**System Programming Project 4**

담당 교수 : 김영재

이름 : 이예진

학번 : 20181668

1. **개발 목표**

프로젝트 4에서는 리눅스 쉘 프로그램을 구현한다. 리눅스 쉘은 interactive command line terminal program으로 사용자가 입력하는 명령을 실행하고 프로그램을 실행한다. 쉘은 반복적으로 프롬프트를 출력하고 명령을 기다렸다가 명령의 내용에 따라 작업을 수행한다. 본 프로젝트를 통해 system-level process control, process isgnaling, interprocess communication, running process, jobs in the background에 대해 배우게 된다.

MyShell은 3개의 phase가 있는데 각 phase에서 쉘의 기능을 확장하기 때문에 차례대로 수행해야 한다. 첫 번째 phase에서는 fork(), exec(), wait() 등의 시스템 콜을 사용하여 입력 받은 명령을 child 프로세스에서 실행하는 simple shell을 작성한다. 프로세스에 대한 이해를 필요로 한다.

두 번째 phase에서는 redirection 과 piping을 추가한다. pipeline에 있는 각 명령을 위한 새로운 프로세스를 생성하고 parent 프로세스가 마지막 command를 기다리게 한다. 핵심 아이디어는 한 프로세스의 출력을 다른 프로세스의 입력으로 전달하는 것이다. dup(), dup2() 등의 시스템 콜을 사용하도록 한다.

세 번째 phase에서는 프로세스를 백그라운드에서도 실행할 수 있도록 한다. 리눅스 쉘은 job control 개념을 지원하므로 job들을 백그라운드와 포그라운드 간에 이동이 가능하고 job 내에 있는 프로세스의 상태를 변경할 수 있다. 백그라운드 명령과 job control을 지원하는 다양한 내장 명령을 제공하는 프로그램을 만들어야 한다. 시그널에 대한 개념을 필요로 한다.

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

리눅스 쉘은 사용자의 명령을 입력 받고 입력 받은 명령에 알맞은 작업을 수행할 수 있다. 쉘은 quit나 exit 명령을 받기 전까지 반복적으로 프롬프트를 출력하여 입력을 받는다. cd, ls, mkdir, rmdir, touch, cat, echo, exit 명령어를 수행할 수 있다. 명령어의 원래 기능과 동일한 기능을 수행한다. 기능에 대한 설명은 아래와 같다.

- ch : 쉘에 있는 디렉터리들을 탐색하여 출력한다.

- ls : 디렉터리 내용들을 출력한다.

- mkdir, rmdir : 쉘에 디렉터리를 만들거나 삭제한다.

- touch, cat, echo : 파일을 만들거나 읽거나 파일의 내용을 출력한다.

- exit : 쉘을 나가서 원래의 프로그램으로 돌아간다. 종료시 프로세스들을 회수한다.

명령은 child 프로세스로 실행이 된다. fork()를 통해 프로세스를 생성할 수 있고 exec() 사용하여 프로세스를 새 프로그램으로 대체한다.

1. Phase 2

phase 1에 파이프 기능이 추가되었다. 파이프 기호 ‘|’ 를 사용한 명령어를 수행할 수 있다. 아래와 같은 명령어들을 수행할 수 있다.

- ls -al | grep filename : ls -al의 결과 중에 filename에 해당하는 파일의 정보를 출력한다.

- cat filename | less : filename에 해당하는 파일의 내용을 less를 통해 보여준다.

- cat filename | grep -v “abc” | sort -r : filename에 해당하는 파일을 “abc” 패턴이 존재하지 않는 행만 출력 순서를 역순으로 출력한다.

파이프 기호를 기준으로 앞에 위치하는 프로세스의 출력이 뒤에 나오는 프로세스의 입력으로 전달되어 명령이 실행된다. 프로세스는 fork()를 사용하여 만들고 exec()를 사용하여 프로그램으로 대체한다. 파이프는 pipe()를 통해 파이프를 생성하고 dup()을 통해 파일 디스크립터를 연결하고 close()를 사용하여 사용하지 않는 파일 디스크립터는 닫는다.

1. Phase 3

phase 2에 job control과 백그라운드 실행 기능이 추가되었다. ‘&’ 기호를 명령 뒤에 붙이면 명령을 백그라운드에서 실행한다. 아래와 같은 명령어들을 수행할 수 있다.

- phase 2에서 수행 가능한 모든 명령어의 끝에 ‘&’을 붙인 명령 : 백그라운드에서 명령이 실행된다.

- ls -al | grep filename & : 백그라운드에서 명령을 수행하고 ls -al의 결과 중에 filename에 해당하는 파일의 정보를 출력한다.

- cat sample | grep -v a & : 백그라운드에서 명령을 수행하고 sample이라는 파일에서 a 문자가 존재하지 않는 행만 출력한다.

- jobs : running, stopped 백그라운드 job들을 출력한다.

- bg job : 중지되어 있는 백그라운드 job을 running 백그라운드 job 상태로 바꾼다. job은 job id로 들어올 수도 있고 process id로 들어올 수도 있다.

- fg job : 중지되어 있거나 실행중인 백그라운드 job을 running 포그라운드 job으로 바꾼다. job은 job id로 들어올 수도 있고 process id로 들어올 수도 있다.

- kill job : job을 종료시킨다.

ctrl-c를 누르면 SIGITN 시그널이 포그라운드 job에 있는 각 프로세스에 전달이 되고 SIGINT를 받은 프로세스는 종료된다. ctrl-z를 누르면 SIGTSTP 시그널이 포그라운드 job에 있는 각 프로세스에 전달이 되고 SIGTSTP을 받은 프로세스는 중지 상태가 된다. 중지된 프로세스는 SIGCONT 시그널을 받으면 다시 실행이 된다.

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

csapp.c에 구현되어 있는 Fork 함수를 사용했다. Fork 함수에서는 fork()를 호출하여 반환값을 받고 반환값이 0보다 작으면 에러를 출력한다. (fork가 실패하면 -1이 반환된다. 성공 시에는 부모 프로세스에서는 자식 프로세스의 PID 값이 반환되고 자식 프로세스에서는 0이 반환된다.) fork() 시스템 호출은 새로운 프로세스를 위한 메모리를 할당하고 fork를 호출한 프로세스를 새로운 공간으로 전부 복사한다. 자식 프로세스는 부모 프로세스와 똑같은 코드를 가지고 있다. 반환값이 pid에 저장되는데(코드에서 pid = Fork() 부분이다.) pid가 0이면 자식 프로세스이므로 fork 이후 라인부터 실행이 된다. 자식 프로세스에서는 execve를 곧바로 실행하여 입력 받은 명령을 수행한다. execve는 새로운 프로세스를 위한 메모리를 할당하지 않고, execve를 호출한 프로세스의 PID가 그대로 새로운 프로세스에 적용되며 execve를 호출한 프로세스는 새로운 프로세스에 의해 덮어 쓰여지게 된다. 부모 프로세스는 fork 이후에 원래 프로세스대로 작업을 실행한다. fork 함수 호출 이후 코드부터 부모와 자식 프로세스는 각자의 메모리를 사용하여 실행된다.

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

foreground 작업은 작업 수행 중에 다른 명령을 입력할 수 없고 작업이 끝날 때까지 기다려서 입력해야 한다. 그러므로 csapp.c에 구현되어 있는 Wait 함수를 사용하여 자식 프로세스가 끝날 때까지 부모 프로세스가 기다리도록 구현했다. 자식 프로세스가 종료되면 부모 프로세스에게 SIGCHLD 시그널을 보낸다. 다른 정보도 같이 보내주는데 Wait 함수에서는 pid\_t wait(int\* statloc)을 호출하는데 인자로 종료 상태를 저장할 변수를 넘겨준다. 오류가 발생하면 -1을 반환하고 자식 프로세스가 정상적으로 종료되었거나 비정상적으로 종료된 경우에는 프로세스 PID를 반환한다. 자식 프로세스가 동작 중이면 종료 상태를 얻어올 때까지 대기한다. 자식 프로세스가 종료되면서 exit 함수의 인자 값, 리턴한 값이 statloc에 저장이 된다. 또한 자식 프로세스가 정상 종료가 되었는 지에 대한 정보도 statloc에 저장이 된다. <sys/wait.h>에 정의되어 있는 매크로들을 사용하면 원하는 정보를 분리할 수 있다. wait 함수가 오류인 경우에 statloc이 ECHILD(호출자의 자식 프로세스가 없는 경우)나 EINTR(시스템 콜이 인터럽트 된 경우)을 값으로 갖는다. 자식 프로세스가 종료된 상태이면 부모 프로세스는 반환된 값들을 보고 자식 프로세스의 종료에 대한 정보를 얻게 된다.

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

명령어에 파이프 심볼(‘|’)이 있으면 없는 경우와 구분하여 따로 함수를 수행하도록 했다. 우선 파이프 심볼을 기준으로 앞에 존재하는 문자열과 뒤에 존재하는 문자열로 나누어서 각 문자열을 토큰화해준다. 함수에서는 pipe()를 통해 파이프를 생성하고 파이프 생성 오류가 생기면 프로그램을 종료한다. pipe 함수는 파일 기술자 2개를 저장한다. fd[0]은 읽기 전용으로 열고 fd[1]은 쓰기 전용으로 연다. 파이프 생성후 Fork()를 통해 자식 프로세스를 생성한다. 자식 프로세스는 부모 프로세스가 생성한 파일 기술자들도 복사한다. 파이프는 단방향 통신이므로 자식 프로세스는 읽기만 할 것이고 부모 프로세스는 쓰기만 할 것이다. 따라서 자식 프로세스에서는 쓰기용 파일 디스크립터, 부모 프로세스에서는 읽기용 파일 디스크립터를 닫아줄 수 있다. dup2()를 사용하여 자식 프로세스는 표준 출력이 파이프로 가게 한다. dup2()를 사용하여 부모 프로세스는 표준 입력이 파이프로부터 오게 한다. 자식 프로세스에서는 해당 명령을 execvp 함수를 사용해서 실행한다. 정상적으로 실행된다면 execvp 다음 줄이 실행되지 않지만 오류가 발생하면 다음줄에 있는 오류 출력과 함께 exit하게 된다. 부모 프로세스에서는 파이프 기준 뒤에 있는 명령에 또 파이프 심볼이 존재하는지 확인한다. 함수를 통해 토큰화하는데 파이프가 존재하지 않았다면 하나의 토큰과 빈 토큰이 발생할 것이다. 따라서 두 번째 토큰이 비어 있지 않다면 재귀함수를 통해 다시 파이프를 생성하고 파일 디스크립터를 알맞게 설정한다. 두 번째 토큰이 비어 있는 경우에는 execvp 함수를 사용해서 실행하고 에러 처리는 자식 때와 같다. 자식 프로세스 종료에 대하여 핸들러를 설정해두어서 종료된 프로세스 회수가 가능하다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

재귀함수를 사용하여 파이프 심볼(‘|’)의 개수에 제한이 없이 파이프 명령을 수행할 수 있게 구현하였다. 입력 받은 명령을 앞에서부터 탐색하여 파이프 심볼을 찾는다. 첫 번째로 찾으면 그 심볼을 기준으로 앞, 뒤 2개로 토큰화한다. cmd\_pipe 함수를 통해 자식 프로세스에서 앞부분 명령을 수행하고 그 수행 결과는 파이프로 가게 된다. 다음으로 부모 프로세스에서 뒷부분 명령을 함수를 통해 파이프 심볼이 더 있는지 확인한다. 더 없다면 파이프에서 입력을 받아 명령을 수행한다. 파이프 심볼이 더 있다면 파이프 심볼 기준으로 또 명령어를 2개로 토큰화한다. 재귀 함수로 cmd\_pipe를 다시 호출하여 프로세스를 새로 생성하고 자식에서는 파이프에 저장된 입력을 받아 출력을 파이프에 저장하고 부모 프로세스에서는 다시 명령어에 파이프 심볼이 있는지 확인한다. 명령어에 있는 파이프 심볼을 모두 탐색할 때까지 재귀를 반복하므로 파이프 라인의 개수에 제약 없이 파이프 명령을 수행할 수 있다.

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

백그라운드 기능은 프로세스가 실행되는 동안 다른 프로세스가 실행 가능하다. 백그라운드 방식으로 명령어를 실행하면 곧바로 다음 명령어를 실행할 수 있다. 이런 특성을 반영하여 구현하였다. 우선 기호의 사용은 ‘sort foo.txt &’, ‘sort foo.txt&’, ‘sort foo.txt & ’ 3가지 경우에 대해 모두 유효하도록 했다. 자식 프로세스에서는 포그라운드와 마찬가지로 명령에 알맞은 작업을 실행한다. 부모 프로세스에서는 포그라운드의 경우 자식 프로세스가 끝날 때까지 부모 프로세스가 기다렸지만 백그라운드 프로세스의 경우에는 자식 프로세스를 기다리지 않는다. 자식 프로세스의 프로세스 아이디, 작업 아이디와 명령에 대한 정보를 출력한 다음 바로 다음 명령을 받는다. 그러는 동안 백그라운드에서 프로세스는 계속 돌아가고 있다. Signal() 함수를 사용하여 백그라운드 프로세스의 종료시 시그널을 받아 종료된 프로세스를 회수할 수 있도록 구현했다. 시그널을 보낼 때 부모의 메인 루틴과 시그널 핸들링 흐름들의 일부 중첩이 일어날 수 있다. 이러한 race라고 하는 동기화 에러를 해결하기 위해 작업을 더하기 전에 지우는 경우가 발생하지 않도록 했다. 자식 프로세스가 종료되었을 때 SIGCHLD 이벤트가 발생하고 Signal 함수를 통해 구현해놓은 sigchld\_handler 함수가 수행된다. 핸들러 함수에는 waitpid 함수를 사용해여 종료된 프로세스를 회수하고 해당 job을 삭제한다. 백그라운드를 쉽게 다루기 위해 jobs, fg, bg, kill 명령어도 사용할 수 있도록 구현했다. jobs는 구조체 배열을 통해 구현했고 fg, bg, kill은 Kill 함수를 사용하여 프로세스에 신호를 보내주도록 구현했다.

* 1. **개발 방법**

baseline 코드로 주어진 csapp.h와 csapp.c, myshell.c(shellex.c를 이름 변경한 것)를 사용하여 개발을 하였다. csapp에 있는 함수를 사용하고 myshell.c 소스코드를 수정하여 개발 내용을 구현하였다.

- phase 1 : 주어진 소스코드의 eval 함수에서 execve 수행 전에 Fork 함수 호출부분을 추가하였다. fork로 자식 프로세스를 생성하고 execve로 명령을 수행한다. if(!bg)는 부모 프로세스인데 foreground job이 종료되기를 기다리는 중이므로 Wait 함수를 사용하여 자식 프로세스의 종료를 부모가 기다리도록 코드를 추가했다.

- phase 2 : 주어진 소스코드에 전역 변수 is\_pipe를 추가하여 pipe symbol이 명령어에 있는지에 대한 정보를 저장하도록 했다. pipetoken 함수를 추가하여 명령어 문자열에서 파이프 심볼을 찾고 파이프 심볼 기준으로 앞 부분과 뒷 부분의 문자열을 각각 토큰화하여 cmd1, cmd2 문자열 배열에 저장하였다. cmd\_pipe 함수를 추가하여 파이프를 생성하고 명령마다 프로세스를 생성하여 실행하도록 했다. 파일 디스크립터로 자식 프로세스의 출력을 파이프에 저장하고 부모 프로세스에서 파이프로부터 입력을 받을 수 있도록 dup2() 함수를 적절하게 사용하였다. cmd\_pipe 함수는 재귀적으로 호출되게 하여 파이프 심볼 개수에 제약 없이 실행할 수 있다. sig\_chldhandler 함수를 추가하여 Signal 함수에서 SIGCHLD 신호에 대해 핸들러로 사용하였다. SIGCHLD 신호에 대해 waitpid를 pid와 상관없이 실행하여 종료된 프로세스를 회수하도록 구현했다. is\_pipe 변수가 1의 값을 가질 경우에는 0인 경우와 다르게 실행하도록 구현했다.

int is\_pipe : 파이프 심볼 존재 정보 저장하는 플래그

void pipetoken(char\* str[], char \*cmd1[], char \*cmd2[]); : 파이프 기준으로 토큰화함

void cmd\_pipe(char\* cmd1[], char\* cmd2[]); : 파이프 명령 수행 함수

void sig\_chldhandler(int sig); : SIGCHLD에 대한 핸들러 함수

- phase 3 :

(1) 백그라운드 작업 : cmd\_jobs 함수로 jobs를 입력받았을 때 job 배열에 있는 작업들을 출력할 수 있다. cmd\_bg를 통해 bg 명령어에 대하여 job 번호로 해당하는 작업을 찾거나 pid로 해당하는 작업을 찾을 수 있다. 작업의 상태를 백그라운드 실행으로 변경하고 kill 함수를 통해 프로세스 그룹에 SIGCONT 신호를 줘서 다시 동작하도록 한다. cmd\_fg를 통해 fg 명령어에 대하여 job 번호 또는 pid로 해당하는 작업을 찾아 작업의 상태를 포그라운드 실행으로 변경하고 kill 함수를 통해 프로세스 그룹에 SIGCONT 신호를 줘서 동작하도록한다. wait\_fg를 호출하여 포그라운드에서 프로세스가 끝날 때까지 기다라게 한다.

(2) 작업 관리 : 작업을 저장하기 위해 구조체 Job\_t를 선언하여 이용했다. 구조체 안에는 프로세스 아이디, 작업 아이디, 작업 상태, 명령을 저장할 수 있다. 구조체 배열을 전역 변수로 선언하여 작업을 추가하고 삭제하고 확인할 수 있게 했다. jidx는 현재 배열에서 다음에 삽입될 위치를 가리키는 변수이다. insertjob, deletejob 함수를 사용하여 배열에 job을 삽입하고 삭제할 수 있다. find\_jidx는 배열을 탐색하여 가장 큰 작업 아이디를 반환한다. pidtojid는 pid를 인자로 받아와서 작업 배열을 탐색하여 해당하는 작업 아이디를 반환한다. jidtojob은 인자로 받은 작업 아이디에 맞는 작업을 반환한다. pidtojob은 인자로 받은 프로세스 아이디에 맞는 작업을 반환한다. find\_fgpid는 작업의 상태가 foreground인 작업의 프로세스 아이디를 찾아 반환한다.

(3) 시그널 관리 : wait\_fg 함수는 실행되고 있는 foreground 작업이 종료될 때까지 기다리는 기능을 수행한다. 특정 시그널을 관리하기 위해 sigchld\_handler, sigint\_handler, sigtstp\_handler를 만들어서 Signal 함수와 사용하였다. SIGCHLD 신호를 받으면 프로세스 아이디에 맞는 작업을 찾고 매크로를 통해 시그널의 원인을 찾는다. 중지라면 작업 상태를 바꿔준다. 종료라면 deletejob으로 작업을 삭제한다. waitpid 세번째 인자로 WNOHANG|WUNTRACED를 사용하여 즉시 종료 상태를 회수할 수 없는 상황에서 차단되지 않을 수 있고 중단된 자식 프로세스가 있으면 프로세스 상태를 받을 수 있다. SIGINT 신호를 받으면 프로세스 그룹에 kill을 통해 해당 신호를 보내준다. SIGTSTP 신호를 받으면 프로세스 그룹에 kill을 통해 해당 신호를 보내준다. 그룹 단위로 그룹 내에 있는 모든 프로세스에게 시그널을 보낼 수 있다. eval 함수에서는 SIGCHLD 시그널을 fork 호출 전에 블록하고 addjob 호출 후에 블록을 해제하도록 했다. 자식은 작업 리스트에 추가된 이후에 삭제될 수 있다.

typedef struct job\_t { : 작업 저장에 사용하는 구조체

pid\_t pid; // process id

int jobid; // job id

char state; // f : fg, b : bg, s : stop

char cmdline[100]; // command

}Job\_t;

Job\_t jobs[20]; // 작업 배열

void wait\_fg(pid\_t pid); // 포그라운드에서 기다리는 함수

void sigchld\_handler(int sig); // SIGCHLD 핸들러

void sigint\_handler(int sig); // SIGINT 핸들러

void sigtstp\_handler(int sig); // SIGTSTP 핸들러

void insertjob(pid\_t pid, char state, char\* cmdline); // 작업을 배열에 삽입

void deletejob(pid\_t pid); // 작업을 배열에서 삭제

int find\_jidx(); // 현재 배열에서 가장 큰 작업 아이디를 반환

int pidtojid(pid\_t pid); // pid를 job id로 변환

Job\_t \*jidtojob(int jid); // 입력받은 jid를 가진 job을 반환

Job\_t \*pidtojob(pid\_t pid); // 입력받은 pid를 가진 job을 반환

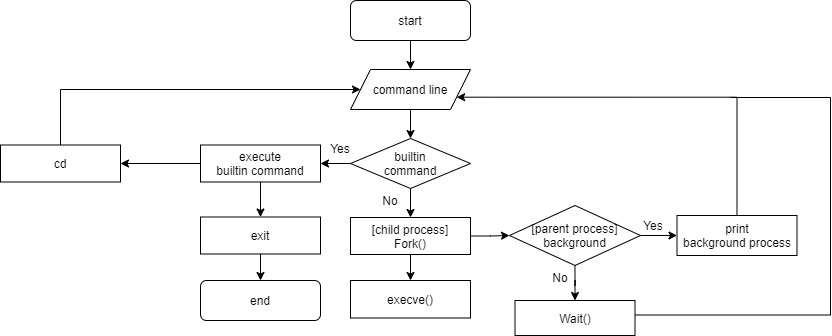
pid\_t find\_fgpid(); // 포그라운드에 있는 pid를 반환

void cmd\_jobs(); // 작업 배열을 출력

void cmd\_bg(char \*\*argv); // bg 명령어 수행

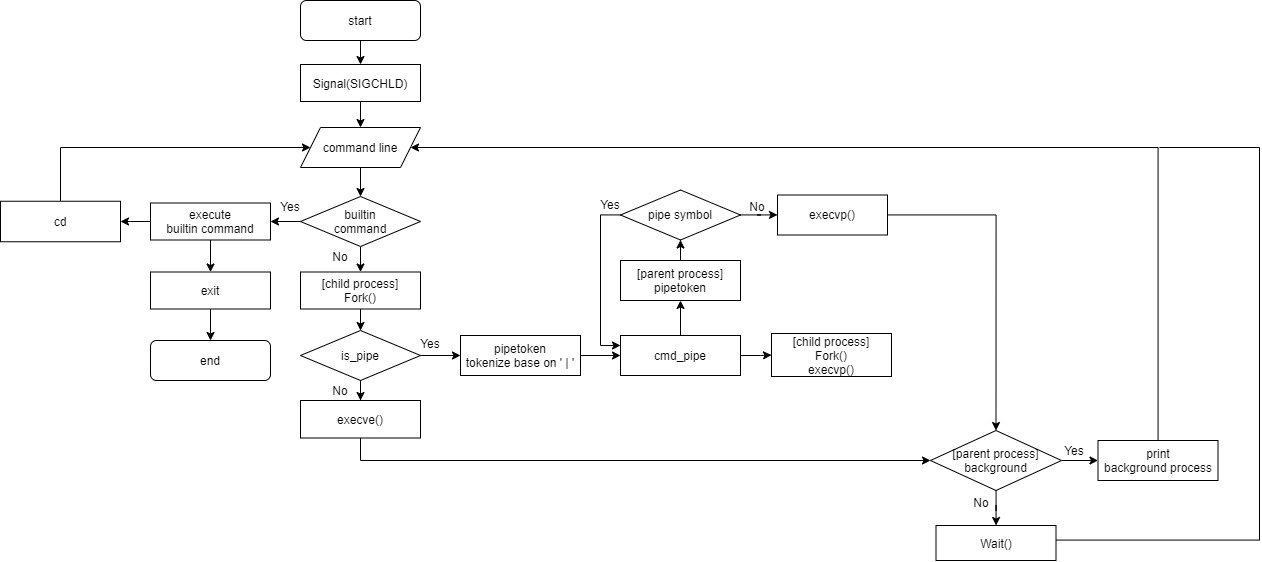
void cmd\_fg(char \*\*argv); // fg 명령어 수행

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

****

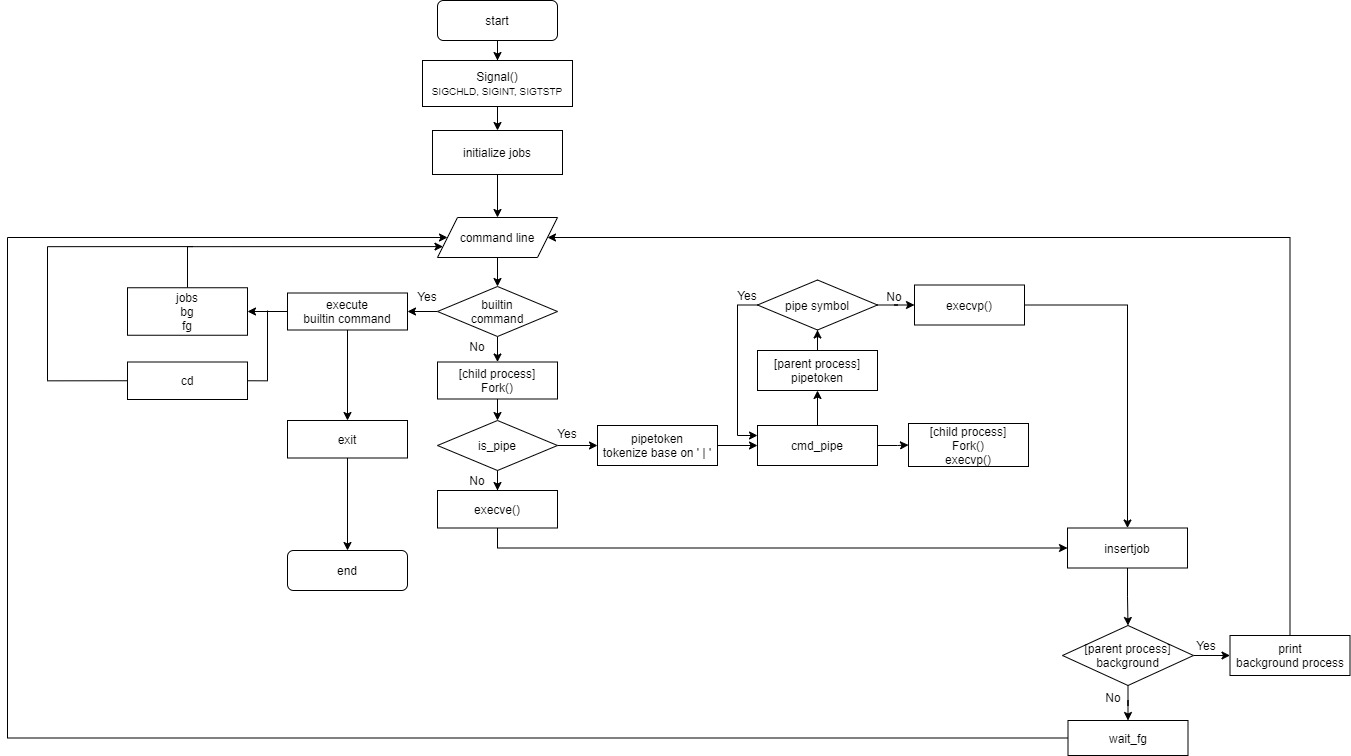
**builtin command인 경우에는 명령을 바로 실행을 하고 아닌 경우에는 fork를 통해 자식 프로세스를 생성하여 execve로 명령을 실행한다. 포그라운드 작업만 가능하므로 wait 함수를 통해 자식 프로세스가 종료될 때까지 부모 프로세스는 기다리게 된다.**

1. **Phase 2 (pipeline)**

****

**fork를 통해 자식 프로세스를 생성하고 pipe symbol이 있는지 확인한다. 있다면 symbol 기준으로 명령어를 두 부분으로 나누고 cmd\_pipe 함수로 인자를 넘긴다. fork, execvp를 통해 자식 프로세스로 첫 번째 명령을 실행하고 두 번째 명령을 다시 symbol 기준으로 나눈다. 나눠지면 cmd\_pipe를 재귀적으로 실행하고 더 이상 나눠지지 않으면 execvp를 통해 명령을 실행한다. signal 함수를 통해 종료된 프로세스들은 회수된다.**

1. **Phase 3 (background)**

****

**signal을 통해 각 신호를 다룰 수 있도록 핸들러를 지정한다. 작업 배열을 초기화한다. builtin command에서 jobs, bg, fg 명령어를 실행할 수 있다. kill은 builtin command로 실행가능하다. 부모 프로세스에서 작업 정보를 배열에 삽입하고 백그라운드 작업인 경우에는 정보 출력 후 바로 입력을 받는다. 포그라운드 작업인 경우 자식 프로세스가 끝날 때까지 기다렸다가 입력을 받는다.**