**System Programming Project 2**

담당 교수 : 박성용

이름 : 정현정

학번 : 20210428

1. **개발 목표**

본 프로젝트는 Unix 환경에서 프로세스 및 작업 관리 기능을 구현한 사용자 정의 쉘(이하 MyShell)을 개발하는 것을 목표로 한다. 이 쉘은 시스템 레벨에서의 프로세스 제어를 활용하여 Bash를 기반으로 한 기능을 모방하였다. 본 프로젝트는 세 단계의 개발 과정으로 구성된다.

**Phase 1**

이 단계에서는 쉘의 기본 명령어를 구현한다. 기본적인 명령어(ls, mkdir, rmdir, cd, echo, exit 등)가 수행되어야 한다. 명령어의 실행은 주로 fork()와 execvp() 함수를 통해 자식 프로세스에서 처리된다. 내장 명령어가 아닌 경우, 해당 명령어의 실행을 직접 구현한다.

**Phase 2**

이 단계에서는 명령어 간 파이프 연결을 구현하여 한 명령어의 출력을 다른 명령어의 입력으로 사용할 수 있도록 한다. 이를 위해 표준 입력과 출력을 dup2() 함수를 사용하여 파이프로 재배치한다. 첫 입력은 터미널을 통해 입력되며 구현된 파이프라인을 통해 마지막 명령어의 출력은 터미널을 통해 사용자에게 전달된다.

**Phase 3**

이 단계에서는 job을 제어하는 기능을 추가한다. 사용자는 명령어를 background에서 실행할 수 있으며 이후 새롭게 명령어를 입력할 수 있다. background에서 동작하는 프로세스는 ctrl+z, ctrl+c와 같은 키보드 입력에 반응하지 않는다. foreground에서 동작하는 프로세스의 경우 해당 키보드 입력에 반응한다. ctrl+z로 일시 중지된 작업은 background에 suspended 상태로 이동하고, ctrl+c로 실행 중인 작업을 종료할 수 있다. fg와 bg 명령어를 사용하여 작업의 상태를 변경할 수 있으며 jobs 명령어로 background에서 실행 중인 작업을 모두 확인할 수 있다. 또한 SIGINT, SIGTSTP, SIGCONT와 같은 시그널을 적절히 처리하여 사용자가 입력한 커맨드에 따라 시스템이 반응하도록 구현한다.

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



이 단계에서 구현된 MyShell은 기본적인 명령어 실행 기능을 제공한다. cd, ls, mkdir, rmdir, touch, cat, exho와 같은 기본적인 명령어를 사용할 수 있다. 위의 사진처럼 모든 명령어라 올바르게 수행되는 것을 확인할 수 있다. cd와 exit 같은 명령어는 builtin\_command로 구현하였고 나머지 명령어는 fork()와 execvp()를 통해 자식 프로세스에서 실행파일을 불러와 수행하도록 하였다. 이후 종료되면 자식 프로세스는 부모 프로세스를 통해 reaping된다.

1. Phase 2

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 화이트, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

위 사진과 같이 모든 명령어가 올바르게 동작하는 것을 확인할 수 있다. Phase 2에서는 파이프라인 기능을 추가하여 사용자가 여러 명령어를 연결해 작업을 수행할 수 있도록 한다. 이를 위해 표준 입출력을 파이프로 리다렉션 하며 이 과정에서 dup2() 함수가 사용된다. 따옴표 처리 또한 구현하였다.

1. Phase 3

텍스트, 스크린샷, 번호이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 번호이(가) 표시된 사진

자동 생성된 설명

위 사진과 같이 요구되는 모든 명령어를 구현하였다. 위 사진을 통해 ctrl+z와 같은 시그널이 foreground 수행 도중 들어왔을 때 suspended 상태로 다시 background 작업에 추가되는 것을 확인할 수 있다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

본 명세서에 따라 pipeline으로 연결된 명령어 또한 background에서 실행이 되도록 구현하였다. cat tmp.txt | grep -I “abc” & 명령어를 실행시키면 실제 bash 쉘 창과 동일하게 작동하는 것을 위의 사진을 통해 확인할 수 있다. tmp.txt는 사진과 같다. ctrl+z와 ctrl+c 또한 foreground에서 실행되는 프로세스를 정상적으로 멈추게 하거나 종료시킨다. 만일 아무런 명령어를 수행하지 않고 프롬프트만 출력되어 있는 상황에서 ctrl+z 또는 ctrl+c를 입력한다면 아무것도 수행하지 않고 다시 프롬프트를 출력한다.

* 1. **개발 내용**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명

fork() 함수를 호출하면 실행 중인 프로세스의 메모리 공간을 그대로 복사하여 새로운 자식 프로세스를 생성한다. 이 자식 프로세스에서는 ls, mkdir, rmdir, touch 등의 명령어가 수행된다. 이러한 명령어들은 /bin 디렉토리에 위치한 실행 프로그램으로 구현되어 있다.

명령어 실행을 위해 쉘에서 execvp() 함수를 사용하여 해당 명령어에 맞는 프로그램을 실행한다. 이 함수는 인자로 받은 이름에 해당하는 실행 파일을 시스템 경로에서 찾아, 현재 프로세스의 메모리 공간에 로드하고 실행한다. 프로젝트에서는 해당 동작을 부모 프로세스가 아닌 자식 프로세스에서 수행하도록 구현하였다.

명령어가 입력되면 main함수에서 fgets()를 통해 명령어를 받고 eval() 함수를 호출한다. eval() 함수는 입력된 명령어를 파싱하고 builtin commmand가 아니라면 fork()를 호출하여 자식 프로세스를 생성한 뒤 execvp()로 프로그램을 실행한다. 실행 실패 시 에러 메시지를 출력하고 자식 프로세스를 종료한다.

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

자식 프로세스에서 프로그램의 실행이 완료되면 연결을 종료해야 한다. 예를 들어 ls 명령어를 실행시키면, 자식 프로세스는 execvp()를 통해 ls 프로그램을 찾아 실행하고 프로그램이 실행을 마치면 터미널에 파일 정보를 출력한 후 종료된다. 프로그램이 종료되면 자식 프로세스는 좀비 상태가 되어 메모리에서 사라지지 않는다.

종료된 후 좀비 상태가 된 자식 프로세스를 정리하기 위해 부모 프로세스는 wait() 또는 waitpid() 함수를 사용하여 reaping을 수행한다. 이 함수들은 자식 프로세스가 종료될 때까지 부모 프로세스의 실행을 중지시키고 종료된 자식 프로세스를 reaping 한다. 따라서 자식 프로세스는 더 이상 자원을 소비하지 않는다.

부모 프로세스는 자식 프로세스의 종료를 감지하기 위해 SIGCHLD 시그널을 받는다. 자식 프로세스가 종료되면 시그널이 부모 프로세스에게 전달되며 wait() 또는 waitpid() 함수가 실행된다. 이 프로젝트에서는 waitpid()를 사용하여 생성된 자식 프로세스가 종료될 때까지 명시적으로 기다림으로써 모든 자식 프로세스가 정상적으로 종료되고 자원이 정리되는 것을 보장한다.

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

현재 코드는 ‘|’ 문자를 포함한 명령어를 감지하여 파이프라인을 구성한다. main 함수에서 사용자의 입력을 받고 strchr() 함수를 사용하여 입력된 명령어에 ‘|’가 포함되어 있는지 확인한다. 포함되어 있지 않은 경우 기존 로직에 따라 eval() 함수를 호출하며, 포함되어 있는 경우 pipe\_execute() 함수를 호출하여 파이프라인에 따른 처리를 수행한다.

pipe\_execute() 함수에서는 입력된 커맨드 라인을 ‘|’ 기호에 따라 분리하고, 각 명령어를 개별적으로 처리한다. 이를 위해 pipe() 함수로 파이프를 생성하고, 각 명령어 실행을 위해 eval() 함수를 통해서 fork()를 수행한다. fork() 함수를 통해 자식 프로세스를 생성한 뒤 dup2() 함수로 필요에 따라 표준 입력 또는 출력, 기타 파일 디스크립터를 파이프에 연결한다. 이렇게 설정된 파이프를 통해 앞선 명령어의 출력을 다음 명령어의 입력으로 전달한다.

‘|’를 기준으로 나눈 각각의 명령어에 대하여 자식 프로세스는 execvp()를 호출해 해당 명령어를 실행한다. 실행 중 생성된 모든 자식 프로세스는 자원을 정리하고 제어를 부모 프로세스에게 반환하기 전에 필요에 따라 파이프를 닫는다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

파이프라인의 개수가 증가함에 따라 프로그램은 각 명령어 사이에 파이프를 동적으로 생성하여 데이터 스트림을 관리한다. 첫 번째 명령어는 표준 입력을 사용하고 마지막 명령어는 표준 출력을 사용한다. 중간에 위치한 명령어들은 이전 명령어의 출력을 입력으로 받고 결과를 다음 명령어의 입력으로 전달한다.

파이프라인 구성 시, 각 명령어 구간 사이에 두 개의 파일 디스크립터 fd1과 fd2를 설정하였다. 예를 들어 ‘ A | B | C ’ 와 같이 세 개의 명령어가 연결될 경우 다음 규칙을 적용시켜 표준 입출력 또는 파일 디스크립터를 연결한다.

- A : (SDTIN – fd1)로 데이터를 처리

- B : (fd1 – fd2)로 데이터를 처리

- C : (fd2 – STDOUT)로 데이터를 처리

이 과정에서 각 명령어 실행 후 파이프라인의 각 부분에 대해 생성된 파일 디스크립터를 적절히 닫아주어야 한다. 이 과정을 통해 파이프라인의 개수와 관계 없이 명령어 사이의 데이터 전송을 관리할 수 있다.

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

background 프로세스의 실행은 사용자의 커맨드 입력에서 맨 뒤에 붙은 ‘&’를 식별하여 이루어진다. 이를 통해 사용자는 다른 작업을 계속 하는 동시에 중단 없이 background에서 프로세스를 실행할 수 있다. background 프로세스 구현은 다음 단계를 통해 이루어진다.

- 명령어 파싱 : 사용자가 커맨드를 입력하면 parseline() 함수를 통해 파싱된다. 이 함수는 명령어 끝에 있는 ‘&’를 확인한다. 이 기호가 존재하면 argv에서 ‘&’를 제거하고 해당 명령어가 background에서 실행될 것임을 나타내도록 정수값을 반환한다. 해당 반환값은 bg 변수에 저장된다.

- 자식 프로세스 생성과 프로세스 그룹 관리

eval() 함수 내에서 Fork()를 호출하여 자식 프로세스를 생성한다. 생성된 자식 프로세스에 대해 setpgid(0, 0)을 수행하여 자식이 새로운 프로세스 그룹에 속하도록 한다. 이를 통해 자식 프로세스는 foreground 프로세스 그룹에서 분리되며 터미널의 foreground 그룹이 받는 시그널(ctrl+z, ctrl+c)에 의해 영향을 받지 않는다. 즉, 새롭게 생성된 자식 프로세스는 새로운 프로세스 그룹에서 실행되기 때문에 터미널에서 발생하는 시그널이 전달되지 않는다.

- 프로세스 상태 관리

background 프로세스 정보는 jobs 배열에 저장되며, 이 배열은 프로세스의 상태, pid, 명령어, 해당 프로세스의 인덱스를 추적한다. 사용자는 jobs 명령어를 실행하여 현재 background에서 실행 중인 작업들의 리스트와 상태를 조회할 수 있다. background에서 실행되는 프로세스의 종료나 상태 변화는 SIGCHLD 핸들러에 의해 감지된다. 이 핸들러는 waitpid() 함수를 사용하여 종료된 자식 프로세스들을 reaping하고, 종료된 프로세스와 중단된 프로세스를 구분하여 jobs 배열의 상태를 업데이트한다.

이렇게 background에서 프로세스를 실행하면 사용자는 명령어의 실행을 기다리지 않고 즉시 다른 커맨드를 입력하거나 다른 작업을 계속할 수 있다.

* 1. **개발 방법**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

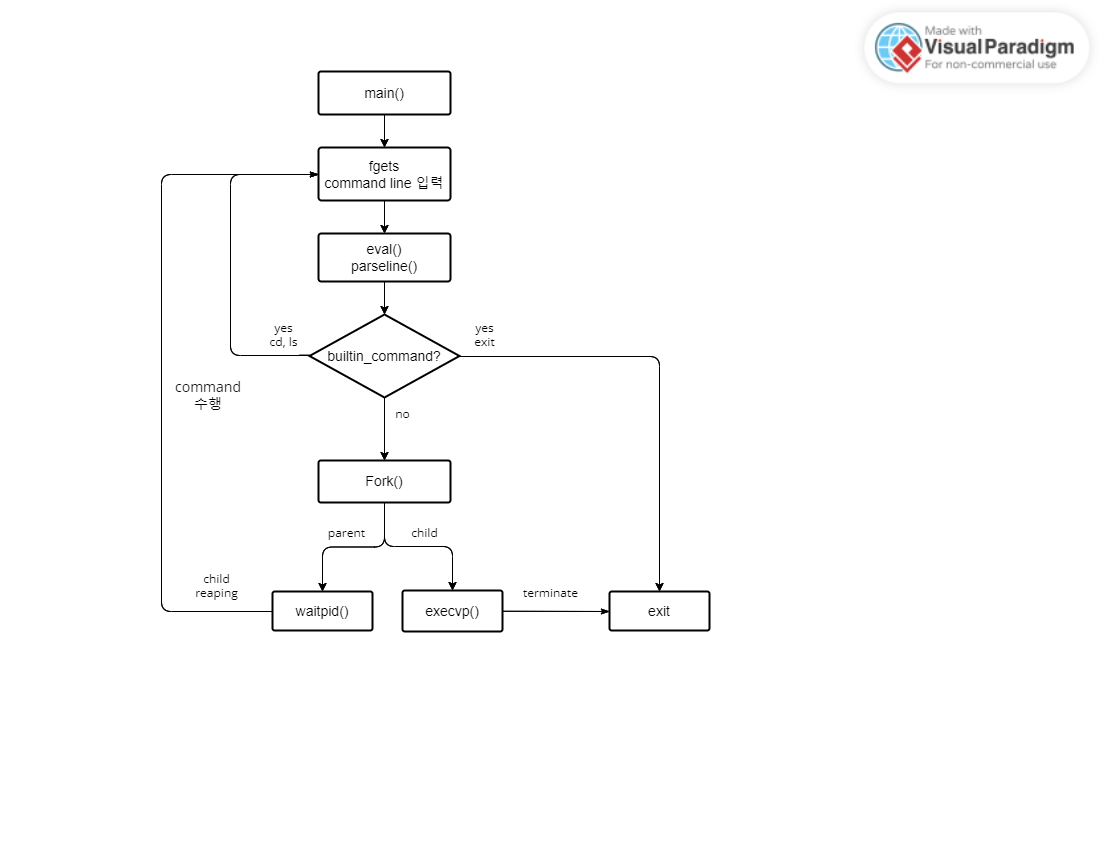
위 사진과 같이 struct job 구조체를 사용하여 background 및 foreground 프로세스를 관리한다. 이 구조체는 프로세스의 상태, pid, 프로세스의 인덱스, 커맨드를 저장하고 struct job 타입의 배열 jobs를 통해 모든 background 프로세스의 정보를 관리한다. 이때 jobs[0]은 foreground에서 실행되는 프로세스의 정보를 고정적으로 저장하도록 구현하였다. jobs[0]을 통해 background에서 실행되던 프로세스가 fg 명령어를 통해 foreground에서 동작하던 도중 ctrl+z로 시그널이 발생하여 중단되면 다시 background에 suspended 상태로 관리될 수 있도록 하였다.

eval() 함수는 개별 커맨드를 실행하는 함수로서 필요에 따라 Fork()와 execvp()를 수행한다. 또 background 프로세스의 경우 foreground group에 적용되는 시그널로부터 독립적으로 동작할 수 있도록 프로세스 그룹을 설정하는 역할 또한 eval() 함수가 수행한다. 파이프라인 기능을 수행하기 위해서는 파일 디스크립터로 입출력을 적절하게 조정하는 Dup2() 함수를 사용해야 하는데, 해당 기능 또한 eval() 함수에 추가되어 있다.

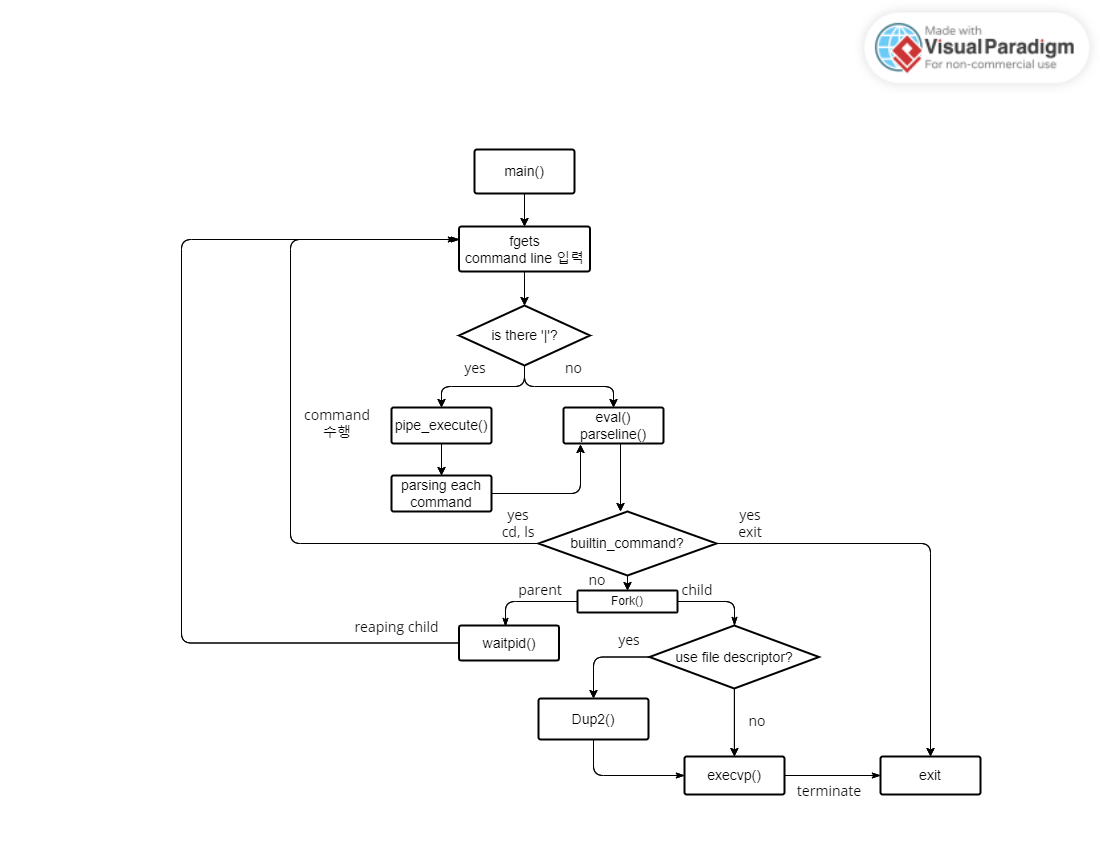
SIGINT 핸들러와 SIGTSTP 핸들러는 키보드 인터럽트에 반응하도록 구현되어 있다. SIFCHLD 핸들러의 경우, 자식 프로세스의 상태 업데이트를 관리한다. SIGCHLD 핸들러를 통해 background 프로세스의 종료를 감지하고 프로세스의 상태를 업데이트 한다. background 프로세스가 실행될 때, 해당 프로세스는 jobs 배열에 저장되고 각 job은 고유한 프로세스 인덱스를 통해 관리된다. 이 인덱스는 fg와 bg, kill을 통해 background 프로세스를 제어할 때 사용된다.

builtin\_command() 함수는 명령어가 내장 명령어인지 판단하고 해당하는 동작을 실행한다. 예를 들어 cd, exit 등과 같은 명령어는 부모 프로세스에서 직접 처리된다. 파이프라인을 구현하기 위해 pipe\_execute() 함수는 ‘|’를 기준으로 명령어를 분리하고 각 명령어를 eval() 함수를 통해 실행한다. 여기서 내부적으로 사용되는 parseline2() 함수는 기존의 shellex.c에 있는 parseline() 함수로 파이프로 연결된 커맨드의 맨 뒤에 ‘&’ 기호의 유무를 판단한다. 만약 ‘&’ 기호가 있다면 파이프로 연결된 각각의 명령어 뒤에 ‘&’를 추가한다. phase 3의 myshell.c 코드에서 eval() 함수에서 내부적으로 parseline() 함수를 사용하고 있는데, 이는 따옴표 처리를 구현한 새로운 파싱 함수라는 점에서 parseline2()와 차이점을 갖는다.

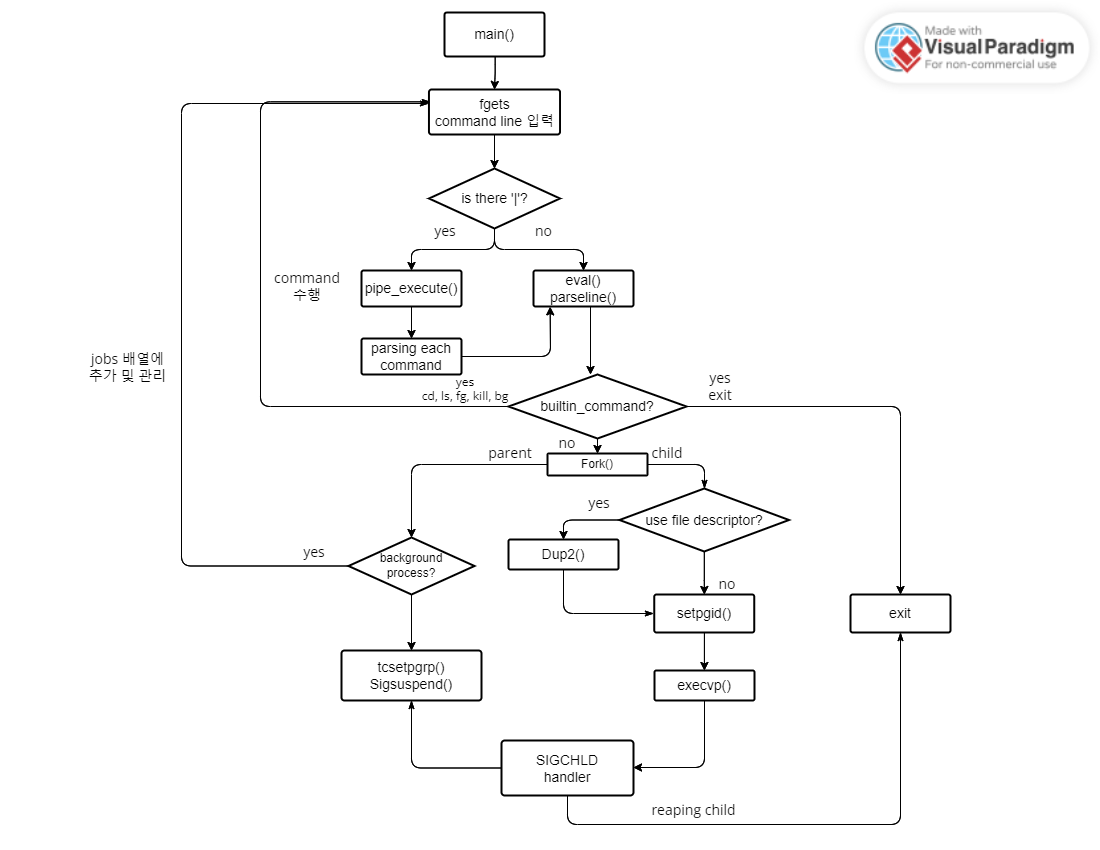
1. **구현 결과**
   1. **Flow Chart**
2. **Phase 1 (fork)**

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

****