

# System Programming Project 1

담당 교수 : 김영재

이름 : 홍주표

학번 : 20181702

## 1. 개발 목표

먼저 Phase 1에서 Fork 함수와 Built-in Command를 통해 자식 프로세스를 생성하고 다양한 명령어를 처리한다.

그리고 Phase 2에서는 Phase 1을 기반으로 작동하는 명령어에 파이프(Pipeline, '|')가 입력될 때도 작동되도록 한다.

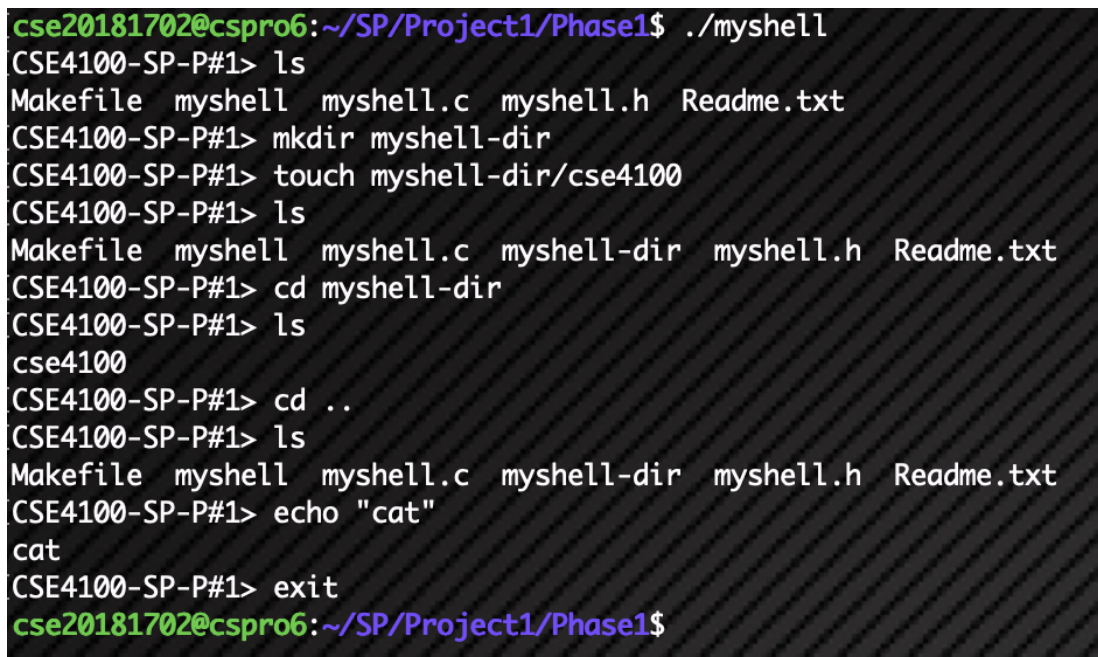
마지막으로 Phase 3에서는 앞에서까지 foreground에서 구현한 것들을 background에서도 작동되도록 한다. 또한 background 개념이 있을 때 사용 가능한 명령어도 처리한다.

이로써 기존의 shell과 기능이 유사한 myshell을 만든다.

## 2. 개발 범위 및 내용

### A. 개발 범위

#### 1. Phase 1



```
cse20181702@cspro6:~/SP/Project1/Phase1$ ./myshell
CSE4100-SP-P#1> ls
Makefile myshell myshell.c myshell.h Readme.txt
CSE4100-SP-P#1> mkdir myshell-dir
CSE4100-SP-P#1> touch myshell-dir/cse4100
CSE4100-SP-P#1> ls
Makefile myshell myshell.c myshell-dir myshell.h Readme.txt
CSE4100-SP-P#1> cd myshell-dir
CSE4100-SP-P#1> ls
cse4100
CSE4100-SP-P#1> cd ..
CSE4100-SP-P#1> ls
Makefile myshell myshell.c myshell-dir myshell.h Readme.txt
CSE4100-SP-P#1> echo "cat"
cat
CSE4100-SP-P#1> exit
cse20181702@cspro6:~/SP/Project1/Phase1$
```

#### <Phase 1 실행 결과>

- Fork 함수와 Built-in Command를 사용하여 ls, mkdir, touch, cd, cd., exit과 같은 다양한 명령어를 정상적으로 처리한다.
- 또한 사진의 echo "cat" 과 같은 상황처럼 따옴표를 무시하는 것도 처리한다.

## 2. Phase 2

```
cse20181702@cspro6:~/SP/Project1/Phase2$ ./myshell
CSE4100-SP-P#1> ls
Makefile myshell myshell.c myshell.h Readme.txt
CSE4100-SP-P#1> ls -al | grep myshell
-rwxr-xr-x 1 cse20181702 under 33016 Apr 11 03:51 myshell
-rw-r--r-- 1 cse20181702 under 5827 Apr 11 03:52 myshell.c
-rw-r--r-- 1 cse20181702 under 29682 Apr 2 22:03 myshell.h
CSE4100-SP-P#1> grep "printf" myshell.c | sort -r | tail -4
    printf("\n");
        printf("%d %s", pid, cmdline);
            printf("%d %s", pid, cmdline);
        printf("CSE4100-SP-P#1> ");
CSE4100-SP-P#1> ls -al | grep myshell | head -2 | sort
-rw-r--r-- 1 cse20181702 under 5827 Apr 11 03:52 myshell.c
-rwxr-xr-x 1 cse20181702 under 33016 Apr 11 03:51 myshell
CSE4100-SP-P#1> exit
cse20181702@cspro6:~/SP/Project1/Phase2$
```

### <Phase 2 실행 결과>

- 명령어에 pipeline이 입력될 때도 정상적으로 처리한다.
- pipeline을 기준으로 왼쪽에 위치한 명령어의 결과를 오른쪽 명령어의 입력으로 받아 최종 결과만을 출력하도록 처리한다.
- 또한 사진의 세 번째 명령어처럼 pipeline 앞 뒤로 공백 여부에 관계 없이 정상적으로 작동하도록 처리한다.

### 3. Phase 3

```
cse20181702@cspro6:~/SP/Project1/Phase3$ ./myshell
CSE4100-SP-P#1> jobs
CSE4100-SP-P#1> sleep 100 &
[1] 143640                sleep 100
CSE4100-SP-P#1> jobs
[1] Running               sleep 100
CSE4100-SP-P#1> sleep 101
^Z[2] Suspended          sleep 101
CSE4100-SP-P#1> jobs
[1] Running               sleep 100
[2] Suspended            sleep 101
CSE4100-SP-P#1> ls -al | grep myshell &
[3] 143650                ls -al | grep myshell
CSE4100-SP-P#1> -rwxr-xr-x 1 cse20181702 under 37448 Apr 11 03:45 myshell
-rw-r--r-- 1 cse20181702 under 11409 Apr 11 03:42 myshell.c
-rw-r--r-- 1 cse20181702 under 29682 Apr 10 00:28 myshell.h
[3] Done                  ls -al | grep myshell
CSE4100-SP-P#1> jobs
[1] Running               sleep 100
[2] Suspended            sleep 101
CSE4100-SP-P#1> sleep 10
^Z[3] Suspended          sleep 10
CSE4100-SP-P#1> jobs
[1] Running               sleep 100
[2] Suspended            sleep 101
[3] Suspended            sleep 10
CSE4100-SP-P#1> bg %3
[3] Running               sleep 10
CSE4100-SP-P#1> jobs
[1] Running               sleep 100
[2] Suspended            sleep 101
[3] Done                  sleep 10
CSE4100-SP-P#1> jobs
[1] Running               sleep 100
[2] Suspended            sleep 101
CSE4100-SP-P#1> fg %2
[2] Running               sleep 101
[1] Done                  sleep 100
[2] Done                  sleep 101
CSE4100-SP-P#1> jobs
CSE4100-SP-P#1> sleep 104 &
[1] 143665                sleep 104
CSE4100-SP-P#1> kill %1
CSE4100-SP-P#1> jobs
CSE4100-SP-P#1> █
```

#### <Phase 3 실행 결과>

- 명령어의 마지막에 앰퍼샌드(&)가 입력될 때 백그라운드로 그 명령어가 실행된다.
- jobs 명령어를 입력하면 background에 있는(Stopped or Running) 명령어가 보여진다.

- 명령어가 foreground에서 실행중일 때 CTRL+Z를 입력하면 해당 명령어가 background에 Stop된 상태로 변경된다.
- bg <job>을 입력하면 background에 멈춰있는 명령어를 background에서 작동되게 한다.
- fg <job>을 입력하면 background에 있는(Stopped or Running) 명령어를 foreground에서 작동되게 한다.
- kill <job>을 입력하면 해당 명령어를 끝낸다.
- 위의 모든 기능은 pipeline()이 존재할 때도 작동된다.

## B. 개발 내용

- Phase1 (fork & signal)
  - ✓ fork를 통해서 child process를 생성하는 부분에 대해서 설명
- 입력한 명령어가 Built-in Command가 아닐 때 Fork 함수를 실행하고 이를 통해 child process를 생성한다.
  - ✓ connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow
- Fork 함수를 통해 child process가 생성되고, 그 안에서 명령어를 수행한다. parent process는 waitpid 함수를 통해 해당 명령어가 종료되는 것을 기다리고, 해당 명령어가 종료되면 종료 시그널을 받고 child process를 reap한다.
- Phase2 (pipelining)
  - ✓ Pipeline( '|' )을 구현한 부분에 대해서 간략히 설명 (design & implementation)
- 처음 명령어를 입력받을 때 해당 명령어에 pipeline이 존재하는지에 따라 실행하는 함수를 나눈다.
- pipeline이 존재한다면 pipe\_parse 함수를 실행하고 입력받은 명령어를 pipeline을 기준으로 나누고 cmd 변수에 넣는다. 그리고 cmd, cmdline 그리고 pipeline의 개수를 parameter로 가지는 eval\_pipe 함수를 실행한다.

- eval\_pipe 함수는 pipeline이 존재하지 않을 때 실행되는 eval 함수와 유사한 기능을 실행한다. 다만 cmd에 할당되어 있는 세부적인 명령어를 실행하고, eval\_pipe 함수를 recursive하게 호출하며 그 출력값을 다음 cmd의 입력값으로 넘긴다는 점이 eval 함수와의 차이점이다.

- 마지막 cmd의 명령어까지 처리하면 해당 결과값은 STDOUT에 출력되게 처리한다.

- ✓ Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

- pipe\_parse 함수에서 pipeline의 개수를 count하고 이에 맞게 eval\_pipe 함수를 recursive하게 호출한다. 이를 통해서 pipeline이 존재할 때 그 개수와는 무관하게 정상적으로 작동하도록 처리한다.

- **Phase3 (background process)**

- ✓ Background ('&') process를 구현한 부분에 대해서 간략히 설명

- pipeline이 없을 때 실행하는 eval 함수와 pipeline이 존재할 때 실행하는 pipe\_parse 함수 안에서 각각 is\_background 함수를 호출한다. 이 함수는 cmdline을 파라미터로 넘기고 해당 명령어(cmdline)의 마지막에 &가 있는지 여부를 확인한다. &가 있으면 1을, 없으면 0을 return한다.

- 이 때, & 뒤에 space가 나오더라도 &로 끝난 것으로 간주하여 정상적으로 동작하도록 처리한다.



## C. 개발 방법

### Phase1 (fork & signal)

- 먼저 builtin\_command 함수에 cd와 exit 명령어가 작동되도록 처리한다. cd 명령어는 chdir 함수를 사용하고, exit 명령어는 기존의 quit과 동일하게 구현한다.
- eval 함수에서 Built-in Command가 아닐 때 Fork 함수를 실행하고 그 child process 안에서 execve 함수를 통해 입력받는 명령어를 실행하도록 처리한다. 이 때 execve의 첫 파라미터로 path1, path2를 각각 받게 하는데 이는 “/bin/”에 있는 함수와 “/usr/bin/”에 있는 함수를 모두 실행하기 위함이다.
- 위의 명령어 실행은 모두 foreground에서 이루어지기 때문에 parent process에서 background가 아닐 때 waitpid 함수를 이용하여 명령어 수행을 끝낸 child process를 reap 한다.
- 또한 echo “cat” 과 같은 명령어가 들어올 때 따옴표를 무시해주는 코드를 parseline 함수에 추가한다.
- 그리고 myshell 내부에서 foreground로 명령어가 작동 중일 때 CTRL+C를 입력하면 myshell은 구동 중인 채로 해당 작업만 중지시키기 위해서 sigint\_handler 함수를 만들어 처리하고, CTRL+Z 입력은 무시한 채로 그대로 진행시키도록 처리한다.

### Phase2 (pipelining)

- 명령어를 입력받을 때마다 해당 명령어 내에 pipeline이 존재하는지를 확인하고 pipeline이 존재한다면 pipe\_parse 함수를, 그렇지 않다면 eval 함수를 부른다. 이 때 pipeline이 존재하지 않는 명령어는 Phase1과 같은 내용이기 때문에 이 조건에서는 추가로 무언가를 구현하지는 않는다.
- pipeline이 존재하는 명령어를 받아서 pipe\_parse 함수로 들어가면 pipeline을 기준으로 명령어를 나눈다. 그리고 나누어진 명령어들은 cmd라는 변수에 순서대로 할당된다. 이 때 pipeline의 개수를 count하여 pipe\_num 변수에 할당한다.
- 위의 parse를 마치고 eval\_pipe라는 함수를 호출한다. 이 함수는 parse를 완료한 명령어들(cmd), 기존에 입력한 명령어(cmdline), pipeline의 개수(pipe\_num) 그리고 index라는 변수를 파라미터로 가진다. 이 때 index는 몇 번째 cmd인지 알기 위한 변수로써, pipe\_parse에서

eval\_pipe를 호출할 때는 0을, 그 이후에 recursive하게 eval\_pipe가 호출될 때는 1씩 증가한 값을 가진다.

- eval\_pipe 함수에서는 fd 변수를 2차원 배열의 형태로 선언한다. recursive 하게 eval\_pipe 함수가 호출될 때마다 현재 index에 맞는 fd를 pipe 함수를 통해 file descriptor로 만든다
- 이후 코드는 pipeline이 없을 때 실행되는 eval 함수와 유사하다. 다만 eval\_pipe 함수에서는 pipeline을 기준으로 왼쪽 명령어의 출력값을 오른쪽 명령어의 입력으로 넘겨준다는 것이 둘의 차이점이다. 이를 수행하기 위해서 Fork 함수를 통해 만들어진 child process 내에서 해당 명령어가 마지막 명령어가 아닐 때 Dup2 함수를 통해 출력값을 넘겨준다. 또한 parent process에서는 child process의 명령어가 끝나고 child process를 reap한 뒤, Dup2 함수를 통해 입력값을 받는다. 그리고 index를 1만큼 증가시킨 채로 eval\_pipe 함수를 recursive하게 호출한다.

### Phase3 (background process)

- DEFAULT, RUNNING, DONE, STOP, KILL, FOREGROUND 순서대로 0~5 값을 할당한다. (순서대로 빈 job, background에서 작업중인 job, 작업을 마친 job, background에서 작업이 멈춘 job, foreground에서 작업중인 job을 가리킨다.)
- background와 foreground의 개념이 생기는 phase이기 때문에 각 명령어들의 상태를 나타내는 것들을 #define을 사용하여 위 설명처럼 단순매크로화한다. 또한 입력받은 명령어의 process id(pid), 입력받은 명령어가 쌓이는 순서(id), 해당 명령어의 상태(state) 그리고 처음 입력받은 명령어(cmdline)를 하나의 struct(job)로 만든다. 그리고 job type을 가지는 jobs와 실행중인(혹은 정지해 있는) 명령어의 총 개수를 다룰 job\_num 변수를 전역변수로 선언한다.
- 입력받은 명령어가 background에서 작동하는지 여부를 알기 위해 eval(pipeline이 존재하면 pipe\_parse) 함수에서 is\_background 함수를 호출한다. 이 때 cmdline을 파라미터로 넘겨서 &를 감지하는데, cmdline의 뒤부터 보면서 space 혹은 개행문자(\n)가 있으면 이는 무시하고 처음 나오는 다른 문자가 &라면 1을, &가 아닌 다른 문자가 나오면 0을 return한다. &가 있을 때는 & 문자를 빈칸으로 바꿔 명령어를 실행하는데 영향을 주지 않게 처리한다.
- eval(또는 eval\_pipe) 함수에서 phase 2까지와 유사하게 명령어를 처리한다. 이전과의 차이점은 foreground에서 작동하는 명령어이든 background이든지 간에 job type의 jobs 변수에 pid, id, state 그리고 cmdline을 입력시킨다. jobs 변수는 배열로 선언되기 때문에 위에서 설명한 job\_num을 jobs의 index로 삼아 알맞은 위치에 입력시킨다. 이 때 foreground에서 실행



행되는 명령어는 state로 FOREGROUND를, background에서 실행되는 명령어는 RUNNING을 입력받는다. jobs에 입력을 한 후에는 job\_num을 1만큼 증가하는데, foreground에서 작동시킨 명령어가 정상적으로 작동을 완료한 경우에는 job\_reinit 함수를 사용해 현재 index에 해당하는 jobs를 다시 초기화한다.

- jobs 함수를 사용하기 위해 builtin\_command 함수에 코드를 추가한다. jobs 명령어를 입력하면 background에서 작동하거나 멈춰있는 명령어를 출력해야 하므로 조건에 맞게 job의 state를 이용하여 알맞게 출력한다.

- kill 함수를 사용하기 위해 builtin\_command 함수에 코드를 추가한다. kill 과 함께 %N(N은 자연수)을 입력하면 N에 해당하는 id를 가진 job을 Kill 함수와 SIGKILL을 통해 죽이고 해당 job의 state를 KILL로 변경한다. 만약 N에 해당하는 job이 없거나 %가 입력되지 않는다면(kill 명령어 형식에 부합하지 않는다면) 각 상황에 맞는 에러메시지를 출력한다.

- background 개념이 있는 상황이면 CTRL+Z를 앞 phase까지와는 다르게 처리해야 한다. foreground에서 어떤 명령어가 실행 중이라면 그 명령어를 중단시킨 채 background 영역으로 전환시킨다. 이를 실행시키는 함수인 sigtstp\_handler를 만들고 Signal 함수를 통해 CTRL+Z 입력을 알맞게 처리하게 한다.

- bg 함수를 사용하기 위해 builtin\_command 함수에 코드를 추가한다. bg와 함께 %N(N은 자연수)을 입력하면 N에 해당하는 id를 가진 job이 background에서 멈춰있는 작업인지를 확인하고, 조건에 부합하면 Kill 함수와 SIGCONT를 통해 background에서 해당 작업을 재개한다. 만약 N에 해당하는 job이 bg 명령어의 조건에 부합하지 않으면 에러메시지를 출력한다.

- fg 함수를 사용하기 위해 builtin\_command 함수에 코드를 추가한다. fg와 함께 %N(N은 자연수)을 입력하면 N에 해당하는 id를 가진 job이 background에서 멈춰있거나 작동중인 작업인지를 확인하고, 조건에 부합하면 Kill 함수와 SIGCONT를 통해 작업을 재개한다. 이 재개된 작업은 foreground 영역에서 작동해야 하므로 while 무한 루프를 통해 기다리고 job의 state가 변경되면(FOREGROUND가 아니면) 무한 루프를 빠져나온다.

- main 함수에서는 jobs 전체를 초기화하는 job\_init 함수를 호출한다.

- main 함수의 while loop 내의 마지막 부분에서는 명령어를 입력할 때마다 print\_done 함수를 호출하여 각 job의 state에 맞는 update를 한다. state가 KILL인 job이 있다면 그 job의 state를 DEFAULT로 만든다. state가 DONE인 job이 있다면 그 job의 state를 DEFAULT로 만들고 끝났다는 문구를 출력한다. 또한 jobs의 뒤부터 훑으며 job의 state가 RUNNING이거나

STOP이면 그 다음 위치로 job\_num 값을 변경한다. jobs의 모든 state가 DEFAULT(혹은 KILL 인데 DEFAULT로 변경되지 못한 상태)이면 job\_num을 0으로 초기화한다.

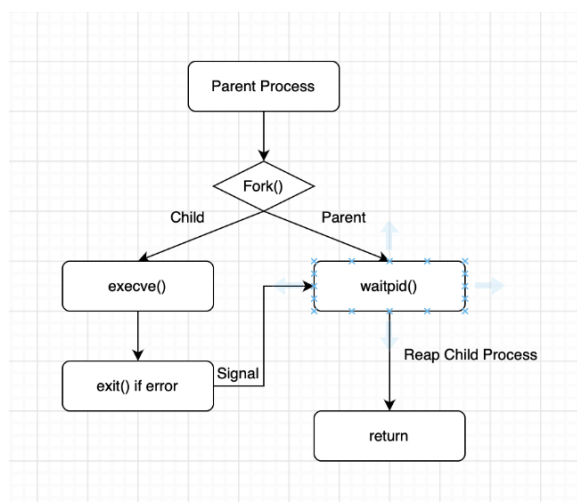
- eval(또는 eval\_pipe) 함수 내에서 waitpid 함수를 통해 parent process가 child process를 기다릴 때 CTRL+Z 입력을 정상적으로 처리하기 위해 waitpid 함수의 세 번째 파라미터를 WUNTRACED로 변경한다. WUNTRACED는 중단된 자식 프로세스 상태를 받는 상수이다.

- sigchld\_handler 함수는 child process가 보내는 signal을 감지한다. background에서 실행한 명령이 정상적으로 끝났다면 해당 job의 state를 DONE으로 변경한다.

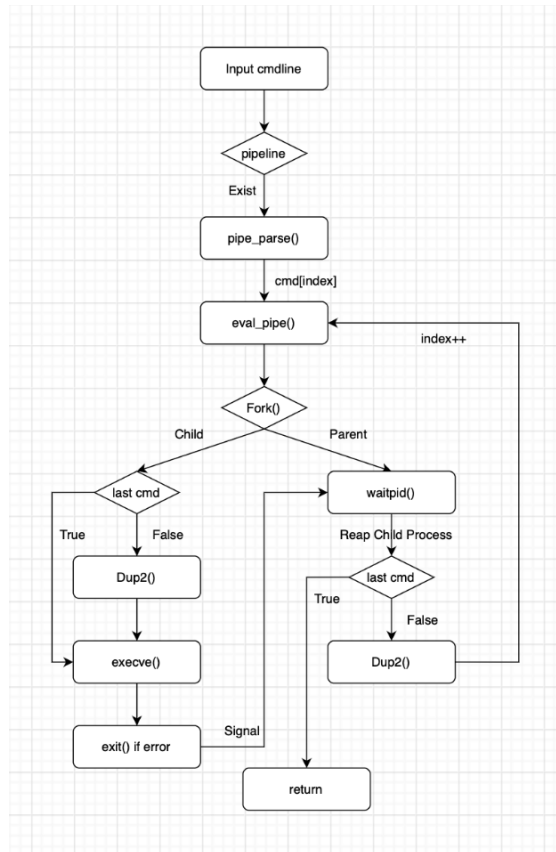
### 3. 구현 결과

#### A. Flow Chart

##### 1. Phase 1 (fork)



##### 2. Phase 2 (pipeline)



### 3. Phase 3 (background)

