**시스템 프로그래밍**

**1차 과제**

**Log Structured File System & Ext4   
File System Profiling**

**group 15**

**컴퓨터학과 2013210107 김정수**

**컴퓨터학과 2014210075 이한얼**

**제출 : 2016.11.07  
사용 freeday – 5일**

**각자 맡은 부분 :**  
김정수 – 전체 이론 , 커널 및 LKM myqueue  
이한얼 – 전체 이론 , 커널 및 LKM myproc  
  
이론이 중요한 부분이라 서로 토의하며 진행함.

**환경**

**가상머신 : Oracle VM VirtualBox  
운영체제 : Ubuntu 16.04 LTS   
커널 : linux-4.4.1  
파일시스템 : Ext4, Nilfs2**

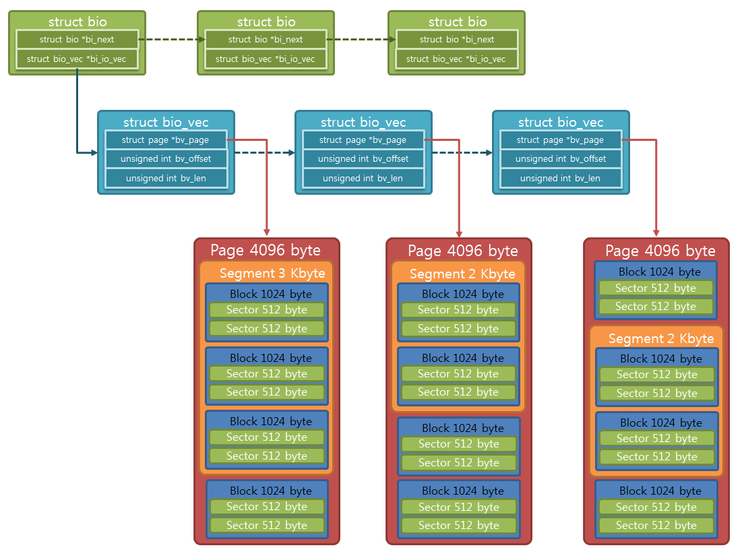
**배경지식**

이번 과제는 Log Structured 파일 시스템과 기존 ext4 파일 시스템 사이의 실제 디스크에 데이터가 쓰이는 방식을 알아보기 위한 과제이다.

기본적으로 sector는 실제 물리적인 디스크의 단위이고 block은 그에 대응하는 linux kernel이 논리적으로 접근하는 단위이다. 보통 block size = 8 \* sector로 이루어진다. 필자의 경우 실제 다음과 같은 명령어를 통하여 sector size = 512 bytes인 것과 block size가 4090 bytes인 것을 알 수 있었다.  
sudo fdisk –l  
C:\Users\김정수\Downloads\sector_size.PNG  
sudo tune2fs –l /dev/sda1 | grep –I ‘block size’  
C:\Users\김정수\Downloads\block_size.PNG

VFS에서 하나의 버퍼는 하나의 블록으로 생각을 한다. 리눅스 버전에서는 buffer\_head 라는 구조체를 통해 디스크 버퍼를 관리하였지만 좀 더 효율적인 관리를 위해 특정 버전 이후로 보다 큰 bio 구조체를 중심으로 디스크 버퍼를 관리하게 된다. 실제 이 bio라는 구조체에 대한 정보는 /linux/blk\_types.h 헤더에 위치해 있다.

Bio에 대한 전체적인 구조는 다음 그림과 같다.

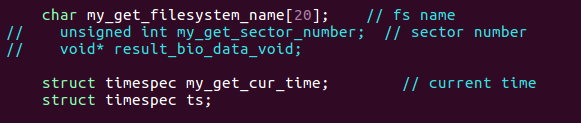
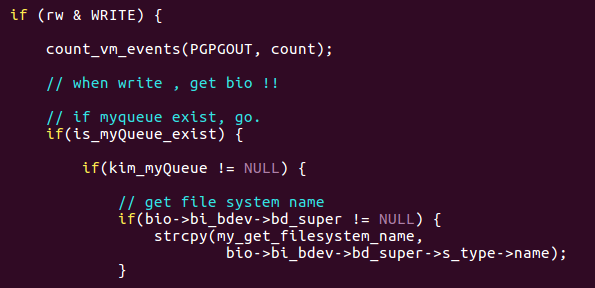
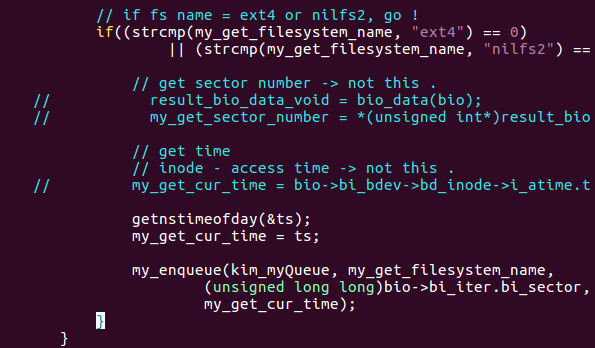


하나의 bio 구조체는 여러 개의 bio\_vec을 선형구조를 가리키게 된다. 이 bio\_vec은 하나의 page를 가리키게 된다. 우리가 접근하고자 하는 sector는 앞서 설명한 block을 이루고 그 block이 다시 page를 이루게된다. bio\_vec 안의 page number와 offset을 이용하여 원하는 sector까지 다다를 수 있다. 다시 말하면 결과적으로 하나의 bio를 통해 접근하고자 하는 sector로 접근이 가능하다.

블록 I/O의 시작 시점은 submit\_bio() 함수이다. 우리가 기본적으로 사용하는 ext4 파일 시스템의 경우 해당 bio를 이 함수로 넘겨준다. 또한, nilfs2 파일 시스템에서도 자체적인 구조를 사용하기는 하지만 결과적으로 자신의 자료구조를 bio형태로 만들어 submit\_bio() 로 넘겨주는 것을 볼 수 있다. 따라서 우리는 어떤 파일 시스템에서든 이 함수에서, 쓰여질 bio를 넘겨받을 때 해당 bio에 대한 정보(sector number) 및 넘겨 받은 시간을 원형큐 자료구조로 저장해 두었다가 후에 커널 메세지로 출력 후 시간에 따른 저장되는 sector number의 순서를 알아보려 한다.

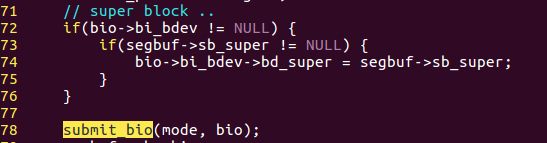
**작성한 부분에 대한 설명**

* linux/block/blk-core.c

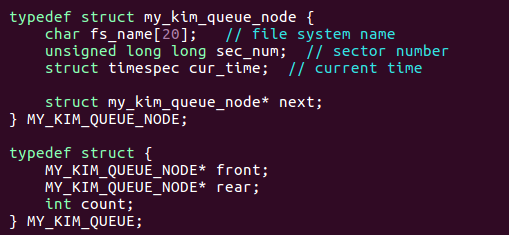
우선 원형큐 자료구조를 사용하기 위한 헤더를 추가시켜준다. myqueue.h  
C:\Users\김정수\Desktop\4-2\system_Pro\1차과제\캡쳐\blkcore1.PNG  
  
사실상 이 부분에서 extern 된 변수들은 이미 헤더에 추가되었기 때문에 pass하도록 한다.  
submit\_bio함수로 넘어가 보면,  
파일 시스템의 이름을 받아오기 위한 변수, 시간을 받아오기 위한 변수를 다음과 같이 선언하여준다.   
  
\* 원래는 bio\_data라는 함수를 통해 값을 가져오면 (void 포인터 형태) page와 offset을 계산하여 해당하는 섹터를 가리킬 것이라 생각하였지만 실질적으로 bio->bi\_iter의 bi\_sector에서 섹터 번호를 가져오기로 결정하였다. 그래서 주석 처리된 변수 부분은 사용하지 않기로 한다.  
rw는 상태에 대한 변수인데 이 변수가 WRITE 상태 값을 가질 때, 우리는 bio안의 값을 큐에 저장하기로 한다. Is\_myQueue\_exist의 경우는 큐 자료구조를 사용할 것인지에 대한 bool 변수 이기 때문에, 단순히 모듈이 켜졌는지 꺼졌는지에 따라 해당 코드가 실행될 것인지를 결정한다. 또 한, 큐가 존재할 경우에만 my\_enqueue함수를 실행하도록 한다.  
  
  
파일 시스템의 이름은 bio->bi\_bdev(디바이스 정보)->bd\_super(슈퍼블럭)->s\_type(파일시스템)->name(파일시스템 이름) 에서 받아오도록 한다.  
파일 시스템이 ext4 이거나 nilfs2일 때에만 시간을 계산하여 파일 시스템 이름과 섹터 번호 및 시간을 큐 자료구조에 추가한다.  
  
  
( 시간을 inode의 access 시간에서 가져올까 생각하여 inode의 i\_atime 변수에 접근해 보았지만 모두 0이 찍힘을 알 수 있었다. 따라서 그냥 현재 시간을 가져오는 getnstimeofday함수를 이용하였다. )

* linux/fs/nilfs2/segbuf.c

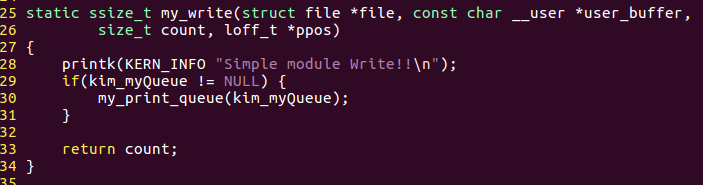
nilfs2는 자체적인 자료구조, nilfs\_segment\_buffer ( = segbuf)를 가지지만 결과적으로 bio형태로 submit\_bio로 넘겨주게 된다. 따라서 bio의 연관된 디바이스가 있는 지 확인, segbuf의 슈퍼블럭이 존재하는지 확인 후, segbuf의 슈퍼블럭 값을 bio의 슈퍼블럭 값으로 넘겨준다.



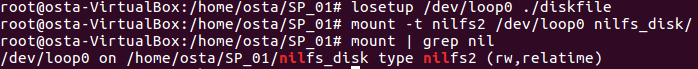
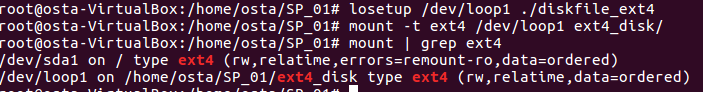
* QUEUE 폴더

원형큐에 대한 소스파일들이다. 우선 myqueue.h, myqueue.usr, myqueue.c, Makefile  
이렇게 이루어진다.  
\* myqueue.h 에는 원형큐 자료구조 및 관련 함수가 정의되어있다.   
  
우선 커널에서 만들어질 큐를 하나 미리 선언한다. 또한 is\_myQueue\_exist bool타입 변수를 통해 커널에서 모듈을 올리고 내릴 때, 오류가 나지 않도록 하였다. 보통 동적할당을 위한 malloc 대신 커널에서 쓸 수 있는 vmalloc 함수를 사용하였다.  
크게 큐를 생성하는 my\_createQueue, 자료를 추가하는 my\_enqueue, 자료를 출력 및 초기화 하는 my\_print\_queue, 그리고 큐를 삭제하는 my\_destroyQueue 함수가 있다. 이 헤더는 submit\_bio가 있는 c파일에 포함되어 처음 부팅 시 함께 실행되도록 되어있다.   
my\_enqueue의 경우, 원래 큐 형태였던 구조가 자료의 값이 5만개를 넘으면 rear를 front와 이어 비로소 큐 형태를 이루게 된다. 이 후, 꽉 찼을 경우에 대해 새로운 데이터에 대해서는 front의 데이터가 사라지며 그 자리를 rear가 차지하게 된다.  
my\_print\_queue 경우, front의 값부터 출력되는 동시에 해당 노드를 초기화하면서 rear까지 이동하는 식이다.  
이 후 미리 선언된 큐 변수나 각각의 함수의 경우 다른 클래스에서 사용하기 위해 EXPORT\_SYMBOL 선언이 되어있다.  
\* myqueue\_usr.h 는 myqueue.h 에서 EXPORT\_SYMBOL 처리 때문에 다른 클래스가 또 같은 헤더를 이용할 수 없기 때문에(중복 EXPORT\_SYMBOL 선언) 큐에 대한 구조체만 가지고 있는 헤더이다. 이는 각각의 적재가능 커널 모듈에서 사용하기 위한 헤더이다.  
\* myqueue.c 는 모듈을 키는 순간 kim\_myQueue 변수에 실제 큐를 만들어준다.  
모듈을 내릴 때에도 같은 방법으로 큐를 삭제하게 된다.

* procAPI 폴더

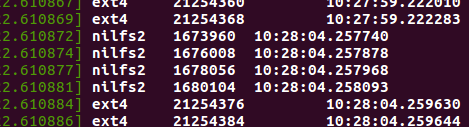
적재 가능 모듈 myproc.ko 에 대한 소스코드.  
변수사용에 에러가 나서 Makefile에 다음과 같은 부분을 추가시켰다. 해당 부분은 큐 LKM에서의 변수를 참고한다는 의미이다.  
C:\Users\김정수\Desktop\4-2\system_Pro\1차과제\캡쳐\procapi.PNG  
\* myproc.c 는 QUEUE/myqueue\_usr.h 헤더를 추가하였으며, /proc/myproc/myproc에 어떠한 값이 아무거나 쓰일 때마다 kim\_myQueue의 데이터들이 출력되는 동시에 초기화 되도록 하는 LKM이다.  


**실행 방법에 대한 간략한 설명, 실행 결과 캡쳐 화면**

Nilfs2와 ext4에 대한 공파일 생성 및 마운트  
C:\Users\김정수\Desktop\4-2\system_Pro\1차과제\캡쳐\nil1gb.PNG  
C:\Users\김정수\Desktop\4-2\system_Pro\1차과제\캡쳐\nil2set.PNG  
  
ext4 역시 같은 방식으로 /dev/loop1 에 마운트 시켰다.  


이후 insmod 를 통해 myqueue와 myproc LKM을 커널에 올려둔다.  
C:\Users\김정수\Desktop\4-2\system_Pro\1차과제\캡쳐\myqueue실행.PNG  
C:\Users\김정수\Desktop\4-2\system_Pro\1차과제\캡쳐\myproc실행.PNG  
  
/proc/myproc/myproc 에 미리 write하고 dmesg –c 를 통하여 출력버퍼를 비워둔다.  
C:\Users\김정수\Desktop\4-2\system_Pro\1차과제\캡쳐\echo_write.PNG  
  
iozone 을 nilfs2 시스템으로 마운트된 디스크에 write 한다.  
C:\Users\김정수\Desktop\4-2\system_Pro\1차과제\캡쳐\iozone_nilf.PNG  
/proc/myproc/myproc 에 write함으로 써 버퍼내용 출력 후,  
C:\Users\김정수\Desktop\4-2\system_Pro\1차과제\캡쳐\echo_write.PNG  
nilfs2 부분만 nilfs2.txt에 저장한다.  
C:\Users\김정수\Desktop\4-2\system_Pro\1차과제\캡쳐\grep_nilfstxt.PNG  
  
이후, ext4도 같은 방식으로 실행.  
C:\Users\김정수\Desktop\4-2\system_Pro\1차과제\캡쳐\iozone_ext4.PNG  
C:\Users\김정수\Desktop\4-2\system_Pro\1차과제\캡쳐\echo_write.PNG  
C:\Users\김정수\Desktop\4-2\system_Pro\1차과제\캡쳐\ext4writetxt.PNG

**결과 및 그래프**

x축은 시간이며(같은 시간이 아닌 순서), y축은 sector number이다.  
예상한 대로, nilfs2는 쌓이는 방식이 순서대로 차곡차곡 쌓이는 반면, ext4의 경우 섹터의 이동이 심함을 알 수 있었다. 하지만, 다만 아쉬운 점은 기본적으로 ext4가 파일 시스템이기 때문에 nilfs2를 실행하여도 다음과 같이 계속해서 ext4가 지속적으로 관여함을 알 수 있다.  
  
따라서, 그 이유인지 실제 받아온 nilfs2값은 ext4에 비해 턱없이 부족했다. 다음에는 기회가 된다면 따로 큐를 만들어 따로 저장해 두면 더욱 좋을 것 같다는 생각이 든다.

**과제 수행 시 어려웠던 부분과 해결 방법**  
기본적으로 커널을 다루기 때문에 너무 양이 방대했다. 더군다나 어떤 변수가 정확히 어떤 의미인지 알 수가 없어 데이터처리도 힘들었다. 또한, 정확히 오류가 나는 위치를 알 수 없었다. (addr2line을 이용하려 했지만, 막상 사용해보니 정확한 위치는 ??:? 식으로 나오지가 않았다.) 다만, 차근히 포인터에 대해서 하나하나 exception처리를 해가면 오류를 없앨 수 있었던 것 같다.