**System Programming Project 2**

담당 교수 : 박성용

이름 :

학번 : 20xxxxxx

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* **(MyShell을 만드는 전체적인 개요에 대해서 작성하면 됨.)**
  1. Phase 1에서는 cd, exit, ls, mkdir와 같은 단일 명령어를 처리하는 것이 목표다. 특히 cd, exit의 경우 shell의 built-in command이기 때문에 exec()를 통한 실행이 불가능하므로 직접 구현해야 한다. 나머지 명령어들의 경우 fork()를 통해 자식 프로세스를 생성하고 해당 명령어를 실행시키고, 부모 프로세스에서 자식 프로세스가 종료될 때까지 기다린다. 그리고 SIGINT, SIGCHLD에 대한 간단한 signal handler를 구현한다.
  2. Phase 2에서는 Piping된 명령어에 대해 처리하는 것이 목표다. run\_pipe 함수를 호출하여 명령어줄에 속한 명령어를 하나씩 수행하고 다음 명령어에 대해 run\_pipe 함수를 다시 재귀로 호출하여 모든 명령어를 수행한다. 여기서 핵심은 프로세스 간의 소통을 위해 pipe fd를 이용하여 명령어 수행 결과를 다음 명령어의 입력으로 받을 수 있도록 하는 것이다. 그리고 아직 명령어의 실행이 완료되지 않았는데 다음 파이프된 명령어가 먼저 실행되는 것을 방지하기 위해 Phase 1에서 그러한 것처럼 자식 프로세스가 종료될 때까지 기다린 후 run\_pipe를 호출하도록 한다. 마지막으로 SIGTSTP에 대한 간단한 signal handler를 구현한다.
  3. Phase 3에서는 프로세스를 background으로 돌리는 것에 대한 구현이 필요하다. 명령어줄에 &(ampersand)가 붙으면 이를 background에서 돌아가도록 하고 background 상태에서 종료되면 이를 reaping하는 작업이 필요한데, SIGCHLD에 대한 handler 함수를 수정하여 종료한 자식 프로세스에 대해 정상적으로 reaping 할 수 있도록 한다. 그리고 piping된 명령어에 대한 작업을 수행중인데 SIGINT, SIGTSTP와 같은 신호가 발생하는 경우 관련된 자식 프로세스들을 한번에 종료 또는 정지할 수 있도록 프로세스 grouping에 대한 구현을 한다. 명세서에 명시된 jobs, bg, fg, kill 명령어는 bash shell의 built-in command이므로 이는 직접 구현해야 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

명세서에 명시된 cd, ls, mkdir, rmdir, touch, cat, echo, exit 모두 정상적으로 작동하는 것을 확인하였다. 따옴표(“ 혹은 ‘)로 감싸진 부분을 하나의 문자열로 정상적으로 인식하였다. 이 외에도 pwd, sleep과 같은 exec()에서 실행 가능한 명령어들도 정상적으로 동작하였다. Ctrl+C를 입력하였을 때 현재 foreground인 프로세스가 있는 경우 terminate되었고 명령어 입력 대기 상태인 경우 다음 줄로 넘어가면서 아무 일도 발생하지 않았다. exit 명령어를 통해서만 myshell을 종료할 수 있다.

1. Phase 2

명세서에 명시된 “ls | grep filename”, “cat filename | less”, “cat filename | grep -i “abc” | sort -r”와 같은 파이프된 명령어들이 모두 정상적으로 작동하는 것을 확인하였다. SIGTSTP에 대한 signal handler를 구현하였기 때문에 Ctrl+Z를 누르면 myshell에서 foreground로 실행중인 프로세스가 있는 경우 정상적으로 정지되는 것을 확인할 수 있었다.

1. Phase 3  
   명세서에 명시된 명령어들 모두 정상적으로 동작하는 것을 확인할 수 있다. 명령어줄에 &가 붙은 경우 백그라운드로 돌려 두고 바로 다른 명령어를 실행할 수 있다. jobs, bg, fg, kill 명령어들 모두 정상적으로 작동하는 것 또한 확인하였다. Signal handler들 또한 정상적으로 작동한다.
   1. **개발 내용**

* **아래 항목의 내용만 서술**
* **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명

fork()를 호출하고 반환된 값에 따라 if문으로 분기하여 자식인 경우(반환값 0) execvp()를 호출하여 파싱한 명령어 실행, 부모인 경우 (반환값 자식pid) 자식이 끝날 때 까지 기다려 정상적으로 reaping할 수 있도록 한다. 이는 zombie 프로세스가 발생되는 것을 방지하기 위함이다.

* + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow  
    자식 프로세스가 종료되면 SIGCHLD 신호를 발생시키는데 이를 직접 구현한 signal handler에서 처리할 수 있도록 한다. 우선 부모 프로세스에서 signal blocking을 통해 모든 signal을 막고 atomic variable인 pid\_atomic을 0으로 설정, while문과 Sigsuspend()를 통해 SIGCHLD 신호를 명시적으로 기다린다. 그리고 SIGCHLD 신호가 발생하면 handler에서 waitpid를 통해 종료한 자식 프로세스를 정상적으로 reaping할 수 있도록 하고, 반환값을 pid\_atomic에 저장한다. 따라서 부모 프로세스는 while문을 탈출하면서 자식이 종료되었다는 것을 알게 되고 모든 signal을 다시 허용한 후 추가적으로 필요한 작업을 수행한다 (flag 설정 등)
* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)  
    기존에 제공된 skeleton code의 parseline 함수의 parsing은 “ls -al”이라는 명령어가 있다면 “ls, -al, NULL”과 같이 나누는 방식이었는데, 이를 조금 수정하여 끝에 NULL을 한 번 더 붙인다. 그리고 파이프(|)를 만나면 NULL로 바꾼다. 어떤 말이냐면, “ls -al | grep abc”라는 명령어줄이 들어오면 “ls, -al, NULL, grep, abc, NULL, NULL”과 같이 parsing을 하는 방식이다. 이를 통해 NULL 다음에 또 NULL이 나타나야 명령어가 끝난 것이고 아니라면 파이프된 다음 명령어가 존재한다고 인식하게 된다. 그리고 run\_pipe 함수를 재귀적으로 호출하면서 순차적으로 각 명령어에 대한 자식 프로세스를 생성 및 종료시킨다. 이 과정에서 한 프로세스의 명령어 출력 결과가 다음 프로세스의 입력으로 들어갈 수 있게끔 pipefd를 통해 구현하였다. pipefd의 write end를 표준 출력에 dup하여 명령어 실행 결과가 커널에 buffer된다. 이후 다음 프로세스에서 pipefd의 read end를 표준 입력에 dup하여 buffer된 출력이 표준 입력으로 들어가게 된다. 만약 다음 명령어가 없다면 pipefd에 연결할 필요 없이 바로 실행 결과가 표준 출력으로 나오게 된다.
  + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명  
    Pipeline 개수에 상관 없이 파싱된 명령어에 따라 더 이상 명령어가 없을 때 까지 재귀적으로 run\_pipe 함수를 호출하게 된다. 이 과정에서 pipefd를 통해 프로세스의 출력을 다음 프로세스의 입력으로 넣어주는 것 또한 함수가 호출된 횟수만큼 반복적으로 발생한다.
* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명

Foreground process의 경우 fork() 이후 부모 프로세스에서 waitpid()를 통해 자식 프로세스가 종료할 때 까지 대기하였는데, background process의 경우 기다리지 않고 바로 다음 명령어 입력을 대기해야 하므로 wait 하지 않은 상태로 evaluation을 끝낸다. 그리고 background 상태에서 그대로 종료하는 경우 SIGCHLD handler에서 해당 프로세스를 reaping하고 종료되었음을 화면에 알린다. fg, bg를 통해 프로세스를 foreground로 가져오거나 background에서 돌릴 수 있고, kill을 통해 특정 job을 terminate 할 수 있다. jobs를 통해 background 혹은 정지된 job을 확인할 수 있다. 이러한 job의 개념을 구현하기 위해 Project 1의 doubly linked list를 가져와 활용한다. SIGINT 또는 SIGTSTP 신호가 발생하면 해당 job에 속한 모든 자식 프로세스에게 신호가 전달되어야 하는데, fork() 이후 setpgrp() 함수를 통해 job의 pid를 pgid로 설정하고, job에 대한 자식 프로세스는 이 pgid를 상속받는다. 그리고 kill() 함수에 인자를 -pid로 넘겨줌으로써 해당 그룹에 속한 모든 프로세스에 신호가 전달된다.

* 1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

1. Job의 개념을 구현하기 위해 Project 1의 doubly linked list 활용. list\_item 멤버에 job에 관련된 데이터 추가(pid, job\_id, condition, cmdline). Job 자체의 요소를 담은 job\_list. 실제 job의 경우 가장 최근에 foreground 혹은 상호작용한 job에 ‘+’가 붙고 그 다음 우선 순위에 대해 ‘-‘가 붙는데, 이러한 job들의 우선 순위가 stack 자료구조와 유사하다고 판단되어 job의 list\_elem을 데이터로 가지는 job\_stack 선언. job이 추가되면 job\_stack의 front에 삽입됨. 이후 fg, bg 동작에 따라 특정 job이 front로 올라옴.
2. parseline 함수  
   기존의 parseline 함수는 “ls, -al, NULL”과 같이 파싱. Piping에 대한 처리를 위해 뒤에 NULL을 한 번 더 붙임. “ls -al | grep filename”이 입력되면 “ls, -al, NULL, grep, filename, NULL, NULL”과 같이 파싱한다. 따옴표로 감싸진 경우 하나의 인자로 취급하도록 구현. Pipe 혹은 Ampersand가 space 없이 입력된 경우에 대한 처리 구현.
3. get\_next\_argv  
   위의 parseline 함수에서 파싱한 결과를 기반으로 NULL 다음 인자에 대한 주소 반환. 이를 통해 piping된 명령어의 끝이 어딘지 파악 가능.
4. run\_pipe 함수  
   Piping된 명령어의 파싱 결과를 인자로 받아 fork() 후 첫 번째 명령어를 실행시키고 다음 명령어에 대한 정보를 get\_next\_argv 함수를 통해 얻어 재귀적으로 다음 명령어를 실행함. Pipe()와 dup2()를 활용하여 출력을 다음 프로세스의 입력으로 넣어주도록 구현. argv\_next가 더 이상 없으면 그냥 표준 출력에 실행 결과 출력.
5. check\_group 함수  
   명령어 조합에 따라 grouping이 필요한지 여부를 반환하는 함수.
6. add\_job 함수  
   프로세스의 pid, condition, cmdline을 담은 요소를 job\_list에 추가. job의 list\_elem을 담은 요소를 job\_stack의 front에 삽입.
7. remove\_job 함수  
   job을 삭제하는 함수. job\_list, job\_stack에서 요소를 제거하고 동적으로 할당된 메모리를 해제.
8. get\_last\_job 함수  
   job\_stack의 front 요소가 가리키는 job item을 가져오는 함수.
9. get\_job\_by\_jobid 함수  
   job\_list를 순회하면서 인자로 들어온 id를 가진 job item을 반환하는 함수.
10. get\_job\_by\_pid 함수  
    job\_list를 순회하면서 인자로 들어온 pid를 가진 job item을 반환하는 함수.
11. set\_last\_job 함수  
    인자로 들어온 job item의 list\_elem을 담은 요소를 job\_stack의 front로 옮기는 함수.
12. print\_one\_job  
    인자로 들어온 job에 대한 정보를 출력하는 함수. job\_stack의 첫 번째, 두 번째 요소와 비교하여 ‘+’, ‘-‘를 붙이거나 아무것도 붙이지 않음
13. print\_jobs 함수  
    jobs 명령어가 들어온 경우 job들의 정보를 출력하기 위한 함수. job\_list의 모든 job들에 대해 print\_one\_job 수행
14. check\_jobs 함수  
    백그라운드에서 수행이 완료된 job을 체크하여 존재한다면 print\_one\_job 수행 후 리스트에서 제거하는 함수.
15. clear\_jobs 함수  
    job\_list, job\_stack의 모든 요소들을 제거하는 함수. 동적 할당 메모리 해제. exit 명령어가 들어온 경우 수행됨.
16. toggle\_ampersand 함수  
    BG 여부에 따라 job의 cmdline에 &를 붙이거나 제거하는 함수.
17. sigint\_handler  
    FG 프로세스가 실행중이라면 get\_last\_job을 통해 가져온 job에 속한 프로세스에 대해 SIGINT 신호를 보내고 remove\_job 수행. FG 프로세스가 없다면 다음줄 출력.
18. sigchld\_handler  
    BG 프로세스가 종료되면 waitpid를 통해 자식을 reaping하고 자식의 pid로 job을 찾아 condition을 DONE으로 바꿈.
19. sigtstp\_handler  
    FG 프로세스가 실행중이라면 sigint\_handler와 동일하게 FG인 job에 속한 프로세스에 SIGTSTP 신호 전달.
20. **구현 결과**
    1. **Flow Chart**

* **2.B.개발 내용에 대한 Flow Chart를 작성.**
* **(각각의 방법들에서 추가된 내용(fork, pipeline, background)만 특성이 잘 드러나게 그리면 됨.)**

1. **Phase 1 (fork)  
   텍스트, 스크린샷, 직사각형, 도표이(가) 표시된 사진

   자동 생성된 설명**
2. **Phase 2 (pipeline)  
   텍스트, 스크린샷, 라인, 도표이(가) 표시된 사진

   자동 생성된 설명**
3. **Phase 3 (background)  
   도표, 텍스트, 라인, 폰트이(가) 표시된 사진

   자동 생성된 설명  
   도표, 라인, 스케치, 폰트이(가) 표시된 사진

   자동 생성된 설명  
   도표, 라인, 스케치, 폰트이(가) 표시된 사진

   자동 생성된 설명  
   도표, 라인, 폰트, 텍스트이(가) 표시된 사진

   자동 생성된 설명  
   도표, 라인, 폰트, 화이트이(가) 표시된 사진

   자동 생성된 설명**