**System Programming Project 4**

담당 교수 : 박운상

이름 : 한찬희

학번 : 20182204

1. **개발 목표**

이번프로젝트는 총 3단계로 나누어서 제작되었다. 첫번째에서는 ‘cd’, ‘ls’, ‘mkdir’, ‘rmdir’, ‘touch’, ‘cat’, ‘echo’, ‘exit’와 같은 기본 shell 명령어 실행이 가능하도록 한다.

두 번째는 이러한 기본 명령어를 가지고 파이프라인기능 구현이다.

세 번째는 background 프로세스와 foreground 프로세스가 기능 구현이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

* ‘ls’, ‘mkdir’, ‘rmdir’, ‘touch’, ‘cat’, ‘echo’및 기타 기본 명령어는 fork함수로 child process를 만들어 child process에서 execvp함수를 이용해서 제작
* ‘cd’ 명령어는 chdir 함수를 이용해서 실행되기 때문에 따로 함수를 제작
* ‘exit’ 의 명령어는 쉘 접속 끊기가 아니기 때문에 \_exit(1)를 사용해서 프로그램 종료
* 각 fork별로 child 프로세스가 종료될때까지 wait3함수로 대기하였고 종료시 프로세스 삭제

1. Phase 2

* pipeline, grep를 활용한 파이프연산이 가능하다. pipe() 함수를 이용하여 명령어의 수행결과를 그 다음 명령어의 입력으로 넣어준다.

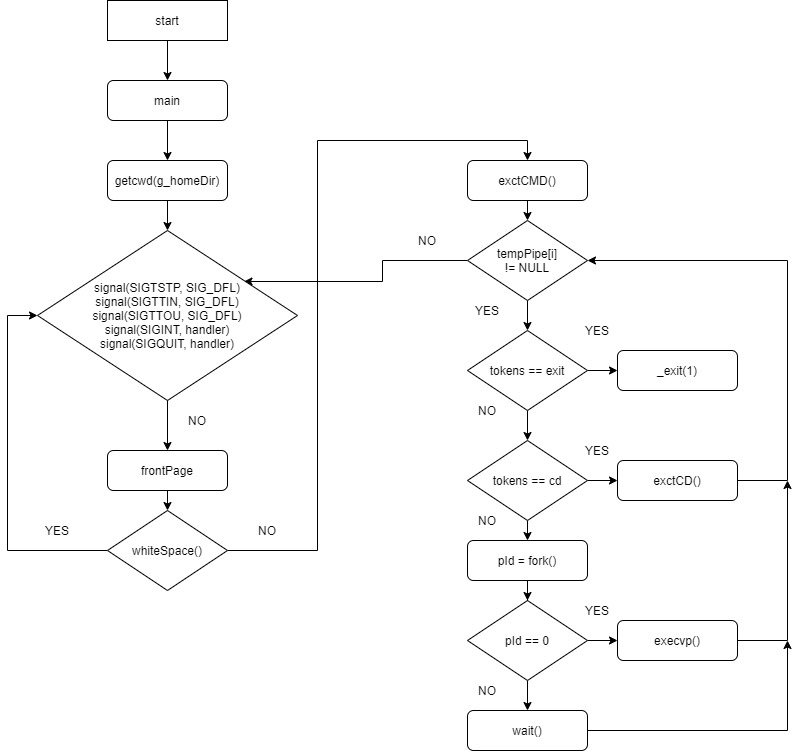
1. Phase 3

* jobs, bg, fg, kill를 활용한 프로세스 처리 가능
* job을 처리하기 위해 jobs 구조체 배열 g\_job를 만들어 명령어, job의 상태 그리고 프로세스 id를 저장하고 있다.
* jobs는 현재 g\_job 에 있는 내용을 알맞은 형식으로 모두 출력한다.
* fg에서는 끄고자하는 프로세스에 SIGCONT 시그널을 보내어 프로세스를 계속 진행한다. 그리고 job과 관련된 모든 status를 바꾸고 wait한다.
* bg 끄고자하는 프로세스에 SIGCONT 시그널을 보내어 프로세스를 계속 진행한다. 대신 프로세스를 기다리지 않고 상태만 변경한다.
* kill은 끄고자하는 프로세스에 SIGCONT 시그널을 보내어 프로세스를 계속 진행한다. 그리고 해당 프로세스의 상태를 바꾸고 삭제한다.
  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
  + connection을 종료할 때 parent process에게 signal을 보내는 signal handling하는 방법 & flow
    - cd, exit 를 제외한 다른 명령어들은 fork를 만들고 child 프로세스가 실행한다. 생성된 프로세스명은 pid\_t pId 변수에 저정하였다. ‘pId == 0’인 경우에는 자식 프로세스로 execvp() 시스템 콜을 이용하여 작동하도록 하였다. 그리고 부모 프로세스에서는 wait() 함수를 이용해서 자식 프로세스 종료를 기다리도록 하였다.
    - signal과 같은 경우 SIGTTIN, SIGTTOU는 기본 handler인 SIG\_DFL을 사용하여 처리하였다. 그와 반면의 SIGTSTP, SIGINT, SIGQUIT, SIGCHLD는 handler함수에서 처리하도록 하였다. SIGTSTP, SIGINT, SIGQUIT, SIGCHLD는 입력화면을 다시 띄우고 기존에 있던 출력값을 비우는 것으로 처리했다.
* **Phase2 (pipelining)**
  + Pipeline( ‘|’ )을 구현한 부분에 대해서 간략히 설명 (design & implementation)
  + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명
    - Pipeline을 구현하기 위해서 pipe() 함수를 이용해서 입력과 출력을 만들었다. 만들어진 파이프들은 int fd[2]에 저장해 놓았다. 그리고 dup2 함수를 사용해서 STDOUT\_FILENO로 나오는 모든 데이터를 fd[1]으로 보내서 그 다음 파이프라인이 읽을 수 있도록하였다.
    - Pipeline을 50개까지 사용하여 테스트 하였으나 큰 문제는 발견되지 않았다. 하지만 입력 받을 수 있는 명령어의 길이가 최대 10만을 넘기지 않도록 입력시에 제한을 걸어놓아 이를 넘길 경우 그 명령어를 무시하도록 하였다.
* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명
    - 이전 단계에서 pipeline을 기준으로 나누었던 tokens에 &가 있는지 확인한다.
    - &가 없는 경우 parent process에서는 해당 프로세스가 종료되기를 계속 기다린다.
    - &가 있는 경우 parent process에서는 해당 프로세스 정보를 g\_job 전역 변수에 저장한다. 그리고 자식 프로세스의 SIGTTOU를 기다린다.
    - jobs를 입력시 g\_job을 전체적으로 돌면서 프로세스들을 출력해준다.
    - fg를 입력시 사용자의 입력으로부터 job 번호와 프로세스 번호를 찾아내어 job\_id, pid에 저장한다. pid를 기준으로 kill(pid, SIGCONT)의 신호를 보낸다. 프로세스가 종료된다면 set\_job\_status로 해당 프로세스의 상태를 변경하고 출력한다. 그리고 wait\_for\_job을 호출하여 해당 프로세스가 종료되기를 기다리고 상태를 status 변수에 저장하여 상태에 알맞게 status를 변경해준다. 정상 종료시 해당 프로세스를 전역변수에서 삭제한다.
    - bg를 입력시 job번호와 프로세스 번호를 저장하고 kill(pid, SIGCONT)의 신호를 보낸다. fg와 다르게 프로세스의 종료를 기다리지 않고 프로세스의 상태만 변경 후 출력해준다.
    - kill을 입력시 job번호와 프로세스 번호를 저장하고 kill(pid, SIGKILL)의 신호를 보낸다. 해당 프로세스의 상태를 TERMINATED로 변경 및 출력해주고 프로세스가 정상 종료될때까지 기다리고 전역변수에서 삭제한다.
  1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**
* Phase1
  + main
    - getcwd() 함수로 우선 현재 dir의 위치 값을 전역변수 g\_homeDir에 읽어들인다.
    - while(1) 안에서 입력 요청 화면과 명령어를 전역변수 g\_line에 읽어들인다. 발생할 수 있는 모든 signal들을 처리하고 있는다.
    - exctCMD를 호출하여 명령어를 수행한다.
  + void exctCMD(void)
    - 받아들인 명령어를 ‘;’와 ‘|’를 기준으로 나누어 배열에 각각 저장한다.
    - ‘;’을 기준으로 명령문을 나눈 tempSemi 배열을 기준으로 while문을 돌고 그 내부에서 ‘|’를 기준으로 명령문을 나눈 tempPipe 배열을 기준으로 while문을 돌린다.
    - “<” 혹은 “>”의 명령어 입력이 있을 수 있기 때문에 이걸 기준으로 다시 나누어 tokens에 저장한다.
    - exit, cd와 toekns[0]이 같은지 확인하고 exit와 같다면 \_exit(0)을 사용해 프로그램을 종료한다. cd와 같다면 exctCD함수를 사용한다.
    - 그외의 명령어는 fork()를 사용하여 프로세스를 만들고 자식 프로세스에서는 execvp 함수를 사용하여 받은 명령어를 처리한다. 부모 프로세스에서는 자식 프로세스를 wait()함수로 기다린다.
  + void exctCD(char \*\*tokens)
    - main에서 읽어들인 g\_homeDir 전역변수를 지역변수 home\_tmp에 저장한다.
    - 만약 이번에 받은 명령어 1번이 NULL 이라면 chdir(g\_homeDir) 시스템콜을 이용해서 원래의 주소로 돌아간다.
    - 만약 이동 위치가 ~으로 시작한다면 home\_tmp에 token의 0번 다음 1번 성분부터 끝까지 삽입하고 chdir(home\_tmp)로 이동한다.
    - 정상적인 입력의 경우 chdir(tokens[1]) 로 이동한다.
    - 만약 이동시 에러가 생기면 에러 메시지를 띄어준다.
  + handler(int sig)
    - SIGINT, SIGTSTP, SIGQUIT, SIGCHLD에 대해 에러처리를 한다. frontPage()를 다시 띄우고 원래 출력하려고 했던 결과를 무마시킨다.
    - 하지만 대다수의 경우 SIG\_DFL로 Default handler가 처리하도록하였다.
  + frontPage()
    - 입력 요청화면을 띄어준다.
  + trim(char \*cmd)
    - cmd로 받은 문자열의 맨 앞과 뒤의 빈칸을 제거해준다.
  + parse(char \*cmd, char \*\*tokens, char \*limit)
    - limit를 기준으로 cmd의 명령어를 나누어 tokens에 저장한다.
* Phase2
  + excuCMD(void)
    - 명령어가 cd인지 확인한 다음에 pipe() 함수를 사용하게 된다. pipe함수를 사용하기 위해 int fd[2]를 선언해 놓는다.
    - pipe(fd)로 두개의 input, output으로 사용될 변수들을 선언해 놓는다.
    - fork()로 나온 pId == 0인 child process 내부에서 이전에 나누어 놓은 tempPipe를 기준으로 그 다음 Pipe가 있는지 확인한다. 만약 그 다음 Pipe가 있다면 dup2를 사용하여 STDOUT\_FILENO로 나오는 결과값을 fd[1]으로 설정해 준다. 즉 그 다음 child process의 입력으로 넣어준다. 그리고 close(fd[1]) 해주어 입력을 닫아준다.
    - parent process에는 close(fd[1])해주어 자식에게 들어갈 수 있는 입력을 막아준다.
* Phase3
  + 사용한 구조체와 전역변수
* typedef struct jobs
* {
* char    pName[100];
* int     status;
* pid\_t   pid;
* }s\_JOBS;
* const char    \*g\_status[] = { *// JOB의 상태를 저장하는 변수*
* "running",
* "done",
* "suspended",
* "continued",
* "terminated"
* };
* int     g\_JOBCNT = 0; *// JOB의 갯수를 저장하고 있는 변수*
* s\_JOBS  g\_job[MAX\_JOB]; *// JOB을 저장하고 있는 배열*
  + exctCMD()
    - 명령문을 | 와 스페이스바로 나눈 다음 backgroundChk 함수를 사용해 &가 입력되어 있는지 확인한다. 만약 있었다면 bgFlag를 1로 바꾸어준다.
    - 그 이후 명령어에 따라 처리하는 과정에서 parent process과정에 오게 된다면 job 개수가 제한에 걸리지 않는 한 자식 프로세스의 id, status, 명령어등을 g\_job 전역변수에 저장한다. 이때 g\_job에서 남아있는 job 공간에 정보를 넣어준다.
  + remove\_job(int id)
    - 삭제 요청을 받은 id(job의 번호)를 받아서 해당 프로세스를 기준으로 뒤에 있는 모든 job을 한칸씩 앞으로 당기어 주고 job의 개수를 저장하고 있는 g\_JOBCNT의 개수를 줄여준다.
  + set\_job\_status(int id, int status)
    - 변경 요청을 받은 id의 g\_job[id].status를 사용자로부터 변경 요청 받은 status로 변경해준다.
  + print\_job\_status(int id)
    - id로 받은 job번호의 정보를 출력해준다.
  + wait\_for\_job(int id)
    - id로 받은 job 번호의 process id에 해당하는 자식 프로세스에 waitpid를 요청한다. 이때 WUNTRACED를 설정하여 중단된 자식 프로세스의 상태도 받아 status에 저장한다.
    - status에 따라서 set\_job\_status를 호출하여 해당 프로세스의 상태를 변경해준다. WIFEXIETED는 DONE, WIFSIGNALED는 TERMINATED, WSTOPSIG는 SUSPEND롤 변경해준다.
  + exctJobs(void)
    - g\_job전체를 출력해준다. 이때 g\_job에 저장되어 있는 값이 NULL이 아닌 job에 대해서만 print\_job\_status를 사용하여 출력해준다.
  + exctFg(char \*\*tokens)
    - tokens에는 fg <job>문자열이 space를 기준으로 나누어 들어가 있다. 만약 <job> 이 NULL로 설정되어 있다면 기능을 수행해주지 않는다.
    - %로 시작하는 입력을 넣었을 경우 해당 job번호를 추출하여 job\_id에 넣어준다. 그리고 job\_id에 해당하는 process id는 pid에 저장해 준다.
    - 만약 %로 시작하지 않는다면 바로 job\_id와 pid에 해당하는 정보를 g\_job에서 구해서 저장해준다.
    - kill(pid, SIGCONT)를 호출하여 해당 프로세스가 계속 작업 후 삭제되도록 signal을 보낸다.
    - g\_job[job\_id] 에 해당하는 status를 continued로 변경해 주고출력해준다. 그리고 프로세가 끝날때까지 wait\_for\_job 함수를 호출해서 기다려주고 정상 종료시 remove\_job함수를 호출하여 g\_job에서 해당 프로세스의 정보를 삭제해 준다.
  + exctBG(char \*\*tokens)
    - tokens에는 fg <job>문자열이 space를 기준으로 나누어 들어가 있다. 만약 <job> 이 NULL로 설정되어 있다면 기능을 수행해주지 않는다.
    - %로 시작하는 입력을 넣었을 경우 해당 job번호를 추출하여 job\_id에 넣어준다. 그리고 job\_id에 해당하는 process id는 pid에 저장해 준다.
    - 만약 %로 시작하지 않는다면 바로 job\_id와 pid에 해당하는 정보를 g\_job에서 구해서 저장해준다.
    - kill(pid, SIGCONT)를 호출하여 해당 프로세스가 계속 작업 후 삭제되도록 signal을 보낸다.
    - set\_job\_status를 호출하여 CONTINUED 상태로 바꾸어주고 출력한다.
    - 이번에는 job을 기다려주지 않는다.
  + exctKill(char \*\*tokens)
    - tokens에는 fg <job>문자열이 space를 기준으로 나누어 들어가 있다. 만약 <job> 이 NULL로 설정되어 있다면 기능을 수행해주지 않는다.
    - %로 시작하는 입력을 넣었을 경우 해당 job번호를 추출하여 job\_id에 넣어준다. 그리고 job\_id에 해당하는 process id는 pid에 저장해 준다.
    - 만약 %로 시작하지 않는다면 바로 job\_id와 pid에 해당하는 정보를 g\_job에서 구해서 저장해준다.
    - kill(pid, SIGCONT)를 호출하여 해당 프로세스가 계속 작업 후 삭제되도록 signal을 보낸다.
    - set\_job\_status를 호출하여 TERMINATED 상태로 바꾸어주고 출력한다.
    - job의 프로세스가 종료되기를 기다려준다. 그리고 정상종료시 해당 프로세스를 remove\_job을 호출하여 삭제해 준다.

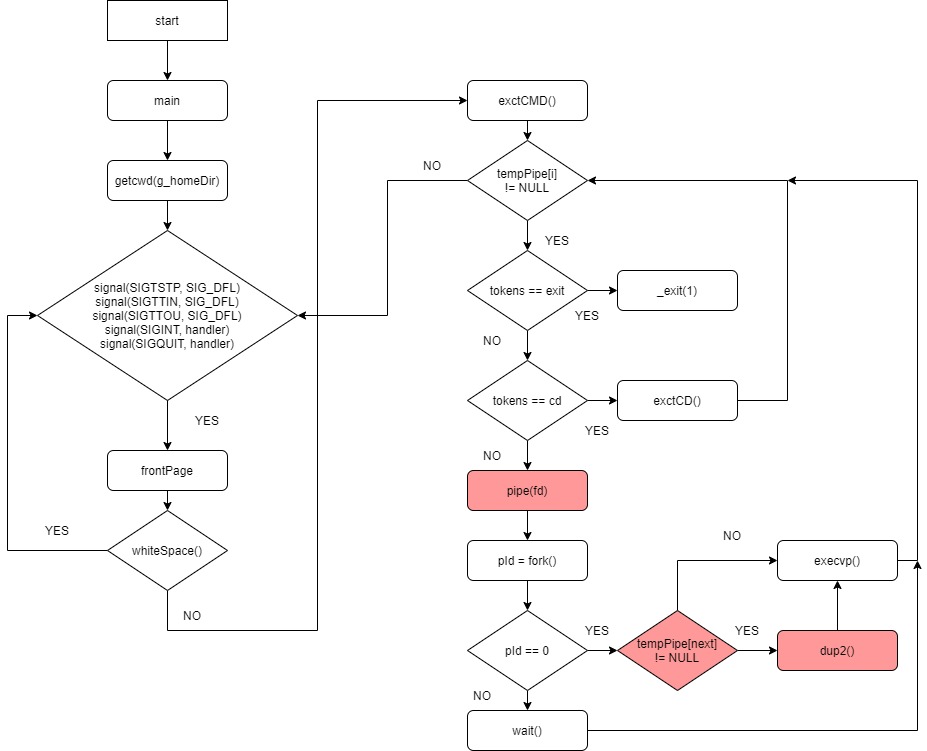
1. **구현 결과**
   1. **Flow Chart**

* **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)**

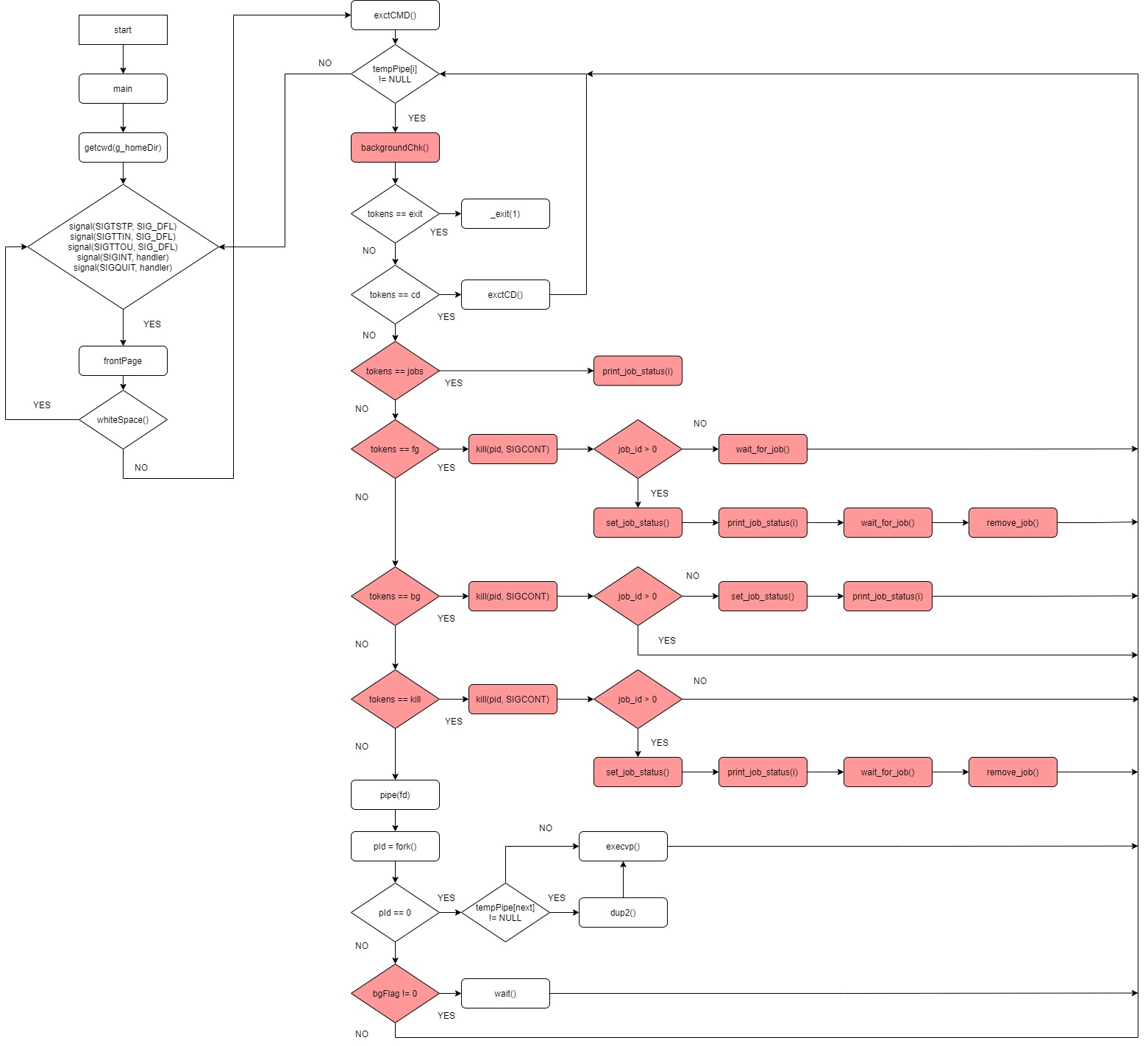
****

1. **Phase 2 (pipeline)**

****

**붉은 색 부분이 phase1과 다른 부분입니다.**

1. **Phase 3 (background)**

** 붉은색으로 칠해진 부분이 phase2와 달라진 부분입니다.**