**System Programming Project 1**

담당 교수 : 김영재 교수

이름 : 조명재

학번 : 20192138

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* **(MyShell을 만드는 전체적인 개요에 대해서 작성하면 됨.)**

1. Phase 1 에서는 단일 명령어에 대한 처리가 필요하다.

parseline 함수로 [”ls”, “-al”, NULL] 과 같은 2차원 배열의 argv 를 생성하여 해당 명령어를 fork() 를 띄워서 자식 프로세스에게 해당 명령어를 실행하도록 한다.

이때 핵심은 fork() 를 띄워서 자식 프로세스가 명령어를 실행할 때 까지 부모 프로세스는 기다리고 있다는 점이다.

2. Phase 2 에서는 파이프에 대한 처리가 필요하다.

파이프를 처리하는 방법에는 iterative, recursive 두 가지가 존재하는데 Phase 2 에서는 간단하게 iterative 을 이용하여 구현하였다.

우선 pipe() 함수를 이용하여 output 을 그 다음 프로세스에서 처리가 가능하도록 해야 한다.

fork() 를 호출하여 자식 프로세스에서 파이프 단위로 각각의 명령어를 실행함으로써 그에 대한 output 을 파이프에 담아주고 부모 프로세스가 이와 같은 작업이 끝나면 그 다음 자식 프로세스를 생성하여 이전에 담아둔 output 을 이용하여 그 다음 명령어를 덧붙여 실행하는 과정을 파이프 가장 마지막 명령어에 도달할 때까지 실행하게 된다.

3. Phase 3 에서는 & 에 대한 처리가 필요하다.

& 를 통해 자식 프로세스에서 작업을 처리하게 하고 현재 프로세스(부모 프로세스)는 계속 해서 다른 작업을 처리하도록 할 수 있는데 이를 구현하기 위해서 Signal 에 대한 처리가 필요하다.

그리고 pipe 구현을 recursive 하게 구현함으로써 각각의 명령어를 수행할 수 있는 자식 프로세스들을 생성함으로써 이를 Grouping 하는 작업을 하여 SIGTSTP 와 같은 시그널이 올 경우 grouping 된 모든 프로세스들이 동일하게 작업이 중지하도록 하는 것이 핵심이다.

우선, [ctrl + z] 키를 누르면 SIGTSTP 시그널을 현재 진행중인 Foreground process 에게 보냄으로써 그룹화된 모든 프로세스들이 작업을 중단하도록 해야 한다.

그리고 [ctrl + c] 키를 누르면 SIGINT 시그널을 현재 진행중인 Foreground process 에게 보냄으로써 그룹화된 모든 자식 프로세스들이 죽어서 Kernel 영역에서 죽어버린 모든 자식 프로세스들을 reaping 하는 작업이 이뤄져야 한다.

그리고 bg, fg 와 같은 명령어를 통해서 bg 같은 경우는 중단된 임의의 작업을 background 에서 실행하도록 하는 것이고 fg 같은 경우는 중단된 임의의 작업 또는 백그라운드 프로세스를 포그라운드 프로세스로 실행하도록 하는 것이다.

Kill 명령어는 해당 작업을 terminate 시킴으로써 위에서 말했던 [ctrl + c] 키와 동일하게 처리해주도록 해야 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

명령어 cd, ls, mkdir, rmdir, touch, cat, echo, exit 등 모든 명령어들이 제대로 돌아가는 것을

확인할 수 있다.

그리고 echo “ abc” 이거나 echo ‘ hello ’ 와 같이 여러 따옴표가 들어간 경우에

정확하게 문자열을 파싱하여 결과가 쉘에서 실행한 명령어와 동일하게 실행되는 것을

확인할 수 있다.

1. Phase 2

파이프에 대한 여러 작업들이 정상적으로 실행되는 것을 확인할 수 있다.

그리고 중간에 이상한 명령어가 들어갈 경우 builtin\_command() 함수를 통해서 명령어가

유효한지를 체크하여 해당 명령어가 없는 명령어라는 문구 또한 띄우도록 하였다.

그러나 ls -al | hello | wc 와 같이 중간에 이상한 명령어가 들어갈 경우는 정상적으로 동작

하게 되는데 그 이유는 ls -al | hello 에서의 문구가 명령어가 없는 문구이기 때문에

이러한 output 이 정상적으로 넘어감으로써 결과가 제대로 나오는 것을 알 수 있다.

실제로 리눅스에서 ls -al | hello | wc 와 같은 명령어를 실행해보면 에러 문구가 뜨기는 하나

wc 에 대한 결과가 뜨는 것을 볼 수 있다.

1. Phase 3

Jobs, bg, fg, kill 에 대한 명령어들이 정상적으로 실행되는 것을 볼 수 있다.

또한 & 을 가장 뒤에 붙여서 명령어를 실행하면 background 에서 해당 명령어가 실행되며 정상적으로 다른 작업이 이뤄지는 것을 볼 수 있다.

그리고 [ctrl + z], [ctrl + c] 에 대한 처리도 정상적으로 실행되는 것을 알 수 있다.

* 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() 함수를 통해 child process 를 생성할 수 있는데 parent process 는 child process 가

전부 실행될 때까지 기다려줘야 한다.

즉, zombie process 가 생성되는 것을 방지하기 위함이고 child process 에서는 파싱한 명령어

들을 가지고 execvp() 함수를 통해 실행하게 된다.

* + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow

자식 프로세스가 종료하게 될 경우 부모 프로세스에게 SIGCHLD 시그널을 보내게 되는데

부모 프로세스는 fork() 를 띄웠을 때 pid 값이 자식 프로세스 pid 값이기 때문에 pid 값이

0 이 아닐 경우에 대해서 wait() 함수를 호출함으로써 signal handling 을 할 수 있다.

즉, fork() 를 띄웠을 때 자식 프로세스는 pid = fork() 값이 0 이기 때문에 명령어에 대한

수행이 이뤄지고 부모 프로세스는 pid = fork() 값이 자식 프로세스의 pid 즉, 0 이 아닌

값이기 때문에 wait() 함수를 호출해둠으로써 자식 프로세스가 종료하여 signal 을 보내게

될 경우 자연스럽게 wait() 함수가 수행되면서 signal handling 이 이뤄지게 된다.

* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)

이전에 말한 것 처럼 recursive 방법과 iterative 방법 중 iterative 방법을 통해 구현하였다.

예를 들어서 ls -al | grep csapp | wc 와 같은 파이프가 2개인 명령어를 통해 설명하자면,

1. ls -al 를 처리하기 위한 자식 프로세스를 생성하고 부모 프로세스는 ls -al 에 대한 처리가

끝날 때까지 기다린다.

우선 pipe() 함수를 통해 fd = [in ~> 4, 3 ~> out] 에 대한 파이프를 형성하고 output 에

대한 fd를 dup2() 함수를 통해 fd = 4(in) 을 가르키게 함으로써 execvp() 명령어를 실행한다.

따라서 파이프 [in ~> 4, 3 ~> out] 내부에 ls -al 에 대한 output 이 들어가게 된다.

작업이 끝나면 더 이상 파이프 fd = 4(in) 에 들어가지 못하도록 닫아버리고 그다음 input 에

대한 fd가 파이프 fd = 3(out) 을 받도록 한다.

2. grep csapp 를 처리하기 위한 자식 프로세스를 생성하고 부모 프로세스는 grep csapp

에 대한 처리가 끝날 때 까지 기다린다.

pipe() 함수를 통해 fd = [in ~> 5, 4 ~> out] 에 대한 파이프를 형성하고 input 에 대한 fd를

dup2() 함수를 통해 fd = 3(out) 을 가르키게 하고 output 에 대한 fd 를 dup2() 함수를 통해

fd = 5(in) 을 가르키도록 하여 execvp() 명령어를 실행한다.

여기서 두 개의 파이프가 존재하는 것을 알 수 있는데,

[in ~> 4, 3 ~> out] === [in ~> 5, 4 ~> out] 와 같이 연결되었다.

파일디스크립터 fd = 0 은 3 을 가르킴으로써 ls -al 에 대한 결과를 ( ) 라고 할 때,

grep csapp ( ) 와 같이 실행하게 함으로써 그에 대한 결과가 결국

파이프 [in ~> 5, 4 ~> out] 에 들어가게 되는 것을 알 수 있다.

3. wc 를 처리하기 전에 파이프가 더 이상 존재하지 않는 것을 알 수 있다.

따라서 output 에 대한 fd 를 dup2 함수를 통해 다른 fd 를 가르키도록 하지 않게 함으로써

execvp() 함수가 실행할 때 정상적으로 화면에 출력이 되도록 하는 것이 핵심이다.

그리고 이전에 만들었던 누적된 결과들을 기반으로 하여 execvp() 함수를 통해 결과가

나타나게 되는 것을 알 수 있다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

Pipeline 개수가 n 개가 있다면 (n - 1) 개의 자식 프로세스는 위에서 언급한 것처럼

Pipe 에 output 을 담고 그에 대한 output 을 input 으로 받아들여서 또 다시 pipe 에

Output 을 담는 작업을 반복하였으며 1 개의 자식 프로세스는 누적된 pipe 결과물에 대해서

파일디스크립터 fd = 1 즉, output 을 처리하는 fd 를 더 이상 dup2() 함수를 통해 다른 fd

에 pointing 하지 않도록 하여 execvp() 함수를 호출하여 정상적으로 화면에 결과가 나타나게

하도록 하였다.

* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명

Background process 를 구현하기 위해 pipeline 을 recursive 하게 구현하였다.

recursive 하게 구현한 이유는 iterative 하게 구현할 경우에 pipeline 의 개수가 n 개라고

할 경우 (n + 1) 개의 자식 프로세스가 한번에 생성되는 것이 아니라 step by step 으로

생성되기 떄문에 이로 인한 자식 프로세스들의 grouping 이 힘들었다.

따라서 recursive 하게 구현한다면 fork() 를 연달아 띄운 후에 wait() 을 호출하게 함으로써

각각의 모든 명령어들을 수행할 수 있는 자식프로세스들이 생성되기 때문에 이로 인해

Grouping 이 가능하게 되고 자연스럽게 SIGTSTP, SIGINT 에 대한 시그널이 해당 백그라운드

프로세스에게 보내게 된다면 grouping 된 모든 프로세스들이 kill() 함수를 통해서 적용이되며

자동으로 SIGTSTP, SIGINT 에 대한 처리가 가능하게 된다.

그리고 setpgrp() 함수를 통해서 백그라운드 프로세스의 grouping 을 돕는데 역할을 했다.

SIGTSTP, SIGINT 를 kill() 함수를 통해 백그라운드 프로세스를 중단시키거나 종료시킬 경우

SIGCHLD 시그널이 발동되는 것을 여러 실험을 통해서 확인이 가능했다.

그때 SIGCHLD 에서 wait() 함수를 통해 얻어온 pid 값이 양수값이 아니여서 정상적으로 해당 프로세스가 끝나지 않았음을 암시할 수 있었다.

그리고 job 들을 담기 위해서 Single Linked List 를 생성하여 job 들을 담았으며 메모리 누수가 일어나지 않도록 job 이 종료될 경우에 대해서 free 를 해줬으며 프로그램이 완전히 종료될 경우에 모든 job 들을 탐색하여 free 를 해주고 종료하도록 하였다.

* 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. Job 들에 대한 const 형 변수의 Condition 변수들을 선언하였다.

FG : 포그라운드 프로세스 BG : 백그라운드 프로세스

DIE : 프로세스가 죽은 경우 (Kill)

STOP : Suspend 된 상태 (FG 에서 [ctrl + z] 누른 경우)

DONE : 명령어에 대한 처리 완료한 경우

SINT : 시그널 SIGINT 을 받은 상태 (FG 에서 [ctrl + c] 키를 누른 경우)

dummy : fork 한 자식 프로세스들이 파싱한 명령어들을 참조하기 위해 선언한 2차원 배열

2. Job 들을 담기 위한 구조체를 선언하였다.

가장 마지막 job 을 가르키도록 하고 다음 job 을 가르키도록 back, next 를 선언하였으며 job 의 pid 를 담기 위한 변수, 상태를 담기 위한 변수, 가장 마지막 job 의 숫자를 담기 위한 변수, job 의 숫자를 나타내기 위한 변수, 실행중이거나 중지된 모든 job 들의 개수를 확인하기 위한 변수, 현재 stop 된 job 이 몇 개인지 확인하기 위한 변수, 현재 job 의 명령어가 무엇인지를 확인하기 위한 변수를 설정하였다.

3. 백그라운드 프로세스로 전환하거나 포그라운드 프로세스로 전환하거나 할 때의 명령어 마지막에 & 를 붙이거나 빼도록 하는 함수 cmd\_change(char \*cmd, int cond) 를 선언하였다.

예를 들어서 bg %3 을 하게 된다면 명령어 뒤에 & 가 붙게 되면서 출력이 되고 백그라운드 프로세스가 실행이 끝나거나 suspend 되거나 할 때는 명령어 뒤에 & 가 붙지 않는다.

이를 위해 명령어를 변경할 수 있도록 하는 함수를 구현하였다.

4. 프로그램이 종료되면 모든 job 들을 free 해주는 함수 free\_all\_jobs() 를 선언하였다.

독특하게도 job 에 달린 명령어는 따로 strdup() 함수를 통해 만든 문자열이다.

따라서 이 또한 heap 영역에서 특정 문자열을 포인팅하고 있기 떄문에 메모리 누수를

방지하기 위해 문자열 또한 free 해주도록 하였다.

그리고 현재 job 중에 죽지 않거나 종료되지 않은 경우 grouping 된 프로세스들을 죽이도록 구현하였다. (kernel 측에서 죽이지 않고 직접 부모 프로세스인 main 프로세스가 해결하도록 구현하였음)

5. 명령어를 실행시키는 job 을 추가하는 함수 add\_job(pid\_t pid, int cond, char \*cmd) 를 선언하였다.

parameter로 pid 값과 FG 인지 BG 인지를 판단하는 cond, 명령어 문자열 cmd 를 받아와서 전역변수로 선언한 jobs 에 job 을 추가하도록 하는 함수를 구현하였다.

6. 특정 job 을 삭제하기 위한 함수 delete\_job(pid\_t pid) 를 선언하였다.

parameter로 pid 값이 동일할 경우 해당 job 을 삭제하도록 하는 함수이다.

이 함수는 유일하게 포그라운드 프로세스가 정상적으로 실행될 경우에 호출되는 함수이며 포그라운드 프로세스가 정상적으로 실행되서 종료될 경우에는 jobs 에서 뺴도록 하여 나중에 jobs 출력시 해당 job 이 출력되지 않도록 하기 위함이다.

그리고 strdup() 함수를 통해 만든 문자열에 대해서 포인팅하고 있기 떄문에 메모리 누수를

방지하기 위해 문자열 또한 free 해주도록 하였다.

7. 실행중이었던 프로세스가 종료되거나 죽거나 SIGINT 시그널을 받게 된 job 들을 삭제하기 위한 함수 clear\_jobs(int op) 를 선언하였다.

명령어를 입력하고 자동으로 job 들에 대한 clear 작업이 이뤄지도록 해야 한다.

이 함수는 main 함수에서 실행되며 jobs 명령어를 입력할 때에도 실행하도록 하였다.

그 이유는 좀 더 리눅스 환경의 jobs 들에 대한 출력 문구가 동일하도록 처리하기 위함이다.

8. 프로세스가 끝난 경우를 처리하는 함수 set\_done\_job(pid\_t pid) 를 선언하였다.

명령어 실행이 끝나서 자식 프로세스가 종료하게 될 경우 SIGCHLD 시그널을 통해 핸들러가 호출하게 된다.

핸들러는 위와 같은 함수를 통해 종료한 job 들을 DONE 으로 마킹함으로써 나중에 clear\_jobs() 함수를 통해 실행이 종료된 job 들의 free 가 이루지도록 한다.

9. jobs 명령어를 처리하는 함수 prin\_all\_jobs(Job\* job, int stop\_count) 를 선언하였다.

쉘에서 여러 번의 시행착오를 통해 suspend 된 job 들의 개수에 따라서 [n]+, [n]- 이런식으로 붙여지는 것을 확인할 수 있었다.

따라서 +, -, 공백문자 이런식으로 [n] 에 붙이는 작업을 하기 위해서 stop\_count 을 고려하여 처리하였다.

그리고 job 의 실행이 끝나서 DONE 으로 마킹될 경우 백그라운드 프로세스가 마지막에 붙여있던 & 문자가 사라지는데 이러한 경우를 고려하기 위해 가장 윗부분에 job 이 DONE 인 경우를 고려해서 백그라운드 프로세스인 경우 ‘&’ 을 ‘\0’ 으로 변경시켜줌으로써 정상적으로 출력이 되도록 하였다.

10. SIGCHLD 시그널은 받을 때 처리하는 함수 sigchld\_handler(int sig) 를 선언하였다.

만약 정상적으로 백그라운드 프로세스가 종료하게 될 경우 SIGCHLD 시그널을 부모 프로세스가 받게 되면서 set\_done\_job 함수를 통해 job 을 DONE 으로 마킹하여 clear 하도록 할 수 있으며 비정상적으로 SIGTSTP 시그널이 호출될 경우 pid 값이 양수가 아닌 값이 들어오게 됨으로써 이는 무시된다.

11. SIGTSTP 시그널을 처리하는 함수 sigtstp\_handler(int sig) 를 선언하였다.

[Ctrl + z] 를 처리하기 위한 대표적인 시그널이며 Foreground process 에서만 위와 같은 시그널을 받아서 호출하게 되는 함수이다.

따라서 모든 job 을 탐색하여 FG 인 job 을 찾아서 해당 job 의 상태를 STOP 상태로 변경하고 해당 job 의 grouping 된 모든 process 들을 SIGTSTP 을 보냄으로써 kernel 측에서 grouping 된 실행중인 모든 프로세스들을 중단시키고 해당 job 이 중단되었다는 문구를 띄우고 종료하게 된다.

12. SIGINT 시그널을 처리하는 함수 sigint\_handler(int sig) 를 선언하였다.

대표적으로 [ctrl + c] 에 대한 시그널로 포그라운드 프로세스만이 허용되므로 포그라운드 프로세스에 대한 job 을 찾아서 해당 job 의 상태를 SINT 으로 변경하여 grouping 된 모든 실행중인 프로세스들에 대해서 SIGINT 을 하여 죽인다.

그러면 main () 함수의 while 문에서 eval() 함수 실행이 종료되면 clear\_jobs() 함수를 호출하게 됨으로써 해당 job 이 삭제됨과 동시에 free 되게 된다.

13. 파이프라인을 처리할 수 있는 함수 operate(char \*cmd[][MAXARGS], int fd, int depth) 를 선언하였다.

Phase 3에서는 Recursive 하게 구현하였으며 built\_command() 함수를 통해서 각 명령어들이 유효한지에 대한 체크와 pipe 를 이용하여 iterative 에서 했던 처리와 동일하게 구현하였다.

그리고 builtin\_command 함수를 계속해서 체크함으로써 각각 명령어가 유효한지 체크도 해주었다.

Pipe 에 대한 자세한 설명은 위에서 하였으므로 패스한다.

14. 단일 명령어만을 처리할 수 있는 함수 operate\_one(char \*\*cmd) 를 선언하였다.

단지 명령어 실행을 위한 execvp() 함수를 호출하여 오류 발생 시 exit(0); 을 호출하도록 하였다.

15. 현재 명령어가 grouping 이 유효한지 아닌지 체크하는 함수 check\_grouping(char \*\*argv, char \*(cmd[][MAXARGS]), int pipe) 를 선언하였다.

이 함수를 Phase 3에서 구현하게 되면서 굉장히 많은 issue 가 존재하였다.

우선, 파이프라인일 경우 cat filename | less 와 같은 명령어가 실행되지 않았다.

그리고 단일 명령어일 경우 cat, grep, less 또한 실행되지 않았다.

fork() 를 띄우자 마자 자식 프로세스 전부를 Grouping 을 해버렸는데 그로 인해 제대로 실행되지 않는 issue 가 발생하였고 이를 해결하기 위해 단일 명령어일 경우와 파이프라인의 명령어들에 대해서 cat, grep, less 와 같은 경우가 들어올 경우에 grouping 을 하지 않도록 하며 그 외에는 grouping 을 허용하도록 함수를 구현하였다.

16. main 함수에서 받아온 cmdline 을 가지고 명령어를 실행시킬수 있도록 하는 함수 eval(char \*cmdline) 을 수정하였다.

우선 백그라운드 프로세스인지 아닌지를 판별하여 조건을 둘로 나눴으며 동일하게 fork() 를 띄워서 명령어를 처리할 수 있도록 하는 자식 프로세스를 대표로 생성한다.

이때 단일 명령어일 경우와 파이프라인이 있는 명령어일 경우를 고려해서 조건을 또 나눴으며 grouping 이 될 경우 setpgrp() 함수를 호출하여 현재 groupid 가 현재 pid 값이 되도록 설정하였다.

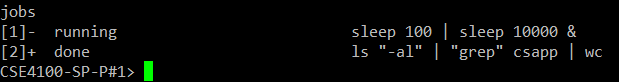
그리고 백그라운드 프로세스일 경우에는 단순히 job 을 추가만 시키고 포그라운드 프로세스일 경우에는 job 을 추가시켜서 waitpid() 함수를 호출하는데 세 번째 인자에 WUNTRACED 를 넣어준다.

그 이유는 비정상적으로 suspend 될 경우에 호출이 되게끔 하여 빠져나오도록 하여 ctrl+c 가 실현되게끔 하고 status 값을 통해 정상적으로 포그라운드 프로세스가 실행이 되었다면 job 을 삭제하도록 하였다.

17. job 의 command 문자열을 만들기 위한 함수 set\_cmdline(char \*cmdline, char \*\*argv, int bg) 를 선언하였다.

아래 사진과 같이 jobs 명령어를 입력하면 command 문자열이 오른쪽에 나타나게 되는데 이러한 command 를 보기 좋게 만들기 위한 함수이다.

즉, 수많은 공백 문자가 중간에 들어와도 단 하나의 공백문자로 처리되며 파이프가 붙여있어도 출력될 때 붙여있지 않고 공백을 붙여주도록 하였다.



18. 따옴표로 들어오는 명령어들을 따옴표 없이 각각의 명령어들을 담도록 하는 함수 set\_command(char \*cmd[][MAXARGS], char \*\*argv) 를 선언하였다.

command 를 pipeline 을 처리할 때 보기 좋게 하기 위해서 위와 같이 처리하였다.

그리고 따옴표에 대한 경우 execvp 함수는 따옴표를 처리하지 않기 때문에 이를 방지하기 위해서 여러 조건을 둬서 해결하였다.

19. jobs, bg, fg, kill 명령어를 처리하기 위한 builtin\_command(char \*\*argv) 를 수정하였다.

Jobs 같은 경우는 job 이 하나 이상일 경우에 모든 job 들을 출력하도록 했으며 실행이 다 된 백그라운드 프로세스가 있을 수 있으니 clear\_jobs() 을 호출함으로써 실행이 다 된 job 들을 clear 해주도록 하였다.

bg 명령어 같은 경우는 현재 삽입된 모든 job 들을 탐색하여 해당 job의 숫자와 동일할 경우에 이미 백그라운드일 경우에는 에러 문구를, 아닐경우에는 suspend된 job이 BG 이 되므로 stop\_count 값을 1 감소 시켜서 명령어가 ‘&’ 가 붙도록 변경시켜줬으며 상태도 BG 로 변경시킴으로써 해당 pid 에 SIGCONT 시그널을 보냄으로써 커널측에서 해당 프로세스를 다시 실행시키도록 하였다.

fg 명령어는 bg 와 동일하게 job의 num 을 비교하여 같을 경우에 해당 job 의 상태를 FG 으로 변경시킴으로써 SIGCONT 시그널을 해당 프로세스에게 보내도록 하였다.

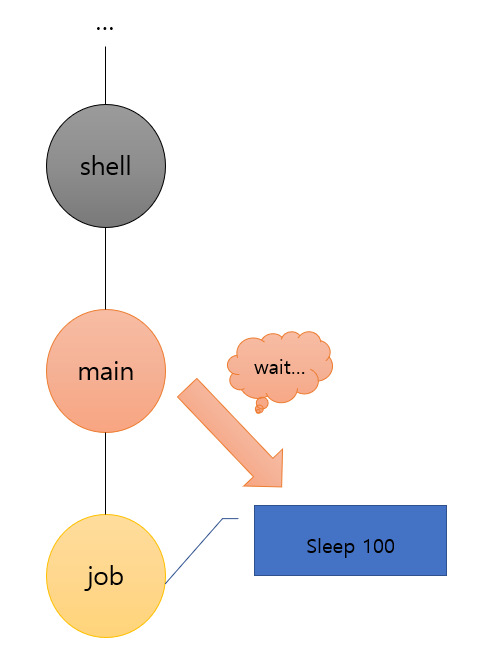
그리고 나서 waitpid() 를 eval() 함수에서 호출하는게 아니기 때문에 단순하게 while 루프를 실행시킴으로써 현재 실행중인 포그라운드 프로세스의 condition이 FG 가 아닐때까지 계속해서 main 프로세스는 기다리고 실행이 끝나게 되면 자연스럽게 SIGCHLD 시그널이 발동하여 시그널 핸들러를 통해 waitpid() 함수를 호출하여 해당 프로세스를 reaping 하면서 동시에 job condition 이 DONE 으로 마킹되므로 while 루프를 빠져나오게 된다.

Kill 명령어는 해당 job 에 대해서 SIGINT 시그널을 보냄으로써 job condition 또한 DIE 로 마킹하여 eval() 함수가 끝나고 나서 clear\_jobs() 를 호출할 때 해당 job 을 제거하도록 해준다.

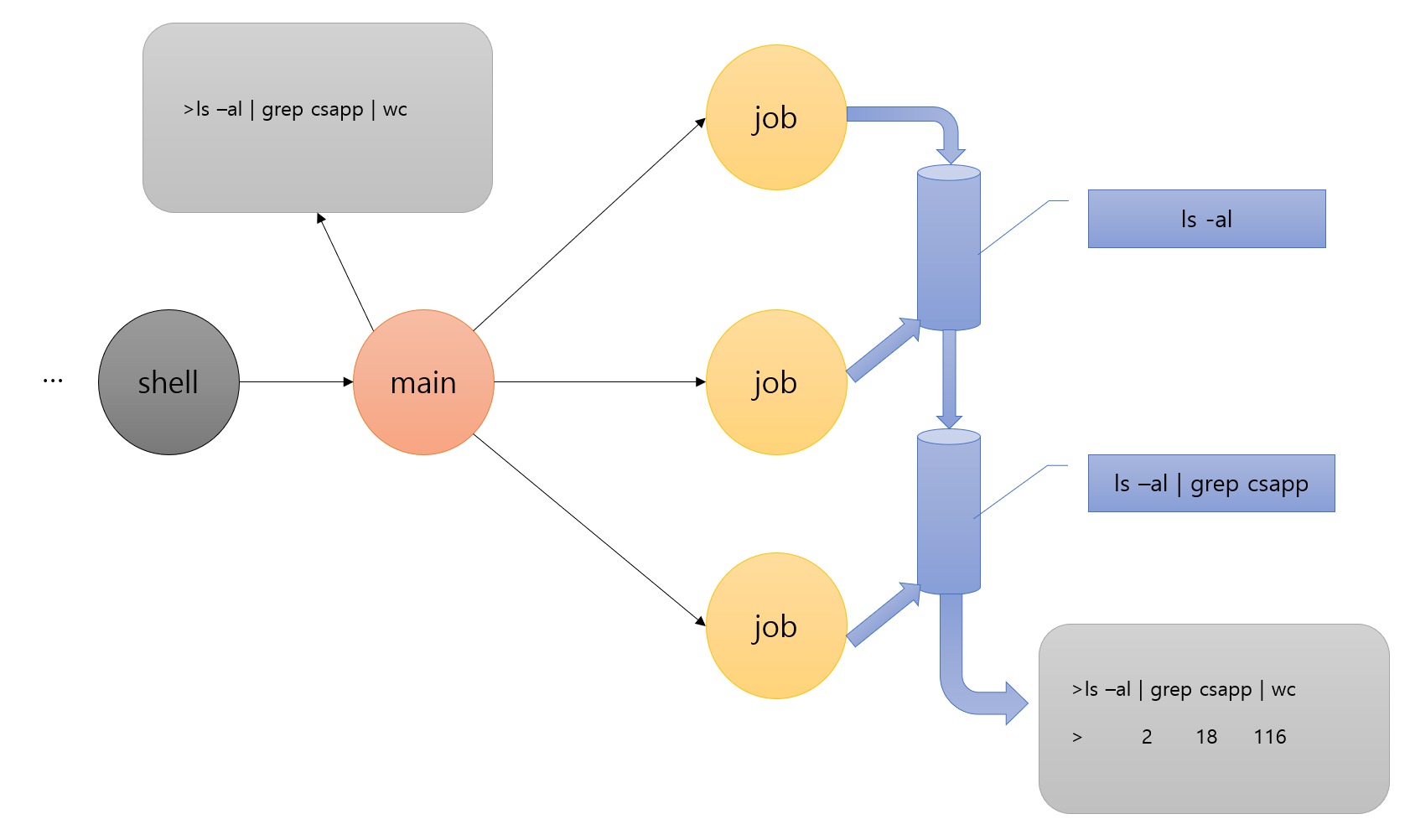
1. **구현 결과**
   1. **Flow Chart**

* **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)**

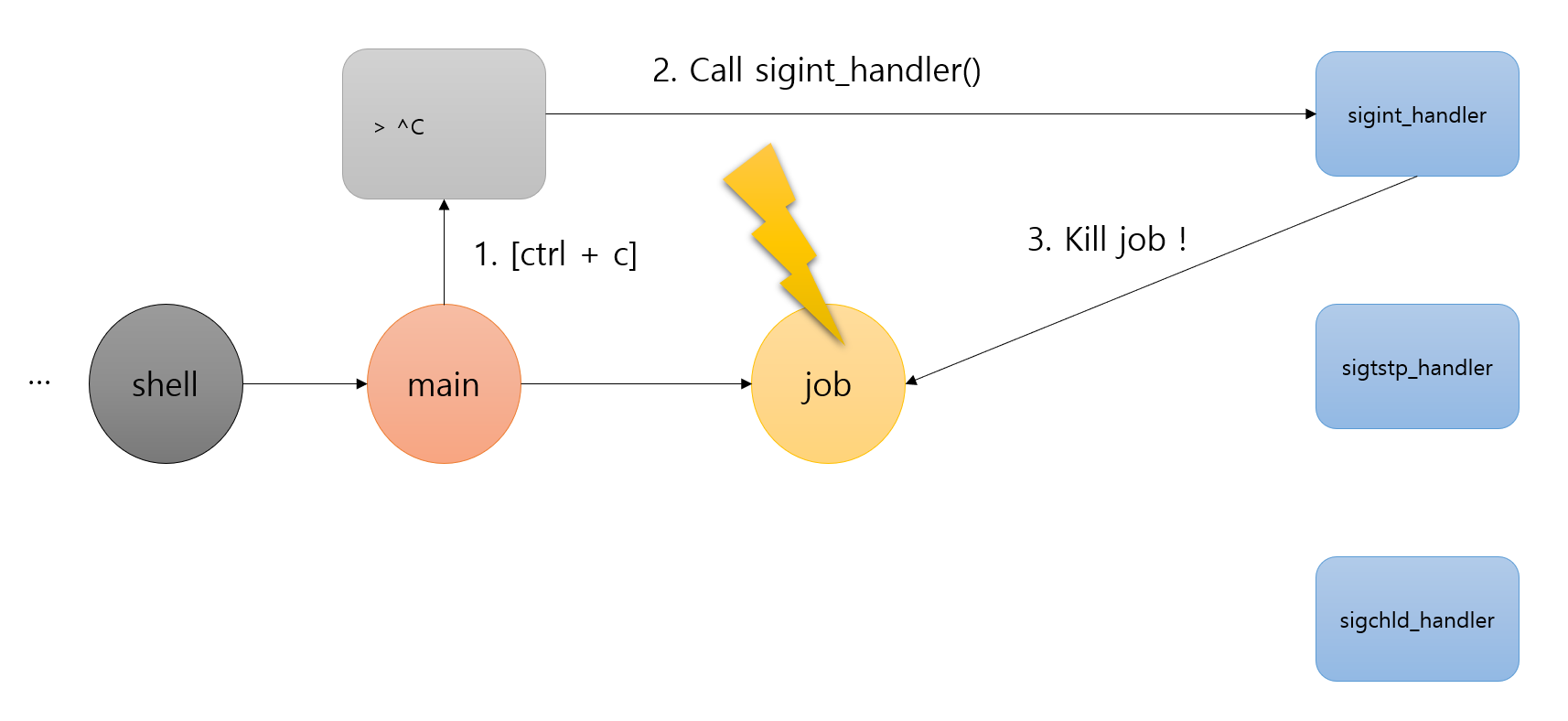


1. **Phase 2 (pipeline)**

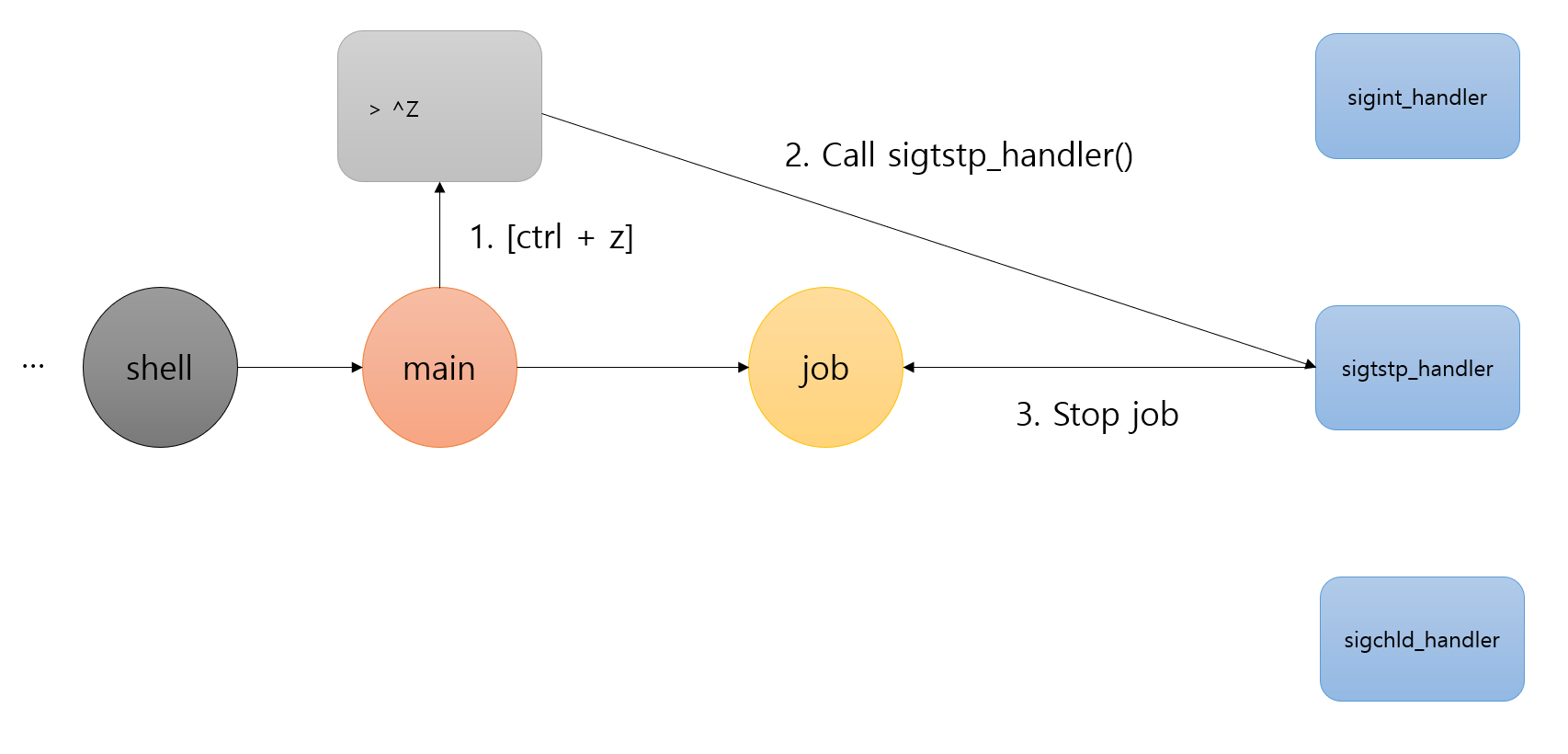


1. **Phase 3 (background)**

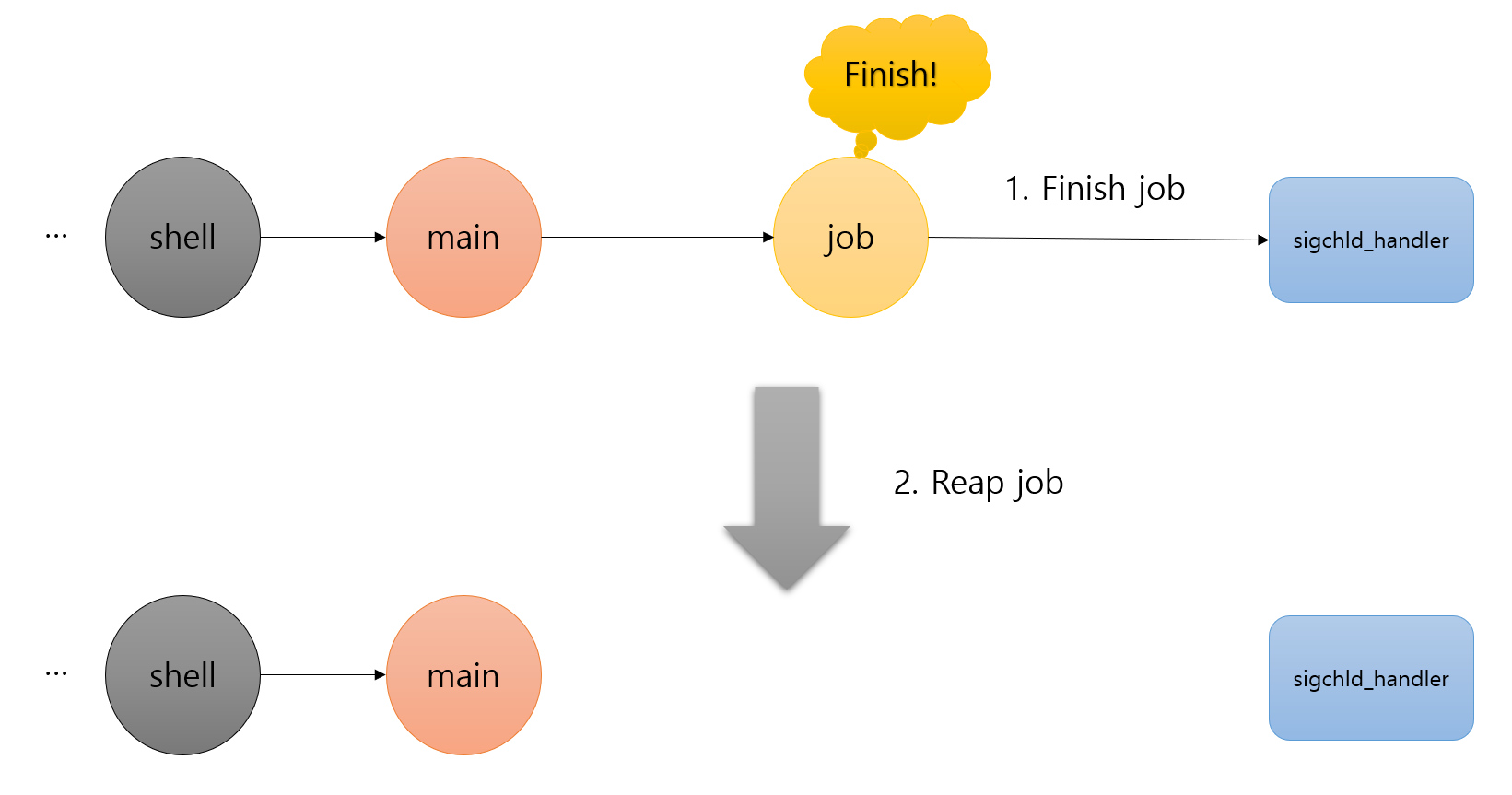
**(1) SIGINT**



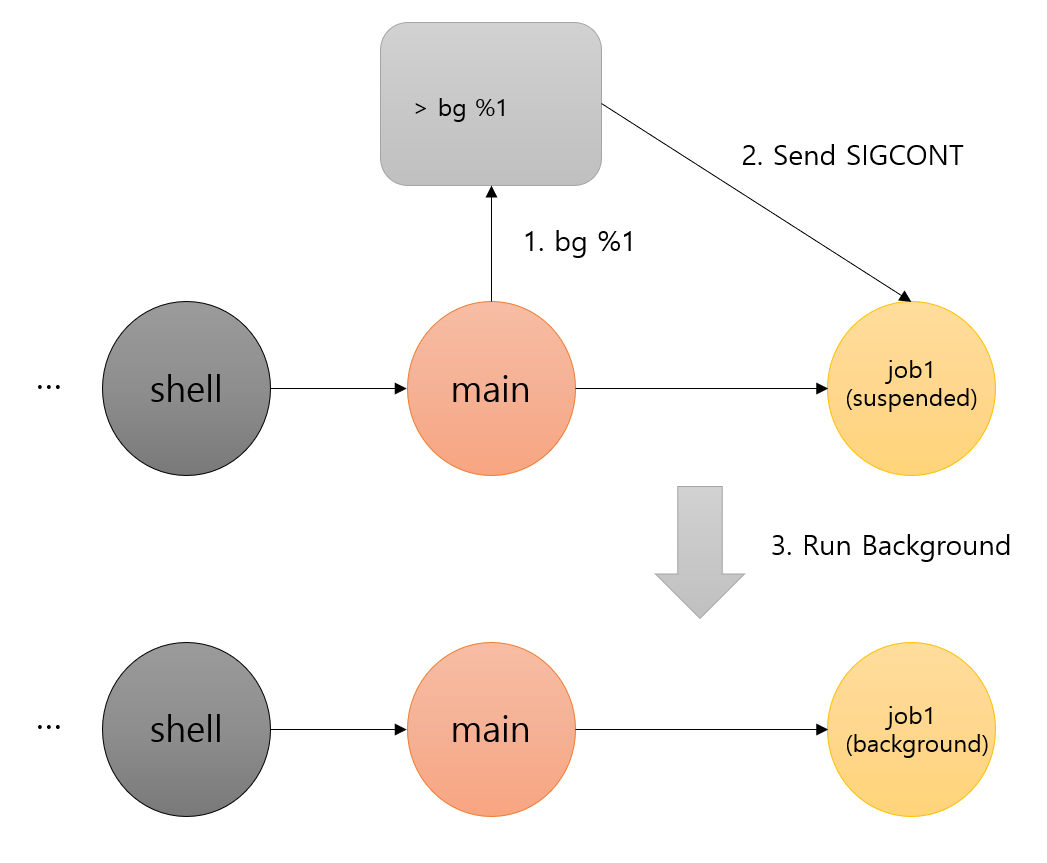
**(2) SIGTSTP**



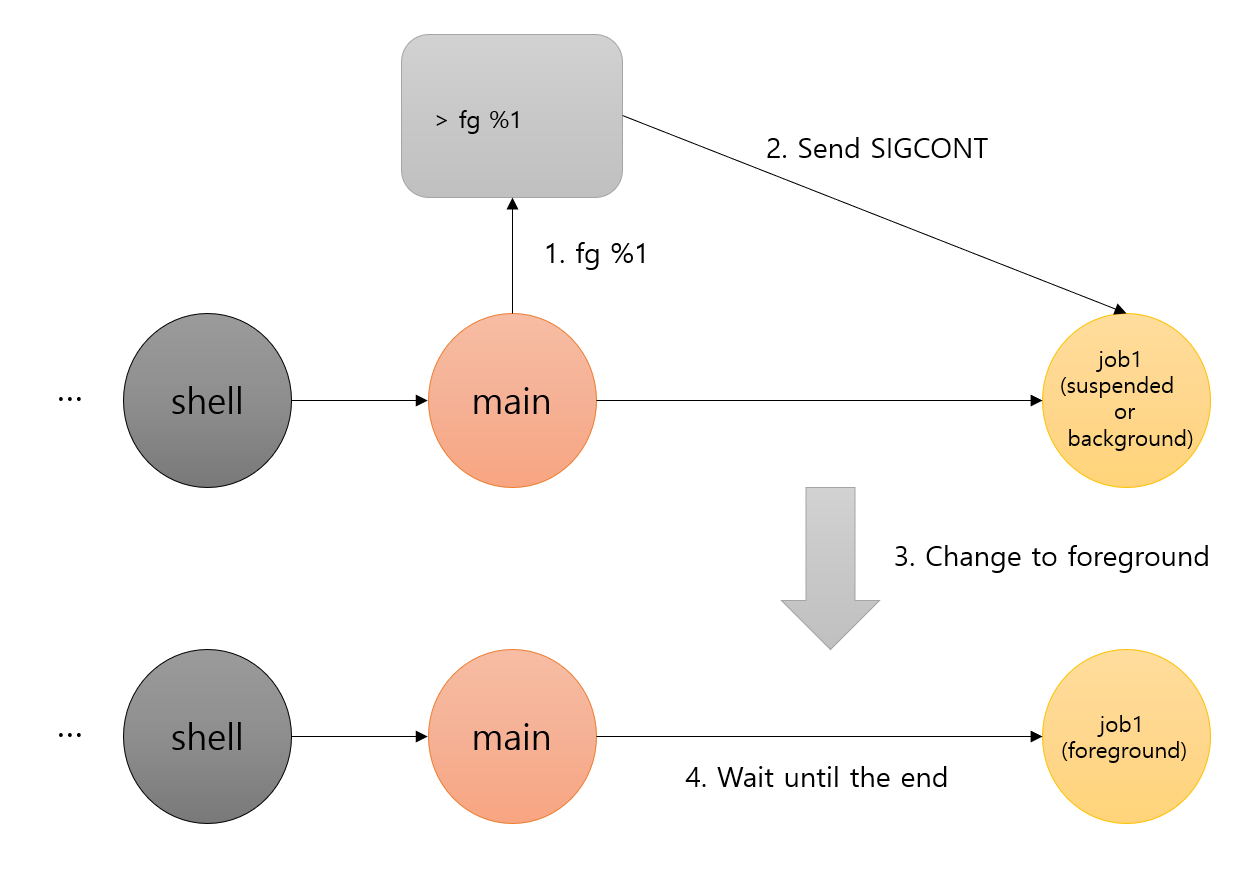
**(3) SIGCHLD**



**(4) bg %n**



**(5) fg %n**



**(6) kill %n**

