**System Programming Project 2**

담당 교수 : 이영민 교수님

이름 : 김상협

학번 : 20210354

1. **개발 목표**

자신만의 리눅스 쉘을 만들어 process 제어, signal , process 간 signal 전달, 실행중인 process 와 job 들에 대해서 배웁니다. Phase 1 에서 기본 shell 명령어를 실행 할 수 있도록 하고, phase 2 에서는 phase 1 의 구현 내용을 확장시켜 명령어의 command 개수 제한이 없는 pipeline 을 구현합니다. Phase 3 에서는 background 와 foreground process 들의 행동을 구현합니다. 사용자가 둘 모두 자유롭게 사용할 수 있도록 mini shell 을 구현합니다.

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

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

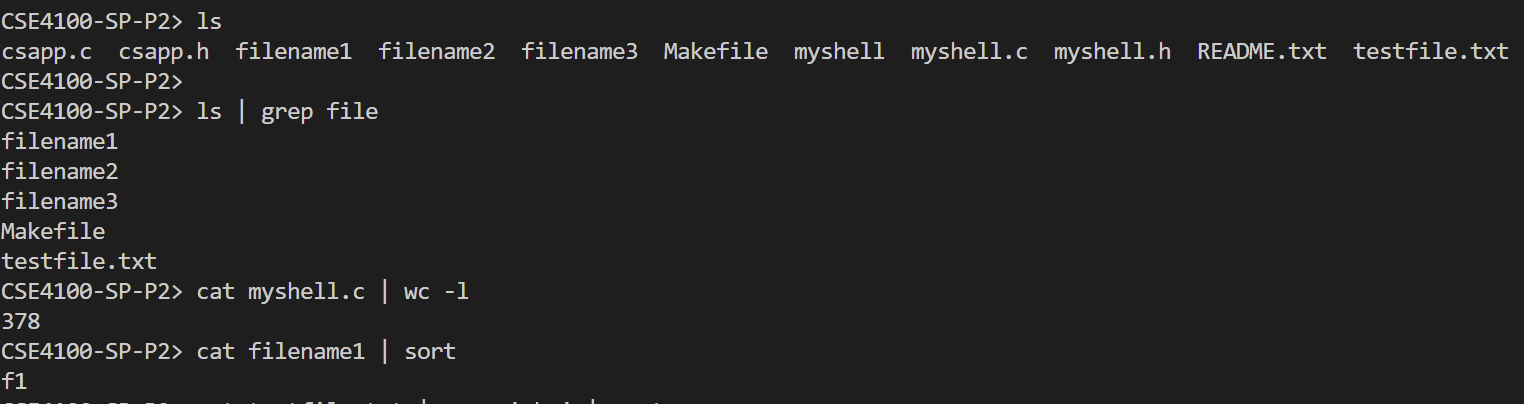
텍스트, 폰트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

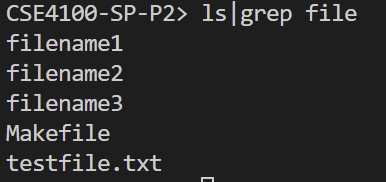
Linux shell 의 기본 내장 명령어를 실행하도록 개발하였습니다. 위는 phase1 의 myshell 실행 예시입니다. cd, ls, mkdir, rmdir, touch, echo, cat, exit 명령을 수행할 수 있도록 하였습니다. 이때 명령어들은 Fork 를 통해서 자식 process 를 생성해 수행합니다.

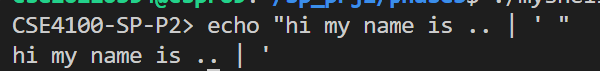
1. Phase 2

리눅스 쉘의 pipeline 을 구현합니다. 명령어를 parsing 한 뒤 built in command 를 제외한 명령어는 fork 를 통해 Pipeline 의 왼쪽 명령어의 output 이 오른쪽 명령어의 입력으로 들어가도록 설계합니다. 아래는 명령어를 입력한 결과들 입니다.



이런식으로 pipe line 이 잘 작동하는 것을 볼 수 있습니다.





이렇게 | 와 다른 문자열이 붙어있는 경우도 리눅스 쉘과 동일하게 잘 처리할 수 있도록 parsing 하였습니다. 추가적으로 “ 말고도 ‘ 같은 경우도 잘 처리할 수 있도록 구성하였고. Phase 3 에서 요구하는 문자열에 & 가 붙은 경우에도 명세서의 요구사항대로 처리할 수 있도록 구성했습니다.

1. Phase 3

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Background process 를 구현합니다. 추가적으로 background 를 fg bg kill jobs 와 같은 커맨드를 통해 자유롭게 조정할 수 있도록 합니다. 또, signal safe handling 할 수 있도록 구현하였습니다.

* 1. **개발 내용 (기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
* **Phase1 (fork & signal)**
  + fork를 통해서 child process를 생성하는 부분에 대해서 설명

shell 에서 command 를 입력 받아 parsing 하고 , 파싱 후 명령어를 수행합니다. Builtincommand 명령어를 제외한 명령어의 경우는 fork 를 이용해서 child process 에서 수행합니다. Child process 는 pid == 0 인 경우로 /bin/ 에 저장되어 있는 명령어를 execve 를 통해 수행합니다. Parent process 에서는 child process 를 waitpid 로 기다렸다가 reaping 해 줍니다.

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

child process 가 종료될 시 SIGCLD 시그널을 보내고, 앞서 설명했듯이 parent process 에서는 waitpid 로 시그널을 받고 shell 을 계속해서 실행합니다. 더 자세한 시그널 처리 같은 경우는 phase 3 부분에서 서술하겠습니다.

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

명령어를 입력받은 뒤 명령어에 대해서 pipe line 이 포함되어 있는 명령어인지 아닌지 먼저 판단한 뒤 서로 다르게 처리하였습니다. Pipe line 이 존재하지 않는 경우는 phase 1 과 동일하게 처리하였고, pipe line 이 있는 경우에는 총 pipe 가 몇 개 존재하는지 count 를 먼저 해둔 후 pipe line 을 기점으로 파싱해서 각 인자를 분리하였습니다. 파싱을 할 때 “|” 기호 처리도 중요하지만 무엇보다 quotes 처리에 신경을 많이 썼습니다. 기존 parseline 함수를 참고해서 parseline 함수자체를 다시 재구성하였습니다. 재구성한 부분에 대해서는 C. 개발방법 에서 더 자세히 설명하겠습니다.

* + Pipeline 개수에 따라 어떻게 handling했는지에 대한 설명

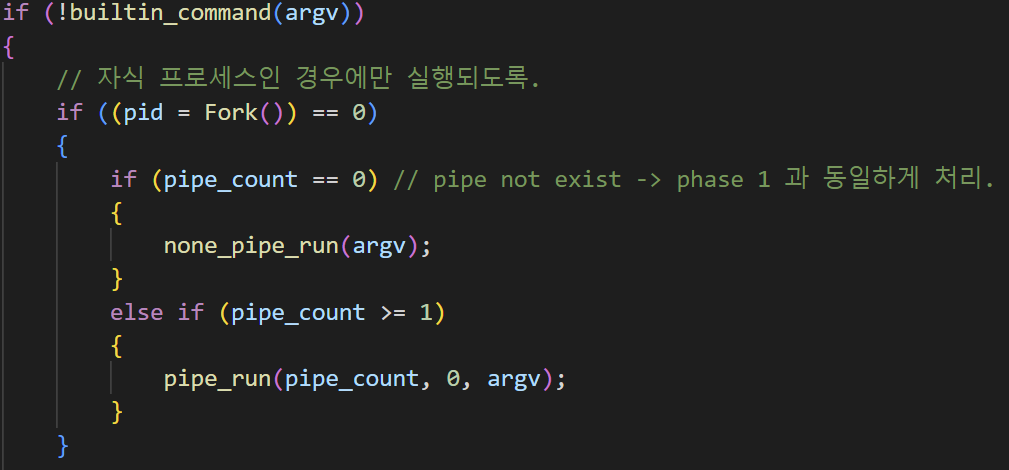
먼저 총 파이프 개수에 대해서 count 한 뒤에 , parsing 이 된 문자열을 argv 에 저장 후 재귀적인 방식을 통해서 handling 했습니다. 총 파이프 라인 개수를 count 한 이유는 마지막 base case (pipe line 에서 마지막 명령어) 에서는 실행 방식을 달리해야 하기 때문입니다. 재귀적인 방식으로 구현했기 때문에 command의 개수가 MAXAGRS 넘어가지 않는 선에서는 명령어의 개수 제한 없이 잘 작동할 것 입니다.

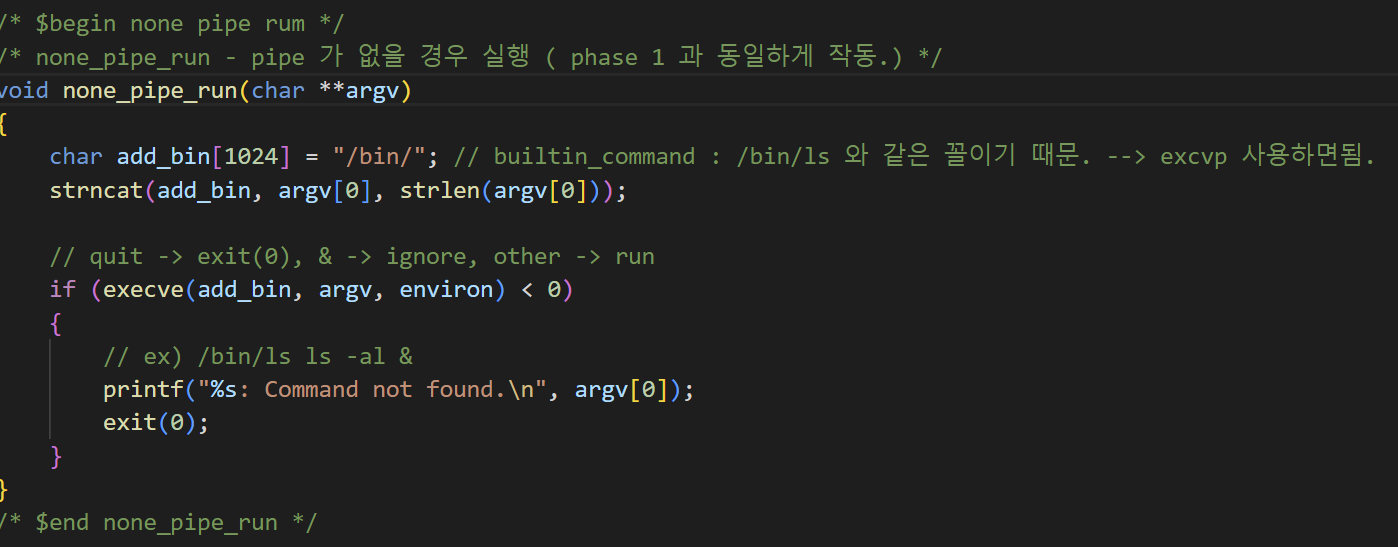
* **Phase3 (background process)**
  + Background (’&’) process를 구현한 부분에 대해서 간략히 설명

백그라운드 프로세스를 관리하기 위해서 job 구조체를 만들고 jobs 라는 리스트를 통해서 구조체들을 관리하였습니다. ‘&’ 를 사용자가 백그라운드에 넣고 명령어를 입력하면 이 문자는 제거하고 파싱후에 jobs 리스트에 넣어주도록 구현했습니다. Job 명령어는 백그라운드 프로세스들의 번호,상태, 명령어를 출력하고 bg 는 해당 번호와 일치하는 백그라운드 프로세스를 SIGCONT 시그널을 보내서 정지되었던 프로세스가 다시 실행되도록 하였고 fg 는 해당 프로세스를 job 리스트에서 제거하도록 하였다. fg 는 구현에 실패했는데 waitpid 같은걸로 부모 프로세스에서 기다린 뒤 reaping 해야 할 것 같은데 왜인지 동작하지 않아서 일단은 SIGKILL 로 강제종료 하는 것으로 일단 구현해 두었다. Kill 은 해당 번호에 일치하는 프로세스를 SIGKILL 로 종료한뒤 해당 번호에 해당하는 jobs 에서 삭제해 주었다. 더 자세한 구현방법은 아래에서 설명하겠다.

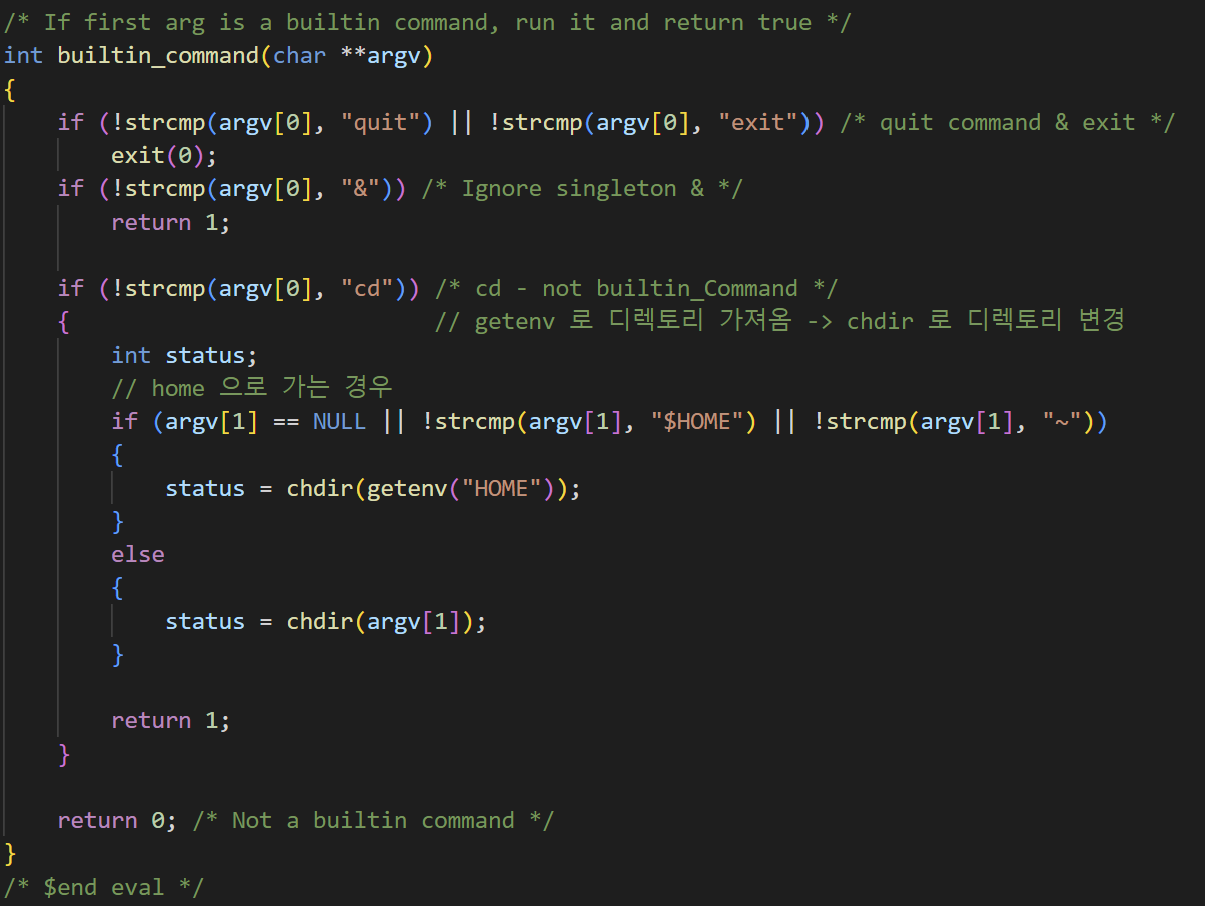
* 1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

Phase<1>



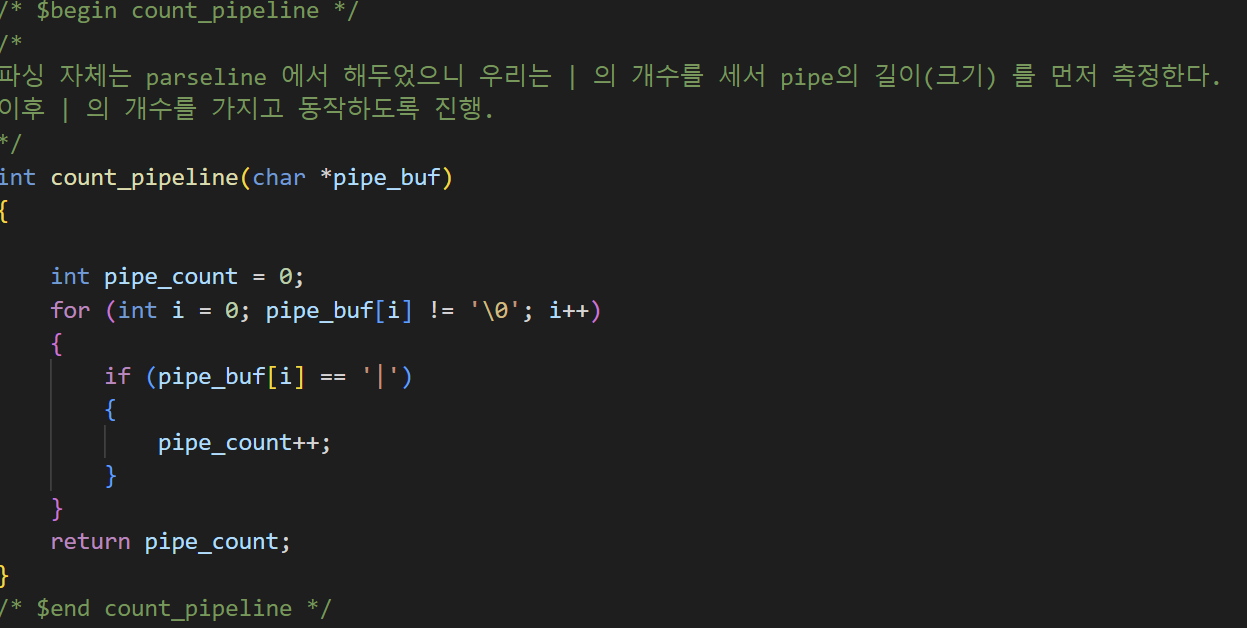
Builtin command 인지 먼저 확인합니다. 만약 builtin\_command 인 경우에는 fork 를 할 필요 없이 현재 프로세스에서 처리해 주는 경우 이기 때문입니다. Phase 1 의 경우는 none\_pipe\_run 에 해당하는 경우만 존재합니다.

Fork 로 child process 를 생성합니다. Child process 에서 None\_pipe\_run 함수를 실행하면 파싱한 첫번째 명령어 앞에 /bin/ 을 추가한 뒤 excve 함수를 사용합니다.



Builtin\_Command 는 다음과 같이 구성했습니다. Exit 인 경우에는 exit(0) 을 호출하고. 명령어가 “cd” 인 경우 현재 프로세스에서 파일을 변경해야 합니다. Linux shell 의 처리와 최대한 비슷하게 구성하기 위해서 적절한 예외처리들을 통해서 (cd , cd ~, cd $HOME 🡪 home directory 로 바로 이동) cd 관련된 명령어들을 처리 하였습니다.

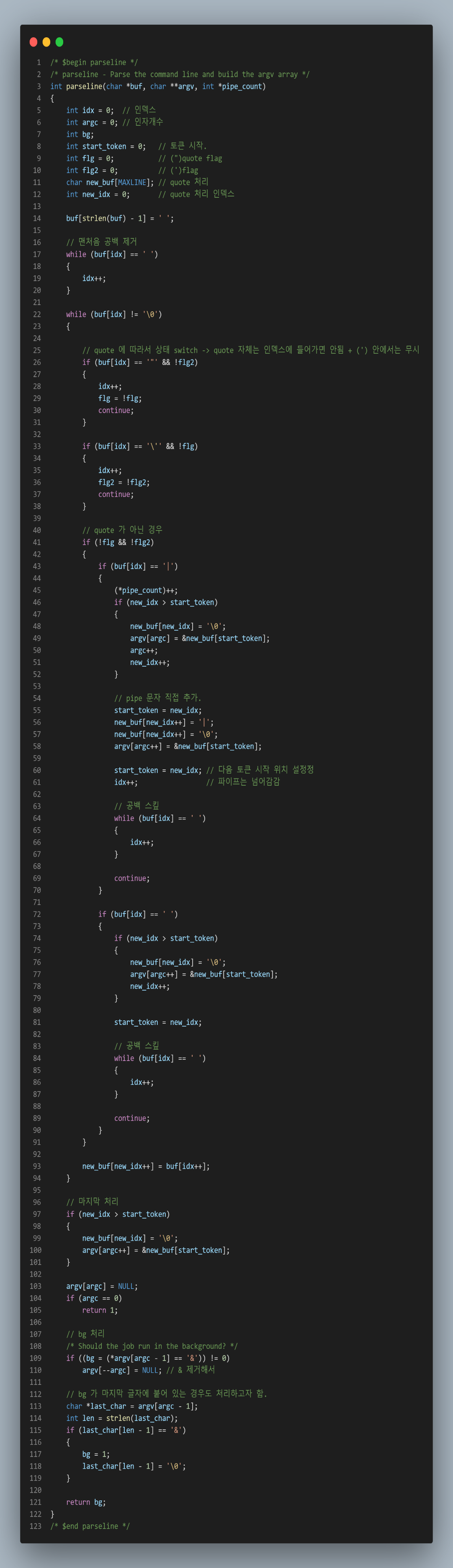
Phase<2>

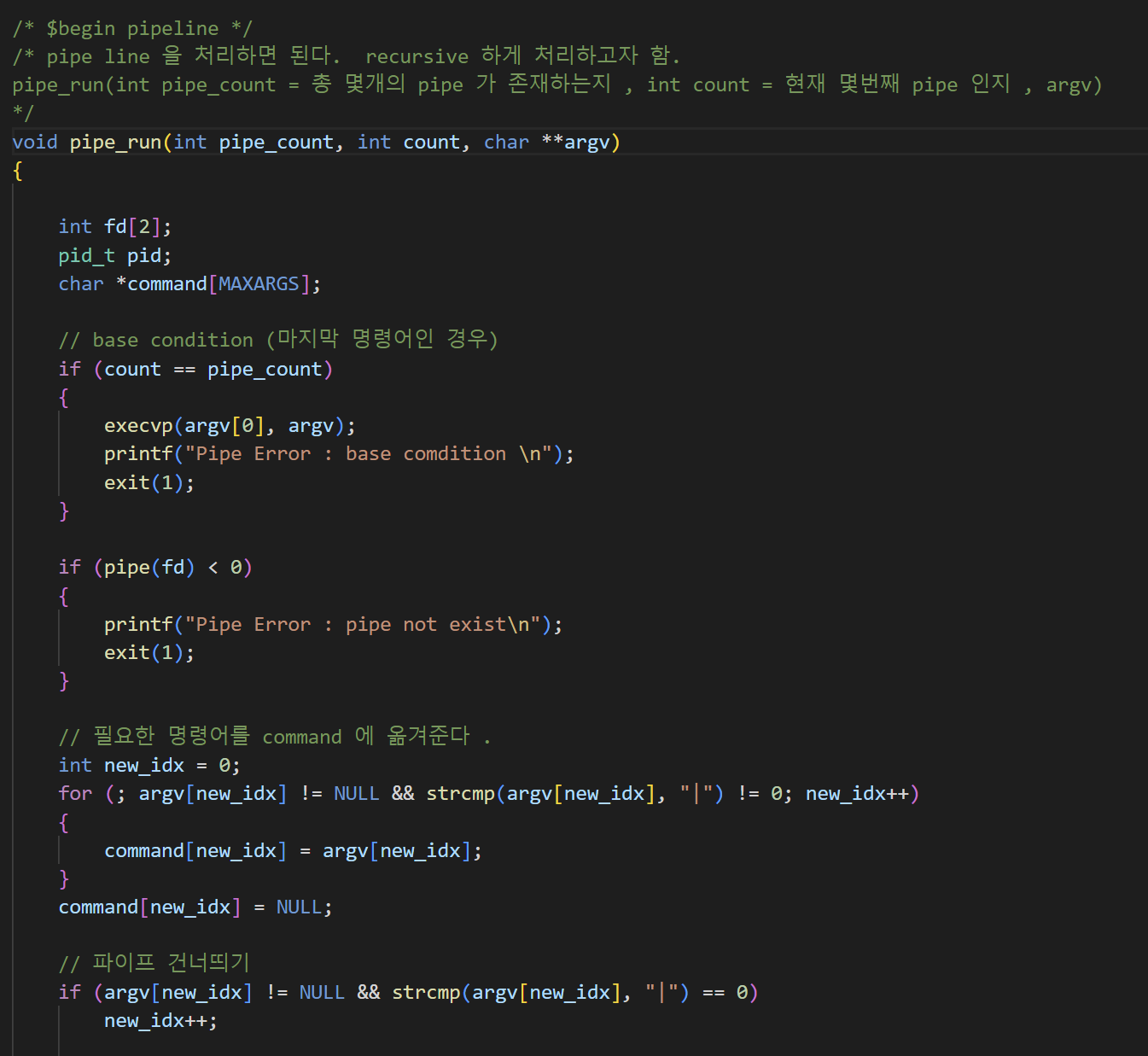


처음 구상을 했을 때 위와 같이 pipeline 의 개수를 먼저 count 해야 한다고 생각했지만

이런 경우에 echo “|” 같은 경우를 처리하지 못할 것이라는 생각이 들었습니다. 따라서 pipe 의 개수역시 parseline 함수를 호출 하면서 동시에 count 해야 한다는 생각이 보고서 쓰는 중간에 들어서 다시 수정했습니다. 다음장부터 parseline 함수 서술하겠습니다.

parseline 함수입니다. 이 함수는 커멘드 입력을 | (pipe) 단위로 나눠서 파싱할 뿐만 아니라 “(quoates) or ‘ 와 같은 따옴표도 Linux shell 환경과 동일하게 처리할 수 있도록 구성했습니다. 추가로 phase 3 에서 필요한 background (&) 표기도 잘 처리할 수 있도록 구성했습니다. 추가로 pipe 의 개수도 count 할 수 있도록 구성해 이후에 pipe\_run 에서 사용할 수 있도록 하였습니다. 먼저 커맨드에서 앞쪽 공백을 제거합니다. Flg 와 flg2 는 “ 와 ‘ (quotes) 가 열려있는 경우 ( ex “ asdf ~ ) 인 경우를 체크할 수 있습니다. 이제 커멘드를 한 글자씩 돌면서 pipe 와 quotes 를 처리할 수 있습니다. 조금 더 자세히 보면, “ 혹은 ‘ 가 시작되는 부분에서는 인덱스를 하나 더해주고(“ 와 ‘ 는 문자열 안에 들어가지 않기 때문 ) 현재 상태를 변화시켜줍니다 (!flg , !flg1). 만약 따옴표가 열린 상태가 아닌 경우부터 서술하자면, pipe ( | ) 에 해당하는 인덱스라면 pipe\_count를 하나 올려주고 새로 만들어주는 토큰이 진행중이라면 해당 토큰을 종료후 argv 배열에 추가합니다. 그 후에 pipe (|) 를 하나의 토큰으로 처리해 argv 에 넣어 준 뒤 시작 토큰 위치를 갱신해줍니다. 이후에 다시 공백이 나오는 경우는 건너띄어줍니다. 또, quotes 가 아니 아닌 경우에는 현재 문자열이 공백이라면 앞과 유사가하게 new\_buf 에 생긴 문자열을 하나의 토큰으로 argv 에 추가하고 인덱스를 이동시켜 준 뒤 공백을 제거합니다. 만약에 quote 안에 있는 경우라면 어떤 문자이든지간에 상관없이 추가시켜주어야 하기 떄문에 new\_buf[new\_idx] 에 buf[idx] 를 복사해줍니다.

위 루프 종료후에 남아있는 new\_buf 토큰을 argv 배열에 추가한뒤 마지막 배열을 NULL 로 설정해줍니다. 이제 bg 처리를 하면 파싱이 마무리 되는데 기본적으로 shellex 에서 제공한 bg 처리 이외에도 저희는 phase3 에서 sleep 5& 와 같이 & 기회가 문자열과 붙어있는 경우를 추가로 처리해주어야 했습니다. 그래서 마지막 argv 문자열의 마지막 문자가 & 인 경우에는 bg 로 돌아가도록 체크하였고 &문자가 있다면 삭제해주었습니다.

이제 파싱을 마치고 pipe\_run 을 실행합니다. 파이프를 생성한 다음에 이전 명령어 수행 결과가 파이프를 통해 다음 명령어의 입력으로 전달될 수 있도록 파일 디스크럽터인 fd 를 선언하고 pipe(fd) 로 pipeline 을 만들었습니다.

일단 재귀에서 자주 사용하듯이 base condition 을 설정해 주었습니다. 현재 호출횟수(count) 가 pipe\_count(파이프 개수) 가 같아지면 즉, 파이프라인의 마지막 명령어라면 이제 더이상 재귀호출을 하지 않도록합니다.

그 다음으로는 command 를 분리합니다. | 가 나오기 이전은 이번에 fork 할 child process 가 사용할 명령어이고, 나머지 | 이후 오른쪽 부분은 이후에 다시 재귀적으로 사용할 수 있어야 합니다. Child 가 사용할 명령어는 command 배열에 저장해줍니다. 예를들어서 cat test.txt | grep “hi” 같은 경우를 보면 cat test.txt -> grep “hi” 이런식으로 진행이 되는데 | 자체는 필요가 없기 때문에 인덱스를 한칸 띄어줍니다. 이후에 사용할 커맨드를 다시 tmp 에 저장합니다. ( 첫 | 이후 오른쪽 argv )

이후에 child process 에서 read pipe (fd[0]) 은 닫고 , fd[1] 을 출력(stdout) 으로 복제 합니다. 이렇게 구성하면 child process 에서 출력이 파이프를 통해서 다음 재귀 함수의 child 의 입력으로 들어갈 수 있도록 하였습니다. 만약, fork 된 child process 가 아닌경우는 위와 반대로 writ pipe(fd[1] 을 닫고 waitpid 를 통해 child process 가 종료된 다음 reaping 한 후에 재귀적으로 pipe\_run 을 실행합니다. 첫 | 이후 명령어들을 모아둔 tmp 를 다음 pipe\_run 으로 보냅니다. 이렇게 재귀적으로 구성할 경우 pipe 의 개수에 구애받지 않고 실행할 수 있습니다.

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.Phase<3>

일단 먼저 수업시간에 배운 safe signal handling 을 최대한 적용하고자 하였습니다. 따라서 fork 를 하기 이전에 SIGCHILD 를 block 함으로써 race 를 방지하였습니다. 이후에 child process 에서는 다시 이전 prev 로 복구해두었습니다. Setpgid 함수를

통해서 프로세스를 그룹으로 관리 함으로써 시그널 전달을 그룹단위로 할 수 있도록 하였습니다. 이후 다시 parent 에서는 SIGCHLD 를 차단한걸 해제합니다. 만약 background 가 아니라면 waitpid 를 통해서 forground 를 reaping 하고 background 인 경우는 add\_job ( jobs 리스트에 등록) 하고 넘어갑니다. 추가적으로 fgpid 를 설정했는데 이는 SIGINT 나 SIGSTP 같은 경우는 forground 프로세스에만 신호를 보내야 하는데 이를 식별하기 위해서 설정하였습니다. 따라서 !bg 인 경우에 현재 forground pid 를 fgpid 로 선언하였습니다. 이후 sigint 나 singstp handler 에서는 이 fgpid 를 사용해서 foreground 인 경우에만 시그널이 작동할 수 있도록 하였습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Job 구조체 입니다. MAX\_JOBS 는 10으로 설정했습니다. background 의 pid , job\_id , status ( 실행중이면 1 아니면 0 )그리고 명령어를 저장할 job\_cmd로 구성하였습니다.

이 구조체들을 리스트로 설정한 jobs 를 통해서 이후 명령어들에 알맞은 처리를 할 수 있도록 하였습니다.

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

ㄷ

다음으로는 add\_job 과 delete\_job 입니다. Add\_job 은 background command 를 jobs 에 추가할 수 있도록 구성했습니다. 실행했을때 overwriting 이 발생했어서 이걸 해결해주기위해 CMD 같은 경우는 사용하기 이전에는 다시 초기화 해두었습니다.

Delete\_job 은 pid 에 해당하는 background process 를 삭제해주는 함수입니다.

텍스트, 스크린샷, 디스플레이, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

다음으로는 sigchld\_handler 입니다. 핸들러 수행동안에 다른 시그널을 모두 막은 뒤 delete\_job 을 통해서 background 인경우에 jobs 배열에서 제거를 해준 뒤 다시 signal 을 prev 로 복구시켜둡니다. 제 생각에는 이렇게 구성한다면 safe signal handling 될 수 있다고 생각해서 이런식으로 구성했습니다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

다음으로는 sigint\_handler 와 sigstp\_handler 입니다. sigint\_handler 는 SIGINT 가 발생했을때 foreground 그룹에 시그널을 보내도록 합니다. Fgpid 가 0이 아닌 프로세스만 sigint 를 보내기 때문에 background 에는 영향을 미치지 않을 것으로 예상됩니다. 또한 앞과 마찬가지로 시그널을 보내는 도중에는 모든 신호를 차단함으로써 safe signal handling 까지 해주었습니다. Sigstp 역시 비슷하게 구성했으므로 자세한 설명은 생략하겠습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Built in command 에서 jobs 를 처리하는데 이때 jobs 리스트에 가지고 있던 내용들을 모두 출력합니다.

텍스트, 스크린샷, 디스플레이, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Kill 같은 부분은 저는 프로세스가 max 10 개라고 가정하고 구성하였기 때문에 앞에 %를 뺀 숫자를 job\_cmd라고 설정 해 두었고 커멘드에 해당하는 부분에 SIGKILL 을 해 주어서 background 를 저희가 원하는대로 kill 해주었습니다.

텍스트, 스크린샷, 디스플레이이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 디스플레이, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

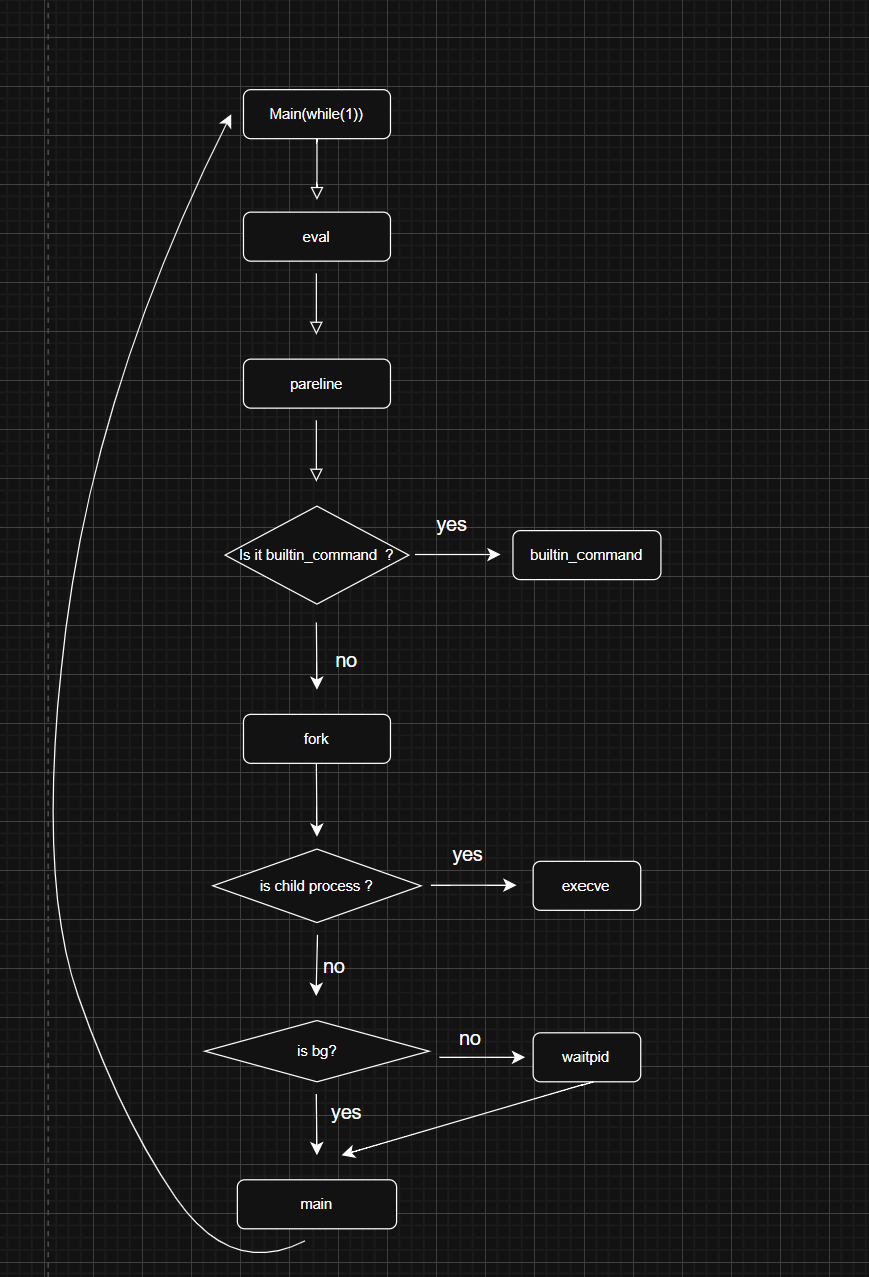
Bg 인 경우에는 kill 함수와 유사하게 전처리를 해주고 난 뒤 정지상태인 경우 sigcont 를 사용해서 stopped 된 job 을 다시 running 해 주면 될 것 같아서 저렇게 구성해 주었습니다.

아쉽게도 fg 같은 경우는 제가 임의로 일단은 SIGCONT 을 보내고 난 뒤에 status 를 실행하지 않는 함수로 바꾸었습니다. 백그라운드 프로세스를 foreground 로 바꾸는 방법이 잘 생각나지 않아서 일단 임의로 이렇게라도 구성했습니다. 백그라운드를 foreground 로 바꾸면 Waitpid 를 통해서 parent process 에서 reaping 해주어야 할 것 같기는한데, 이를 어떻게 바꿀지 여러가지 고민을 해보다 jobs 에 있는 jobs[job\_cmd] 를 재실행 시켜야 하나? 싶었지만 이는 이전의 함수의 실행과 다른 것이여서 생각을 접고 아쉽게도 구현을 하지 못하였습니다.

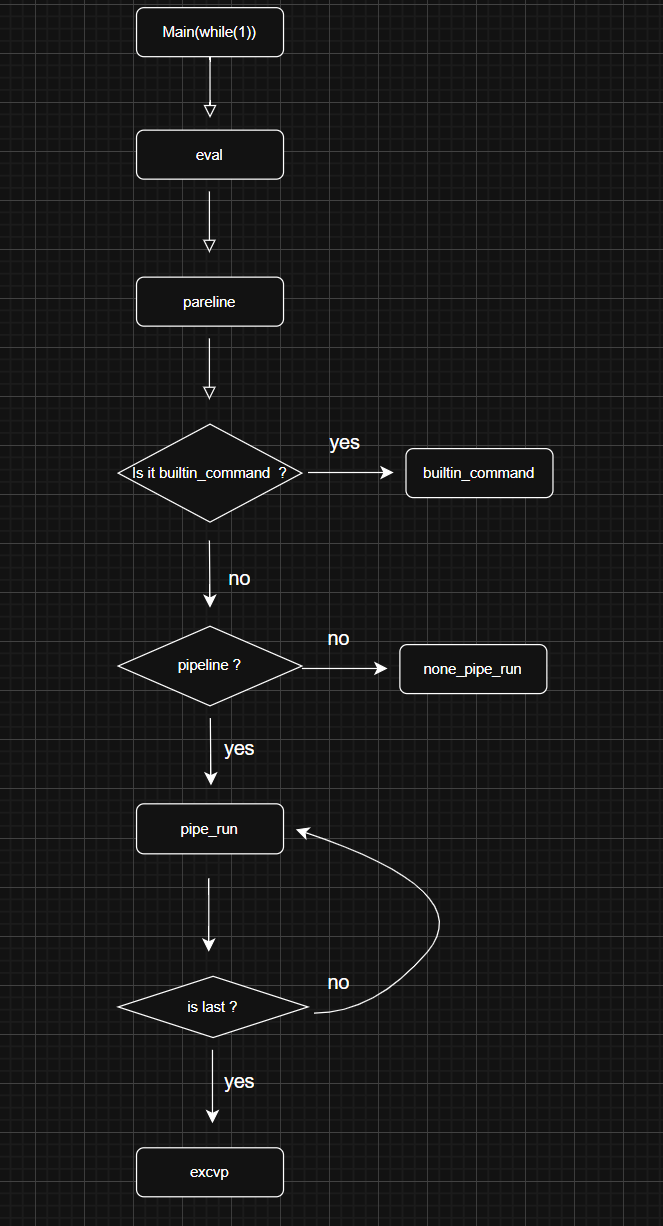
1. **구현 결과**
   1. **Flow Chart**

* **2.B.개발 내용에 대한 Flow Chart를 작성.**
* **(각각의 방법들에서 추가된 내용(fork, pipeline, background)만 특성이 잘 드러나게 그리면 됨.)**

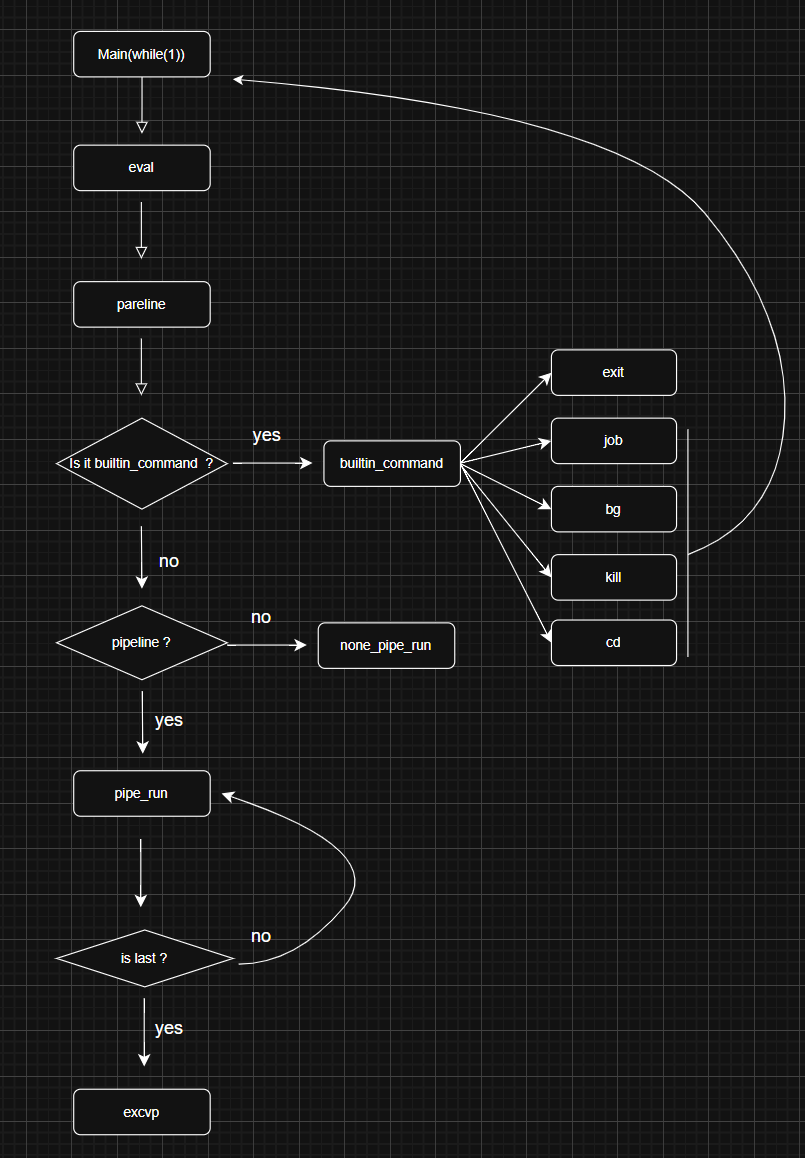
1. **Phase 1 (fork)**

****

1. **Phase 2 (pipeline)**

****

1. **Phase 3 (background)**

****