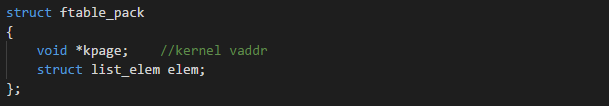
**Project 3 Final Report**

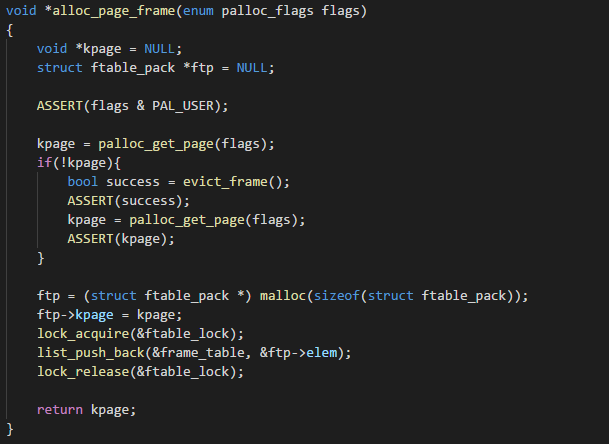
성해빈, 송창근

**1. 문제 해결 (구현)**

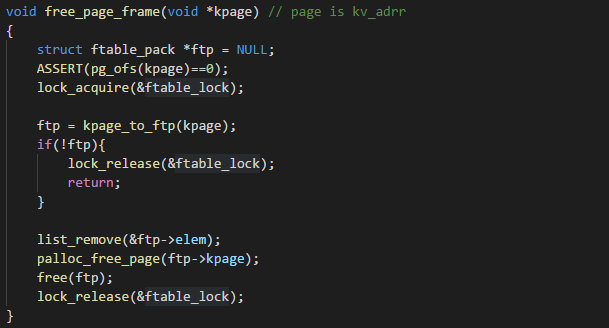
1. **Frame Table** (5 points)



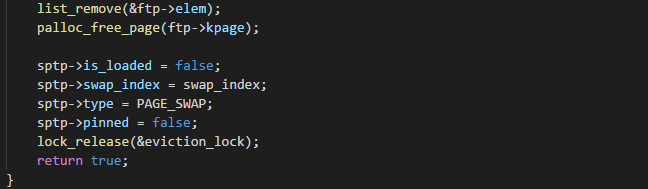
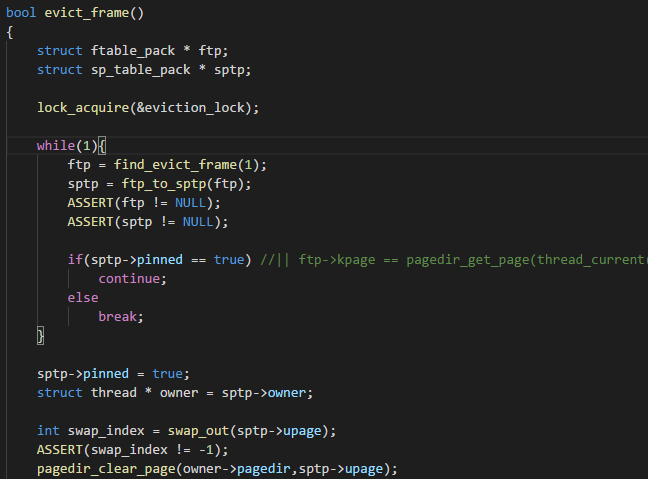
프레임 테이블을 리스트를 이용하여 구현하였다. 현재 사용하고 있는 프레임(커널 페이지)를 모아두어 관리하기 쉽도록 하였다.



프레임 테이블 원소의 추가는 위와 같이 진행된다. palloc\_get\_page를 이용해 커널 페이지를 할당받는다. 만약 페이지를 더 이상 할당할 수 없는 경우, eviction을 수행하여 빈 공간을 만들어준 뒤 다시 할당받는다.(자세한 것은 이후 설명) 이후 리스트 원소를 만들어 페이지 주소를 넣어주고 전역으로 선언된 프레임 테이블 리스트에 넣어준다. 마지막으로 할당받은 페이지 주소를 리턴한다.

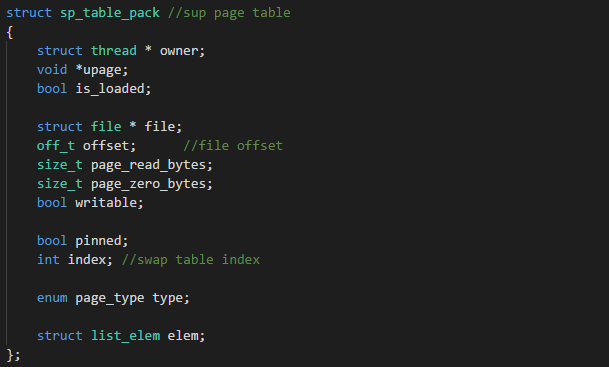


프레임 테이블 원소의 삭제는 이 역순으로 진행된다. 커널 페이지 주소를 입력으로 받아 프레임 테이블에서 해당 프레임 원소를 찾고, 테이블에서 제거한 뒤 해당 주소의 페이지를 할당 해제해준 뒤 프레임 테이블 원소까지 삭제해준다.

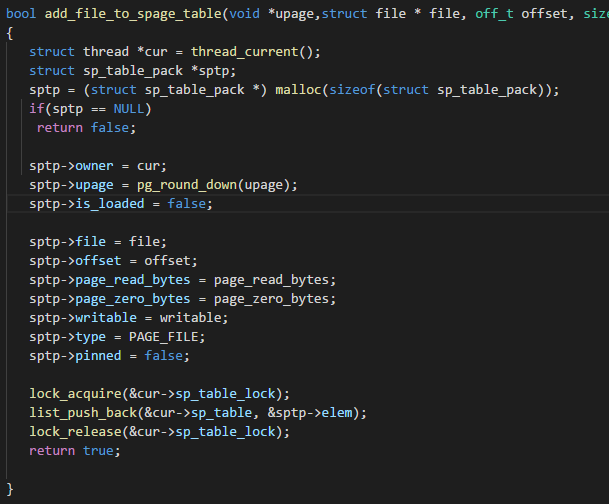


위에서 설명했다시피, 프레임을 할당받을 때, 공간이 부족하면 프레임을 적절히 선택하여 swap table에 옮기는 eviction을 진행한다. 먼저, evict 될 프레임을 선택한다. 선택하는 방법은 Clock 알고리즘을 따른다. 이 때, pinned된 프레임은 선택해서는 안 된다. 선택된 프레임을 swap\_out 함수(Swap Table에서 설명)를 통해 스왑 테이블로 옮기고, 페이지를 할당 해제하여 공간을 만들어준다. 이후 페이지 테이블의 변수들을 상황에 맞도록 조정해준다.

1. **Supplemental Page Table** (5 points)



페이지 테이블의 기능을 확장한 S-페이지 테이블을 리스트로 구현하였다. S-페이지 테이블은 가지고 있는 쓰레드, 페이지 주소, 파일 로드에 사용하는 정보들(offset, page\_read\_bytes, page\_zero\_bytes, writeable), 스왑 가능 여부, 스왑에 필요한 인덱스, 페이지가 무슨 역할을 하는지 정보를 포함한다.



S-페이지 테이블에 파일을 할당할 경우, 일단은 로드를 하지 않는다. 우선은 포인터만 이용하여 받아둔 뒤, 나중에 할당하는데, 이러한 Lazy Loading에 대한 설명은 아래 해당하는 항목에서 설명하겠다. 스왑 테이블, 파일 메모리 매핑 역시 마찬가지로 해당 항목에서 설명하도록 한다.

1. **Stack Growth** (5 points)

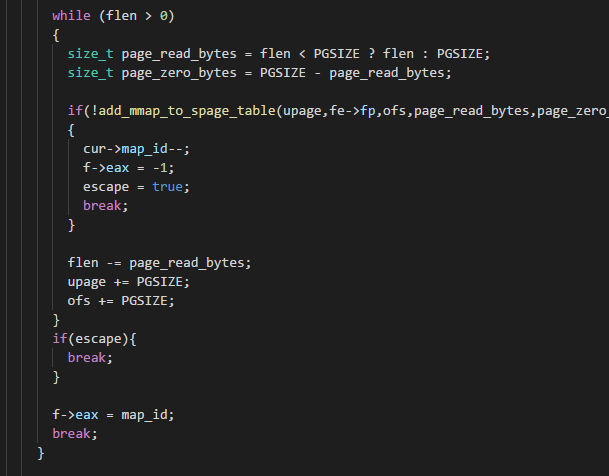
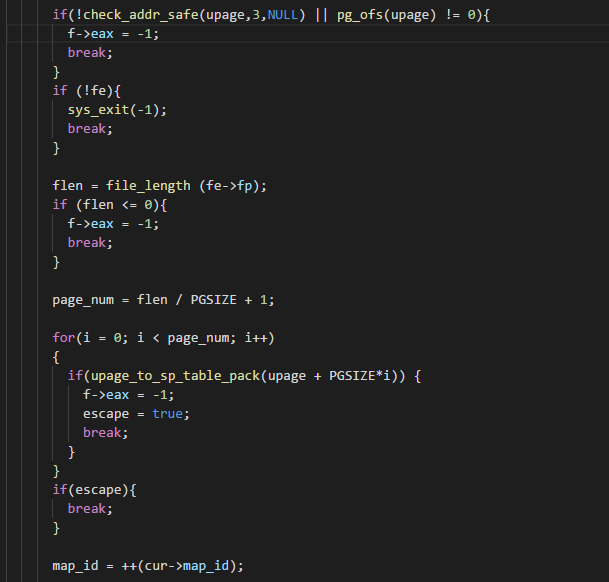


페이지를 더 할당하여 스택의 크기를 증가시키는 grow\_stack을 구현하였다. 이 함수는 페이지 폴트나 read/write 상황에서 원하는 페이지가 할당되지 않았을 때 사용된다.

1. **File Memory Mapping** (5 points)



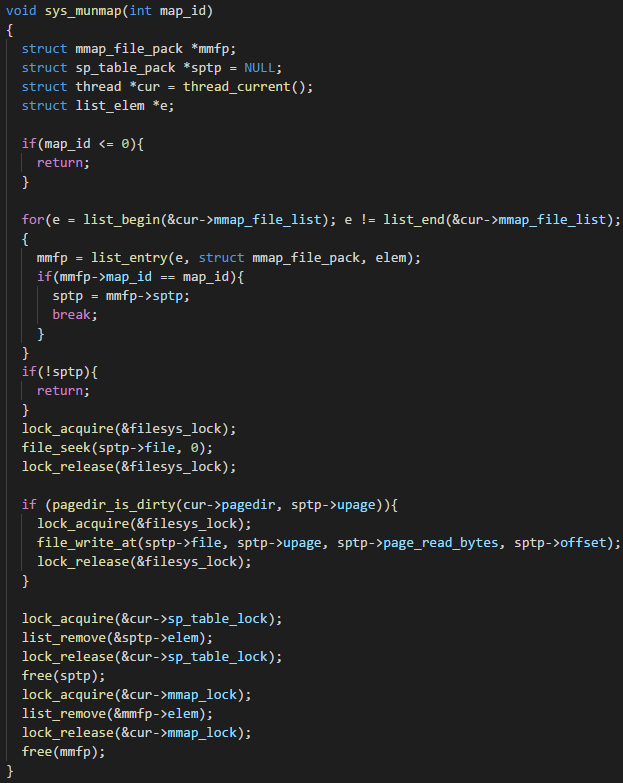
파일을 메모리에 실제로 올리기 위한 파일 매핑 테이블을 리스트로 구현하였다. 이 리스트는 mmap과 munmap 두 개의 시스템 호출에 의해 관리된다.



mmap에서는 우선 입력받은 주소를 검사한 뒤, 할당받아야 하는 페이지의 개수를 계산한다. 그 뒤 총 파일의 길이에서 페이지의 크기만큼 점차 빼 가면서 각각 페이지에 할당한다. 마지막에 남은 파일의 크기가 한 페이지의 크기보다 작다면 남는 부분은 0으로 채우고 사용하지 않는다.

mmap함수는 file이 zero byte일때, addr가 page-align이 안 돼있을 때, addr부터 할당될 page가 다른 page와 충돌을 일으킬 때, addr가 0일때, fd가 0또는 1일때 fail한다.

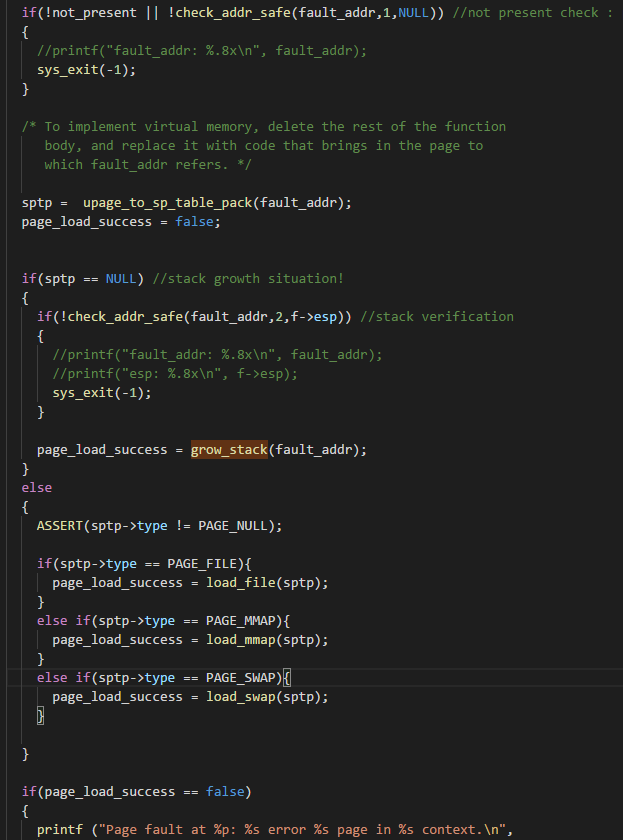
여기서 호출하는 add\_mmap\_to\_spage\_table 함수는 add\_file\_to\_spage\_table과 동일하게 우선은 load를 하지 않는다. 실제 로드는 페이지 폴트 핸들러에서 진행되며, 자세한 것은 lazy\_loading에서 설명한다.



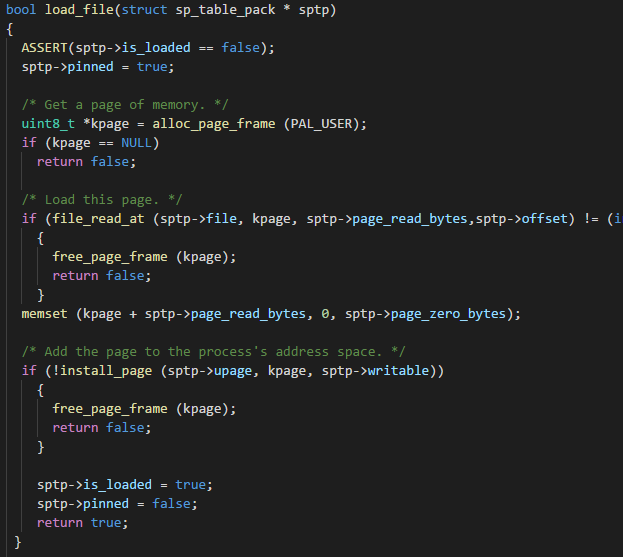
munmap에서는 순서대로 map\_id에 맞는 원소를 찾아 리스트에서 삭제하고 할당을 해제해준다. 그 과정에서 dirty를 체크하여 파일이 수정되어 있었는지 확인하여 만일 수정되었다면 파일에 수정된 사항을 적용한다.

1. **Lazy Loading** (5 points)

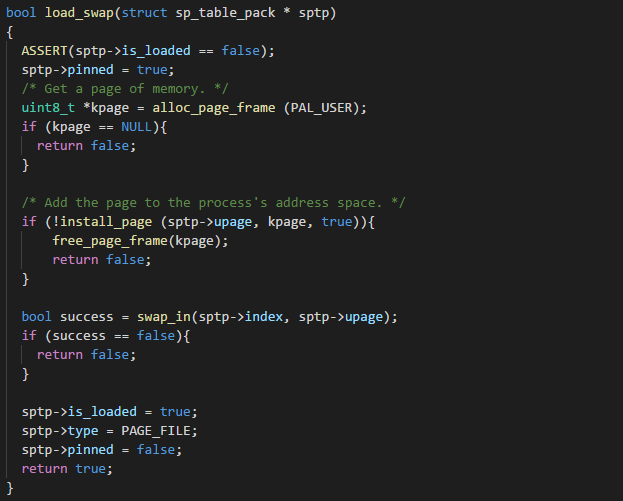
메모리에 페이지를 로드하기 위해 Lazy Loading을 구현한다.



위에서 설명한 대로, 각 함수들은 바로 메모리에 로드를 하지 않는다. 우선은 페이지만 할당해 둔 뒤, 페이지의 주소를 불러 페이지 폴트 핸들러가 호출되면 로드 과정이 실행된다. 해당 주소에 페이지가 할당되어있지 않을 경우에도 grow\_stack을 이용하여 할당하며, 이외에는 주소에서 페이지를 찾아 각 page의 타입별로 각각 로드하는 함수를 호출한다.

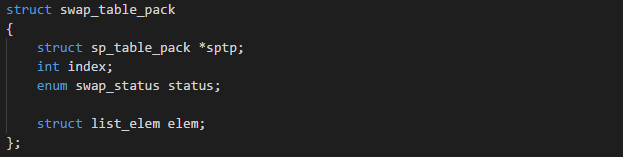


load\_file은 프레임을 할당받고 파일을 읽어 페이지에 넣는다. 그 뒤 페이지를 프로세스의 주소 영역에 넣는다. 이 과정에서 실패할 경우 false를 리턴하며 이 경우 페이지 폴트가 나게 된다. load\_mmap 역시 이와 동일하다.

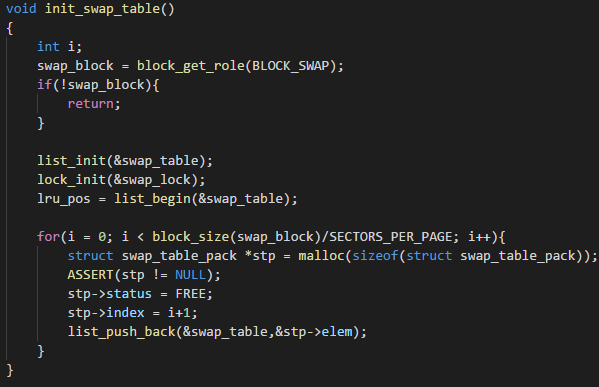


load\_swap의 경우 파일이 아니므로 파일을 복사하지는 않고, 프레임을 할당받고 페이지를 프로세스의 주소 영역에 넣은 뒤 swap\_in을 통해 필요한 페이지를 스왑한다.

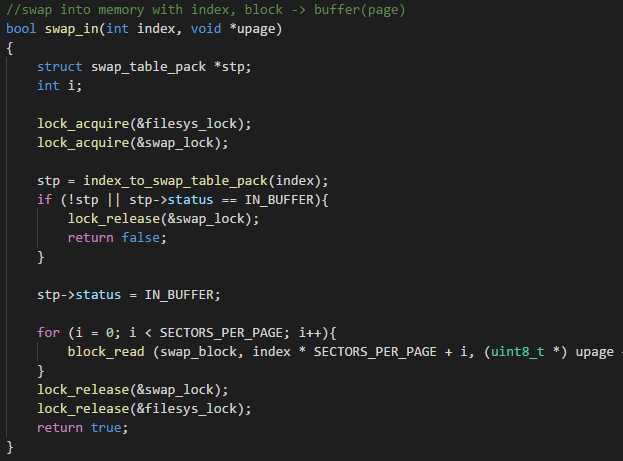
1. **Swap Table** (5 points)



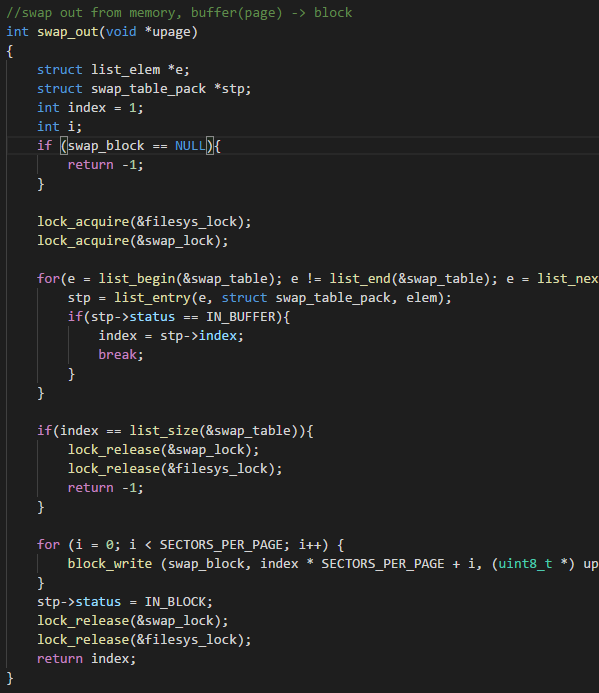
eviction 당한 페이지는 swap table에 의해 관리된다. 스왑 디스크를 만들고 스왑 디스크를 관리하기 위한 스왑 테이블을 리스트로 구현하였다. 스왑 테이블은 swap\_in과 swap\_out 두 가지 함수에 의해 관리된다. devices/block.h에 구현되어 있는 섹터 기반으로 구현하였다.



스왑 테이블의 크기는 처음에 프로그램을 실행할 때 인자로 받아 결정하는데, 그 크기만큼 스왑 테이블을 만들어준다.

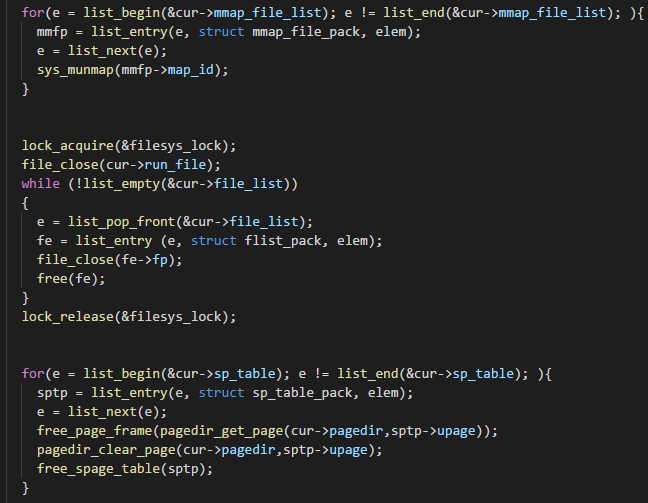


swap\_in은 스왑 디스크에 있는 페이지를 새 프레임에 할당한다. 페이지 폴트가 발생했을 때 load\_swap에서 이 함수를 호출한다. 이 함수는 페이지가 할당되어있어야 정상적으로 작동한다.



swap\_out은 페이지를 선택하여 스왑 디스크에 넣고 페이지 할당을 해제하여 여유 공간을 만든다.

1. **On Process Termination** (5 points)



process\_exit에서 프로세스가 종료될 때, 위에서 구현한 S-페이지 테이블, 프레임 테이블, 스왑 테이블의 관련 내용을 삭제하고 파일 역시 전부 닫는다. munmap 과정에서 수정된 파일들 역시 전부 변경 상황이 적용된다.

**2. Conclusion**

처음 시작할 때는 제공하는 틀도 없고 처음부터 아주 많은 것들을 구현해야 해서 아주 막막 하였으나 노력 끝에 결국 모든 테스트를 통과할 수 있었다. 페이지, 프레임, 스왑, mmap 등 많은 기능들이 얽혀 있어 구현도 어려웠지만 디버그를 하기가 매우 어려웠다. 특히 이번에는 메모리를 다루는 함수들이 많아 오류 메시지와는 전혀 다른 곳에서 문제가 발생했던 경우가 많아 더욱 고생하였다. 마지막에는 스택 영역을 테스트 코드가 침범하여 문제가 생겼는데, 오류 메세지는 process\_exit에서 문제가 생겼다고 나와 상당히 고생하였다. 기능 구현 자체는 그래도 여유가 있었으나, 안정적인 프로그램을 만드는 것의 어려움을 정말 잘 느낀 프로젝트였던 것 같다. 그래도 결국에는 이번에도 올 패스를 이루어내었다.

매뉴얼 권장사항대로 해시나 비트맵을 사용하지 않고 전부 리스트로 구현했고, LRU 알고리즘 역시 사용하지 않았는데, 시간적인 여유가 더 있었다면 이것을 적절히 바꿀 생각이었으나 이번에는 가뜩이나 2주라는 짧은 시간이 주어졌는데, 지진으로 인해 더 줄어들어 아쉽게도 수정은 하지 못했다.

