료를 대기
257         if(WIFSIGNALED(child_status)){  // 자식프로세스를 종료시킨 시그널이 있는지 체크
258             printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid, WTERMSIG(child_status));
259             deletejob(jobs, pid);
260         }else if(WIFSTOPPED(child_status)){
261             // pid에 해당하는 자식 프로세스의 작업 정보를 jobs로부터 포인터로 얻어서
262             // state에 접근하여 상태를 ST로 변경
263             getjobpid(jobs, pid)->state = ST;
264
265             printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid, WSTOPSIG(child_status));
266         }else{
267             deletejob(jobs, pid);
268         }
269     }
270     return;
271 }
```

Trace 번호 (11)

```
[redacted]@2020sp:~/shlab-handout$ ./sdriver -V -s ./tsh -t 11
Running tracell.txt...
Success: The test and reference outputs for tracell.txt matched!
Test output:
#
# tracell.txt - Child sends SIGINT to itself
#
tsh> ./myints
Job [1] (17733) terminated by signal 2
tsh> quit

Reference output:
#
# tracell.txt - Child sends SIGINT to itself
#
tsh> ./myints
Job [1] (17741) terminated by signal 2
tsh> quit
```

각 trace 별 플로우 차트



trace 해결 방법 설명

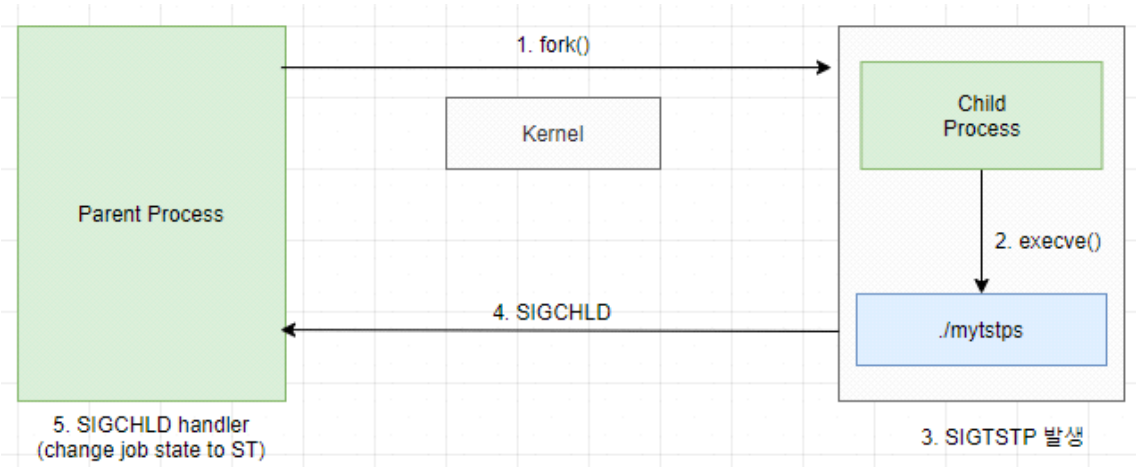
trace11은 자식 프로세스가 스스로에게 SIGINT를 전달하였을 때 부모 프로세스에서 자식 프로세스의 작업을 종료시키는 것이 목적이었다. 그런데 자식 프로세스에는 SIGINT handler가 없기 때문에 스스로에게 SIGINT를 전달하면 스스로 종료되면서 SIGCHLD를 부모 프로세스에게 전달하고, SIGCHLD handler가 구현되어 있는 부모 프로세스는 종료된 자식 프로세스를 정상적으로 처리하게 된다. 따라서 이는 trace08을 통해 이미 구현한 내용에 포함되므로 trace11은 자동적으로 통과하게 된다.

Trace 번호 (12)

```
[redacted]@2020sp:~/shlab-handout$ ./sdriver -V -s ./tsh -t 12
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] (13838) stopped by signal 20
tsh> jobs
(1) (13838) Stopped    ./mytstps

Reference output:
#
# trace12.txt - Child sends SIGTSTP to itself
#
tsh> ./mytstps
Job [1] (13846) stopped by signal 20
tsh> jobs
(1) (13846) Stopped ./mytstps
```

각 trace 별 플로우 차트



trace 해결 방법 설명

trace12는 자식 프로세스가 스스로에게 SIGTSTP를 전달하였을 때 부모 프로세스에서 자식 프로세스의 작업을 종료시키는 것이 목적이었다. 그런데 자식 프로세스에는 SIGTSTP handler가 없기 때문에 스스로에게 SIGINT를 전달하면 자식 프로세스가 중단되어 SIGCHLD를 부모 프로세스에게 전달하고, SIGCHLD handler가 구현되어 있는 부모 프로세스는 중단된 자식 프로세스의 작업 상태(state)를 ST로 바꿔준다. 따라서 이는 trace09를 통해 이미 구현한 내용에 포함되므로 trace12은 자동적으로 통과하게 된다.