main문을 살펴보면 ret변수에 들어 있는 값이 0이 아니면 계속 while문이 유지되는 상태이다. ret 변수에 들어가는 값은 __process_command 함수의 반환 값이고 __process_command 함수의 반환 값은 run_command 함수의 반환 값인 코드이다. Exit외에 쉘이 종료되지 않도록 ret값에 0이 아닌 수를 넣을 수 있도록 했다.

1. Outline how programs are launched and arguments are passed

fork는 현재 실행중인 process를 복사해서 다른 process를 생성하는데 복사하는데 의미가 있듯이 가지고 있는 시스템 자원을 원래 process와 공유하기 때문에 코드가 실행되면 fork()를 해줘서 자식 process를 생성해 기존 시스템에서 실행 가능한 ls, pwd, cp 등의 기능이 자식 process에서 가능하게 해줬다. 또한 fork()는 자식에게 0을 반환하기 때문에 이 점을 이용하여 자식 process 안에서 입력이 잘못되어 결과값이 없어 실패하는 경우 execvp는 -1을 반환하는 점 또한 이용하여 if문을 통해 입력된 문자는 실행될 수 없다는 경고문을 보여줄 수 있게 해줬다. exec 함수군은 path나 file에 지정한 명령이나 파일을 실행하는데 그 중 execvp를 이용해 자식 프로세스에서 tokens[0]에 들어있는 명령어가 실행 가능하게 했다. 부모 프로세스는 좀비가 되는 것을 막기 위해 자식 프로세스가 끝나는 것을 기다리도록 wait()을 사용했다.

2. How the command history is maintained and replayed later

입력된 data를 저장할 수 있도록 구조체를 선언해주고 명령어와 그 명령어의 번호를 저장할 수 있도록 했다. 명령어의 번호는 함수안에 지역 변수로 선언할 경우 계속 초기화 될 수 있기 때문에 전역 변수로 사용하여 저장했다. 이렇게 생긴 data가 저장된 struct를 list에 넣어주었고 (INIT_LIST_HEAD) 기존에 코드에 선언 되어있던 history list에 이 list를 추가할 수 있도록 하였다 (list_add_tail). list_add가 아닌 list_add_tail을 사용하여 연결리스트의 마지막이 아니라 head부분 전에 연결할 수 있도록 하였다. 이렇게 history 리스트를 유지하였다.

저장된 history를 <! Num>으로 replay 시키기 위해서 나는 일단 history의 data를 순방향으로 순회(list_for_each)하여 Num에 맞는 번호에 해당하는 명령어를 가지고 오도록 하였다. 여기서 history라는 list의 주소에 접근하기 위해서, 엔트리 접근 방식으로 container_of()를 사용하였는데 pos라는 포인터만 가진 구조체를 선언하여 실제 메모리를 가리키고 있는 것에서 history 구조체의 주소까지의 오프셋을 구하여 실제 주소를 도출해서 member에 접근 가능하게 해줘 저장된 번호 데이터와 들어온 Num을 비교할 수 있게 하였다. 이렇게 비교했을 때 같은 경우 해당 entry의 command를 가져와 command를 실행시키기 위해서 __parse_command에 직접 넣어 실행할 수 있도록 하였다. (처음에는 run_command를 생각했지만 run_command 안에서 run_command 호출보다 run_command를 따로 포함하고 있는 __parse_command에 넣었다.)

3. Your strategy to implement the pipe

tokens[]에 "|"가 있는지 확인하도록 먼저 구현했다. 그리고 후에 "|" 기준으로 명령을 구분을 할 수 있도록 했다. 파이프의 수행을 위해서 pipe 함수의 입출력 모두의 파일 디스크립터인 fd[2]를 넣어서 각각의 모드를 실행하도록 했다. Fork는 두 번 수행하여 프로세서로 데이터를 전달하도록 구현하였다. 처음 fork를 통해서 앞쪽 명령어의 프로세스를 생성하여 수행하게 해주고 앞쪽 파이프에서는 데이터를 쓸 것이기 때문에 읽기 위한 파이프는 닫고 fd[1]에 데이터를 쓰게 했다. 그후 execvp로 앞쪽 명령어를 수행하게 하였고 fd[1]을 닫도록 했다. 두번째 fork에서는 앞쪽 명령어에 의해 실행된 명령어의 결과 데이터를 읽어서 연산할 수 있도록 하였다. 전체적으로 자식프로세스에서 쓰고 부모 프로세스에서 읽는 방식으로 파이프를 구현했다.

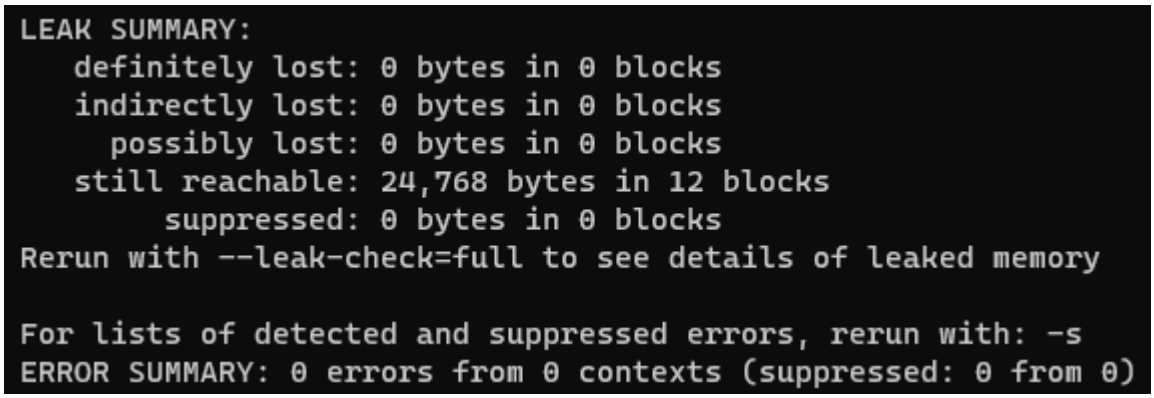
4. And lessons learned

- 처음 pipe를 | 하나로만 접근했을 때는 만약 파이프가 여러 개 있다면 어떻게 되는건지에 대한 개념이 잘 정리가 되어있지 않았는데 직접 구현을 해보고 난 뒤 알게 된 내용을 정리해 보자면 "1 | 2 | 3"이라면 1번 명령의 결과를 2에 전달해주고 2번까지 거친 결과를 3번 명령어에 전달함을 확실히 알 수 있었다.
- 이론 수업으로 들었을 때 굳이 fork과정을 거치고 exec를 하지? Exec만 사용하면 안되나? 생각했지만 fork후에 exec를 사용해야 프로세스의 위치를 기억해 부모프로세스로 돌아올 방법이 생김을 알 수 있었다.
- History를 구현하는 과정에서 history명령은 먼저 입력된 순서로 먼저 출력되는 마치 Queue와 같이 작동하는데 이때 list_add_tail을 사용하면 Queue처럼 작동시킬 때 매우 편하다는 것을 알 수 있었다. 처음에는 list_add를 사용했다가 물론 순회를 역방향을 하면 마찬가지지만 좀더 들어온 순서대로 처리해준다는 느낌으로 list의 First부분을 넘겨주면 처리가 쉬울 수 있음을 알 수 있었다.
- Debian 환경에서 주의해야 할 점으로 교수님이 전에 말씀해주신 변수가 선언되었을 때 사용이 안되면 debian에서 컴파일할 때 오류가 생기며 typedef로 구조체를 선언할 경우 구조체 별칭을 설정해줘야 오류가 안 생긴다는 점을 알 수 있었다.

```
typedef struct entry {
    struct list_head list;
    char* command;
    int index;
}entry;
```

- History를 기록하기 위해서 새로운 entry를 만들고 malloc으로 공간을 할당하는 식으로

구현하였는데 이 프로그램에 종료할 때 할당해준 공간을 처리하지 않았기에 pa0와 같이 memory leak과 같은 issue는 없을지 궁금하여 valgind ./posh로 확인해 보고 ls명령 cp명령 ls명령 rm명령 순으로 입력하고 exit를 해봤는데 leak 부분 결과가 아래 사진과 같았다.



Valgrind 상으론 leak부분이 없다고 뜨는 것 같은데 난 history append할 때 새로운 entry를 만들어 malloc으로 공간을 할당해주고 프로그램이 종료될 때 free를 따로 해주지 않았는데 왜 leak이 없는지 이해가 잘 되지 않았었는데 memory leak은 '필요하지 않은 메모리'를 계속 점유하고 있는 현상만을 check해줘서 pa0의 pop과 같이 정보가 나가 메모리가 이유 없이 생성되어 있는 경우에 처리를 해줘야 함을 알게 되었다. 이 전에는 단순히 memory leak을 없애는 것에만 집중해 내가 과제를 하면서 할당한 공간을 프로그램이 종료되면 모두 삭제해줘야 한다고 생각했던 것 같다.