**System Programming**

**HW : In-memory / On-disk**

2013210111 남세현

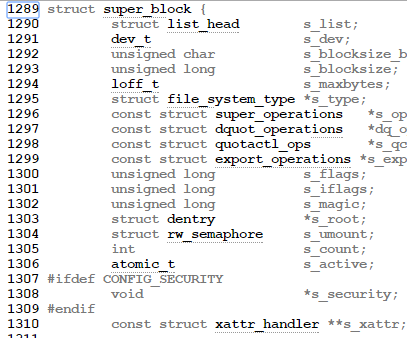
1. On-disk 자료구조가 어떻게 In-memory Object로 관리되고 있는가?

On-disk 자료구조에는 다음과 같은 요소가 있다.

* Boot Block
* Super Block
* Inode
* Data Block

이 중에서 Boot Block은 제외하고, Super Block부터 In-Memory로서의 Object 정보를 찾아보자.

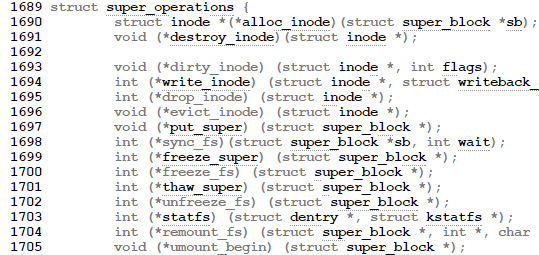
File System을 담당하고 있는 /linux/fs.h 헤더파일에 보면 super\_block에 대한 정의가 나와있다.



약 코드라인이 100줄이여서 이 보고서에 다 담기에는 부적절해, 중요한 멤버들만 간추려 보면 다음과 같다.

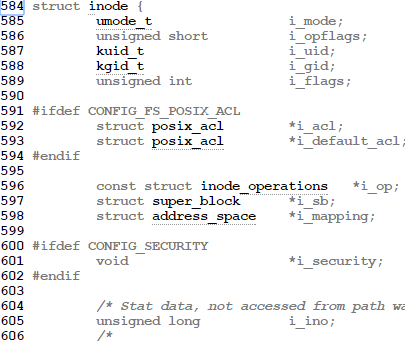
* struct list\_head s\_inodes
* const struct super\_operations \*s\_op

다른 멤버들은 특별한 것이 없었다. Super Block에서 가장 중요한 Inode Table이 s\_inodes라는 멤버변수로 저장되고 있음을 확인할 수 있었다. 그 다음으로 가장 특별한 것은 바로 super block에 대한 operation들을 저장해 놓는 s\_op이다.



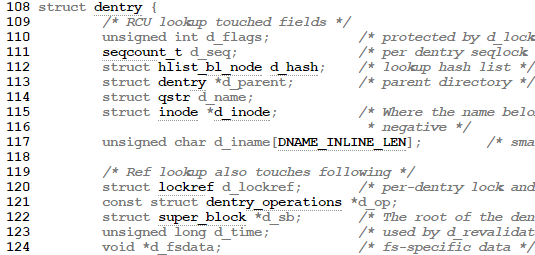
struct\_operations는 역시 linux/fs.h에 정의되어 있는데, destory\_inode, write\_inode 같은 super block이 가져야 할 operation들이 정의되어 있다. 가장 특이한 점은 이 함수들이 함수포인터로 만들어져 있다는 것인데, 그 말은 자신만의 함수로 대체를 할 수 있다는 것이다.

자신만의 File Structure를 만들었을 경우 그것에 가장 최적화된 operation을 custom으로 사용할 수 있도록 만들어 놓은 것 같다.



struct inode도 역시 /linux/fs.h에 저장되어 있다. 구분하기 위한 i\_uid, super\_operation과 비슷한 inode\_operations \*i\_op, 자신의 super\_block을 가리키는 super\_block \*i\_sb, 그리고 sturct list\_head i\_lru가 있다. Inode LRU List라고 주석이 되어있다.  
on-disk structrue를 in-memory object로 바꾸는 이유는 캐싱을 위해서인데, 그 캐싱의 기법이 LRU여서 그렇게 네이밍을 한게 아닌가 싶다. 그 다음으로는 i\_dentry가 있다. 이 내용은 다음 주제와 연결된다.

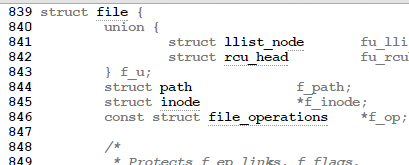
1. In-Memory에만 존재하는 object들은 어떻게 관리되고 있는가?



i\_dentry는 dentry의 리스트인데, dentry는 /linux/dcache.h에 정의되어있다. Dentry는 그 자체의 이름인 struct qstr d\_name, 소속되어 있는 inode인 \*d\_inode, 역시 operation들의 집합인 \*d\_op도 있다. Dentry의 root인 super block을 가리키는 d\_sb, 가장 최근에 수정한 시간을 저장하는 d\_time 등이 있다.

그리고 d\_lru (LRU list), d\_child, d\_subdirs가 있다. 디렉토리 엔트리 내부에 있는 파일이나 서브디렉토리에 대한 항목을 저장하고 있는 것으로 파악된다.

하지만 dentry는 위에 나열한 On-disk 자료구조 4개에는 포함되지 않는다. 그와 비슷한 only in-memory object에는 file이 있다.

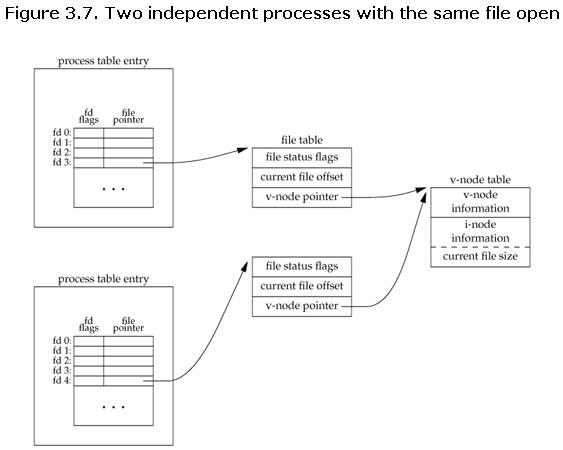


File은 /linux/fs.h에 정의되어 있다. 캐싱해놓은 inode 포인터 \*f\_inode와 파일의 경로 f\_path, operation를 정의해놓은 \*f\_op가 있다. 추가적인 변수로는 f\_owner 등이 있다. 우리가 ls 명령어를 통해 얻을 수 있는 정보들이 저장되어 있는 듯 하다.

File에서 가장 반가운 멤버는 바로 f\_pos다. 우리가 흔히 말하는 파일 포인터의 역할을 하는 것으로 보인다. 역시 File의 concurrency를 위해 mutex로 f\_pos\_lock을 만들어 둠을 확인할 수 있다. 리눅스 개발자들은 바보가 아니었다.

추가적으로, file\_operations에는 커널에 기본적으로 정의되어 있는 함수를 사용할 것인지, 아니면 VFS로서 custom된 함수를 사용할 것인지 정의가 다 되어있다.

1. Open된 파일 디스크립터는 어떻게 관리되고 있는가?



1학기때 진행된 운영체제 수업때 언급한 것 처럼, Proc/PID/FD 형태로, 즉 Process당 FD Table이 따로 존재한다. 하지만 그 FD는 바로 파일에 일대일 이 아니라 정확하게는 다대일 대응이 된다.

우리가 파일을 열거나 만들면 위 1, 2에서 언급한 내용들이 메모리에 캐싱되게 된다. 그럼 그 캐싱된 테이블에 대해 Proc/PID/FD에 포인터가 저장되고, 유저 입장에서는 FD로만 그 지점까지 접근이 가능한 것이다.

File의 count 멤버변수를 이용하면 스마트 포인터처럼 현재 reference count를 체크하여 캐싱을 계속 하고 있어야만 하는지를 결정할 수도 있다는 이야기다.

Inode에 대해서는 어차피 File이 inode에 대한 포인터를 가지고 있으므로 크게 문제는 없지만, 확실한 것은 Super Block 단위로 테이블이 만들어져 있으며 새로운 mount에 대해서도 Super Block 1개로써 관리된다고 명시되어 있다.

세번째 주제에 대해서 실습시간에도 정확하게 교수님이 말씀하신 주제가 무엇인지 혼란이 있었는데, 당시 교수님이 숙제를 내기 전에 한 학생이 “FD는 프로세스별로 관리되는지 커널별로 관리되는지”를 질문하였으므로, 그것에 대한 주제라 생각하여 위와 같이 작성하였다.