M1522.000800 System Programming
Fall 2018

System Programming Shell Lab Report

Daeeun Kim
2014-15395

1. Shell Lab

Shell Lab은 간단한 Unix Shell을 작성해 보며 기본적인 프로세스 실행과 관리, 시그널 처리에 대한 이해를 높이는 것이 목적이다.

2. Implementation

먼저 CSAPP 교재에 나와있는 eval 함수대로 구현을 하고 수정하는 방향으로 시작했다. execve함수를 이용한 프로그램 실행 부분은 그대로 유지했으며, built-in command와 bgfg전환 함수, waitfg함수를 이용해 프로세스를 관리한다. 또한 ctrl+c, ctrl+z입력 각각에 대해 interrupt와 stop 시그널을 foreground프로세스에 전달하기 위한 핸들러와, 프로세스 종료 관리를 위한 핸들러를 사용했다. 마지막으로, system-call 함수에 대한 에러 처리를 위해 wrapper 함수를 만들어 기존의 system-call을 대체했다.

2.1 eval

eval 함수는 CSAPP교재에 나와있는 함수를 그대로 구현한 뒤, 이번 shell lab 스펙에 맞도록 수정하는 방향으로 진행했다. 먼저, 이번에 구현해야 할 tsh는 실행되고 있는 job을 관리하기 때문에 execve를 통해 프로세스가 실행되고 난 뒤, addjob을 통해 job 리스트에 프로세스를 추가하도록 수정했다. 그리고 &를 이용한 background 실행 커맨드가 들어왔을 때와 명령어를 찾을 수 없을 때(Command not found) 쉘의 동작을 스펙에 맞도록 변경해주었다. 또한 SIGINT, SIGTSTP 시그널 처리시에 process group id를 이용하게 되기 때문에 shell과 다른 프로세스 그룹에 속하게 만들어야 했다. 따라서 fork직후, execve함수 실행 전에 setpgid(0, 0)함수를 이용하여 프로세스의 pgid를 그 자신의 pid로 변경하였다.
이렇게 작성한 shell을 실행하고 습관적으로 ls, ps와 같은 명령어들을 입력했는데, 이 명령들에 대해 Command not found는 잘 출력되었지만, jobs를 통해 프로세스 목록을 열어보면 실행에 실패한 프로그램들(ls, ps)이 job list에 들어있음을 확인할 수 있었다. 이는 프로그램 실행 실패를 따로 구분하지 않고 항상 addjob을 수행했던게 원인이었다. Shell의 관점에서는 프로그램이 정상적으로 실행되었다는 신호를 받을 수 없기 때문에 execve함수가 실패하고 exit을 통해 child프로세스가 종료되었는지를 확인해야했다. child프로세스 종료 확인이 addjob함수보다 나중에 실행되어야 정상적으로 job을 삭제할 수 있기 때문에 fork전에 SIGCHLD 시그널을 block하고, fork와 execve를 진행하도록 하였다. child프로세스에서는 fork직후에 해당 시그널을 다시 unblock했고, 기존 shell 프로세스에서는 addjob을 수행한 뒤에 SIGCHLD를 unblock하였다. 이렇게 구현하게 되면, 프로세스 실행 여부와 관계없이 job을 추가하지만, 프로세스가 종료되자마자 다시 리스트에서 삭제하기 때문에 프로세스 관리가 정상적으로 이루어진다.

함수 마지막 부분에서 foreground와 background여부에 따라 shell의 작동 방식을 결정하게 되는데, foreground일 때는 waitpid대신 waitfg함수를 호출하는걸로 변경했고, background일 때의 출력메세지를 tshref스펙과 동일하게 변경하였다.

2.2 builtin_cmd

이번 랩에서 구현하는 빌트인 커맨드는 quit, jobs, bg, fg 이렇게 네 종류이다. CSAPP교재에 나와있는 builtin_command함수를 가져와서 jobs, bg, fg명령에 대한 처리를 추가해 주었다. jobs의 경우, 미리 구현된 listjobs 함수를 이용했으며, bg나 fg가 들어왔을 때는 do_bgfg함수를 호출하도록 구현했다. eval에서 올바른 리턴값을 받을 수 있도록 빌트인 커맨드가 정상적으로 실행되었다면 1을 리턴하고, 아닌 경우 0을 리턴하도록 구현하였다.

2.3 do_bgfg

bg와 fg커맨드 모두 job id 혹은 process id를 인자로 받아야 하기 때문에 argv[1]을 확인하여 atoi함수를 이용해 jid 와 pid를 추출해냈고, getjobjid, getjobpid를 이용해 인자로 받은 프로세스의 job 구조체를 얻었다. 이 구조체로부터 pid값을 가져와 해당 프로세스 그룹에 SIGCONT 시그널을 보내고, 해당 job의 state를 커맨드에 맞게 변경해주었다.
에러 핸들링에 관련된 14번 trace 테스트를 수행하면서, fg와 bg 커맨드에 잘못된 인자가 들어오는 경우 각각에 대한 에러메세지를 tshref프로그램과 같은 출력값을 내도록 수정하였다. 에러케이스는 크게 네 가지로, 인자가 없는 경우 (argv[1] == NULL), '%'로 시작하지만 job을 찾을 수 없는 경우, 숫자로 시작하지만 해당 pid를 job목록에서 찾을 수 없는 경우 그리고 마지막으로 argv[1]이 숫자 혹은 %로 시작하지 않는 경우이다.

2.4 waitfg / signal handlers

waitfg함수의 최초 구현에서는 교재 eval함수 마지막부분에 waitpid함수를 사용했던 코드를 그대로 함수로 옮기기만 하는 형태로 구현했다. 또한, 시그널 전달에 있어서는 SIGINT와 SIGTSTP시그널이 들어왔을 때, kill 함수를 통한 시그널 전달 성공 여부에 따라 각각의 signal handler에서 job을 삭제하거나 상태를 변경하도록 구현하였으며, sigchld_handler 함수는 비어있는 채로 구현하였다. 하지만 background 프로세스가 시그널 없이 정상 종료되었을 때도 처리를 해주어야 했기 때문에 sigchld핸들러에 waitpid함수를 추가하였다.
이렇게 구현하고 실행한 뒤, 터미널을 통해 시그널을 보냈을 때, 시그널이 제대로 처리되지 않음은 물론이고 쉘이 아무런 출력도 없이 정지된것같은 동작을 보였다. 명령어 하나하나 실행 과정을 분석한 결과, waitfg와 sigchld_handler 두 곳에서 모두 waitpid system-call을 사용하고 있었기 때문에 프로세스가 정지한것처럼 나타났었음을 알았다. 그래서 child프로세스에 대한 모든 처리를 sigchld핸들러로 통합했고, sigint, sigtstp핸들러에서는 단순히 child 프로세스

그룹으로 signal만 전달하는 형태로 변경하였다. 새로 구성된 sigchld_handler는 WNOHANG과 WUNTRACED를 사용하여 이미 정지된 프로세스에 대한 처리가 가능하도록 구현하였다. 또한 SIGINT와 SIGTSTP 시그널에 대해서는 WIFSIGNALED, WIFSTOPPED매크로를 이용해 job에 대한 처리를 해주었다. 마지막으로, 여러 개의 child 프로세스가 terminated될 경우를 처리하기 위해 while루프에서 waitpid의 리턴값이 양수인지 체크하도록 하였다.
마지막으로 waitfg함수는 waitpid없이 구현해야했다. sigchld핸들러에서 job에 대한 모든 처리를 마치기 때문에 waitfg함수에서는 인자로 받은 pid가 foreground인지만 확인하면 되었다. 따라서 while반복문에서 fgpid를 이용해 현재 foreground인 pid와 비교하도록 구현했으며, cpu 리소스 낭비를 막기 위해 sleep(1)을 추가하였다.

2.5 Wrapper Functions

16개의 trace파일에 대해 레퍼런스 쉘과 같은 출력을 보이는 것을 확인하였다. 이렇게 작성된 tsh가 안정적으로 구동될 수 있도록 구현에 사용한 system-call 각각에 대한 wrapper 함수를 작성해 자체적으로 에러처리가 이루어지도록 구현하였다. 사용한 system-call 중 에러처리가 별도로 되지 않은 5개에 대해서 wrapper를 작성했으며, 그 목록은 다음과 같다.

fork, waitpid, setpgid, kill, sigprocmask

함수 이름은 CSAPP책에서 나왔던 대로 첫 글자를 대문자로 변경한 이름을 사용하였으며, 기존 system-call이 음수를 반환하는 에러 케이스에 대해 미리 구현된 unix_error함수를 이용해서 에러처리를 하였다.
이렇게 에러 처리를 하였을 때 waitpid함수에서 문제가 발생했는데, sigchild핸들러에서 while루프를 돌며 reaping 할 child프로세스가 없는 경우에 -1을 리턴하여 unix_error함수를 호출했다. 이 경우를 예외적으로 처리하기 위해 waitpid의 wrapper함수에 예외적으로 errno 값이 ECHILD가 아닌 경우에만 unix_error함수를 호출하도록 변경하였다.

3. Conclusion

교재에 eval함수와 builtin_cmd함수의 기본적인 형태가 미리 구현되어 있었기 때문에 이전의 랩들에 비해 프로그램 작성을 시작하는데 있어 비교적 적은 시간이 걸렸다. sigchld_handler 와 waitfg에서 프로그램이 정지했던 버그를 해결하는데 많은 시간이 소요되긴 했지만, 제공된 handout에 있던 hint가 문제 해결에 많은 도움이 되었다. 또한 addjob, deletejob과 같이 프로세스 목록을 관리하는 함수들이 모두 구현되어 있었기 때문에 각각의 함수 작성 과정에서 세세한 부분들에 대한 처리보다 보다 Shell이라는 프로세스 전체적인 큰 흐름에 집중할 수 있었다. 16개나 되는 trace파일이 제공되어서 프로그램 실행 결과를 비교해 볼 수 있는 예제가 많아서 좋았지만, 매 실행마다 pid가 바뀐다는 점 때문에 레퍼런스 쉘과 직접 작성한 쉘에 대해 diff 명령어를 이용한 출력결과 비교가 어려웠다.