**System Programming Project 4**

담당 교수 : 김영재

이름 : 임정호

학번 : 20171682

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* **(MyShell을 만드는 전체적인 개요에 대해서 작성하면 됨.)**  
  자체 프로그래밍된 shell을 구현함으로써 프로세스 제어, 프로세스 신호 전달, 프로세스 간 통신 및 프로세스 및 작업 실행등의 시스템 수준의 개념을 배우고 익힌다. 구현된 shell은 기본적인 shell command의 수행과, redirection, piping, 백그라운드 실행을 지원할 수 있게 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Phase 1

cd, ls, mkdir, rmdir, touch, cat, echo, exit의 command를 실행할 수 있다.

command에 맞는 수행을 끝내면, 다시 prompt를 출력해서 다음 명령을 키보드로부터 받을 수 있다.

1. Phase 2

Phase 1에서 구현된 기능에서 pipelining 기능을 확장한다.

ls -al | grep filename과 pipe command를 수행할 수 있다.

한 개뿐만 아니라 여러 개의 pipe로 이루어진 command도 수행할 수 있다.

1. Phase 3

구현한 shell이 job control를 지원할 수 있다.

jobs, kill %n, fg %n, bg %n의 명령어를 수행할 수 있다.

foreground와 background에서 명령어를 수행시키고, 멈추거나 종료시킬 수 있다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술**
* **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
* **Phase1 (fork & signal)**

**fork를 통해서 child process를 생성**

redir\_pip()함수를 구현하여 child process를 fork하고, 명령어를 수행할 수 있게 하였다. redir\_pip()함수에서 fork()를 호출하여 자식 프로세스를 생성하고, 자식 프로세스이면 (pid가 0인 경우) execvp()함수를 이용하여, /bin에 있는 명령을 수행할 수 있게 하였다. cd나 exit의 명령어인 경우 builtin함수에서 처리해주었고, 부모 프로세스이면 (pid>0인 경우) 자식 프로세스를 foreground 명령인 경우 waitpid()함수를 이용하여 pid를 기다리게끔 하였고, background 명령인 경우 기다리지 않고, job queue에 enqueue하고 백그라운드에서 실행되는 index와 pid를 출력한다.

**connection을 종료할 때 사용되는 signal handling 방법 & flow**

child process가 종료되면 sigchld 시그널을 보내게 되고, 부모 프로세스는 그 시그널을 waitpid()를 통해 기다린다. foreground 실행의 경우 부모 프로세스가 waitpid()를 이용하여 명시적으로 기다리지만, background 실행의 경우는 명시적으로 기다리지 않기 때문에 sigchld\_handler를 이용하여 sigchld를 handle할 수 있게 하였다. 더 자세한 내용은 Phase 3에서 설명할 것이다.

* **Phase2 (pipelining)**

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

우선 | 를 구분하기 위해 eval()함수에서 ‘|’ 를 기준으로 parseline()를 반복적으로 호출하여 명령어를 나눈다. pipe 5개까지 지원할 수 있도록 명령어를 저장하는 배열을 할당하였고, eval()에서 ‘|’ 기준으로 분리된 명령어를 매개변수로 넘기면서 해당 명령을 수행하는 redir\_pip()를 구현하였다. redir\_pip()함수는 파이프 기준으로 더 수행할 명령어가 없을 때까지 재귀호출되는 재귀함수이다. pipelining의 가장 핵심적인 내용은 이전 파이프 명령의 결과가 그 다음 파이프 명령의 입력으로 들어가는 점인데, 이를 위해서 pipe()와 dup2()함수를 이용하였다. redir\_pip()함수는 크게 4부분으로 나누어 볼 수 있다. 마지막 명령어인지 아닌지 구분하고, 그 다음은 자식 프로세스인지, 부모 프로세스인지 구분하기 때문이다. 마지막 명령어가 아닌 경우는 dup2()를 이용하여 stdout을 가리키는 file descriptor를 fd[1]로 변경하고, 마지막 명령어인 경우는 정상적으로 stdout에 출력해야한다. 그렇기 때문에 redir\_pip()함수 내에서 구분하였다. 현재 명령어을 수행하는 프로세스에 입력은 그 전에 수행했던 프로세스의 결과이므로, 이를 위해 함수 매개변수로 file descriptor를 넘겼다. 마지막 명령어인 경우는 stdin을 가리키는 file descriptor를 매개변수로 받은 file descriptor로 연결해주었고, stdout은 건들지 않았다. 마지막 명령어가 아닌 경우는 stdout을 fd[1]로 연결하여, 현재 프로세스의 실행결과가 화면이 아닌 pipe에 쓸 수 있도록 구현하였다. 사용하지 않는 파이프의 file descriptor는 close()를 하였는데, 이 close()의 위치에 따라 bad file descriptor 에러가 나기도 하였고, 실행이 멈추기도 하였다. 부모 프로세스인 경우 fd[1]을, 자식인 경우 fd[0]을 close하였다. 그리고 dup2()를 이용하여 file descriptor 설정이 끝나면 close()하였다.

**Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명**

앞서 설명하였듯이, redir\_pip()함수는 재귀함수로써, 더 수행될 파이프 명령이 없을 때까지 수행된다. pipe의 개수가 5개까지 실행될 수 있게 변수를 할당하였는데, 할당하는 숫자를 바꾼다면 더 많아도 재귀함수이기 때문에 별 문제 없이 수행될 것이다.

* **Phase3 (background process)**

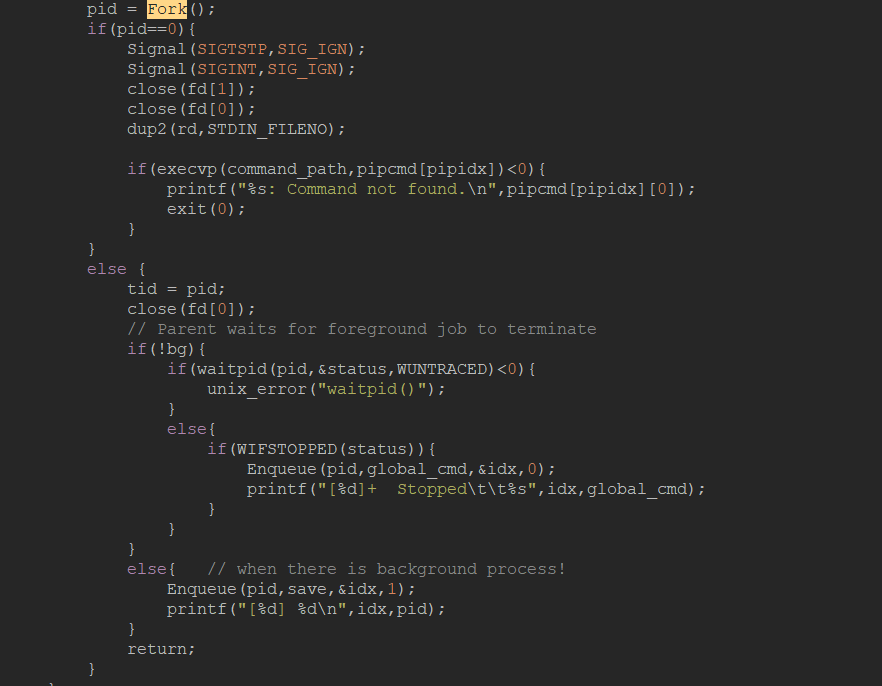
**Background (’&’) process를 구현한 부분에 대해서 간략히 설명**

우선 eval()함수에서 입력된 명령어의 끝부터 읽기 시작하여 ‘&’를 만나면 전체 명령어를 save라는 변수에 저장한다. 이는 redir\_pip()에서 enqueue할 때 전체 명령어를 알 수 없기 때문에 그렇다. 이 save 변수를 매개변수로 redir\_pip()에 넘긴다. background 실행인 경우 마지막 명령어의 경우는 waitpid로 명시적으로 기다리지 않는데, 아닌 경우는 waitpid()에 WUNTRACED를 옵션으로 설정한 뒤 기다린다. 이렇게 한 이유는 내 코드에서는 예를 들어 ls | grep csapp &을 실행하면 ls가 defunct가 되어버리고, 부모 프로세스가 reap을 제대로 못 해주는 상황이 발생해서 그렇다.

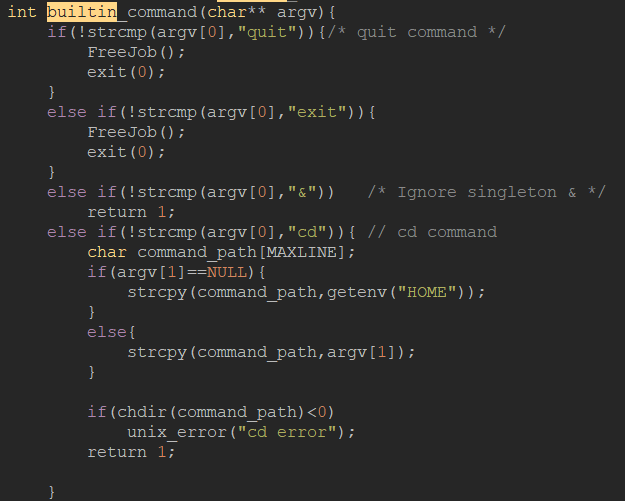
**ctrl+z, ctrl+c, jobs, kill, bg, fg 기능 구현**

ctrl+z, ctrl+c의 입력을 처리하기 위해 sigint\_handler와 sigtstp\_handler를 구현하여 각 시그널을 처리할 수 있게 하였다. 전역변수 F와 tid를 설정하여 각 상황에 맞게 결과를 수행할 수 있게 조정하였다. F가 1이면 promt의 경우이고, 0이면 어떤 명령을 수행하던 중 시그널을 받은 것이다. 그에 맞게 처리했다. jobs, kill, fg, bg는 built\_in command로 built\_in 함수에서 구분하여 각 명령어에 맞게 함수를 호출하였다. jobs를 위해 background에서 실행되거나 멈춘 명령어를 저장하기 위해 구조체를 설정하였고, 이를 queue로 구현하여 관리한다. fg와 bg의 함수에서는 SIGCONT 시그널을 Kill()에 매개변수로 보내 멈춰있던 프로세스를 다시 실행시켰고, fg 명령어의 경우 다시 ctrl+z를 받으면 그에 맞게 출력을 해주었고, 아닌 경우는 background 명령들을 저장하는 큐에서 dequeue해주었다. kill 명령어의 경우 Kill()에 sigkill신호를 보내 해당 프로세스가 죽게 처리하였다.

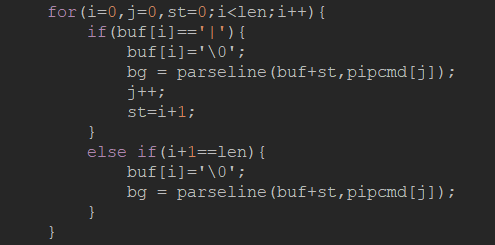
* 1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**
* **Phase1 (fork & signal)**



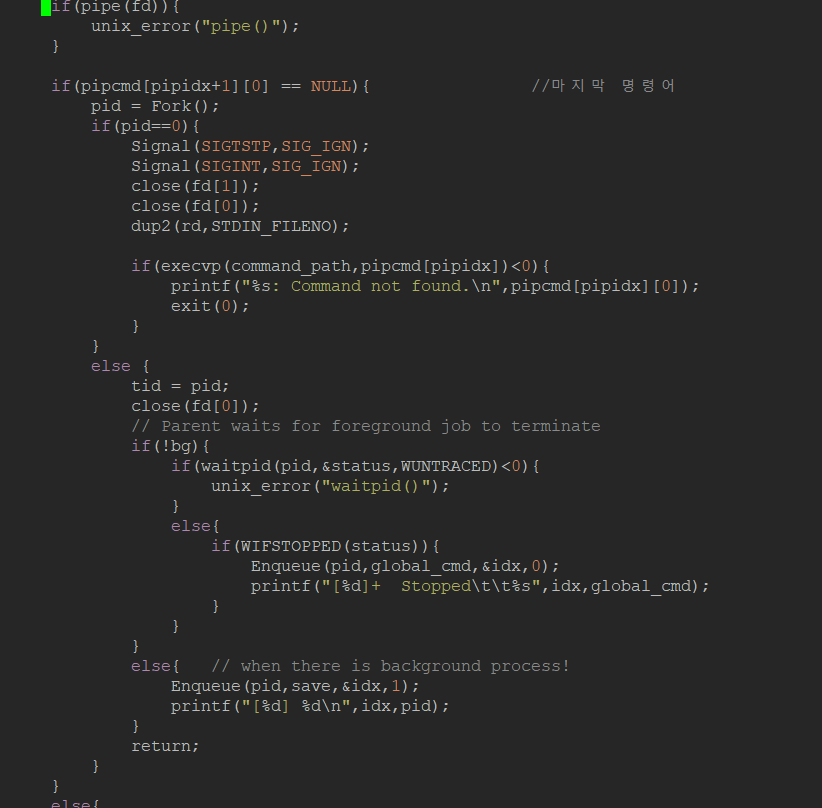
Fork()를 호출하여 자식 프로세스를 생성하여 부모와 자식을 구분하였고, Signal()부분은 phase 3 부분에서 설명할 것이다. 우선 자식 프로세스인 경우, 입력받은 command가 builtin command가 아닌 경우 execvp()를 이용하여 /bin에 있는 명령을 찾아 실행할 수 있게 하였다. 부모프로세스인 경우에서는 waitpid()를 이용하여 sigchld signal을 기다린다.



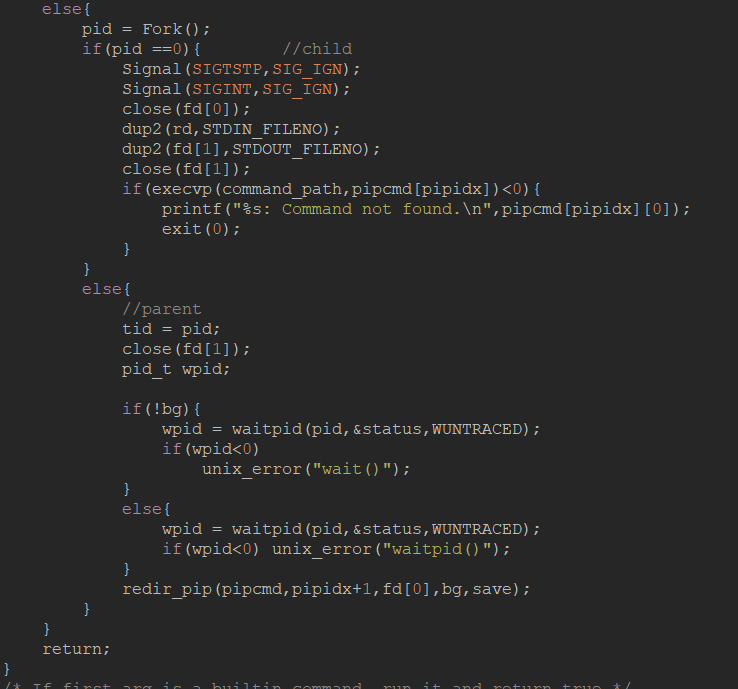
builtin command는 위처럼 처리하였다. cd의 경우는 chdir()를 이용하여 현재 디렉토리의 정보를 바꾸었다.

* **Phase2 (pipelining)**
* 

eval()에서 ‘|’를 기준으로 buf에 저장되어 있는 command를 구분한 후, pipcmd[]에 각 파이프 명령어를 저장한다. 그 후 처음 입력된 command가 builtin command가 아닌 경우 redir\_pip()함수를 호출한다.

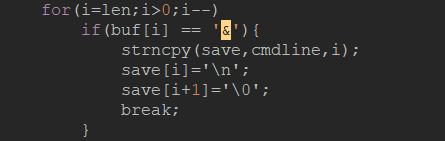


pipe()를 호출하여 file descriptor를 생성한다. 마지막 명령어인 경우는 dup2()를 이용하여 stdin만 매개변수로 받은 file descriptor로 연결해주고, stdout은 건들지 않는다. 그 후 execvp()를 이용하여 처리한다. 마지막 명령어인 경우는 사실 phase 1과 같다. phase1에서 달라지는 점은 마지막 명령이 아닌 경우이다.

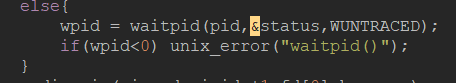
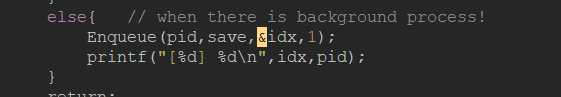


자식 프로세스에서는 dup2()를 이용하여 stdin을 매개변수로 받은 fd인 rd로 바꾸고, stdout을 fd[1]으로 바꿔준다. 부모 프로세스에서는 waitpid()를 이용하여 자식 프로세스의 종료를 기다린다. bg가 1이면 background이고, 0이면 foreground인데, bg가 1인 경우는 phase 3 부분에서 설명할 것이다. 부모 프로세스가 자식 프로세스를 reap해준 다음은 자기 자신을 다시 호출하여 마지막 명령어를 수행한다.

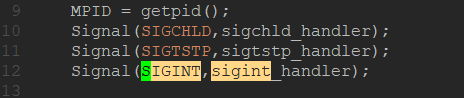
* **Phase3 (background process)**
* **Background (’&’) process를 구현한 부분**



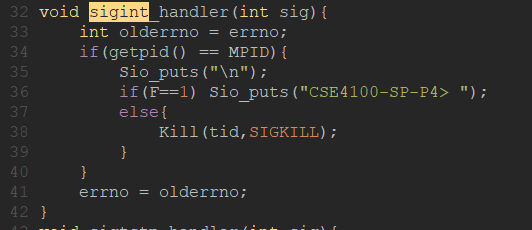
buf에는 입력받은 command가 복사되어 있는데, 뒤에서부터 확인하여 ‘&’가 있는지 확인한 후, 있으면 save에 전체 command를 복사한다. 이는 나중에 enqueue할 때 전체 명령어가 필요하기 때문이다. pipcmd를 0부터 반복문으로 넣어 전체 명령어로 재조합하는 방법도 있었는데, 위처럼 구현하는 것이 더 간편하였다.



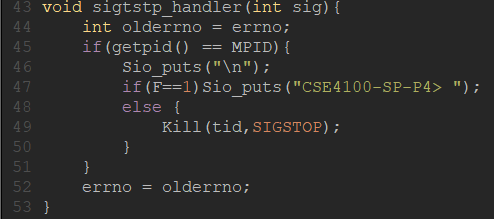
redir\_pip()에 있는 부모프로세스에서 마지막 명령어인 경우와 아닌 경우이다. 마지막 명령어일 때 enqueue하는데, 앞서 설명했던 save를 넣는 것을 확인할 수 있고, &idx는 job queue에서의 job index를 저장하여 출력해주기 위함이다. 마지막에 있는 1은 running임을 알려주기 위함인다. stopped인 경우는 enqueue()할 때 0으로 매개변수를 넘긴다.

* **ctrl+z, ctrl+c, jobs, kill, bg, fg 기능**
* 

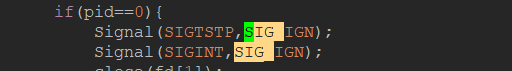
우선 main에서 Signal()를 이용하여 signal handler를 등록한다. main에서 MPID를 부모 프로세스의 pid로 저장하였다.



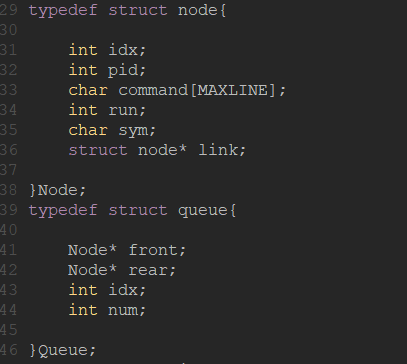
Ctrl+c를 처리하는 handler이다. 우선 MPID와 현재 프로세스의 pid를 비교하여 부모 프로세스인 경우 처리한다. F가 1인 경우는 prompt 창에서 입력을 받은 경우이므로 다시 prompt창이 뜰 수 있게 하였고, 1이 아닌 경우는 어떠한 명령어를 수행하다가 ctrl+c가 입력받은 경우이므로 tid를 죽이는 SIGKILL 시그널을 보내게 된다. 이 때 tid는 전역변수로 현재 처리하는 자식 프로세스의 pid를 저장한다.



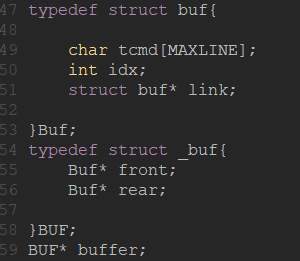
Ctrl+z를 처리하는 handler이다. sigint\_handler처럼 부모 프로세스인지 확인하고, prompt 창에서 입력받은 경우 다시 prompt창이 뜰 수 있게 하였다. 어떤 command를 수행하던 중에 SIGTSTP를 받은 경우는 실행하던 자식 프로세스에게 SIGSTOP을 보내 멈추도록 하였다.



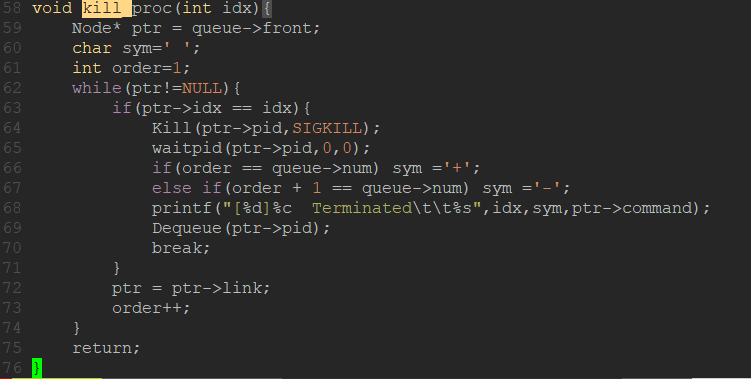
자식 프로세스에서는 SIGTSTP와 SIGINT를 SIG\_IGN으로 처리하여 받지 않게 하였는데, 이는 부모프로세스에서 handle해주기 위함이다. 처음에 getpid()를 이용하여 handler 처리를 하였는데, 이러다 보니 foreground에서 실행되는 프로세스 뿐 아니라 back ground에서 실행되던 프로세스까지 전부 종료되거나 멈춰버리는 에러가 존재하였고, 이를 처리하기 위해 SIG\_IGN으로 signal handler를 자식 프로세스 한해서 다시 등록하였다.



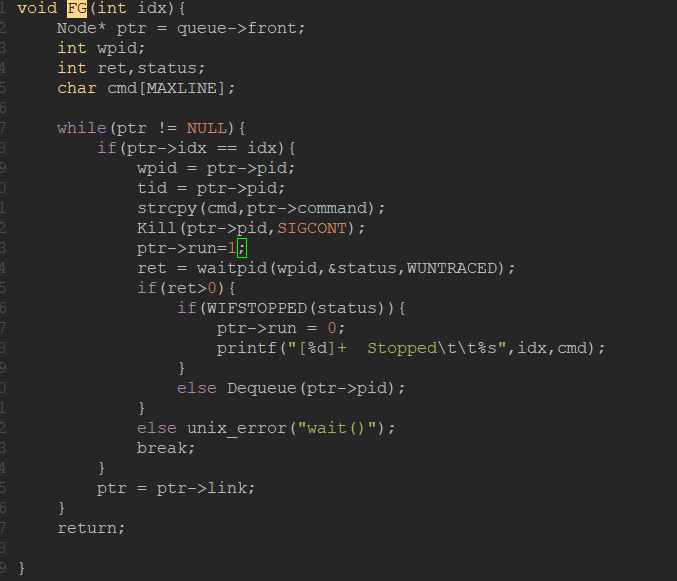
background에서 실행되거나 멈춰있는 프로세스의 정보를 저장하기 위해 큐를 사용하였다고 B에서 설명하였는데, 그 구조체이다. 어떤 명령이 foreground로 실행되던 중에 ctrl+z가 입력된 경우 &를 사용한 명령이 입력된 경우 queue에 저장된다. back ground에서 실행이 종료되면 dequeue된다.



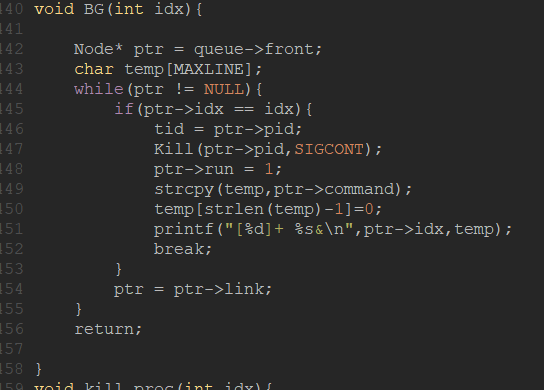
위 구조체는 back ground에서 실행되던 프로세스가 종료되면, 종료되었다는 출력 표시 Done을 다음 command가 입력되고 그 결과가 출력되고 나서 출력하게 되는데, 그 시간차가 존재한다. 그렇기 때문에 프로세스가 background에서 종료되었을 때, 저장해주는데 그 때 사용되는 구조체이다. 출력과 동시에 메모리 해제한다.



kill 명령을 수행할 때 background에서 관리되는 프로세스들 중에 입력받은 job번호에 해당하는 프로세스 노드를 찾아 SIGKILL을 보낸다. 그 다음 제거되었다는 출력을 해준 다음, queue에서 제거해준다.

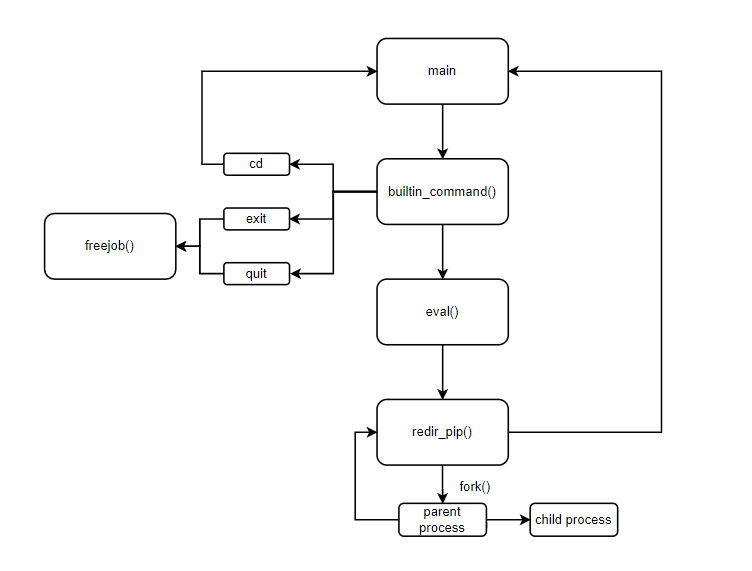


fg의 경우 queue에서 일치하는 job 번호를 가진 프로세스 노드를 찾아 SIGCONT 시그널을 보낸다. 보낸 다음 waitpid()를 이용하여 기다리는데, WUNTRACED를 이용하여 Ctrl+z가 입력된지 확인한다. 다시 멈추면 Stopped 출력을 처리해주고, 아닌 경우는 queue에서 제거해준다.

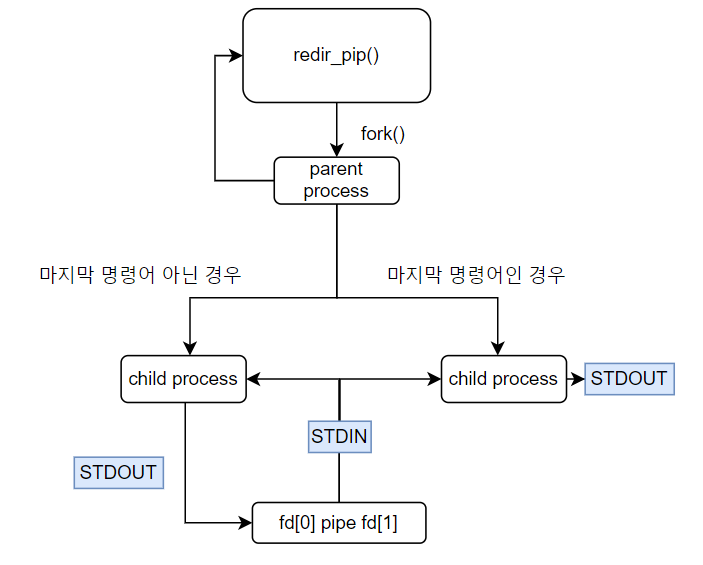


bg의 경우 fg와 동일하게 queue에서 일치하는 job 번호를 탐색한 후 SIGCONT 시그널을 보낸다. 보낸 다음, 다시 running 상태임을 알리는 출력 처리르 해준 다음 종료한다.

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



1. **Phase 2 (pipeline)**



1. **Phase 3 (background)**

