시스템 프로그래밍 실습가이드

Shell Lab

2021년

1. 개요

이번 실습의 목적은 프로세스 제어와 시그널링 개념에 보다 익숙하게 하는 것이다. 여러분은 job 제어를 지원하는 리눅스의 간단한 쉘 프로그램을 작성하게 된다.

2. 실습방법

shlab-handout.tar.gz 파일을 작업을 하려는 디렉토리에 복사한다. 그리고 다음을 수행한다

- tar xvfz shlab-handout.tar를 실행해서 압축을 푼다
- make를 실행해서 테스트 루틴들을 compile&link한다
- tsh.c의 맨 위에 자신의 학번과 이름을 쓴다.

tsh.c(tiny shell) 파일을 보면, 간단한 리눅스 쉘의 함수골격이 들어 있는 것을 알 수 있다. 여러분이 실습을 시작할 수 있도록, 일부 덜 재미있는 함수들은 이미 구현해 놓았다. 여러분이 해야할 일은 아래에 나열한 남아 있는 빈 함수들을 완성하는 것이다. 여러분들이 구현할 때, 참고할 수 있도록 우리가 이미 구현한 정답 코드의 길이를(주석을 포함해서) 표시하였다.

- eval : 커맨드 라인을 해석하고, 파싱하는 메인 루틴(70줄)
- builtin_cmd : 내장 명령들 (quit, fg, bg, jobs)을 해석하고 인식한다(25줄)
- waitfg : foreground job이 완료되기를 기다린다(20줄)
- sigchld_handler : SIGCHILD 시그널을 잡는다(80줄)
- sigint_handler : SIGINT(ctrl-c) 시그널을 잡는다(15줄)
- sigtstp_handler : SIGTSTP(ctrl-z) 시그널을 잡는다(15줄)

tsh.c 파일을 수정할 때마다 make을 실행해서 재컴파일 해야한다. 여러분이 만든 쉘을 실행하려면 tsh을 커맨드라인에 입력해야한다.

unix> ./tsh

리눅스 쉘에 대한 설명

셸은 사용자를 대신하여 프로그램을 실행하는 대화형 명령줄 변환기다. 쉘은 반복적으로 프롬프트를 출력하고 stdin 장치(키보드)에서 명령줄을 기다리고, 명령줄의 입력 내용에 따라 특정 명령을 실행한다.

명령줄은 ASCII 텍스트 단어들의 연속이며, 이들 단어는 공백으로 분리된다. 명령줄의 첫 단어는 내장(built-in)명령이거나 어떤 실행파일의 경로다. 이후에 남은 단어들은 명령줄의 인자들이다. 만일 첫번째 단어가 내장명령이면, 쉘은 즉시 현재 프로세스 내에서 이 명령을 실행한다. 그렇지 않다면, 이 단어는 실행 프로그램의 경로로 간주한다. 이 경우, 쉘은 자식 프로세스를 하나 포크하고, 이 자식의 문맥 내에서 이 프로그램을 로딩하고 실행한다. 한 개의 명령줄을 해석해서 만들어진 자식 프로세스들은 한 개의 작업(job)이라고 부른다. 일반적으로 작업은 리눅스 파이프로 연결된 다수의 자식 프로세스들로 구성된다.

만일 명령줄이 "&" 심볼로 끝난다면, 이 작업은 후면(background)작업이며, 쉘은 이 작업이 종료될 때까지 기다리지 않고, 곧바로 다음 명령줄의 프롬프트를 인쇄하고 다음 명령을 기다린다. 그렇지 않다면, 이 작업은 전면(foreground)작업이며, 쉘은 다음 명령줄을 대기하기 전에 이 작업이 종료될 때까지 기다린다. 그래서, 언제든지 전면에는 최대 한 개의 작업만이 실행되고 있을 수 있다. 그렇지만, 후면에는 여러 개의 작업들이 실행 되고 있을 수 있다.

예를 들어, 다음의 명령줄은 내장명령 jobs를 실행한다.

tsh>jobs

다음의 명령은

tsh> /bin/ls –l –d

ls 프로그램을 전면에서 실행한다. 관습에 의해 쉘은 프로그램이 메인 루틴을 실행할 때 argc와 argv 인자들을 다음과 같은 값으로 초기화한다.

int main(int argc, char **argv)

- argc == 3
- argv[0] == "/bin/ls"
- argv[1] == "-l"
- argv[2] == "-d"

반대로, 다음의 명령줄은 ls 프로그램을 후면에서 실행한다.

tsh> /bin/ls –l –d &

리눅스 쉘은 작업제어 job control 개념을 지원하며, 이것은 유저들이 작업을 전면과 후면으로 이동할 수 있도록 해주며, 하나의 작업내의 프로세스의 상태(running, stopped, terminated)를 변경할 수 있도록 해준다. ctrl-c 를 누르면 SIGINT 시그널이 전면작업내의 각 프로세스들에 전달된다. SIGINT 의 기본동작은 해당 프로세스를 종료terminate 하는 것이다. 유사하게, ctrl-z를 누르면 SIGTSTP시그널이 전면작업 내의 모든 프로세스에 전달된다. SIGTSTP에 대한 기본 동작은 프로세스를 정지상태stopped로 만들어 주며, SIGCONT 시그널이 올때까지 이상태를 유지한다. 쉘은 다음과 같은 다양한 내장명령을 제공한다.

- jobs : running, stopped 상태의 후면 작업들을 보여준다

- bg <job> : stopped 후면 작업을 running 후면 작업으로 변경
- fg <job> : stopped, running 후면 작업을 running 전면 작업으로 변경
- kill <job> : 작업을 종료

tsh 규격

여러분이 구현하는 tsh 쉘은 다음의 기능을 구현해야 한다:

- 프롬프트는 "tsh "로 시작해야 한다
- 유저가 입력하는 명령줄은 한 개의 name과 0 또는 1개 이상의 인자로 구성되어야 하며, 이들은 모두 스페이스로 띄어서 입력한다. 만일 name이 built-in명령이면, tsh은 즉시 처리하고, 다음 명령을 기다려야 한다. 그렇지 않다면, tsh은 name이 어떤 실행파일의 경로라고 간주하고, 이 파일을 로드하고 최초 자식 프로세스의 문맥 내에서 실행한다(이 경우, job은 이 최초의 자식 프로세스를 말한다)
- tsh은 파이프(|)나 I/O재설정(<, >)은 지원하지 않는다
- ctrl-c(ctrl-z)를 누르면 SIGINT(SIGTSTP) 시그널을 현재 전방작업에 보내야 하며, 이 작업의 자손들에게도 보내야 한다(예. 이들이 fork한 자식 프로세스들). 만일 전방작업이 없다면, 이 시그널은 아무 영향을 주지 않는다
- 만일 명령줄이 &로 끝난다면, tsh은 이 작업을 후방작업으로 실행한다. 그 외에는, 작업을 전방작업으로 실행해야 한다.
- 각 작업은 프로세스 ID(PID) 또는 작업 ID(JID)로 구분할 수 있으며, 이 값들은 tsh이 할당하는 양의 정수다. JID는 명령줄에서 접두어"%"로 나타내야 한다. 예를 들어, "%5"는 JID 5를 의미하고, "5"는 PID 5를 의미한다. (여러분이 작업 리스트를 관리하기 위해 필요한 모든 루틴을 제공하였다)
- tsh은 다음의 내장명령어들을 구현해야 한다.
  - quit 명령어는 쉘을 종료한다
  - jobs는 모든 후방작업들을 나열한다
  - bg <job>는 <job>을 SIGCONT 시그널을 보내서 다시 진행시키고, 이것을 후방작업으로 실행한다. <job> 인자는 PID 나 JID가 될 수 있다

- fg <job>명령은 <job>을 SIGCONT 시그널을 보내서 다시 진행시키고, 이것을 전방 작업으로 실행한다. <job> 인자는 PID 나 JID가 될 수 있다

- tsh은 자신의 모든 좀비들을 제거해야한다. 만일 어떤 작업이 자신이 붙잡지 않은 어떤 시그널에 의해 종료되려면, tsh은 이러한 사건을 인식하고 이 작업의 PID와 수신한 시그널에 대한 설명을 출력해야만 한다.

구현한 tsh을 검증하는 방법

정답 쉘 : tshref 프로그램은 미리 구현해둔 tsh의 실행코드다. 여러분의 쉘이 어떻게 동작해야 하는지 잘 모를 때, 이 프로그램을 실행하라. 여러분의 쉘은 이 정답 쉘과 동일한 출력을 만들어야 한다(PID는 다를 수 있다, 물론)

쉘 구동기 : sdriver 프로그램은 한 개의 쉘을 자식 프로세스로 실행해서 trace 파일에 의해 지시된 명령과 시그널을 이 프로세스에 보내주고, 이 쉘에서 나오는 출력을 캡쳐해서 화면에 뿌려준다. –h 인자를 이용해서 sdriver 의 사용법을 익혀라.

```
sys01@systemprogramming17:~/lab08/shlab-handout$ ./sdriver -h
Usage: sdriver [-hV] [-s <shell> -t <tracenum> -i <iters>]
Options
        -h            Print this message.
        -i <iters>    Run each trace <iters> times (default 2)
        -s <shell>    Name of test shell (default ./tsh)
        -t <n>        Run trace <n> only (default all)
        -V            Be more verbose.
sys01@systemprogramming17:~/lab08/shlab-handout$
```

여러분의 쉘이 정확한가를 테스트 할 수 있는 22 trace 파일을 제공한다. 낮은 숫자의 trace파일은 아주 간단한 테스트를 실행하며, 높은 숫자는 좀더 복잡한 테스트를 실행한다.

trace00.txt 파일을 다음과 같이 이용한다:

```
./sdriver -Vt -00 -s ./tsh
```

이와 비슷하게 여러분의 쉘을 정답쉘과 비교하기 위해 다음과 같이 sdriver를 이용할 수 있다

```
./sdriver -Vt -00 -s ./tshref
```

trace 파일의 좋은 점은 여러분이 쉘을 일일이 실행하는 것과 동일한 실행 출력을 만든다는 것이다(trace라는 것을 알려주는 최초의 주석문을 제외하고). 다음의 실행결과를 참고하라.

```
Running trace15.txt...
Success: The test and reference outputs for trace15.txt matched!
Test output:
#
# trace15.txt - Process bg builtin command (one job)
#
tsh> ./mytstpp
Job [1] (30587) stopped by signal 20
tsh> bg %1
[1] (30587) ./mytstpp
tsh> jobs
(1) (30587) Running ./mytstpp

Reference output:
#
# trace15.txt - Process bg builtin command (one job)
#
tsh> ./mytstpp
Job [1] (30596) stopped by signal 20
tsh> bg %1
[1] (30596) ./mytstpp
tsh> jobs
(1) (30596) Running ./mytstpp
```

```
sys01@systemprogramming17:~/lab08/shlab-handout$ ./tshref
eslab_tsh> ./mytstpp
Job [1] (30915) stopped by signal 20
eslab_tsh> bg %1
[1] (30915) ./mytstpp
eslab_tsh> jobs
(1) (30915) Running ./mytstpp
eslab_tsh>
```

힌트

- 교재의 8장의 모든 글자 한자까지 철저히 읽을 것.
- 쉘 개발할 때 trace 파일을 잘 활용할 것. trace01.txt 부터 시작해서 여러분의 쉘이 정답 쉘과 완전히 동일한 출력을 만들도록 할 것. 그 후에 trace02.txt 를 구현하고 하는 식으로 진행할 것
- waitpid, kill, fork, execve, setpgid, sigprocmask 함수가 매우 익숙해질 것이다. waitpid의 WUNTRACED와 WNOHANG 옵션들도 유용하게 사용된다

- 시그널 핸들러를 구현할 때, SIGINT와 SIGTSTP시그널을 전체 전면 프로세스 그룹에 전송되도록 kill함수의 옵션으로 –pid 를 이용해야 한다. pid가 아니라는 점에 주의해라. sdriver.pl 프로그램은 이런 에러를 테스트해준다.

- 이 실습의 한가지 까다로운 점은 waitfg와 sigchld_handler 함수 사이에 일을 어떻게 나눠야 할지를 결정하는 것이다. 다음의 방식을 추천한다:

  - waitfg 에서는 sleep함수 주위에 계속 기다리는 루프를 구현한다
  - sigchld_handler에서는 단 한번의 waitpid를 실행한다.

  waitpid를 waitfg와 sigchld_handler 모두에서 호출하는 방식도 가능하지만, 이렇게 구현하면 혼란스러워진다. 모든 좀비 제거는 이 핸들러에서 처리하는 것이 더 간단하다.

- eval 에서, 부모는 sigprocmask를 이용해서 자식을 fork하기 전에 SIGCHLD를 블록시켜야 한다. 그리고 나서 자식 프로세스를 addjob을 호출해서 작업 리스트에 추가한 후에 sigprocmask 함수를 이용해서 이 시그널을 블록해제해야한다. 자식들은 부모로부터 blocked 벡터를 이어받기 때문에 자식은 자신이 새로운 프로그램을 execve 하기 전에 SIGCHLD 시그널을 블록해제해야 한다. 부모는 SIGCHLD시그널을 이런식으로 블록할 필요가 있는데, 그 이유는 자식이 sigchld_handler에 의해 부모가 addjob을 호출하기 전에 좀비제거되는(그리고 작업리스트에서 제거되는) 곳에서 경쟁상태(race condition)을 회피해야하기 때문이다.

- more, less, vi와 같은 프로그램들은 터미널 설정에 따라 이상하게 동작할 수 있으므로, 여러분의 쉘에서는 이 명령들은 실행하지 말아야 한다. /bin/ls, /bin/ps, /bin/echo 와 같은 명령만 이용하라.

- 여러분의 쉘을 리눅스 쉘에서 실행할 때, 여러분이 만든 쉘은 전방 프로세스그룹에서 실행된다. 만일 여러분의 쉘이 자식 프로세스 한 개를 생성하면, 기본적으로 이 자식은 전방프로세스 그룹의 멤버가 된다. ctrl-c를 키보드에서 누르면, SIGINT가 전방프로세스 그룹의 모든 프로세스에 전송되므로, ctrl-c를 누르게 되면, SIGINT는 여러분의 쉘로 전송될 뿐만 아니라, 여러분의 쉘이 만든 모든 프로세스에게도 전달되는데, 이것은 분명히 잘못된 것이다. 이것을 해결하는 방법은 다음과 같다 : fork 후에, 그리고 execve 전에 자식프로세스는 setpgid(0, 0)을 호출해야 한다. 이것은 자식을 자식의 PID와 동일한 그룹 ID를 갖는 새로운 프로세스 그룹에 포함시킨다. 이렇게 하면 전방프로세스 그룹에는 여러분의 쉘 딱 한 개만 남게 된다. 이때, ctrl-c를 누르면, 여러분의 쉘은 그 결과로 생성된 SIGINT를 붙잡아야 하고, 이것을 적절한 전방작업(보다 정확히는 전방작업을 포함하는 프로세스 그룹)에게 이것을 전달해야 한다.