M1522.000800 System Programming
Fall 2018

System Programming ShellLab Report

Kim Bochang
2014-16757

1. <lab> Shell Lab

Shell Lab 의 목적은 리눅스에서 돌아가는 작은 셸 프로그램을 직접 만들어 보면서,
fork(), execve() 등의 시스템 함수들의 사용법을 익히고, 또 시그널 핸들러가 있는 코드를 직접 짜보면서 어떤 시그널을 블록해야 문제가 생기지 않는지와 같은것들을 생각해 보면서 프로세스와 프로세스간 통신, 시그널 핸들링 개념을 이해하는 것이다.

2. Implementation

<*tsh.c*>
이번 랩에서 구현한 셸 프로그램의 대략적인 구조는 다음과 같다.
처음에 셸의 핸들러 설정, joblist등의 초기화 등을 진행한 다음,
셸은 prompt (tsh>)를 출력한 뒤에 사용자의 입력을 기다린다.
만약 사용자의 입력이 들어오면, 받은 텍스트라인을 인자로 eval()함수를 호출한다.
eval()함수에서는 셸에 들어온 명령을 처리하는 역할을 하는데, 먼저 입력으로 받은 textline을 프로그램이나 명령어의 argument로 사용할 수 있도록 parseline()함수를 통해 가공한다.
만약 입력으로 받은 명령어가 없다면 다시 prompt를 출력하고 사용자의 입력을 기다리는 로직으로 돌아가고, 명령어를 받았다면 buildin_cmd()함수를 통해 셸에 내장된 명령인지 확인한다. 만약 셸에 내장된 명령(quit, bg, fg, jobs)이라면 해당하는 기능을 실행하고,
아니라면 명령어를 파일 이름이라고 가정하고 child 프로세스를 생성, 만약 파일이 존재하지 않는다면 에러메시지를 출력하고, 존재하는 파일이라면 child 프로세스에서 요구받은 파일을 실행하고, 그 프로세스를 셸의 job list에 추가한다.
셸 프로그램은 background와 foreground 작업을 모두 지원하기 때문에, 각 작업유형에 맞는 작업을 해준다. 예를들어, Foreground 작업이라면 작업을 실행한 뒤 waitfg()함수를 호출해 foreground 작업이 끝나기 전까지 새로운 사용자 입력을 받지 않는등의 작업이 있다. 그렇게 프로세스가 끝나면 셸 prompt를 띄우고 새로운 사용자 입력을 기다린다.
이 모든 과정을 하는동안, SIGCHLD, SIGINT, SIGTSTP 시그널이 발생하면 시그널 핸들러를 호출해서 적절한 시그널 처리를 진행하게 된다.

Eval() : eval 함수의 역할은 입력으로 받아온 cmdline을 적절히 처리해서 사용자가 요구한 작업을 수행하는 것이다. 코드 어디에서나 시그널 핸들링이 일어날 수 있고, 시그널 핸들러 중에는 joblist를 수정하는 핸들러도 있기 때문에 eval()함수에서 joblist를 수정하다가 그런 시그널 핸들러가 호출된다면 joblist에 추가하지도 않은 job을 제거하려 한다던가 하는 큰 문제가 발생할 수도 있다. SIGCHLD 핸들러가 그러한 핸들러라서, sigprocmask()함수를

사용해서 joblist에 job을 추가할때 SIGCHLD 시그널을 블록했고, joblist에 job을 추가한 다음에는 블록한 SIGCHLD 시그널을 해제해서 이러한 문제가 발생하는것을 막았다.
또한 받은 명령어가 내장된 명령어가 아닐경우 fork()를 진행하게 되는데, 이렇게 새로 생성된 child process의 경우 group id가 셸의 group id와 똑같다는 문제가 생긴다. 이렇게 생성된 특정 Child process에 대해 시그널을 보내야 할때, child process가 자신만의 child 프로세스를 또 생성하는 경우에는 자식에게까지 시그널을 보내는데 문제가 생기므로 setpgid()를 이용해 child process의 group id를 자신의 것으로 바꾸게 해, 여기에 연결된 모든 프로세스에게 한번에 시그널을 보낼 수 있게 하였다.

builtin_cmd() : builtin_cmd() 함수는 사용자가 입력한 명령어가 셸에 내장된 명령어인지 확인하고, 내장된 명령어라면 해당되는 명령을 실행하게 된다. 구현한 셸의 내장 명령어는 quit, bg, fg, jobs 명령으로, quit는 셸의 종료, bg는 stop된 job을 다시 background에서 실행하도록, fg는 stop되거나 background에서 돌아가는 job을 foreground에서 실행하도록, jobs 명령은 현재 job list를 모두 보여주게 된다. bg나 fg명령어는 후술할 do_bgfg()함수가 처리하도록 했고, 나머지 명령어는 셸을 종료하거나, listjobs()함수를 이용해 해당하는 명령을 수행하도록 했다.

do_bgfg() : do_bgfg() 함수는 사용자가 입력한 명령이 bg 혹은 fg일때의 처리를 담당하게 된다. 먼저 사용자가 입력한 명령어가 형식에 맞는지를 확인한 다음, jid나 pid 로 해당하는 job의 id를 받아서 먼저 stop된 job이라면 그 job의 group에 속해있는 프로세스들에 kill()함수를 통해 SIGCONT 시그널을 보내 다시 활동하게 한 다음, bg명령이라면 그 job을 background job으로 만들고, fg 명령이라면 그 job을 foreground job으로 만든 뒤, waitfg() 함수를 호출해 foreground 작업이 멈추거나 종료될때까지 기다리게 했다. do_bgfg()함수도 joblist에 있는 job의 내용(상태)를 바꾸기 때문에, SIGCHLD 핸들러와 충돌하는 상황을 막고자 sigprocmask() 함수를 이용해 job의 내용을 바꿀때는 SIGCHLD 시그널을 블록했다가, 처리가 끝난 후에 SIGCHLD 시그널을 다시 풀어주는 방법으로 처리를 했다.

Waitfg() : waitfg()함수는 foreground 작업의 pid를 받아서, 그 작업이 terminated 되거나 stopped 되기 전까지 셸에서 또 다른 사용자 입력을 받아 다른 프로그램을 실행하는것을 막게 된다.
현재 foreground 작업의 pid를 받아오는 fgpid()함수를 이용, waitfg가 기다려야하는 job의 pid와 fgpid()로 받은 pid가 달라질때까지 sleep()을 끼고있는 무한루프를 돌게 구현하였다.
시그널 핸들링이 일어나서 job list의 foreground job이 달라지게 되면 루프를 탈출하게 된다.

핸들러들을 만들때는, 핸들러 내부 함수에서 errno를 바꾸는 경우가 발생할 수 있으므로 기존의 errno를 핸들러의 처음과 끝에서 저장/복구 해주었고, 다른 시그널 핸들러가 한 핸들러가 작동하는 도중에 작동하면 문제가 생길 수 있으므로 핸들러의 처음과 끝에서 sigprocmask()함수를 이용하여 모든 시그널을 블록했다 원상복구 시켜주었다.

sigchld_handler() : SIGCHLD 시그널을 핸들링하는 시그널 핸들러다. Shell 내부에서 실행되는 프로그램들은 모두 shell의 child process이므로, child process들이 terminate되거나 stop될때 shell로 SIGCHLD 시그널을 보내게 된다. 이러한 SIGCHLD 시그널이 도착했을때, waitpid()함수를 사용하여 terminate되었거나 stop된 프로세스를 처리해준다.
이때 while문안에 waitpid()함수를 집어넣었는데, 하나의 waitpid()문만 쓰는경우에는 만약 셸에서 SIGCHLD 시그널이 블록되었을때 2개 이상의 SIGCHLD 시그널이 들어온 경우, 시그널은 버퍼에 저장되거나 하지 않으므로 하나 이상의 SIGCHLD 시그널은 무시되고, 무시된 시그널을 발생시킨 프로세스는 waitpid()문에 의해 회수되지 않을것이므로 zombie 프로세스가 생기게 된다. 따라서 while문에 waitpid()를 사용하고, WNOHANG 플래그와 WUNTRACED를 사용하여 모든 프로세스가 종료하기를 무한히 기다리지 않게 하면서 zombie가 회수되지 않는 경우를 막았다.
프로세스가 terminate된경우, 시그널을 받아 종료된경우 관련 메시지를 출력하고, 해당하는 job을 joblist에서 제거한다. stop 된 경우에는 joblist에서 해당 job의 상태를 바꿔주고 안내메시지를 출력한다.
Whileloop이 끝나는 경우는 waitpid()가 리턴하는 값이 0 혹은 -1일때인데, -1이 리턴된 경우는 waitpid()문에서 오류가 발생했을 수 있다. -1이 발생했을 경우, errno를 체크하여 ECHILD (더이상 기다릴 child process가 없음)인 경우가 아닌경우에는 에러처리를 하도록 하였다.

sigint_handler() : SIGINT 시그널을 핸들링하는 시그널 핸들러다. 사용자가 ctrl+c를 입력하면 shell에 SIGINT 시그널이 전달되는데, 이 시그널을 받은경우 현재 foreground에 있는 job에 SIGINT 시그널을 보내게 된다. fgjob()함수를 통해 현재 foreground에서 활동하는 프로세스의 pid를 받아온다음, kill()함수에 -pid를 넘겨주어 그 프로세스 그룹에 속하는 모든 프로세스에게 SIGINT 시그널을 보내게 하였다.

sigtstp_handler() : SIGTSTP 시그널을 핸들링하는 시그널 핸들러다. 사용자가 ctrl+z를 입력하면 shell에 SIGTSTP 시그널이 전달되는데, 이 시그널을 받은 경우 현재 foreground에 있는 job에 SIGINT 시그널을 보내게 된다. fgjob()함수를 통해 현재 foreground에서 활동하는 프로세스의 pid를 받아온다음, kill()함수에 -pid를 넘겨주어 그 프로세스 그룹에 속하는 모든 프로세스에게 SIGTSTP 시그널을 보내게 하였다.

3. Conclusion

랩을 진행하면서, 생각했던것보다 코딩 자체는 어렵지 않았다. 셸의 구조 자체는 그렇게 복잡하지 않고, 사용자가 요청한대로 명령어를 실행하거나, 프로그램을 실행해 주기만 하면 되었기 때문이다. 하지만 까다로운 것들이 여러가지 있었는데, 시그널 핸들러의 사용에서 생길 수 있는 문제점을 체크하는것도 그중 하나였다. 특히, SIGCHLD 시그널 핸들러가 실행될때 여러가지 side effect가 많아서, 코딩을 할때 이 부분에서 SIGCHLD 시그널 핸들러가 실행될때 문제가 생기진 않을까, 계속 체크해 가면서 랩을 진행했었다. jobs라는 공통된 자료구조를 메인로직과 SIGCHLD 핸들러가 공유하기 때문에, 이러한 생각이 하나라도 잘못되는 경우에는 프로그래머가 찾아내기 매우 힘든 영역에서 프로그램이 망가질 수 있었다. 실제로 처음에 짠 코드에서는 SIGCHLD 핸들러 말고도 SIGTSTP 핸들러에서도 jobs를 수정했었는데, 이 경우 셸의 SIGTSTP 핸들러를 거치지 않고 SIGTSTP 시그널을 자식 프로세스가 받았을때 jobs의 갱신이 잘못되어 이미 종료한 jobs을 계속해서 기다리는 문제가 있었다. 이러한 경우 디버깅이 매우 힘들어서 굉장히 당혹스러웠다.

또한 의외로 I/O 함수를 사용할때도 예상하지 못했던 어려움을 겪었었다.

셸을 쓰면서 프로세스가 시그널을 받아서 종료되거나, 정지했을때 사용자에게 메시지를 출력해서 보여줘야 하는경우가 있었는데, 그 과정에서 매우 골치아픈 문제가 생겼었다.

SIGCHLD 핸들러에서 standard I/O 함수인 printf()문을 사용해서 사용자에게 메시지를 보여줄때, 셸이 프롬프트를 출력하기 전에 메시지가 출력되어야 하는데, 프롬프트를 출력한 뒤에 메시지가 출력되는 문제점이 있었다. 다른 코드를 이리저리 바꿔봐도 도저히 문제가 해결되지 않아서 무엇이 문제인지 알아내려고 꽤 많은 시간을 썼었는데, 원인은 fflush()를 사용하지 않아서 생긴것이었다. standard I/O 함수인 printf()는 명시적으로 버퍼를 비우기 전까지 환경에 따라 계속 버퍼에 메시지를 쌓거나 출력을 하는데, 그 때문에 내부적으로 무언가 꼬여서 프롬프트 뒤에 메시지가 출력되었던 것이었다. 따라서 시그널 핸들러에서 메시지를 출력하는 부분의 printf()문 뒤에 fflush(stdout)을 통해 stdout 버퍼를 비워주었고, 정상적인 결과를 얻을 수 있었다. 이러한 문제점은 디버깅을 통해 알아내기가 거의 불가능한 문제점이라 문제를 해결하는데 어려움을 겪었었다.

그 외에도 자잘한 실수가 많았는데, 셸에서 새로운 프로그램을 실행해야 할때 fork()를 해서 child process를 만든 뒤 sigprocmask()로 막아놓은 signal을 child process에서 풀지 않는다던가, 존재하지 않는 파일을 실행하려 할때 exit()를 실행해서 프로세스를 종료시키는게 아니라, return을 해버리는 바람에 셸이 이중으로 돌아가는 문제도 발생하기도 했다.

셸 랩을 진행하면서 리눅스에서 셸이 어떻게 동작하는지 알게 되었고, 프로세스간에 시그널을 통해 통신하는 방법에 대해 깊은 이해를 하게 되었다. 또한 시그널 핸들러를 다룰때는 핸들러의 side effect까지 모두 고려해서, 핸들러의 side effect가 다른 코드에 영향을 미치지 않게 조심스럽게 코드를 짜야한다는점을 배웠다.

또한 standard I/O 함수를 사용할때, 내부의 버퍼를 사용하는 동작방식 때문에 의도치 않은 위치에서 출력이 되는 문제가 발생할 수 있음을 알 수 있었다.