**System Programming Project 1**

담당 교수 : 김영재 교수님

이름 : 전찬

학번 : 20201635

1. **개발 목표**

수업에서 배운 ECF(exceptional control flow), exception, signal을 바탕으로 기본적인 process(cd, ls, mkdir 등의 기본적인 command)을 수행할 수 있는 shell을 직접 구현한다. 추가로 zombie process는 system resources의 낭비를 유발한다. 따라서 shell process에서 종료된 process의 reaping의 수행을 구현한다.

또한 위를 확장하여 서로 다른 process들을 연결시켜줄 수 있는 pipeline을 구현하며 한 process의 output이 다른 process의 input로 연결될 수 있는 형태를 구현한다.

마지막으로 실제 linux shell에서 & 키워드를 통해서 process가 background에서 실행되는 형태처럼, 실제로 구현한 shell에서도 이러한 background 실행을 수행할 수 있도록 shell을 구현한다. jobs, bg, fg, kill 명령어를 구현하며, background process을 제어한다. background에서 실행된 process가 종료된 경우 reaping을 수행할 수 있도록 shell을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Phase 1

간단한 명령어를 사용할 수 있는 shell을 구현했다. 구현한 shell은 exit command로 종료할 수 있으며, cd command를 통해 cd .., cd, cd filename 와 같이 실제 linux에서 사용할 수 있는 cd의 여러 기능을 수행할 수 있도록 구현했다. 또한 ls, mkdir, touch 등의 기본적인 shell command를 사용할 수 있도록 shell을 구현했다. 추가로 구현한 shell이 실행되고 있을 때, ctrl + C 를 입력해도 바로 종료되지 않도록 signal handling을 shell 내부에 구현했다.

1. Phase 2

Phase 1에서 구현한 shell에 pipelining 기능을 추가했다. 따라서 command input에서 “|” 단위로 command을 나누며, 한 process의 output이 다른 process의 input이 되는 형태를 구현했다.

1. Phase 3

background로도 process를 실행할 수 있는 shell을 구현했다. command input의 마지막 문자가 “&” 이라면 입력한 command가 의미하는 process가 shell의 background에서 수행되는 형태이다. 추가로 jobs, bg, fg, kill 등의 command를 통해 background process를 제어할 수 있는 형태를 구현했으며, Phase 1의 ctrl + C의 기능에 더 나아가 ctrl + Z를 통해 현재 foreground에서 실행되고 있는 process를 정지 상태로 변경할 수 있도록 signal handling을 구현했다.

* 1. **개발 내용**
* **Phase 1 (fork & signal)**

1. fork를 통한 child process의 생성

fork는 하나의 추가적인 process를 생성해내는 함수로, return값이 2개라는 특징을 갖는다. fork()가 실행되면, fork()함수를 call한 process는 parent process가 되며, 새롭게 생성된 process는 child process가 된다. 이렇게 생성된 child process는 parent process virtual address space, open file descriptors, parent process의 저장된 register등 대부분의 정보를 복사하게 된다. 이때 가질 수 있는 2개의 return값은 parent process = process id of child process, child process = 0 이 된다.

linux shell에서 어떠한 프로그램을 실행할 때는, fork() -> execute() in child process 와 같은 흐름을 이룬다. 결국 프로그램을 실행하기 위해서 위에서 설명한 fork() function call이 필요하며, fork()는 return 값이 2개라는 큰 특징이 있다. 따라서 child process를 생성하고, 제어하기 위해서 fork()의 return값이 0인 child process을 조건문을 바탕으로 다룰 필요가 있다.

2. connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow

Phase 1에서 signal handling이 필요한 경우는 2가지 경우가 존재한다. 바로 SIGCHLD와 SIGINT에 대한 handling 이다. 우선 SIGINT의 경우에, 키보드의 입력을 통해 발생하는 signal이다. 이러한 SIGINT signal에 별도의 signal handling을 수행하지 않으면, 구현한 shell 또한 종료되어, linux로 돌아가는 것을 파악할 수 있다. 따라서 sigint\_handler()를 작성하여 SIGINT가 발생되었을 때 shell을 종료하지 않도록 구현해야 하는데, 이는 간단하게 sigint\_handler()가 별다른 동작을 하지 않도록 설정하면 된다. 개인적으로 구현한 shell에서는 stdout에 “\n” 를 print 해주는 형태로 sigint\_handler()을 작성했다.

두 번째로 SIGCHLD을 control 해야 한다. SIGCHLD의 경우에는, fork()를 통해 만들어낸 child process가 종료된 경우에 받을 수 있는 signal이다. 하지만 이 SIGNAL의 기본 처리는 ignore이기 때문에 waitpid를 통해 child process를 reaping하는 것 이외에는 별다른 처리를 수행하지는 않았다.

* **Phase 2 (pipelining)**

1. Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)

pipeline을 구현하기 위해서 Phase 1에서 shell에서 변경해야 하는 부분은 크게 두 가지로, parseline(command input을 적절한 형태로 나누는 함수), execute(command을 실행하는 함수) 이다. 우선 parseline 함수에서는 Phase 1과 다르게, “|” 입력을 토대로 command을 추가로 나눌 수 있도록 구현했다. 이후에 command을 array에 저장할 때 string의 1차원 array가 아닌 여러 command 단위로 저장할 수 있는 2차원 string array로 구현했다.

두 번째로 execute 부분을 변경해야 했다. “|” 가 존재하는 경우, 입력된 command가 하나가 아니기 때문에, 많은 command를 실행할 수 있으며, pipe(), dup2() 함수를 통해 각 process의 input, output을 연결시킬 수 있는 pipe\_execute()라는 함수를 새롭게 정의했다. 위 함수의 경우에는 fork()와 함께, 만들어진 child process의 STDIN\_FILENO, STDOUT\_FILENO을 pipe() 함수와 dup2() 함수를 통해 이전에 실행한 process의 output에 현재 process의 input, 현재 process의 output에 다음 process의 input을 연결하는 형태로 구현했다. 이렇게 만들어낸 pipe\_execute() 함수는 재귀 함수로 현재 수행하는 command의 다음 command을 수행하는 pipe\_execute()를 다시 수행하는 형태로 구현했다.

2. Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

기본적으로 남은(이후에 수행해야 할) command의 개수를 pipe\_execute() 함수의 parameter로 사용했다. 이 parameter을 통해, command를 수행하는 pipe\_execute() 함수에서 남은 command가 1개인지, 혹은 2개 이상인지에 따라 조건문을 통해 구현을 다르게 했다. 우선 command가 1개인 경우(pipeline이 존재하지 않는 경우)에는, 새로운 pipe을 만들 필요 없이, dup2() 함수를 통해 이전에 만들어낸 pipe의 output에 현재 process의 STDIN\_FILENO에 연결해주는 형태로 구현했다. 또한 이 경우에는 마지막 command이기 때문에, recursive하게 다시 한 번 pipe\_execute()를 call할 필요 없이 종료하도록 함수를 구현했다.

만약 아직 command가 2개 이상 남아있는 경우(pipeline이 남아있는 경우)에는, 새로운 pipe를 만들어주며, 현재 실행할 command의 input으로 이전에 만들어낸 pipe의 output 부분을 연결하며, 새로운 pipe을 만들어내며, 이 pipe의 input으로 현재 command의 output을 연결하는 과정을 수행했다. 또한 command가 2개 이상인 경우, 아직 남아있는 command를 추가로 수행하기 위해, 위에서 설명한 것처럼 recursive 형태로 pipe\_execute() 함수의 call을 수행했다. 이 경우에 현재 command의 개수에서 – 1을 한 개수를 command 개수로 함수의 parameter에 전달하는 형식으로 구현을 수행했다.

* **Phase 3 (background process)**

1. Background (’&’) process를 구현한 부분에 대해서 간략히 설명

background로 어떠한 process을 실행하기 위해서는, Phase 1, Phase 2와 다르게 실행한command에 대해 reaping을 위한 wait() 함수를 call 하지 않으면 된다. 하지만 이 경우에, 큰 문제가 발생하는데, 실행을 마친 process에 대해 reap을 할 수 없다는 것이다. 따라서 이를 reaping할 수 있도록 함수를 구현해야 하는데, 위에서는 기본 설정이 ignore이기 때문에 별다른 구현을 하지 않은 sigchld\_handler()에서 SIGCHLD가 발생하는 경우에 reaping을 수행할 수 있도록 함수를 구현했다. 하지만 sigchld\_handler()에서 항상 reaping을 수행하는 경우에도 문제가 발생할 수 있는데, foreground 실행에서 reaping하는 것과 겹쳐서 shell에서 sigchld\_handler()을 통해 이미 reaping된 process을 계속 기다릴 수 있다는 점이다. 따라서 fg인지 bg인지를 파악하는 전역 변수를 설정함을 통해서 fg인 경우에는 sigchld\_handler()에서 별다른 reaping을 하지 않으며, shell에서 reaping을 수행하도록, bg인 경우에는 sigchld\_handler()에서 reaping을 수행할 수 있도록 sigchld\_handler()을 구현했다.

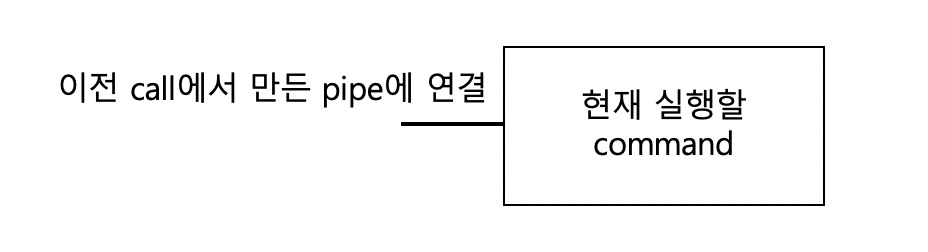
* 1. **개발 방법**
* **Phase 1 (fork & signal)**

위에서 이야기한 것처럼, input command를 실행하기 위해서는, fork(), 그리고 execute()하는 과정이 필요하다. 대부분의 command(ls, cat 등)은 이 방법으로 쉽게 구현할 수 있지만, cd와 같은 경우에는 /bin/cd 형태로 구현되어 있지 않다. 따라서 C에서 사용할 수 있는 함수인 chdir()을 통해서 cd를 구현해야 하는데, 여기에서 추가적인 문제는 chdir() 함수는 내부에서 fork(), execute() 하는 과정을 갖는다는 점이다. 따라서 구현한 shell에서는 fork(), execute() 할 필요 없이, built in command에 cd을 추가해주는 형식으로 shell을 구현했다.

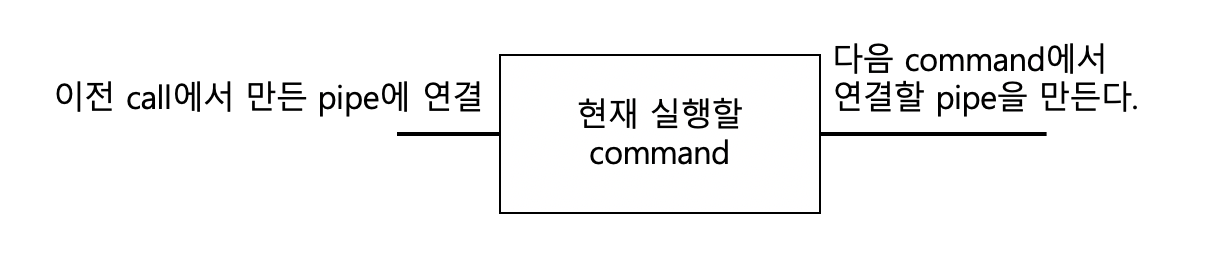
* **Phase 2 (pipelining)**

Phase 1에서 command을 실행하는 execute() 함수를 pipe\_execute() 함수로 확장했다. pipeline 형태를 구현하며, 남은 command에 따라 다른 역할을 수행하는데, 만들어내는 구조는 아래와 같다.

- 남은 command가 1개인 경우



- 남은 command가 2개 이상인 경우



따라서 위 형태를 통해서 pipe로 연결된 여러 process의 실행을 구현했다.

추가로, dup2() 함수를 통해 pipe를 STDIN / STDOUT으로 바꿔준 이후 불필요한 fd table의 data는 close() 함수를 통해 정리하는 형태로 코드를 작성하기도 했다.

* **Phase 3 (background process)**

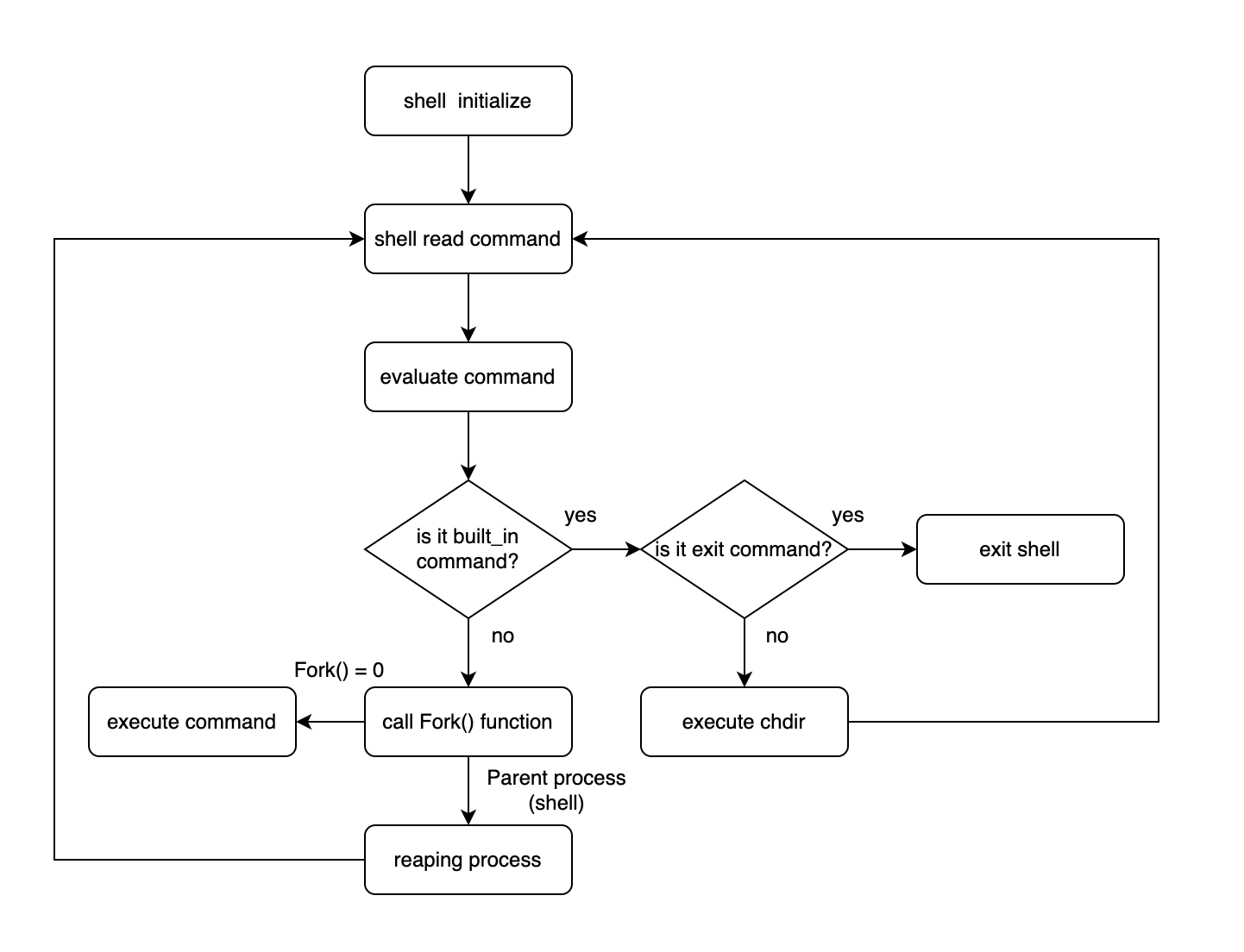
Phase 3에서 처음으로 구현해야 할 것은, 현재 background에서 실행되고 있는 process들이 무엇이 있는지를 파악하는 방법이었다. 우선 background에서 실행될 수 있는 process는 무수히 많을 수 있기 때문에, background process을 하나의 정보 이상을 저장하는 array 형태로 구현해야 했다. 따라서 단순한 static array와 size을 바꿀 수 있는 linked list array 형태 중에서 bg process을 저장하는 방법을 골랐는데, shell이 많은 process을 처리해야 하며, 종료된 process는 배열에서 삭제해주어야 하고, 새로운 bg process는 list에 새롭게 추가해야 되기 때문에, 쉽게 삽입, 삭제를 수행할 수 있는 linked list 형태로 bg process을 유지하는 방법을 선택했다. 각 bg process의 command, pid, background process number 등을 저장하는 struct로 linked list을 구현했다.

더 나아가 bg process가 shell에 추가되다 보니 다양한 문제가 발생했다. foreground에서 실행하고 있는 process를 위한 executing\_process struct 또한 새롭게 정의했으며, sigint\_handler() 을 위에서 설명한 것처럼 현재 실행이 fg, bg인지에 따라 조건문으로 나누었으며, waitpid(-1, &status, WNOHANG) 형태와 함께, 종료된 bg process를 bg process linked list에서 제거하는 작업 또한 sigchld\_handler()에서 수행할 수 있도록 구현했다. 또한 ctrl + Z 입력을 통해 SIGTSTP signal을 보내는 경우에도 shell에 SIGCHLD signal이 설정되어서 sigchld\_handler()가 자동적으로 수행되는 문제가 존재하기도 했는데, 이는 waitpid(pid, &status, WUNTRACED) 형태로 고칠 수 있었다.

또한 built in command에 대한 처리 또한 추가해주어야 했다. jobs, bg, fg, kill command를 추가로 구현했는데, bg, fg로 실행하는 process가 현재 stopped/suspended 된 상태인 경우, kill(pid, SIGCONT) 형태로 running 상태로 바꿀 수 있도록 signal을 전달해 주었다. 또한 shell에서 exit command을 통해 shell이 종료될 때, malloc을 통해 linked list에서 차지한 메모리가 낭비되지 않도록 free\_all\_list() 함수를 구현했다.

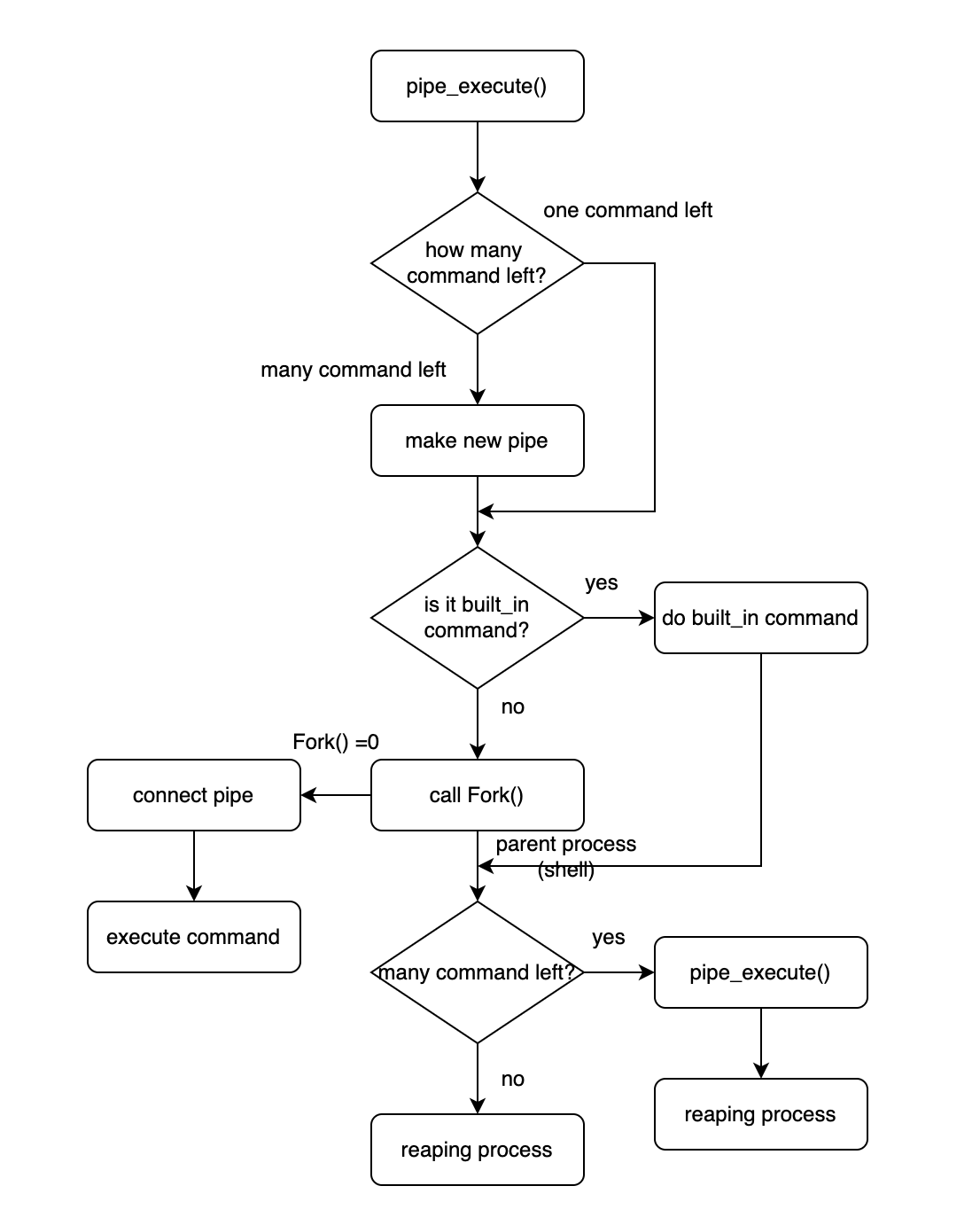
마지막으로 SIGINT, SIGTSTP signal control에 관련해 문제가 존재했다. bg라고 설정한 process 도 실제로는 shell이라는 공통의 parent을 가진 process이기 때문에, SIGINT, SIGTSTP이 발생한 경우, fg process에도 이 signal이 모두 전달되는 형식으로 shell이 실행되었다. 따라서 이를 막기 위해 fork() 로 만들어지는 모든 child process는 SIGINT을 mask 하였고, 만약 fg process에 대해 SIGINT로 종료를 원한다면, 이를 SIGKILL을 통해 종료하는 형태로 shell을 구현했다. 따라서 기능은 정상적이지만, 큰 틀을 먼저 잡지 않고 코드를 작성하다 보니 약간은 비효율적, 혹은 복잡한 형태로 shell을 구현하게 되었다.

1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**

****

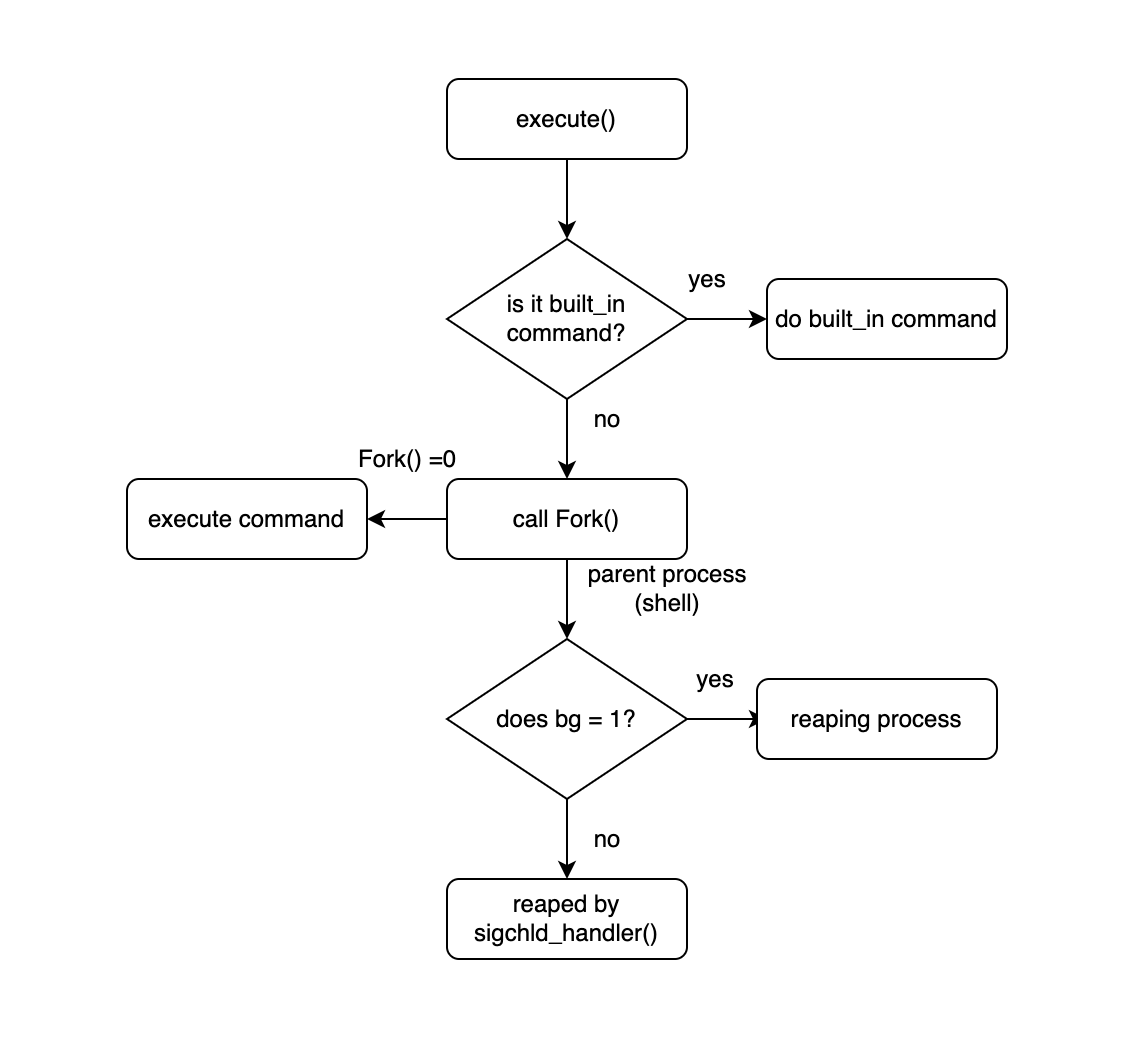
위와 같은 flow로 실행되는 shell을 구현했다. shell은 command input을 읽으며, 이를 실행하게 된다. built\_in command인지, 아닌지에 따라 fork()을 shell에서 수행해주며, fork()로 만들어진 child process은 shell process에 의해 reaping 된다. 이를 exit command input이 존재하기 전까지 계속해서 반복하는 형태이다.

1. **Phase 2 (pipeline)**

****

위는 pipe\_execute가 내부적으로 어떻게 실행되는지를 보여주는 flowchart 이다. pipe\_execute는 남은 command 개수에 따라 새로운 pipe를 생성하며, built\_in command인 경우는 exit, cd 2가지의 command밖에 존재하지 않으며, 곧바로 종료할 수 있다. 이외의 경우에는 pipe을 연결하며, 남은 command 개수에 따라 recursive 형태로 pipelining을 구현할 수 있다.

1. **Phase 3 (background)**

****

위는 shell의 execute 부분을 실행할 때, foreground, background에 따라서 나뉘는 부분의 flowchart 이다. parseline 함수를 통해 bg = 1 / 0 을 파악할 수 있는데, 이를 기반으로 shell에서 직접 reaping을 수행하며 foreground로 기다릴지, 혹은 SIGCHLD handler에게 reaping을 맡기며 shell은 다른 command을 수행할지를 선택할 수 있는 형태로 background / foreground execute을 구별했다.