**Shell lab report**

20210774 김주은

<lab 강의 내용 요약>

1) Shell (Lab10)

- 셸은 명령어와 프로그램을 실행할 때 사용하는 인터페이스다.



-셸은 커널과 사용자간이 다리 역할을 한 것으로 사용자로부터 명령을 받아 그것을 해석하고 프로그램을 실행하는 역할을 한다.

- shell의 기본 기능

1. jobs : list the running and stopped background jobs

2. bg <job> : change a stopped background job to a running background job

3. fg <job> : change a stopped or running background job to a running in the foreground

4. ctrl+c : deliver SIGINT signal to each process in the foreground job

5. ctrl+z : deliever SIGTSTP signal to each process in the foreground job

2) ECF and Signal handling(Lab 11)

- 프로그램 실행 중 중간에 exception이 발생하면 Os가 관리하고 처리해준다.

- signal이란 exception과 비슷하지만 시스템에서 발생한 event를 process에게 알려주는 것, 즉 exception과 비슷하나 방향이 다르다고 볼 수 있다.

- signal은 각자 signal ID를 가지고, 그 ID는 어떤 event인지 나타낸다.

- signal handler를 설치함으로써 기존의 signal handling을 대체하고 사용자가 지정한 action을 취하게 할 수 있다.

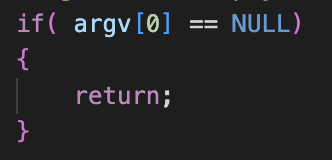
<Shell Lab 풀이 내용>

**(1) eval**

텍스트이(가) 표시된 사진

자동 생성된 설명

먼저, buf에 cmdline의 입력값을 복사하고, buf의 문자열을 잘라서 argument list를 구성한다. (argv)



그 다음, argv[0]이 NULL인지 체크하여 빈 입력값인지 체크하고, 입력값이 비어있는 경우 함수를 종료한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그다음에는 builtin\_cmd인지 아닌지 확인을 하고, 그에 맞게 수행하도록 하는 것이 필요하다. 그래서 builtin\_cmd(argv)를 호출하여 함수를 실행하고 return 값에 따라 수행되도록 한다. Builtin\_cmd(argv)의 결과가 0인 경우, 즉 builin command가 아닌 경우에는 자식 process를 생성한 후 자식 process에서 프로세스를 실행하도록 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

먼저, sigemptyset을 호출하여 signal set을 초기화하고, SIGCHLD를 set에 추가한다. 그 후 SIGCHLD signal을 block한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

fork()를 호출하여 자식 프로세스를 생성하고, 자식 프로세스의 pid로 group pid를 생성한다. 즉, 새로운 프로세스 그룹의 리더가 자기 자신이 되는 것이다. 그리고, SIGCHLD signal을 unblock한다. 이 후에, 입력 받았던 인자리스트와 환경인자를 넘겨서 execve 함수를 실행하고 리턴값이 음수인 경우에는 execve 함수가 에러가 났음을 말하는 것이기 때문에 error 메세지를 출력하고 종료한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그다음에는 bg와 fg의 경우에는 어떻게 처리하느냐가 중요하다. bg라는 변수의 경우 위에 있던 코드에서 parseline 함수의 return값을 저장하고 있는데 cmdline 끝에 &가 붙었는지 저장한 것이다.

&를 붙였으면 background이고, 안 붙이면 foreground 이다. 이를 통해 bg값을 판단하여 조건문으로 실행할 수 있었다. 그래서 foreground의 경우에는 addjob을 통해 현재 process를 job list에 추가하고,

Signal을 unblock한다. 그리고 foreground job이 끝날 때까지 waitfg(pid)를 호출하여 기다린다.

Background의 경우에는 addjob과 unblock하는 경우는 동일하며 여기서는 wait하지 않고 background process의 정보들을 출력한다.

**(2) builtin\_cmd**

argv[0]의 문자열을 비교하여 Builtin\_cmd인 것을 확인하고 맞으면 그에 맞게 함수를 호출하거나 행동을 수행하도록 하는 것이 필요하다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그래서 먼저 quit과 비교하여 quit이라는 입력값이 들어오는 지 확인하고 맞으면 프로그램을 종료한다.

텍스트이(가) 표시된 사진

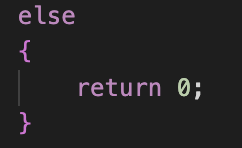
자동 생성된 설명

두번째로는 jobs와 비교하여 jobs인 경우는 listjobs(jobs)를 호출하여 job list를 출력하도록 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

bg랑 fg와 일치하는 지 확인하고 둘 중 하나와 같다면 do\_bgfg(argv)를 호출하여 적절한 행동을 취하도록 한다.



나머지의 경우에는 아무것도 수행하지 않고 종료한다.

**(3) do\_bgfg**

텍스트이(가) 표시된 사진

자동 생성된 설명

먼저, argv[1] 이 NULL이면 id가 입력값으로 안들어온 잘못된 입력값이므로 에러메세지를 출력하고 return 한다.

그 다음에는 id로 입력된 값이 jobid로 들어오는지 pid인지 확인해봐야 한다.

argv[1][0]가 %의 경우, 즉 argv[1]의 첫번째문자가 %의 경우에는 job id이므로, 일치할 경우 job id가 입력된 것이므로 jid라는 변수를 세팅한 후 작업한다.

텍스트, 스크린샷, 화면, 은색이(가) 표시된 사진

자동 생성된 설명

Jid == 0 즉, 잘못 입력된 경우에는 에러메시지를 출력하고 종료한다.

Getjobjid 함수를 호출하여 jid를 가지고 job list에 있는 job을 찾아 job 변수에 저장한다.

텍스트, 스크린샷, 화면, 은색이(가) 표시된 사진

자동 생성된 설명

job 이 NULL인지 확인하여 Job이 없으면 그러한 job이 없다는 메세지를 출력하고 종료한다.

이제 else를 통해 pid로 입력받은 경우이다.

텍스트, 스크린샷, 화면이(가) 표시된 사진

자동 생성된 설명

Jid와 코드 구성은 동일하다.

이제, 이렇게 pid를 설정했으면 argv[0]이 fg인지 bg인지 판단하여 조건문을 통해 각각의 경우에 대해 처리한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

먼저, fg인 경우에는 job->state를 FG로 설정하고, kill함수를 호출하여 해당 Pid의 process에 SIGCONT를 보냄으로써 실행하도록 한다. 그리고 foreground이므로, job->pid의 process가 끝날 때까지 기다린다.

그다음, bg의 경우에는 job->state를 BG로 설정하고, 마찬가지로 SIGCONT signal을 보낸다. 그리고 background이므로, job에 대한 정보만 출력한다.

**(4) waitfg**

먼저, 예외처리로 pid와 job이 제대로된 정보를 담고 있는지 확인하고 아니라면 return한다.

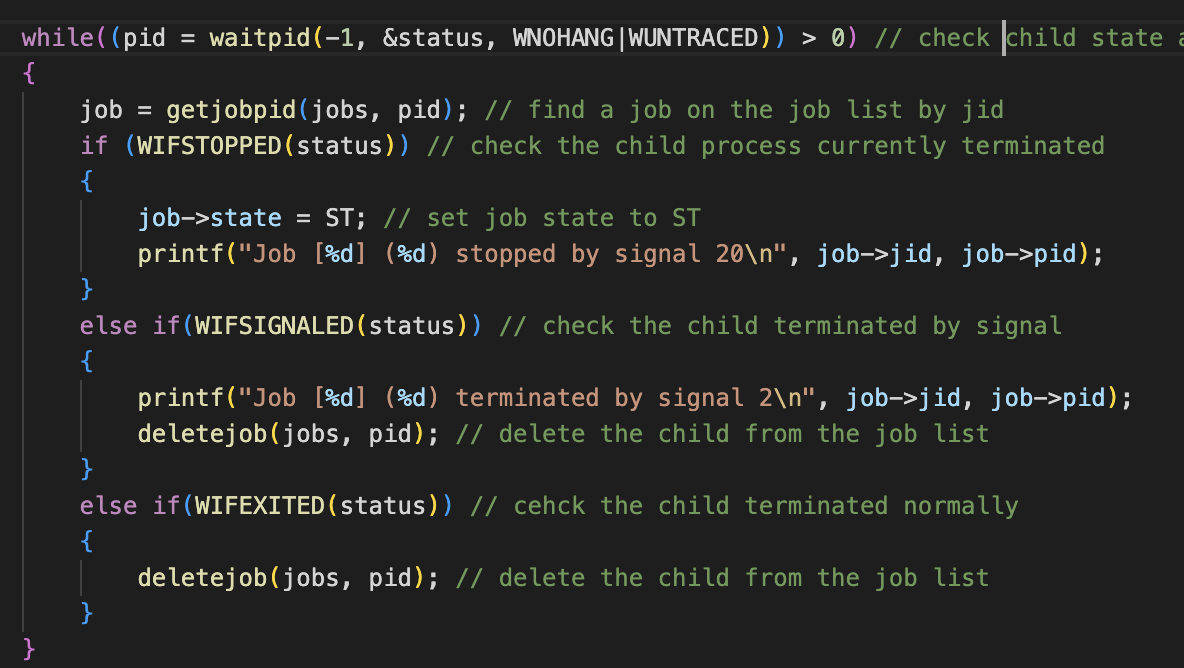
텍스트이(가) 표시된 사진

자동 생성된 설명

이후에 while문을 계속하여 돌리되 fgpid(jobs)가 pid와 다른 경우, 반복문을 나오게 한다. 즉, background인 경우 pid를 return하지 않으므로 다르게 될 것이고, background job인 경우 반복문을 탈출하고 종료한다.

**(5) sigchld\_handler**

먼저 waitpid를 호출하는데 첫번째 인자를 1로 설정하여 호출하므로 임의의 자식 프로세스가 끝나는 것을 기다린다. 그리고 waitpid의 리턴값은 성공하면 pid를 반환하고 오류가 나면 -1을 리턴한다. 그러므로 pid를 반환하는 경우인 0보다 큰 경우로 고려하여 while문의 test를 구성한다.



getjobpid를 호출하여 Joblist에서 pid에 해당하는 job을 찾는다.

그리고 status를 인자로 넘기며 WIFSTOPPED, WIFSIGNALED, WIFEXITED 를 호출하여 각각 child가 signal에 의해 stop되는지, terminate되는지, 정상적으로 terminate되는지 확인한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

WIFSTOPPPED가 true를 return하는 경우 job->state를 ST로 설정하고, 관련 메세지를 출력한다. WIFSIGNALED가 true를 return하는 경우 메세지를 출력하고 joblist에서 child를 삭제한다. WIFEXITED의 경우는 메세지는 출력하지 않은 채 job list에서 child를 삭제한다.

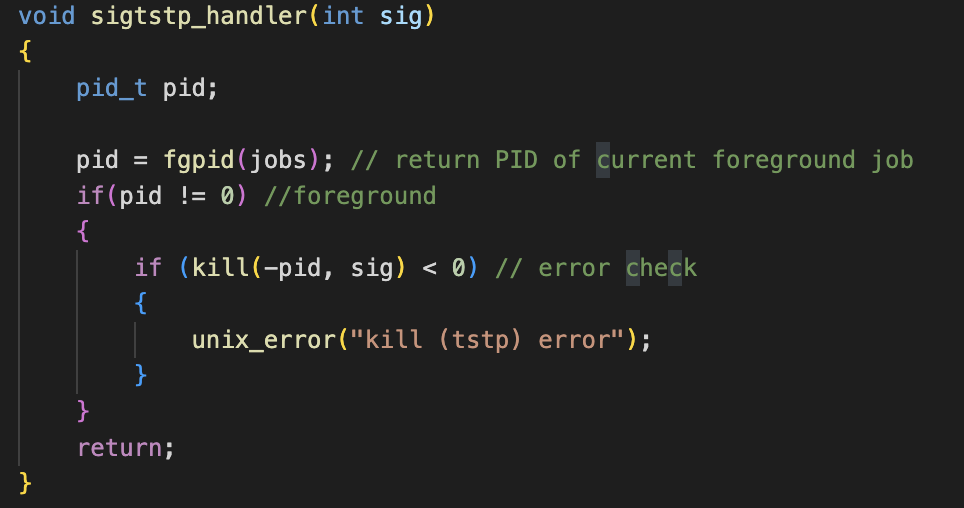
**(6) signt\_handler**

텍스트이(가) 표시된 사진

자동 생성된 설명

pid라는 변수에 fgpid(jobs)의 결과값을 넣어서 현재 foreground job의 pid를 저장한다. pid가 0이 아니면, 즉 foreground job의 경우 Kill 함수를 호출하여 foreground job에 SIGINT signal을 보낸다.

**(7) sigtstp\_handler**



pid라는 변수에 fgpid(jobs)의 결과값을 넣어서 현재 foreground job의 pid를 저장한다. pid가 0이 아니면, 즉 foreground job의 경우 Kill 함수를 호출하여 foreground job에 SIGINT signal을 보낸다. 그리고 kill 함수의 리턴값이 0보다 작을 때는 예외처리를 하여 kill (tstp) error를 출력하도록 한다.

**(8) Analysis the results of test**

위와 같은 방식으로 코드를 짠 후 make test01 ~ make test16라는 명령어를 통해 trace파일들에 대해 tshref와 똑같은 결과를 내는 지 확인한다.

<test01>

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

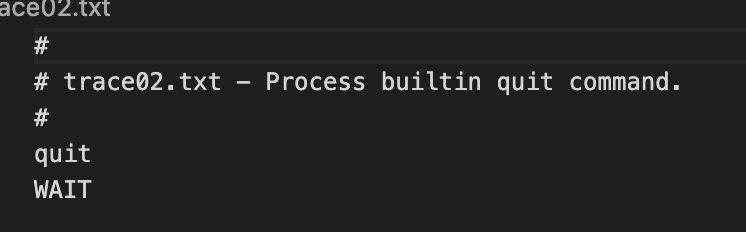
자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

trace01.txt에 대해 테스트한 결과 test01, rtest01의 결과가 동일한 것을 알 수 있다. Close와 wait라는 입력값을 받았을 때는 프로그램을 단지 종료하고 끝낸다.

<test02>



텍스트이(가) 표시된 사진

자동 생성된 설명

‘quit’라는 builin command를 입력 받으므로 quit에 대해 적절한 행동인 프로그램 종료를 진행하는 것이다.

<test03>

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

trace03 파일에 대해서 test, rtest 모두 foreground job을 돌리며 같은 결과를 낸다.

<test04>

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

trace04 파일에 대해서 test, rtest 모두 background job을 돌리며 같은 결과를 낸다.

<test05>

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

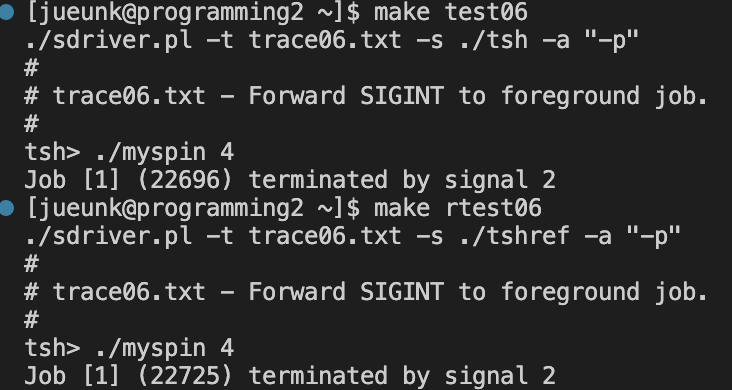
자동 생성된 설명

Trace5파일에 대해서 background process들을 돌리고 jobs명령어를 받아 joblist을 출력하고, 이에 대해 test,rtest의 결과가 동일하다.

<test06>

텍스트이(가) 표시된 사진

자동 생성된 설명



Trace6파일에 대해서 foreground job 을 돌리고 INT입력값을 통해 SIGINT 신호를 보내어 종료시키며 이에 대한 결과는 test, rtest가 같다.

<test07>

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

Trace07에 대해서 background랑 foreground job을 돌리지만 Foreground job에 대해서만 SIGINT 신호가 보내지고 foreground job만 terminate된다. 그리고 이 결과는 test와 rtest가 동일하다.

<test08>

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

Trace08에 대해서 background랑 foreground job을 돌리지만 Foreground job에 대해서만 SIGTSTP 신호가 보내지고 foreground job만 stop된다. 그리고 이 결과는 test와 rtest가 동일하다.

<test09>

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

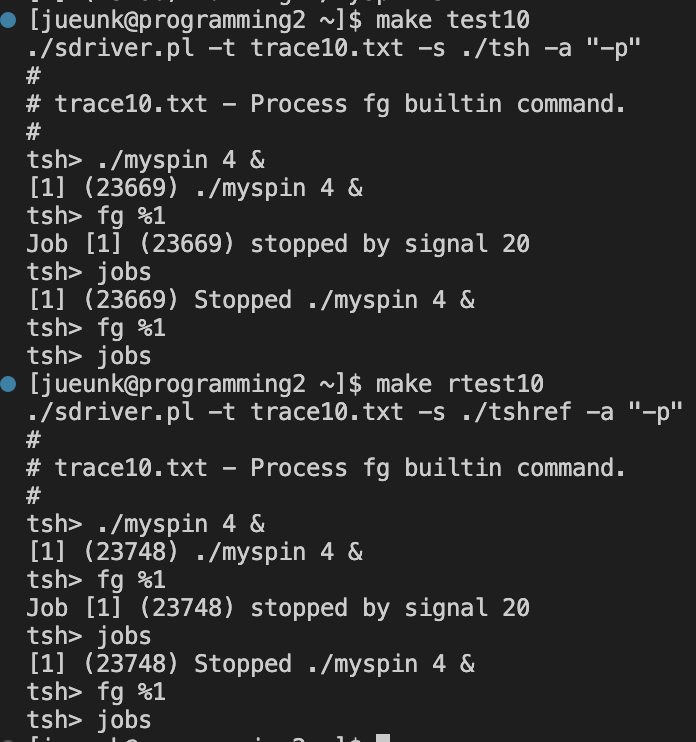
자동 생성된 설명

Trace09.txt에 대해 foreground job과 background job을 실행하고 jobs 명령어를 받아 job list을 출력한다. 이후에 foreground job에 대해 bg 명령어에 대한 처리를 하고 이에 대한 결과를 반영한 job list를 출력한다. 결과는 test,rtest 동일하다.

<test10>

텍스트이(가) 표시된 사진

자동 생성된 설명



trace10.txt의 경우, background job을 실행하고, 이에 대해 fg 명령어에 대한 처리를 한다. Fg 명령어어의 실행 후 foreground job으로 바뀌므로 job list을 출력할 때 출력되지 않는다. 결과는 동일하다.

<test11>

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

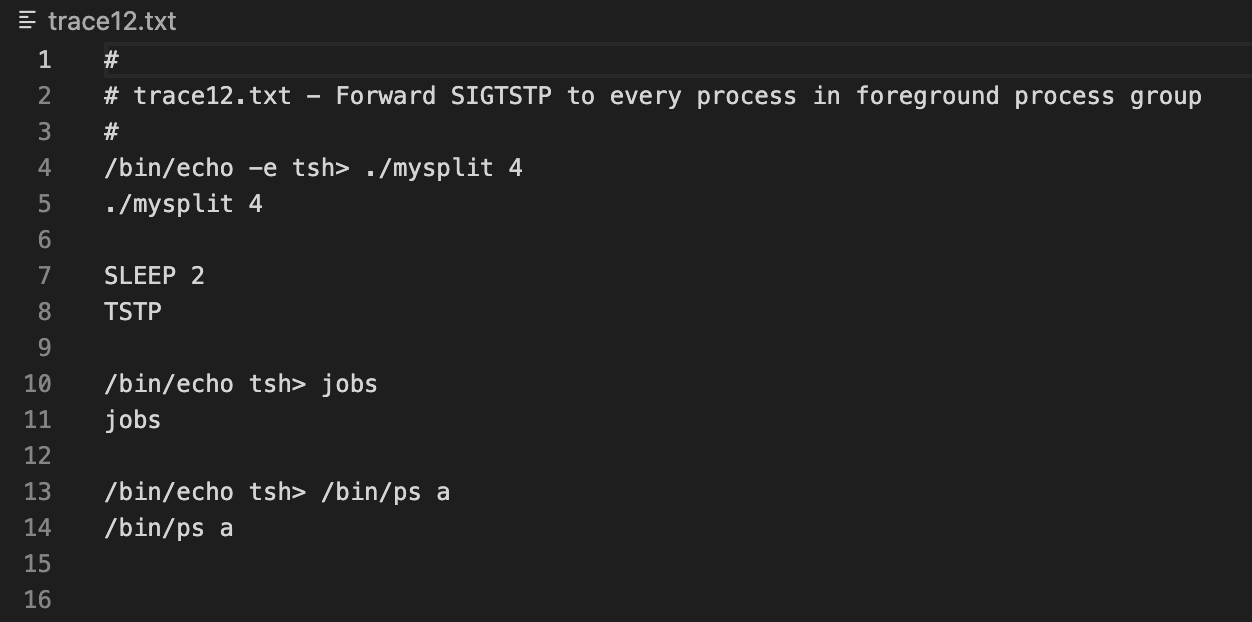
자동 생성된 설명

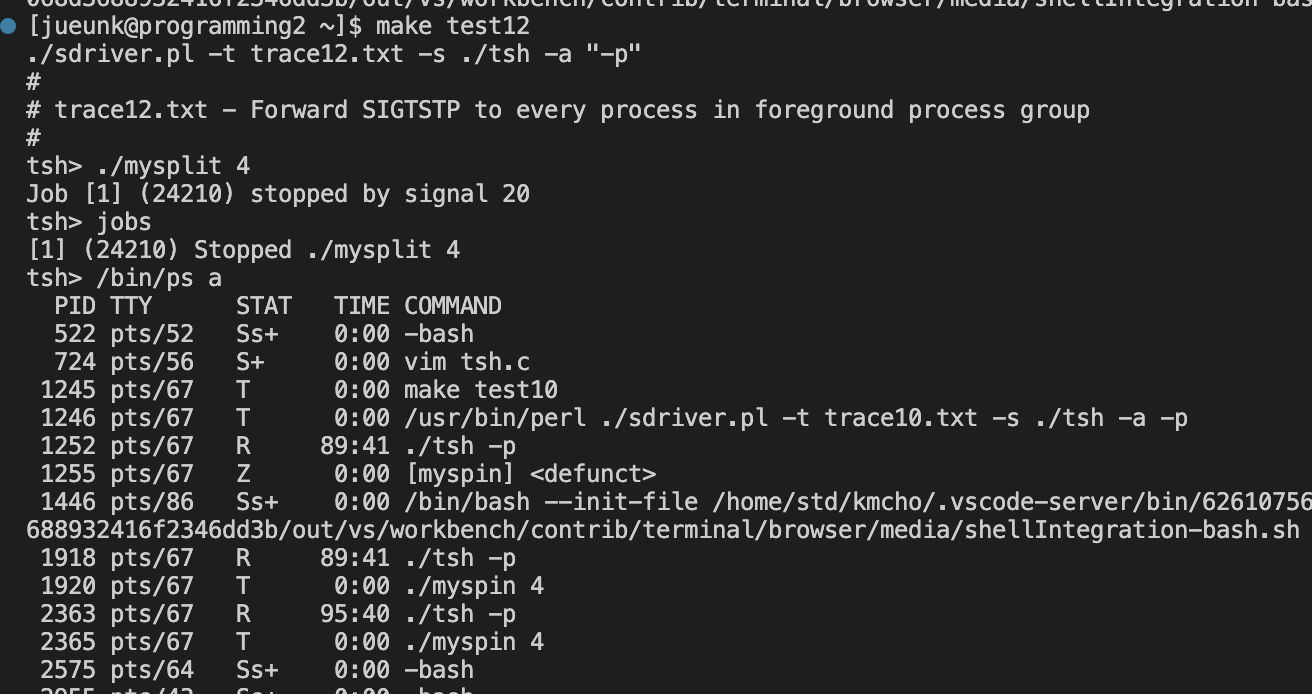
텍스트이(가) 표시된 사진

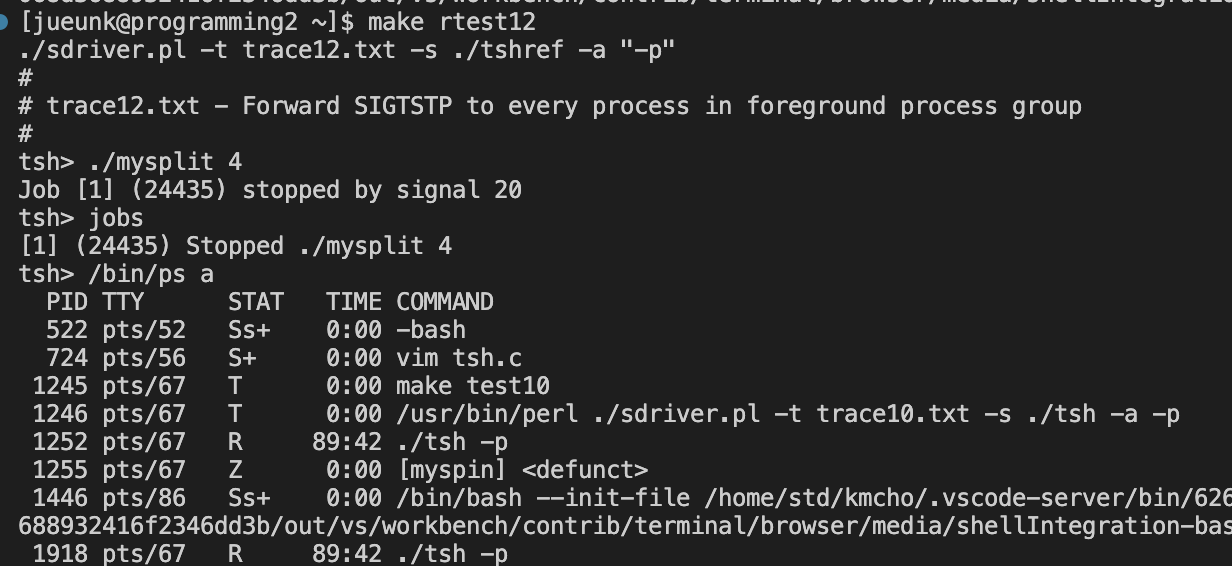
자동 생성된 설명

Trace11.txt에 대해 foreground job이 실행되다가 SIGINT를 받아 종료하고, Ps 명령어를 통해 실행중인 process를 출력한다. Test와 rtest결과 동일.

<test12>





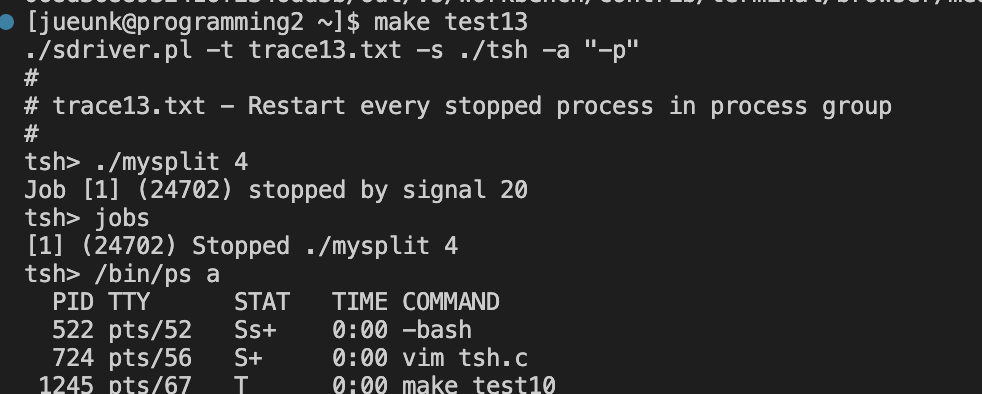


Trace12.txt에 대해 foreground job이 실행되다가 SIGTSTP를 받아 멈추고, jobs라는 명령어를 받고 job list에 대해 출력한다. Ps 명령어를 통해 실행중인 process를 출력한다. Test와 rtest결과 동일.

<test13>

텍스트이(가) 표시된 사진

자동 생성된 설명



텍스트이(가) 표시된 사진

자동 생성된 설명

Trace13.txt에 대해 foreground job이 실행되다가 SIGTSTP를 받아 멈추고, jobs라는 명령어를 받고 job list에 대해 출력한다. Ps 명령어를 통해 실행중인 process를 출력한다. 이 후에 fg %1을 입력받으며 멈췄던 job이 다시 실행한다. Test와 rtest결과 동일.

<test14>

텍스트이(가) 표시된 사진

자동 생성된 설명



Trace 14 파일의 결과에 따르면, fg bg 와 같은 명령어 뒤에 다른 입력값이 안들어오는 에러가 발생하게 되고, 이를 처리하는 결과가 test, rtest가 동일하다.

<test15>

텍스트이(가) 표시된 사진

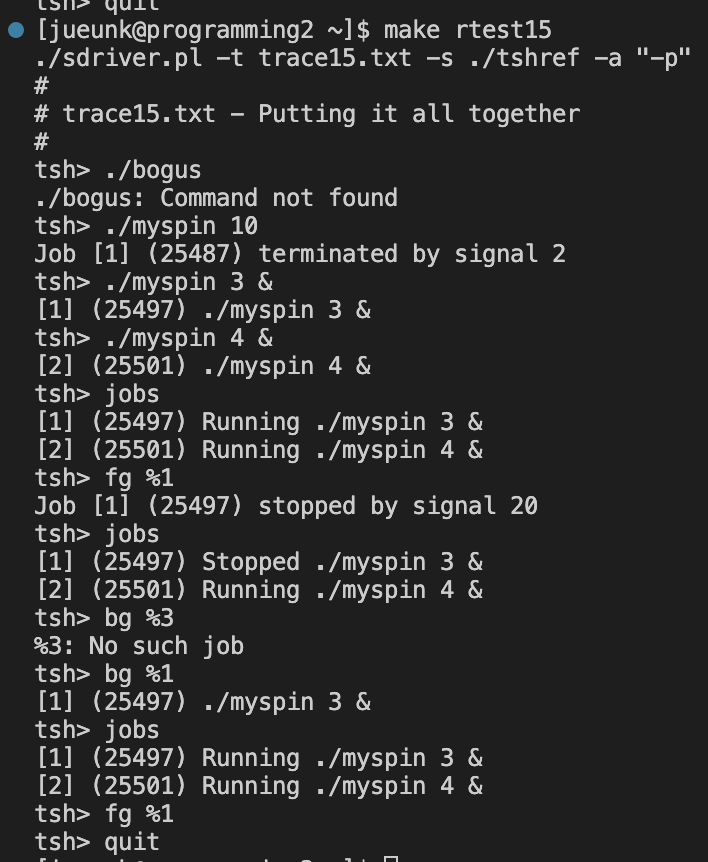
자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

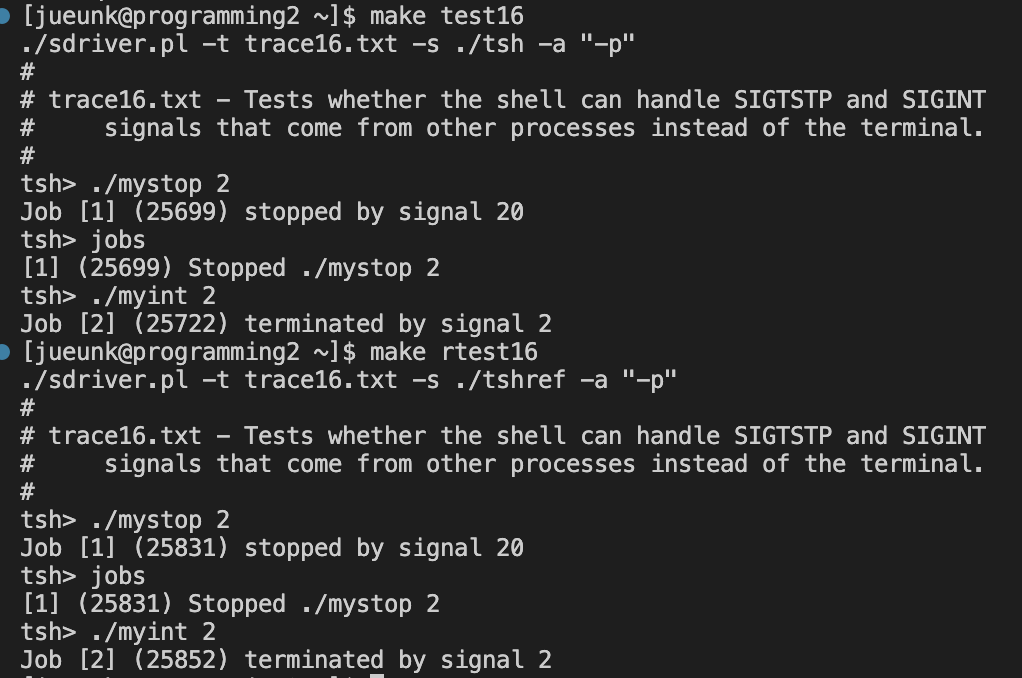


Trace15.txt의 내용에 따라 background job들이 실행되고, job list를 출력할 때 그들이 출력된다. 그리고 fg %1와 bg %1의 입력에 따라 job들의 running or stopping 상태가 바뀐다. 결과는 test, rtest가 동일.

<test16>

텍스트이(가) 표시된 사진

자동 생성된 설명



Trace16.txt의 내용에 따라 ./mystop과 ./myint를 실행시키는 것을 알 수 있으며 거기서 signal을 보내면 이것을 받아 처리한다. 결과는 test, rtest 동일