**Multicore Programming Project 1**

담당 교수 : 박성용

이름 : 남기찬

학번 : 20211180

1. **개발 목표**

이번 MyShell 프로젝트에서는 터미널을 통해 나의 쉘 ‘CSE4100-MP-P1> ‘을 실행하고, 여기에 명령어를 입력 받아서 그 명령어를 실행하는 것을 구현한다. 입력 받은 명령어는 파싱되어 어떤 기능을 수행할 것인지, 옵션에는 어떤 게 있는지 분석된다. ‘history’나 ‘cd’ 등의 기본적인 쉘 명령어나 ‘ls’, ‘echo’ 등의 fork(), execve() 함수를 통해 실행 가능한 명령어들을 구현한다. 그리고 파이프라인 ‘|’이 있을 경우에는 앞 명령어의 출력과 뒤 명령어의 입력이 연결되도록 구현한다. 마지막으로, 명령어의 맨 뒤에 ‘&’가 있으면 그 명령어가 백그라운드에서 실행되도록 구현한다.

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

나의 쉘 ‘CSE4100-MP-P1> ‘에 ‘cd dir’을 입력하면 현재 작업 중인 디렉토리를 dir 디렉토리로 변경한다. ‘ls’를 입력하면 현재 작업 중인 디렉토리의 내용 목록을 출력한다. ‘mkdir dir’을 입력하면 ‘dir’이라는 새로운 디렉토리를 만들고, ‘rmdir dir’을 입력하면 ‘dir’이라는 디렉토리를 삭제한다. ‘touch file’을 입력하면 file이 없는 경우에 새로운 파일을 생성한다. ‘cat file’을 입력하면 ‘file’ 파일의 내용을 읽고 출력하고, ‘cat file1 file2’을 입력하면 두 파일을 병합한다. ‘echo arg’를 입력하면 주어진 인자 ‘arg’를 화면에 출력한다. ‘history’를 입력하면 나의 쉘이 만들어진 이후로 지금까지 실행된 명령어들의 목록을 출력한다. history의 기능들 중 ‘!!’는 가장 최근에 실행된 명령어를 출력하고, 그 명령어를 실행한다. ‘!#’을 입력하면 history의 명령어 중 #번째의 명령어를 출력하고, 그 명령어를 실행한다. 마지막으로 ‘exit’을 입력하면 모든 프로세스들이 끝나고 나의 쉘도 종료된다.

1. **Phase 2**

나의 쉘에 입력된 명령어가 ‘cmd1 | cmd2’, 즉 명령어와 명령어 사이에 파이프라인 ‘|’이 있으면 첫 번째 명령어 cmd1의 출력이 두 번째 명령어 cmd2의 입력으로 전달되어 실행된다. 명령어가 파이프라인 여러 개로 연결되어 있으면 첫 번째 명령어의 출력이 두번째 명령어의 입력으로, 두 번째 명령어의 출력이 세 번째 명령어의 입력으로, 이런 식으로 순차적으로 실행된다.

1. **Phase 3**

나의 쉘에 입력된 명령어의 맨 끝에 ‘&’가 있으면 이 명령어는 포그라운드가 아닌 백그라운드에서 실행된다. ‘jobs’를 입력하면 백그라운드에서 실행 중인 명령어들의 목록과 진행 상태를 볼 수 있다. ‘bg #num’을 입력하면 ‘jobs’를 입력했을 때 나오는 백그라운드 명령어들 목록에서 index가 num인 줄의 백그라운드 명령어의 진행 상태를 running으로 바꿔주고, 이 줄을 출력한다. ‘fg #num’을 입력하면 ‘bg #num’에서와 같이 index가 num인 줄 명령어의 진행 상태를 running으로 바꿔주고, 출력하고, 이번엔 이 명령어를 포그라운드에서 실행한다. ‘kill #num’을 입력하면 index가 num인 백그라운드 명령어 작업을 종료시킨다.

* 1. **개발 내용**

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

fork() 함수는 현재 프로세스의 메모리 공간, 코드, 데이터 등의 거의 모든 정보, 자원을 복제하여 다른 하나의 독립된 프로세스를 생성한다. 원래 있던 프로세스는 부모 프로세스가 되고, 새로 생긴 프로세스는 자식 프로세스가 된다. 이 두 프로세스는 서로 다른 pid값을 가지고, 서로 다른 별개의 작업을 수행할 수 있다. fork() 함수의 return 값으로 부모 프로세스는 자식의 pid를 받고, 자식 프로세스는 0을 받는다. 이 return 값이 다른 것을 이용해 부모 프로세스와 자식 프로세스는 서로 다른 코드 경로를 따른다. 자식 프로세스에서는 Execve() 함수를 이용하여 새로운 프로그램을 실행하고, 부모 프로세스는 자식 프로세스를 제어하거나 프로세스의 종료를 확인한다.

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

자식 프로세스가 종료되거나 중단되면 SIGCHLD 시그널이 발생한다. SIGCHLD 시그널이 발생하면, 이를 처리하기 위해 등록된 SIGCHLD 핸들러에서는 특정 자식 프로세스나 먼저 종료된 자식 프로세스의 상태 정보를 수집하는 waitpid() 함수를 호출한다. 이 함수는 자식 프로세스가 종료될 때까지 blocking되어, 자식 프로세스가 종료될 때까지 부모 프로세스를 기다리게 하고, 자식 프로세스의 종료 여부를 알려준다. 그러면 부모 프로세스는 자식 프로세스의 종료 상태를 확인하고, 종료된 자식 프로세스를 reaping해준다. 만약 부모 프로세스가 종료된 자식 프로세스를 reaping해주지 않으면, 종료된 해당 자식 프로세스는 좀비 프로세스가 되어 시스템의 메모리를 소모하게 된다.

**2. Phase2 (pipelining)**

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

pipe() 함수를 이용해 파이프라인에 사용할 파일 디스크립터를 생성하고 이를 fd1 배열에 저장한다. fd1[0]는 읽기용으로, 파이프의 출구에서 데이터를 읽어들이고, fd1[1]은 쓰기용으로, 파이프의 입구로 데이터를 쓴다. 즉 fd1[1]으로 쓴 데이터는 fd1[0]에서 읽을 수 있다. 이 원리를 이용하여 첫 번째 명령어의 출력을 두 번째 명령어의 입력으로 보낸다. 먼저 fork() 두 번으로 자식 프로세스 2개를 생성하고, 각 자식 프로세스에서 필요 없는 파일 디스크립터는 close() 함수로 닫아준다. 다음 각 프로세스에서 Dup2() 함수로 fd1[1]을 표준 출력으로 바꿔주고, fd1[0]을 표준 입력으로 바꿔주면, 첫 번째 명령어의 출력을 두 번째 명령어의 입력으로 보낼 수 있다. 따라서 이 첫 번째 명령어의 출력을 입력으로 하는 두 번째 명령어가 실행된다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

Input 값을 파싱한 후 PipeExecute 함수에 넣어서 파이프라인이 몇 개가 있는지 저장한다. 그리고 파이프라인의 개수에 따라서 pipe()와 fork(), Dup2(), close()를 부르는 횟수를 다르게 한다. 예를 들어 파이프라인이 2개가 있다면, 먼저 파일 디스크립터 배열 2개 fd1, fd2를 만들고, 자식 프로세스를 3개 생성한다. 다음 첫 번째 자식 프로세스에서는 fd1[1]을 표준 출력으로 바꾸고, 두 번째 자식 프로세스에서는 fd1[0]을 표준 입력으로 바꾸는 것과 함께 fd2[1]을 표준 출력으로 바꾼다. 마지막 자식 프로세스에서는 fd2[0]을 표준 입력으로 바꿔 두 번째 명령어까지의 출력이 세 번째 명령어의 입력에 들어가 실행되도록 한다.

**3. Phase3 (background process)**

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

Input의 마지막에 ‘&’가 확인되면 명령어를 현재 터미널에서 실행하지 않고 이를 백그라운드에서 실행하도록 한다. fork() 함수를 통해 부모 프로세스와 자식 프로세스를 생성하고 Execve() 함수를 사용하여 자식 프로세스에서 프로그램을 실행하는 것까지는 같지만, 부모 프로세스는 자식 프로세스가 백그라운드에서 실행되도록 waitpid() 함수를 호출하지 않는다. 이때 백그라운드에서 실행되는 작업들은 모두 running 상태이다. 부모 프로세스는 자식 프로세스의 종료를 기다리지 않아도 되고, 바로 다른 작업을 수행할 수 있다.

* 1. **개발 방법**

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

먼저 input값을 parse() 함수를 통해 파싱한다. parse()에서는 문자열이 ‘나 “로 둘러싸여 있다면 그 따옴표가 사라지고, 띄어쓰기, 탭, 엔터 단위로 나뉘어져서 char\*\* args에 저장된다. 다음 ‘history’의 구현을 위해 전역 배열 H에 index에 알맞게 input값을 저장한다. 그리고 이제 args[0]이 어떤 값인지에 따라 실행을 다르게 한다. args[0]에 ‘ls’, “mkdir”, “rmdir’, ‘touch’, ‘cat’, ‘echo’가 오게 되면 기본적인 fork(), Execve(), waitpid() 구조를 따라 실행한다. fork() 함수를 실행하여 부모 프로세스와 자식 프로세스를 생성한 뒤, pid가 0인 자식 프로세스에서는 Execve()로 프로그램을 실행한 후 exit()을 통해 종료하고, 부모 프로세스는 else로 가서 waitpid() 함수를 통해 자식 프로세스의 종료를 기다린다. 자식 프로세스가 종료되면 부모 프로세스는 자식 프로세스를 reaping함으로써 메모리를 소모하는 좀비 프로세스를 방지한다. ‘cd’, ‘history’, ‘exit’은 fork(), Execve(), waitpid() 구조를 따르지 않고 chdir(), exit() 등을 이용해서 개별적으로 구현한다. ‘!!’나 ‘!#’가 입력되면 해당하는 명령어를 받아오고 위에서 제시된 방식과 동일하게 구현한다.

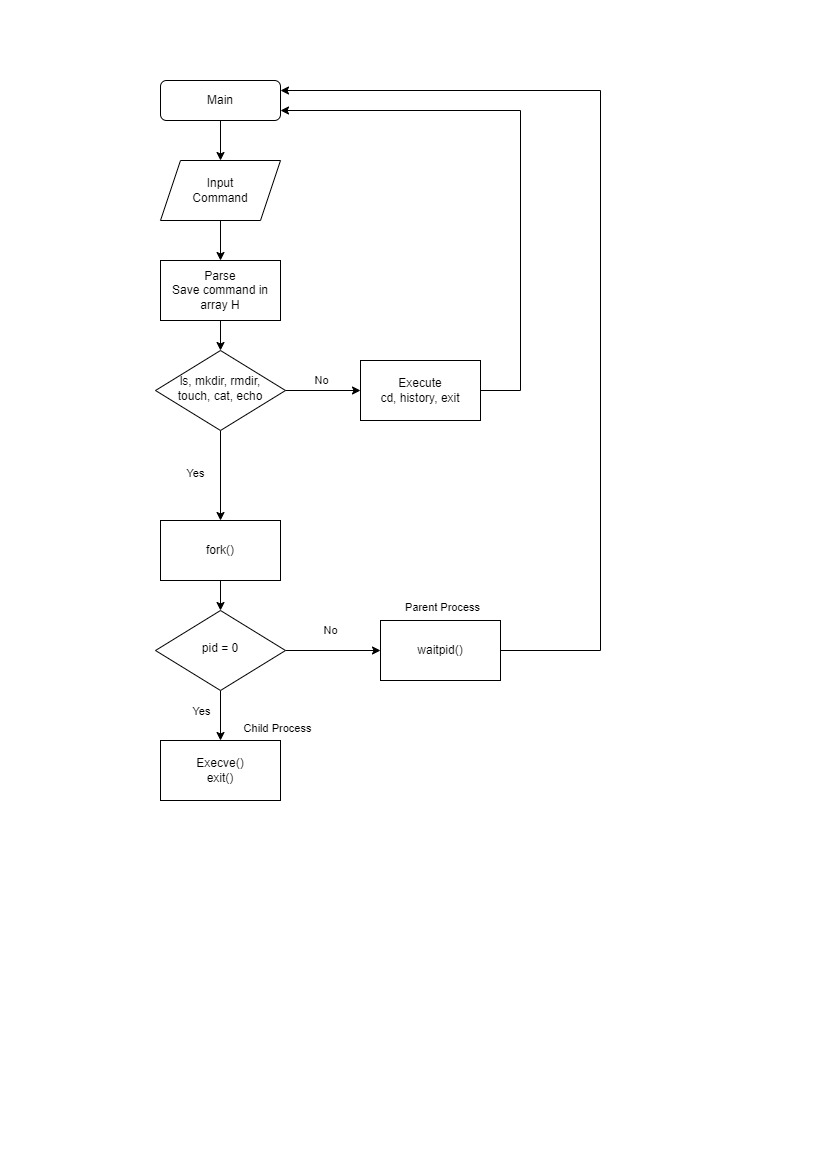
1. **Phase 2 (pipelining)**

Input값을 parse() 함수를 통해 파싱한 결과 char\*\* args에 파이프라인이 존재하면 int pipeexist을 1로 놓고 PipeExecute() 함수를 호출한다. 먼저 input의 모든 파이프라인 개수를 int pipecount에 저장하고, pipecount에 해당하는 조건문으로 이동한다. 그리고 파이프라인이 저장된 args들의 index를 저장하고, 이 index를 이용해 다음 파이프라인이나 ‘NULL’이 나오기 전까지의 명령어를 com1, com2 등의 배열에 저장한다. 예를 들어 pipecount가 2이면 총 3개의 명령어로 나뉘게 된다. 다음, 파이프라인의 개수만큼 pipe() 함수를 실행하고, 나뉜 명령어의 총 개수만큼 fork() 함수를 실행한다. 생성된 자식 프로세스들에서는 각각 Dup2() 함수를 이용해서 파이프 출력을 표준 출력으로, 파이프 입력을 표준 입력으로 바꾼다. 이때 사용하지 않는 다른 파이프들은 close()를 통해 닫아준다. 이렇게 하면 첫 번째 프로세스의 출력 결과가 두 번째 프로세스의 입력으로 들어가고, 두 번째 프로세스의 출력 결과가 세 번째 프로세스의 입력으로 들어가는 방식으로 프로그램이 실행된다. 파이프라인의 개수에 맞게 Dup2()가 실행되고 올바른 파이프 구현에 따라 마지막 프로세스가 실행되면, close()를 통해 모든 파이프를 닫고, 부모 프로세스들은 wait()으로 모든 자식 프로세스들을 reaping해준다. 이 PipeExecute() 함수도 전체 evaluate() 함수에서 fork(), PipeExecute(), waitpid() 구조를 통해 자식 프로세스에서 실행하고, exit()으로 종료되면 부모 프로세스가 이를 reaping해준다.

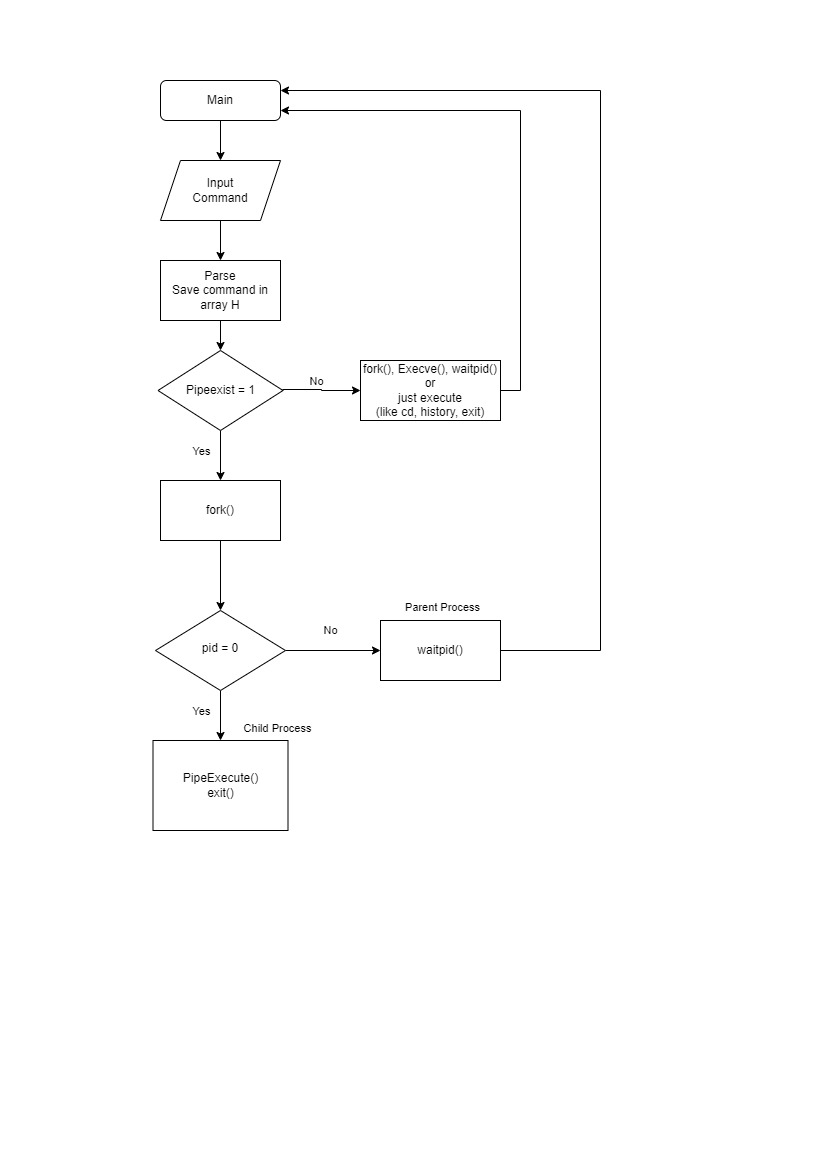
1. **Phase 3 (background process)**

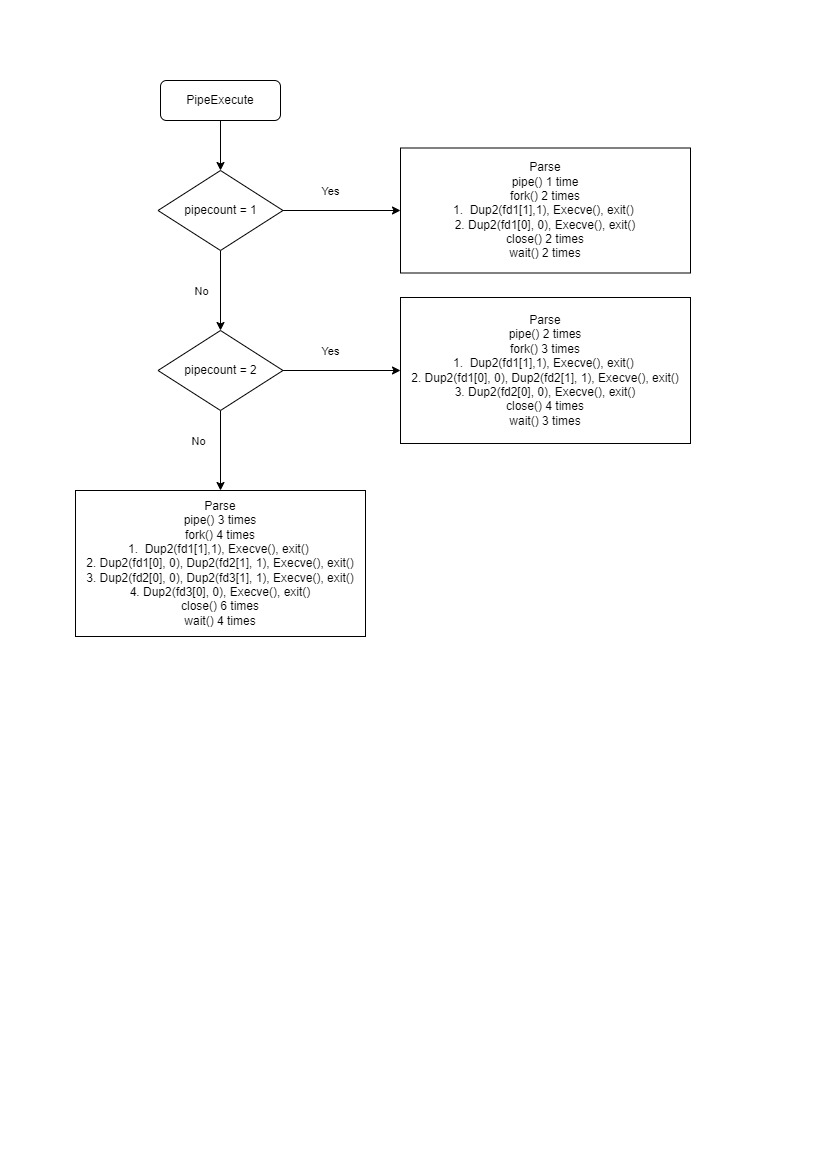
fgets()로 받은 input의 끝에 ‘&’가 있는지 확인하고, 있다면 원래 0이었던 int bg를 1로 바꾼다. 그리고 Phase1과 Phase2의 명령어들과 다른 명령어들을 bg의 값에 따라 다르게 실행한다. bg가 0이면 이전 Phase1, Phase2에서와 같이 일반적인 fork(), Execve(), waitpid() 구조로 실행한다. 만약 bg가 1이면 먼저 백그라운드 명령어들의 목록을 저장하기 위한 구조체 배열 job\_index에 명령어 리스트 번호, pid값, 진행 중인지 멈춰있는지 상태, 그리고 해당 명령어 전체를 addjob()함수를 호출하여 전역 변수 jobs\_count 알맞은 index에 저장한다. 다음 자식 프로세스에서는 백그라운드 명령어가 터미널에 출력되지 않도록 표준 출력을 ‘/dev/null’파일로 바꾼 후 실행하고, 부모 프로세스에서는 waitpid()를 하지 않는다. Input에 ‘jobs’가 입력되면 지금까지 저장된 백그라운드 명령어들의 정보 목록을 출력하고, ‘bg #num’이 입력되면 atoi()를 통해 num값을 구한 후, num index에 해당하는 백그라운드 명령어를 running 상태로 바꿔주고 백그라운드 명령어 목록에서 해당 줄을 출력한다. ‘fg #num’이 입력되면 ‘bg #num’이 입력되는 케이스와 동일한 코드를 넣고, 그 후 백그라운드 명령어 목록의 num index에 저장된 명령어를 터미널에서 실행한다. 이는 명령어를 input이 들어오는 함수 인자 char\* cmd에 넣고 loop를 통해 다시 맨 처음 evaluate 함수로 돌아가서 Phase1, Phase2에서처럼 동일하게 구현한다. 마지막으로 ‘kill #num’이 입력되면 구조체 배열의 int killed를 1로 놓고, 나중에 ‘jobs’가 입력됐을 때 killed가 1이면 num index의 줄이 출력이 되지 않도록 한다. ‘bg’, ‘fg’, ‘kill’을 입력했을 때 atoi()를 통해 얻은 num값이 전역 변수 jobs\_count보다 크다면, “No Such Job\n”을 출력한다.

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

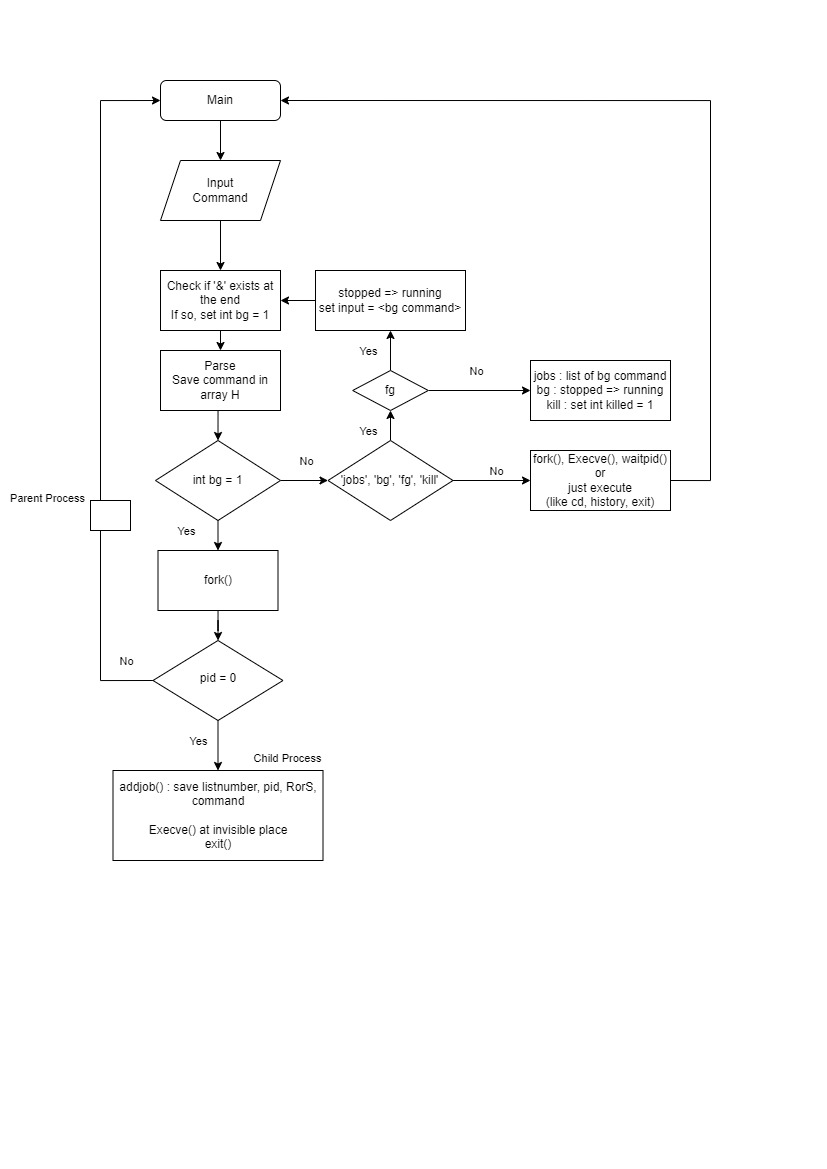
****

1. **Phase 2 (pipeline)**

****

****

1. **Phase 3 (background)**

****