**Pintos Project 5: Filesystem**

담당 교수 : 김영재 교수님

조 / 조원 : 박성우 20171637

개발 기간 : 11.30~12.22

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

- 현재는 파일이 disk 에 저장될 때 consecutive 하게 저장되어서 공간이 있음에도 불구하고 용량이 부족하는 상황인 external fragmentation 이 발생한다. 이런 filesystem 을 indexing 형태로 바꾸어서 disk 를 효율적으로 활용할 수 있게 하고 더욱이 파일의 크기를 바꿀 수 있게 한다.

- 현재의 파일시스템에서는 directory 는 오로지 root directory 한 개 뿐이고 sub directory를 만들 수 없다. Subdirectory 를 만들 수 있게 하고 directory 에 관련된 추가적인 system call 을 구현한다.

- cpu 입장에서는 disk 까지 읽쓰 하는게 너무 느리므로 on-memory 에 buffer cache 를 만들어 수행속도를 빠르게 만들어준다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Extensible file & file growth

Filesystem 을 indexing 형태로 만들어서 disk 의 활용도를 높이고 파일의 크기를 변경할 수 있게 한다.

1. Subdirectory

Directory를 만들 수 있도록 하고 이에 따른 추가적인 system call 을 구현하여 directory 관련 일을 처리할 수 있도록 한다.

1. Buffer cache

Cpu가 disk 에 있는 파일을 찾을 때 disk 까지 갈 필요 없이 on memory 에 있는 buffer cache 를 참조하여 disk 를 참조 하는 것 보다 수행속도를 높일 수 있다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Extensible file & file growth
  + Index structure와 management에 대해서 기술

Pintos의 disk structure 를 보면 inode 하나가 disk block(sector) 하나를 전부 차지한다. 기존의 inode\_disk 를 보면 consecutive 하게 파일이 저장되므로 시작 sector 주소와 파일의 길이, magic number 가 있고 나머지는 sector 하나의 크기인 512 byte 를 맞추기 위한 사용하지 않는 unused 배열이 있다. 효율적으로 disk 를 사용하기 위해 inode disk 구조를 indexing 방법으로 변경한다. 파일의 길이, magic number, 그리고 data sector 를 바로 가리키고 있는 direct\_block[] 배열, 그리고 indirect sector 를 가리키고 있는 변수 하나, 그리고 doubly indirect sector 를 가리키고 있는 변수 하나로 구성한다. 512 byte 를 맞추기 위해 direct\_block 배열은 123개의 entry 를 가지도록 한다. 이렇게 되면 file 이 쪼개져서 disk 의 다양한 곳에 나누어져 저장되게 된다. 이에 따라 한 file 의 원하는 위치를 탐색하려면 direct block 과 indirect, doubly indirect sector 를 따라 들어가면서 탐색을 한 다음 읽고 쓰고 해야 한다. 그리고 이제 한 inode 가 가리키고 있는 block 의 최대 개수는 (direct\_block\_entries + indirect\_block\_entries + indirect\_block\_entries^2) 이므로 이 것 x 512 byte 만큼 파일을 확장할 수 있으나 pintos 의 disk 크기는 8mb 이므로 8mb 근접하게 파일을 확장할 수 있다.

* Subdirectory
  + Directory entry 관리 방법

Subdirectory 를 구현하면 directory를 만들 수 있게 된다. Directory 도 하나의 inode\_disk 이므로 이 inode\_disk 가 가리키고 있는 block(sector) 들은 dir\_entry 들로 채워지게 된다. 이 dir\_entry 구조체 안에는 file 의 이름과 file 의 inode sector 주소, entry가 사용여부의 in\_use 변수가 들어가 있다. Directory entry 를 관리 하는 방법은 기존 방법과 크게 다르지 않다. 다만 이제 subdirectory 를 만들 수 있어서 계층구조가 되어버렸기 때문에 ‘.’ ‘..’ 을 생각해야 하고 directory 의 경로를 parsing 하는 과정이 필요하다. 그 후 각종 directory 관련 함수들을 test case 요구에 맞게 수정한다.

* Buffer cache
  + Buffer cache eviction 방식

Buffer cache 는 disk 크기보다 작기 때문에 적절히 entry 를 내쫓아야 되는데 이때 second chance 방식을 사용한다. Static 변수 clock\_hand 를 만들고 이 clock\_hand 가 buffer cache entry를 순차적으로 가리키게 된다. Buffer cache entry 를 쫓아내야 하는 상황이 오면 우선 이 clock\_hand 가 가리키고 있는 entry의 reference bit 을 확인한다. 이 reference bit 이 1 이면 0으로 만들고 한 번 봐준다 하고 넘어간다. 차례대로 buffer cache entry 를 확인해가면서 reference bit이 0 인 entry 를 찾으면 이 entry가 evict 대상이 된다. 이 때 만약 이 entry 의 dirty bit 가 1이면, 즉 뭔가 수정이 되었으면 flush 함수를 통해 disk 에 기록을 한 다음에 evict 를 해야한다.

* + Buffer cache flush 방식

Buffer cache flush 는 buffer cache 에 있는 entry 내용을 disk 에 기록하는 것이다. Entry 의 dirty bit 을 확인하고 뭔가 수정이 되었으면 block\_write 함수를 통해 disk 에 기록을 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

11월 30일: project 5 강의, project 5, pintos manual 정독

12월 1일~5일: buffer cache 구현

12월 6일~17일: 기말고사 공부 및 시험

12월 17일~21일: file growth, subdirectory 구현

12월 22일: 보고서 작성

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

**Buffer Cache**

Buffer cache entry 라는 구조체를 만든다. 여기에는 사용여부에 관한 valid bit, Evict 를 위한 reference bit, 수정 여부에 관한 dirty bit, 그리고 buffer 에 있는 내용이 disk 어디에 있는지 알기 위한 sector 주소, 그리고 buffer 가 들어가 있다. 이 buffer cache entry 를 담고 있는 배열 buffer\_cache\_entry cash[64] 를 만든다.그리고 cache 관련 함수들을 만든다.

Cache.c

Buffer\_cache\_init()

Synchronization 을 위한 lock\_init 을 하고 cache 배열의 valid bit 을 처음이니 다 사용안하고 있다는 뜻으로 false 로 만든다.

Buffer\_cache\_terminate()

Filesystem 이 종료될 때 cache 배열에 남아 있는 데이터 중에 만약 수정된 데이터가 있으면 disk 에 기록해준다.

Buffer\_cache\_read()&write()

Evict 함수를 통해 Buffer cache entry 를 하나 잡고 주어진 sector 에 해당하는 buffer를 읽거나 buffer 에 쓴다.

Buffer\_cache\_lookup()

주어진 sector 주소에 해당하는 entry 가 있는지 확인하고 있으면 그 entry 를 반환하고 없으면 null 을 반환한다.

Buffer\_cache\_select\_victim()

reference bit 을 확인해 reference bit 이 0 이면 그 entry를 반환한다. 만약 그 entry의 dirty bit 이 1 이었으면 먼저 disk 에 기록하고 반환한다.

Buffer\_cash\_flush\_entry()

해당 entry 의 dirty bit 이 1 이면, 즉 수정이 되었으면 block\_write 함수를 통해 영구적으로 저장될 수 있게 기록한다.

Inode.c

Inode\_read\_at()&write\_at()

기존에는 file 을 읽을 때 block\_read 함수로 disk 를 바로 읽었지만 이제는 buffer cache 에 데이터가 있으면 buffer cash 에 데이터를 읽는 게 더 빠르므로 block\_read 함수 대신 buffer\_cache\_read 함수를 써서 먼저 cache 를 확인해서 읽는다. Write 도 마찬가지로 먼저 buffer cache 에 기록한 후 나중에 disk 에 기록되도록 한다.

Filesys.c

Filesys\_init() & filesys\_done()

File system 이 시작할 때 buffer\_cache\_init() 함수도 넣어주어 buffer\_cache 도 준비 될 수 있게 한다. 끝날 때도 buffer\_cache\_terminate() 함수를 불러준다.

**Extensible file & file growth**

Inode\_disk 구조를 수정한다. Direct\_blocks[], indirect\_block, doubly\_indirect\_block 변수를 추가한다. 그리고 indirect block(sector) 에는 데이터 또는 또다른 indirect block 의 주소가 저장되어 있으므로 그것을 받아오기 위한 구조체인 inode\_indirect\_block\_sector 구조체를 만들고 이 구조체 안에는 blocks[] 배열이 들어가 있다. 여기에 indirect sector 에 담겨져 있는 내용을 받아올 것이다.

Byte\_to\_sector()

기존에는 file 이 consecutive 하게 저장되어 있으므로 읽고 싶은 file 의 offset 이 들어오면 offset 을 block\_size 로 나누고 file 의 시작 주소를 더 하면 짜잔 하고 읽어야 하는 block 주소가 나왔으나 이제는 indexing 방식으로 바꾸었기 때문에 offset 에 따라 direct block 에 저장되어 있는지, indirect block 이나 doubly indirect block 에 저장되어 있는지 확인해서 타고 들어가서 읽어야 하는 주소를 반환하게끔 바꾸어 주어야 한다.

Inode\_create()

기존의 free\_map\_allocate 부분을 새로 만든 함수인 inode\_allocate 로 바꾸어 file 이 disk 에 더 많이, 그리고 분할되어서 저장될 수 있게끔 한다.

Inode\_open()

주어진 주소의 inode 를 기존에서 block\_read 를 통해 disk 에서 읽어 오던 것을 buffer\_cache\_read 함수를 통해 on memory 에 해당 data 가 있으면 먼저 buffer 에서 읽어 올 수 있도록 하여 더 빠르게 IO 를 처리할 수 있게 한다.

Inode\_close()

기존 on disk 자료구조인 inode\_disk 를 지우기 위해 freemap\_release 함수를 사용하던 것을 새로 만든 함수인 inode\_free 함수로 바꾼다. Indexing 구조로 바뀌었으므로 기존과 다른 free 함수가 필요하다.

Inode\_write\_at()

만약 file 에 file size 를 넘어서는 size 만큼을 기록하려고 하면 file size 를 키워서 기록할 수 있게 해주는 부분을 추가하여 준다. 새로 만든 inode\_allocate 함수를 통해 추가적으로 sector 를 가질 수 있게 해준다.

Inode\_allocate() & Inode\_indirect\_allocate()

Indexing 구조로 바꾸었으므로 file 이 disk 에 여러 군데 나뉘어서 저장되도록 바꾸어야 한다. 우선 기존의 bytes\_to\_sector 함수를 통해 file 의 크기를 sector 크기로 나누어 총 몇 개의 sector 가 필요한지 받아오고 나서 direct\_block 배열부터 넣기 시작한다. Direct\_block 은 123개의 entries 를 가지므로 만약 파일크기가 123 x sector size(512 byte) 를 넘으면 이제 indirect\_block 으로 가서 여기에 넣기 시작한다. 이것도 부족하면 doubly\_indirect\_block 을 이용해 타고 타고 들어가서 넣는다. Indirect block, doubly indirect block 에 넣기 위해 inode\_indirect\_allocate 함수를 불러서 넣는다.

Inode\_free() & Inode\_free\_indirect()

Inode 를 배정할 때와 마찬가지 방법으로 inode 를 타고 들어가서 다 free 해준다.

**Subdirectory**

Thread.h 의 thread 구조체에 struct dir \*cwd 를 하나 넣어준다. 여기에 스레드가 현재 작업중인 directory 가 들어간다.

Inode\_disk 구조체에 is\_dir 변수가 들어간다. Directory를 만들면 data block 에 file 과 들어가는 내용이 다르고 다르게 처리를 해야하기 때문에 구별해야 한다.

Inode.c

Inode\_create()

Inode\_create 함수의 인자로 bool is\_dir 을 추가한다. Is\_dir 을 받아 file 이면 0, directory 이면 1 을 inode\_disk 구조체 is\_dir 에 넣어준다.

Filesys.c

Filesys\_create()

파일을 만드는 함수다. 파일을 만드는 것이므로 이 함수안에서 inode\_create 함수를 부를 때 인자로 false 를 추가한다.(file = 0, directory = 1) 그리고 이제는 subdirectory 로 인해 계층 구조이므로 파일을 만드려는 path 를 받아와 directory 부분과 file 부분으로 parsing 해주어야 한다.

Filesys\_create\_dir()

Directory 를 만드는 함수이다. Directory를 만드는 것이므로 이 함수안에서 inode\_create 함수를 부를 때 인자로 True 를 추가한다.(file = 0, directory = 1) 그리고 이제는 subdirectory 로 인해 계층 구조이므로 어디에 directory 를 만드는지 path 를 받아와 parsing 해주어야 한다.

Filesys\_open()

인자로 받아온 경로를 parsing 을 한 다음 해당 directory 에서 찾아서 inode 를 열어 반환한다. 만약 이미 지워진 file 을 열려고 하면 안된다고 한다.

Filesys\_remove()

경로를 parsing 하는 부분을 추가한다.

Parse\_path()

주어진 path 를 directory 와 file(directory 를 만드는 거면 directory) 로 분리한다.

Directory.c

Dir\_create()

Sector 주소와 size 를 받아 inode\_create 함수를 불러 inode 를 만든다. Dir\_add\_first\_entry 함수를 불러서 그 directory 를 열고 첫 번째 dir\_entry 에는 부모 directory 의 sector 주소를 넣어버린다.

Dir\_open()

Directory 의 첫 번째 entry 는 부모 directory 를 가리키고 있으므로 directory 의 pos 를 두 번째 entry 부터 시작하게 한다.

Dir\_open\_path()

주어진 path 를 따라가서 directory 를 연다. 만약 이미 지워진 directory 를 열려고 하면 막는다.

Lookup()

Dir 안에 entry 들을 순회하는데 첫번째 entry 는 부모 directory 를 가리키고 있으므로 순회할 때 두번째 entry 부터 순회하도록 바꾸어 준다.

Dir\_is\_empty()

Directory 를 지울 때 비었는지 확인하는 함수이다. Entry 를 순회하면서 in\_use 변수를 체크한다. 역시 두번째 etnry 부터 순회한다.

Dir\_lookup()

‘.’ 과 ‘..’ 을 찾는 것이 들어오는 부분을 추가해준다. ‘.’ 은 현재 directory 이므로 현재 inode 를 reopen 해서 넘겨주고 ‘..’ 이 들어오면 첫 번째 entry에 들어있던 부모 directory 의 sector 를 열어서 넘겨준다.

Dir\_add()

File 이나 directory 를 만들면 부모 directory에 그것을 추가 해주어야한다. File 을 만들면 기존과 동일하고 만약 directory 를 만들면 이 directory 의 첫번째 entry에 부모 directory 의 sector 를 추가해주는 부분을 작성해준다.

Dir\_remove()

만약 지우는 것이 directory 이고 directory가 비어있지 않으면 지우지 않는다.

File 을 지우는 경우나 directory가 비어있으면 지울 수 있게 한다.

Dir\_chdir()

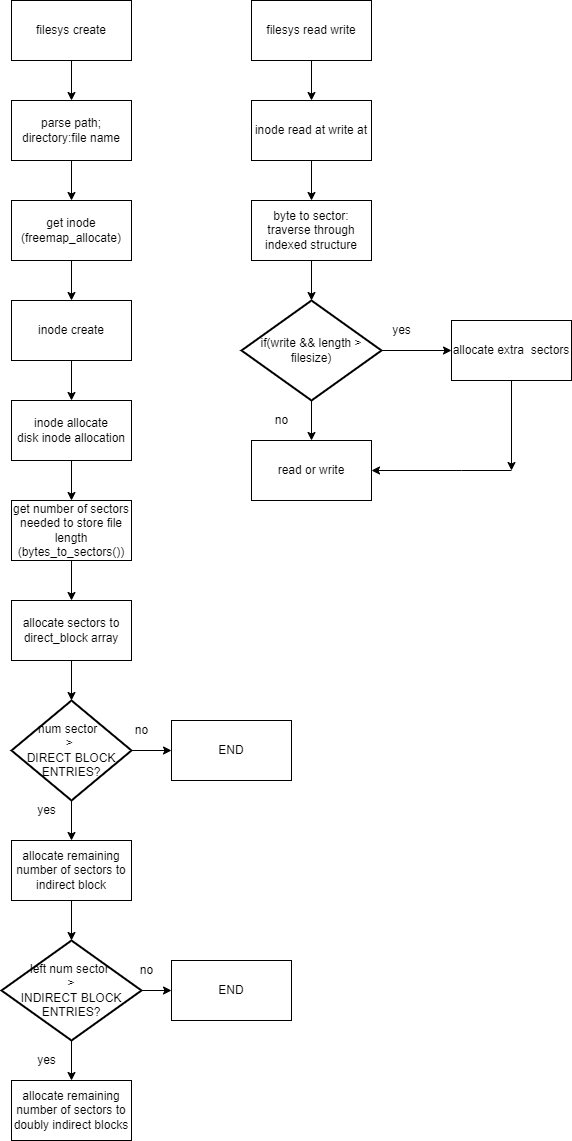
현재 쓰레드의 working directory 를 인자로 받은 directory 를 parsing 한 후 바꾸어 준다.

Dir\_add\_first\_entry()

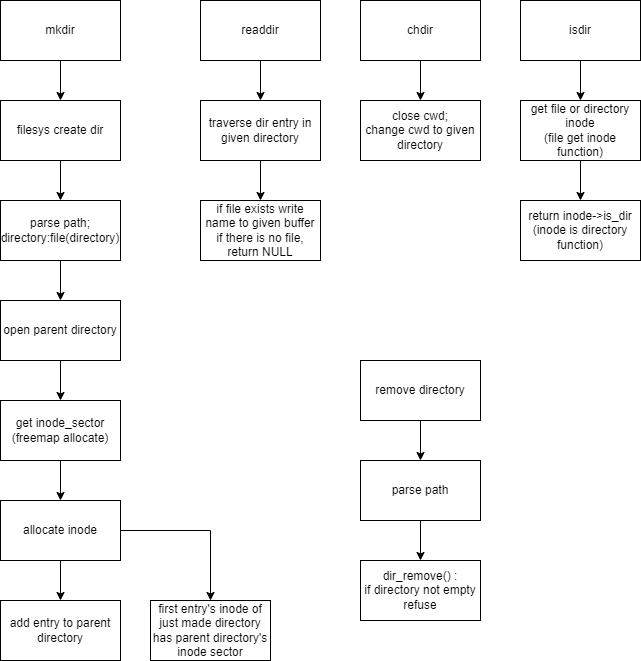
Directory 에 첫번째 entry 는 부모 directory 의 sector 를 가리키고 있게 한다. ‘..” 을 처리하기 위함이다.

1. **연구 결과**
   1. **Flow Chart**

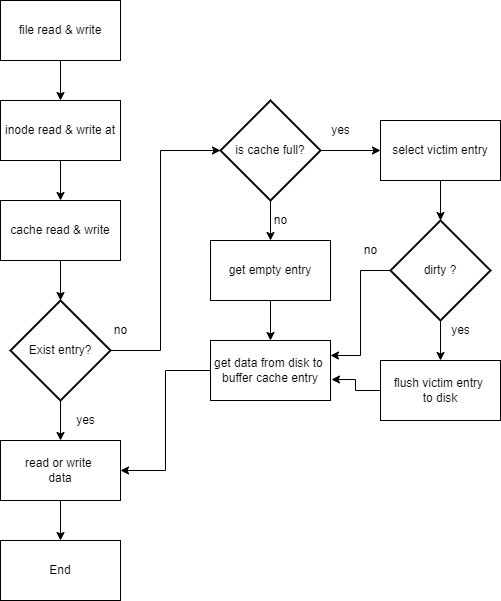
* **II. B. 개발 내용에 대한 Flow Chart를 작성**
* Extensible file & file growth



* Subdirectory

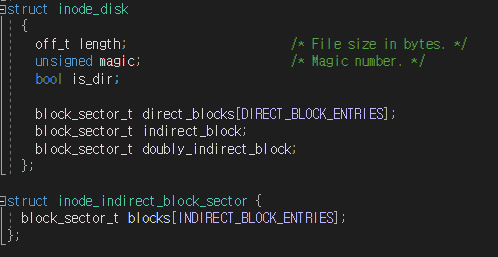


* Buffer cache  
  **구현한 경우만 작성**



* 1. **제작 내용**
* Extensible file & file growth

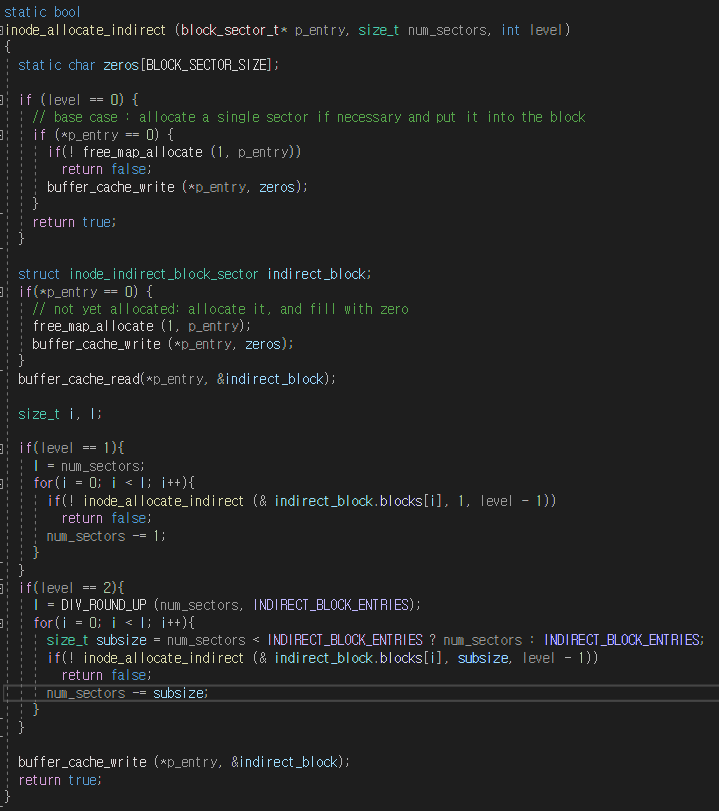
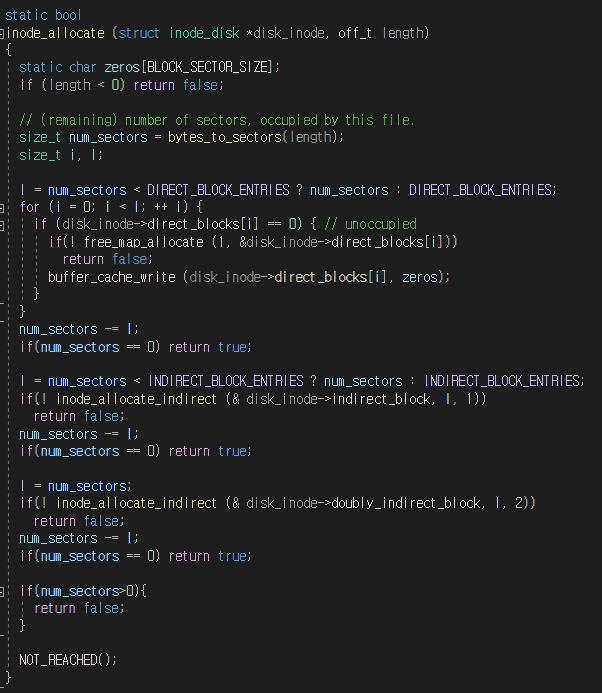
Inode\_disk 의 구조체를 확인해보자.



Direct\_blocks 배열은 바로 data block 들의 주소들을 가지고 있고 indirect block 변수는 data block들을 가리키고 있는 indirect block 의 주소를 가리키고 있고 doubly indirect\_block 은 한번 더 들어간다. 밑의 inode\_indirect\_block\_sector 는 indirect block 들이 가지고 있는 data block 들의 주소를 저장하기 위해 만들었 다.

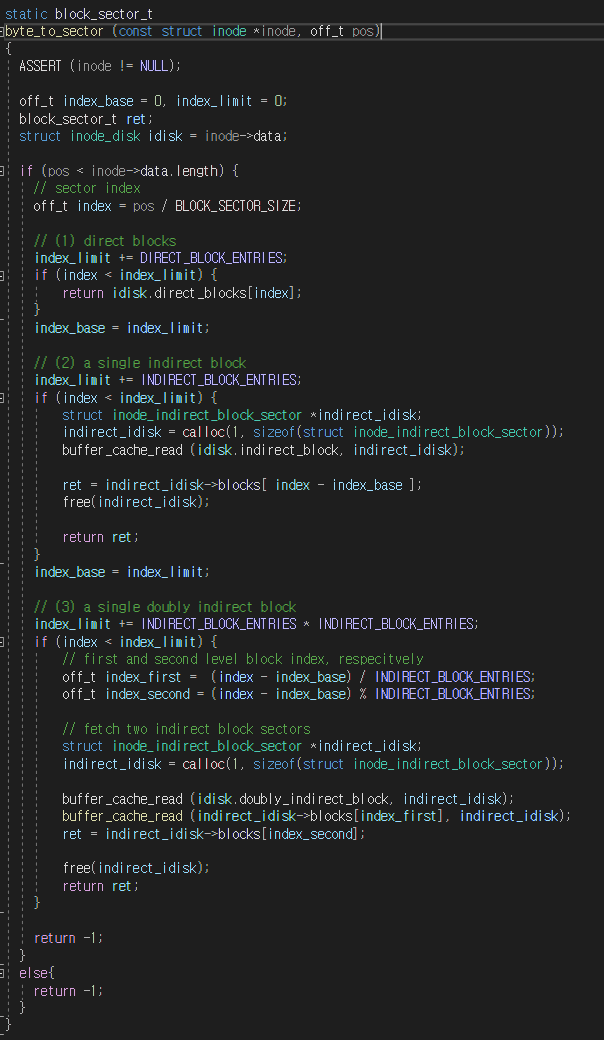
이제 inode\_create 함수를 통해서 inode 가 만들려고 하면 inode\_allocate 함수를 호출하여 만드는데 inode\_allocate 함수를 보자.

Inode allocate() & Inode allocate indirect()



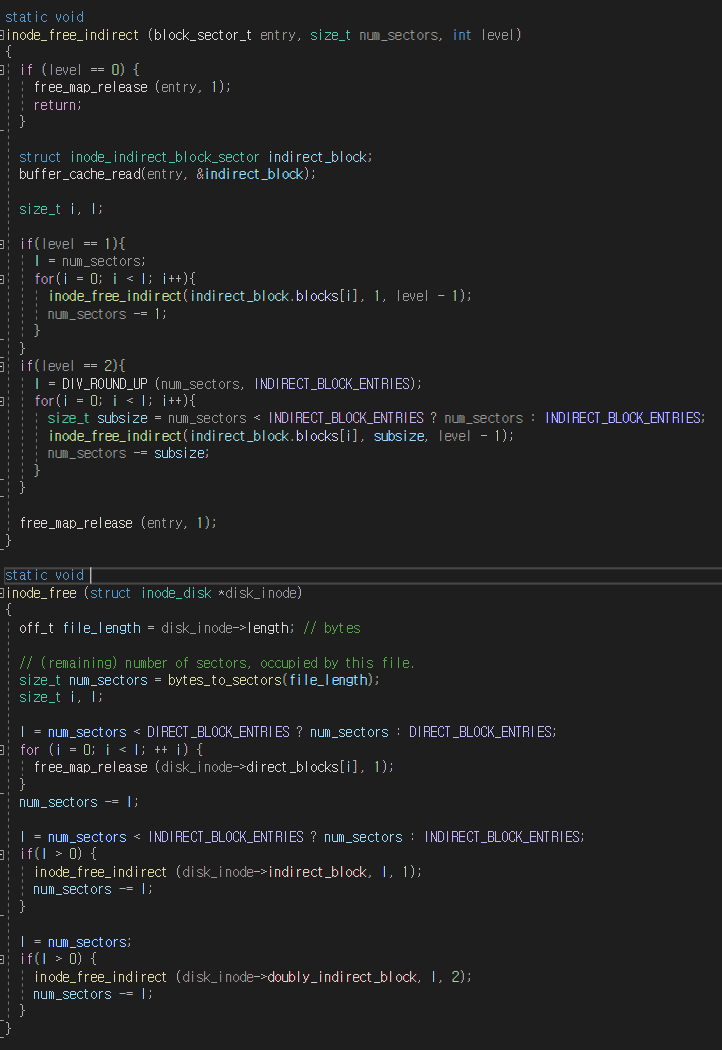
기존의 bytes\_to\_sectors 함수를 통해 file 을 저장하는 데 있어서 총 필요한 block(sector) 의 수를 가져온다. 이제 이 sector 수 만큼 direct block 배열에, 초 과하면 indirect block 에, 또 초과하면 doubly indirect block 에 저장한다. 코드를 보면 먼저 num sectors 를 direct block 배열에 저장한다. 만약 이걸로 충분하면 true 를 리턴하고 끝나게 된다. 부족하면 다음으로 넘어가서 inode\_allocate\_indirect 함수를 level = 1(depth) 로 호출하여 indirect block 에 저 장하게 된다. 아직도 sector 가 남으면 inode\_allocate\_indirect 함수를 level = 2 로 호출하여 doubly indirect block 에 저장하도록 한다.

Byte to sector()



File 의 원하는 위치를 읽으려면 이제 file 이 여러 곳에 흩어져 저장되어 있어서 찾아서 들어가서 읽어야 한다. 찾으려는 위치인 pos 가 direct\_block\_entry 보다 작으면 direct block 배열 안에서 찾아주고 pos 가 그것보다 크면 indirect\_block 을 타고 들어가서 찾아서 반환해준다.

Inode\_free() & Inode\_free\_indirect()

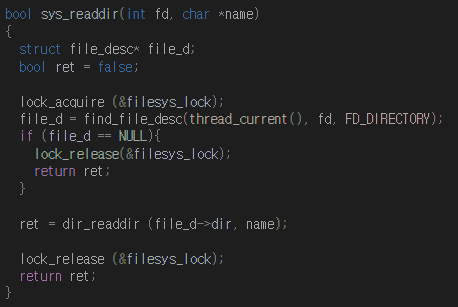


Free 하는 것도 위와 마찬가지로 타고 들어가서 free 해준다.

* Subdirectory

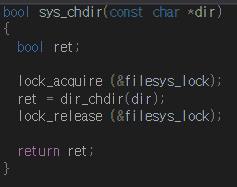
Thread 구조체에 struct dir \* cwd 를 추가해주고 inode\_disk 구조체의 is\_dir 변수 를 추가하고 시작한다. 추가적인 system call 을 구현해야 하는 데 간단한 것들 부터 살펴보자.

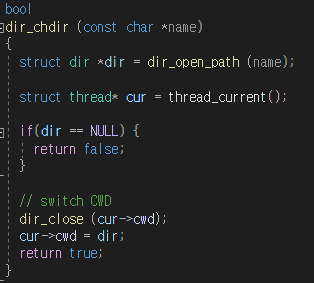
우선 readdir system call 이다.



Readdir system call 은 directory.c 에 kernel api 가 이미 구현이 되어있다. 따라 서 userprog system call 에서 연결해주기만 하면 된다. Dir\_readdir() 함수를 불러 읽어주자.

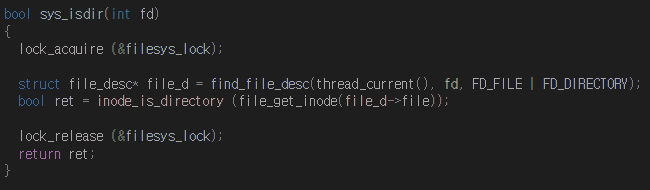
Chdir system call

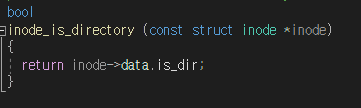




Chdir system call 이 들어오면 userprog syscall.c 에서 dir\_chdir kernel api 를 부 른다. 그러면 dir\_chdir 함수에서 현재 작업하는 directory 를 인자로 받은 name 을 parsing 해서 열고 parsing 한 directory 로 바꾸어준다.

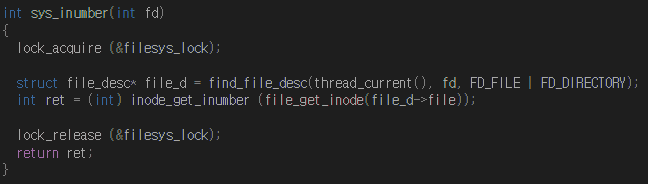
Isdir system call

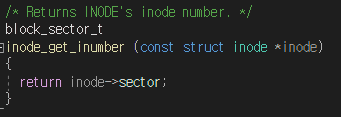




Isdir system call 이 들어오면 inode is directory 커널 함수를통해 inode 의 is\_dir 변수를 반환한다.

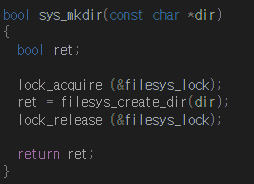
Inumber system call



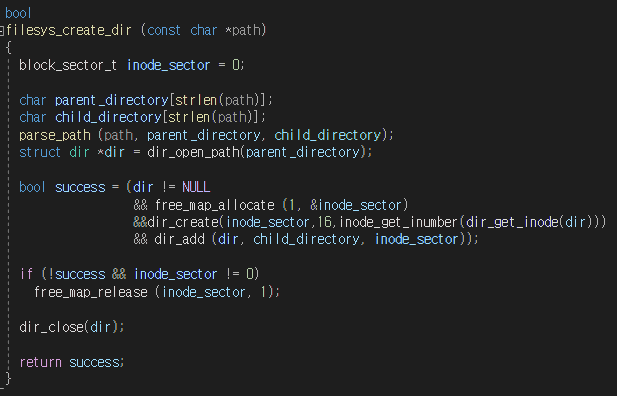


Inumber system call 도 같은 방식이다.

이제 가장 중요한 mkdir, 즉 subdirectory 를 만드는 상황을 알아보자.

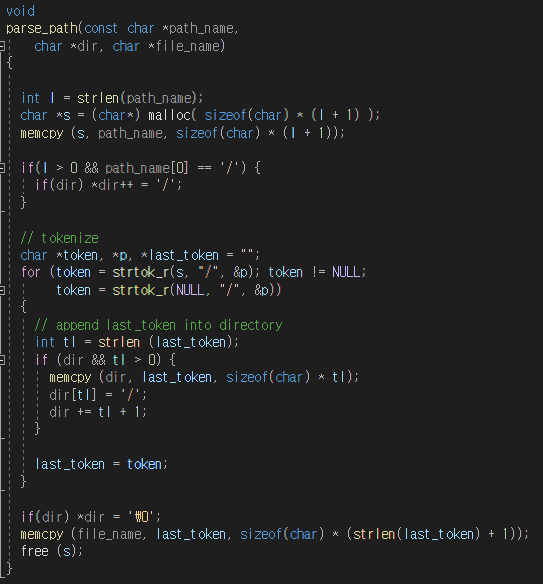
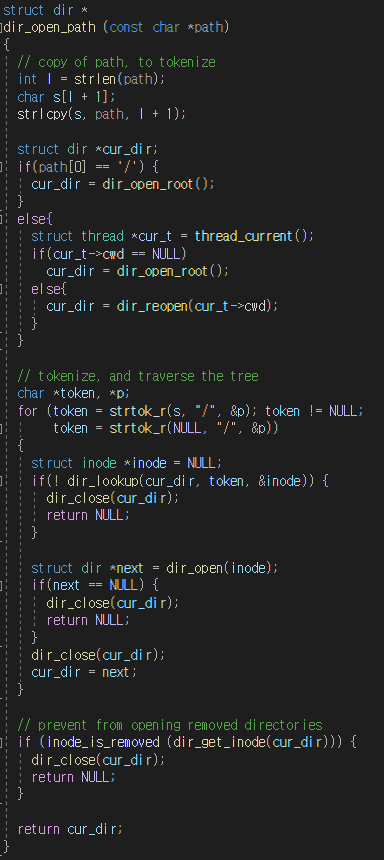


새로 만든 Filesys\_create\_dir kernel api 를 사용한다.



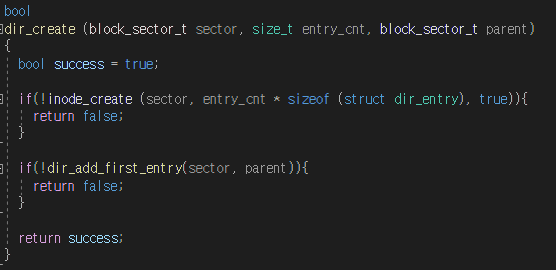
먼저 인자로 받은 path 를 parent directory 와 child directory 로 분리 해주어야 한다. 분리하는 함수 두개를 살펴보자.

Parse path dir\_open\_path

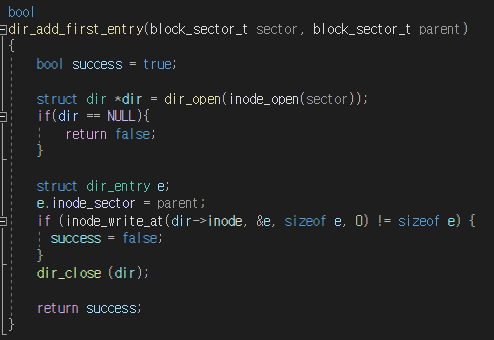
 

먼저 parse path 함수를 통해 (상위 directory 부분) : 나머지 생성하려는 directory 이름으 로 분리한다. 그 후 dir\_open\_path 함수를 통해 (상위 directory 부분) 을 열면서 들어간 다. 그리고 만드려는 directory의 바로 위 부모 directory 를 반환한다.

다시 filesys\_create\_dir 함수를 보면 위 두 함수로 child directory[] 배열에는 만드려는 directory의 이름이, dir 포인터에는 만드려는 directory 의 부모 directory 를 가리키게 된 다. 이렇게 해놓고 inode sector 하나를 받아오고 dir\_create 함수를 불러서 만들어주고 부모 directory 인 dir 에 entry 를 추가해주면 된다. Dir create 함수를 한번 보면

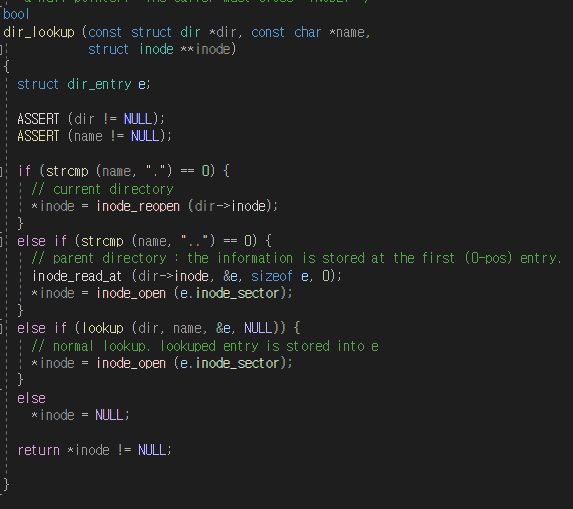


우선 inode 를 만들어준다. 그리고 여기서 ‘..’ 을 처리해주기 위해 중요한 일을 하는데 각 directory 의 첫번째 directory entry 는 부모 directory의 sector 주소를 가지게 한다. Root directory 는 첫 번째 entry 로 자기 자신을 가리키게 한다. Dir add first entry 함수 를 확인하자.



Directory 를 열어주고 첫 번째 entry 에 부모 directory 의 sector 를 넣어준다.

큰 흐름은 끝났다. 하나 더 중요한 ‘.’ 과 ‘..” 을 handling 하는 부분을 보자.

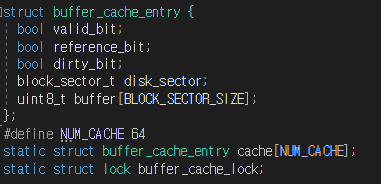


User 가 ‘.’ 을 사용하게 되면 ‘.’ 자기 자신을 가리키므로 자기자신의 inode 를 열어서 반 환해주면 된다. 그 다음 ‘..’ 을 사용하면 directory 의 첫 번째 entry 에는 항상 부모 directory 의 sector 주소를 넣어놨으므로 첫 번째 entry 를 읽어서 이 inode 를 open 하 고 반환해주면 된다. 그 외에는 기존과 같다.

나머지 부분은 user 가 비정상적인 행동을 하는 것을 막아주는 코드들을 군데 군데 삽입 하였다. 이는 3-B 함수 설명에서 포함하였으므로 생략한다.

* Buffer cache  
  **구현한 경우만 작성**

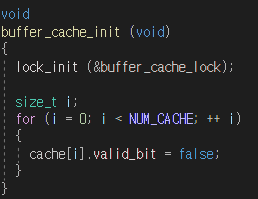
Buffer cache 를 위한 구조체 코드부터 시작하자.



Cache 배열을 만드는 데 배열에 저장될 구조체에는 사용 여부의 valid bit, eviction 을 하기 위한 reference bit, 수정 여부의 dirty bit, 그리고 buffer 에 있는 data 가 disk 어디에 있는지를 위한 disk\_sector, data 가 저장되는 buffer 가 있다.

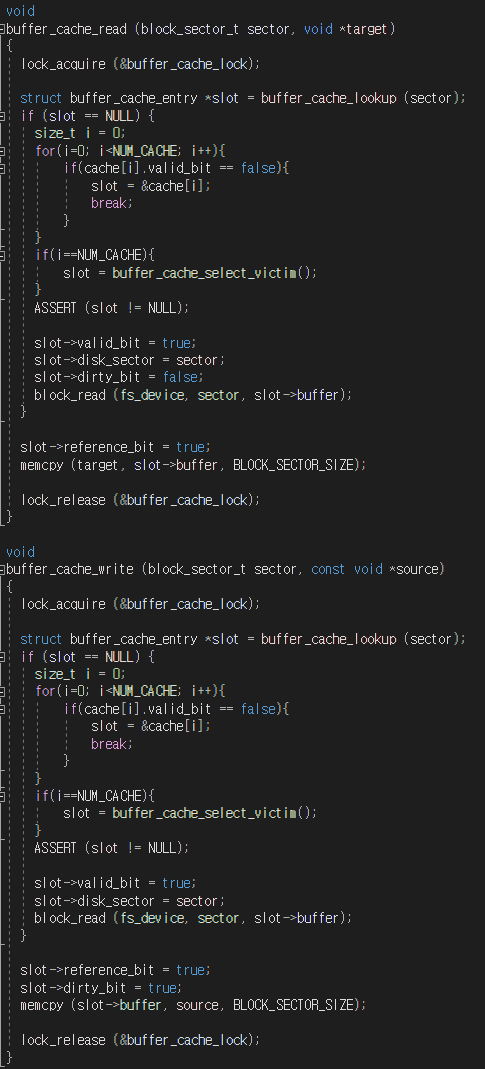
Pintos 의 manual 에 cache size 는 64 라고 했으므로 64개로 만든다.

Filesystem 이 시작할 때 buffer cache\_init 함수가 실행되면서 buffer cache 도 시작된다. 따라서 buffer cache init 함수를 보면



Synchronization 을 위한 lock 을 만들어주고 초기에는 모든 entry 들이 사용하지 않는 상태이므로 valid bit 을 다 false 로 만들어준다.

Buffer cache read write 함수를 보자.

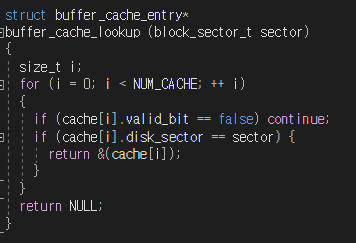


먼저 lock 을 잡고 들어가서 buffer cache lookup 함수를 통해 주어진 sector 를 가지고 있는 cache entry 를 찾는다.

있으면 reference bit 을 1 로 바꾸고 만약 write 이면 dirty bit 까지 1 로 바꾼다. 그리고 read 이면 target 에 buffer 에 있던 것을 넣어주고 write 이면 source 를 buffer 에 넣어준다.

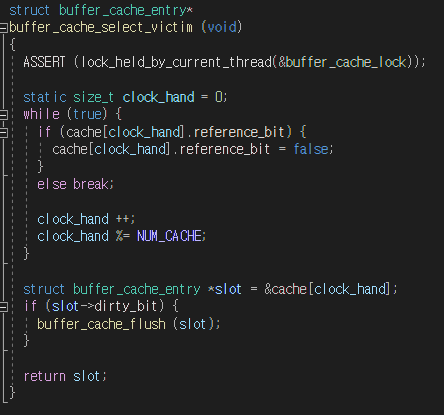
없으면 사용하지 않고 있는 entry 를 찾아서 거기에 read write 을 하고 만약 모두 사용하고 있으면 evict 함수를 불러서 evict 할 entry 를 고르고 거기에 block read 로 필요한 데이터를 가져온 후에 처리를 해준다.

Cache lookup



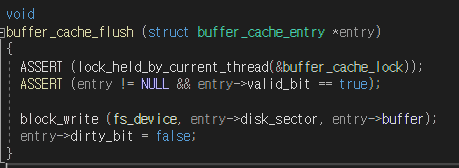
주어진 sector 를 가지고 있는 cache entry 를 반환한다. 없으면 null 을 반환한다.

Buffer cache select victim



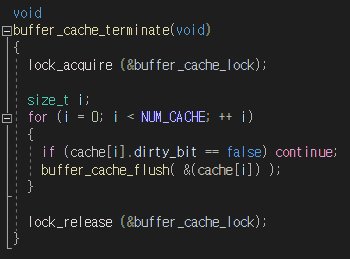
Second chance 알고리즘을 사용한다. Reference bit 을 확인하여서 1 이면 한번 봐주고 0 이면 evict 대상이 된다. Evict 대상이 된 entry의 dirty bit 이 1 이면 수 disk 에 flush 해준 다음에 반환한다.

Buffer cache flush



Disk 에 인자로 받은 entry 를 기록하는 함수이다.

Buffer cache terminate



Filesystem 이 끝날 때 buffer 에 있는 수정된 data 들을 disk 에 기록하고 끝내 게 된다.

이렇게 cache buffer 를 구성한 후 흐름도에서 보여준 대로 inode\_read write 부 분에서 바로 disk 를 읽는 것이 아닌 cache 를 read write 하는 것으로 변경해주 면 cache 를 제대로 사용하게 된다.

* 1. **시험 및 평가 내용**
* **Src/filesys make grade 수행결과를 캡처 하여 첨부.**

