**Pintos Project 5: Filesystem**

담당 교수 : 김영재

조 / 조원 : 20171659/유경연

개발 기간 : 12/1~12/22

1. **개발 목표**

이번 프로젝트에서는 현재 핀토스의 file system 문제를 해결하기 위해서 indirect block을 추가해 external fragmentation를 해결하고 file size를 좀 더 유연하게 한다. subdirectory도 사용할 수 있도록 한다. 추가로 buffer cache를 구현하여 메모리 효율성을 높인다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Extensible file & file growth

현재 핀토스의 file system은 file을 처음 생성할 때에 size가 정해지고 contiguous하게 할당된다. 이 때문에 file을 관리하는데 여러 문제가 발생하는데 대표적으로 external fragmentation이 있다. indexing 방법을 사용하면 indirect block을 이용해서 file의 data들을 관리할 수 있다. 이런 방식으로 구현하면 file의 size를 유연하게 조절할 수 있고 block을 contiguous하게 할당하지 않아도 되므로 fragmentation 문제를 해결할 수 있다. 또한 file size가 고정되어 있지 않으므로 여분 공간이 남아있는 경우에 file size를 더 증가시킬 수 있게 된다.

1. Subdirectory

현재 핀토스는 root directory 하나밖에 사용할 수 없다. 코드를 수정하여 root directory 밑의 subdirectory들을 사용할 수 있도록 한다.

1. Buffer cache

매번 disk에 접근하는 것은 매우 시간이 오래 걸리는 일이다. 따라서 자주 접근하는 data를 buffer cache에 저장해서 disk access time을 줄일 수 있다. 이때 buffer cache에 빈 공간이 없는 경우에 빈 공간을 만들기 위해서 data를 evict하는 알고리즘이 필요하다.

* 1. **개발 내용**
* Extensible file & file growth
  + Index structure와 management에 대해서 기술

현재 핀토스는 file을 contiguous하게 할당한다. 이렇게 file system을 구현할 경우 필연적으로 external fragmentation이 발생하게 된다. 이를 해결하기 위해서 index structure를 사용한다. index structure는 inode에 direct index, indirect index, doubly indirect index을 추가하여 file을 관리한다. direct index은 직접적으로 disk 상의 data secotor을 pointing하는 index이다. 반면 indirect index와 doubly indirect index는 바로 data sector을 pointing하지 않는다. indirect는 indirect block을 1번 거쳐서 data sector을 pointing하고, doubly indirect는 2번 거쳐서 data sector을 pointing한다. 이처럼 구현하는 경우엔 할당할 수 있는 data sector의 개수가 늘어나서 file의 최대 크기가 증가하고 contiguous하게 할당하지 않아도 돼서 file size를 변경시킬 수 있다.

* Subdirectory
  + Directory entry 관리 방법

현재 핀토스는 subdirectory를 지원하지 않는다. subdirectory를 사용하기 위해서는 입력되는 문자열을 filename과 path로 구분해야 할 필요가 있다. path에는 absolute와 relative 두 종류가 있으므로 이에 대해 적절하게 처리해야한다. directory가 비어 있지 않은 경우엔 삭제가 불가능하므로 subdirectory가 추가되면 directory를 삭제하기 위해선 directory가 empty 한지에 대해 검사도 필요하다. directory도 file의 일종이므로 file처럼 관리한다. inode를 부여하고 disk의 data block에 write하여 관리한다.

* Buffer cache
  + Buffer cache eviction 방식

가상 메모리에서 사용한 방식과 유사한 방법을 사용한다. Pseudo LRU 알고리즘인 second chance 알고리즘을 사용하였다. evict할 후보를 찾기 위해 cache의 data에 flag를 만들어서 관리한다. data에 access되는 경우에 그 flag를 set한다. 후보를 선정하는 과정에서 data의 flag가 true라면 flag를 false로 만들고 cache의 다음 data를 검사한다. 이런 식으로 검사하다가 false인 data를 발견한다면 그 data를 evict 후보로 선정한다.

* + Buffer cache flush 방식

buffer cache에서 data를 제거하기 전에 data가 유효하고 dirty bit가 set되어있는지 검사한다. 만약 그렇다면 그 data는 cache에 올라온 이후로 수정되었으므로 disk에 다시 써준다. dirty bit가 set되어 있지 않다면 disk에 있는 원본 데이터와 동일하므로 따로 disk에 써주지 않아도 된다. buffer cache에서 evict하거나 핀토스가 종료되는 경우에 flush를 사용한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

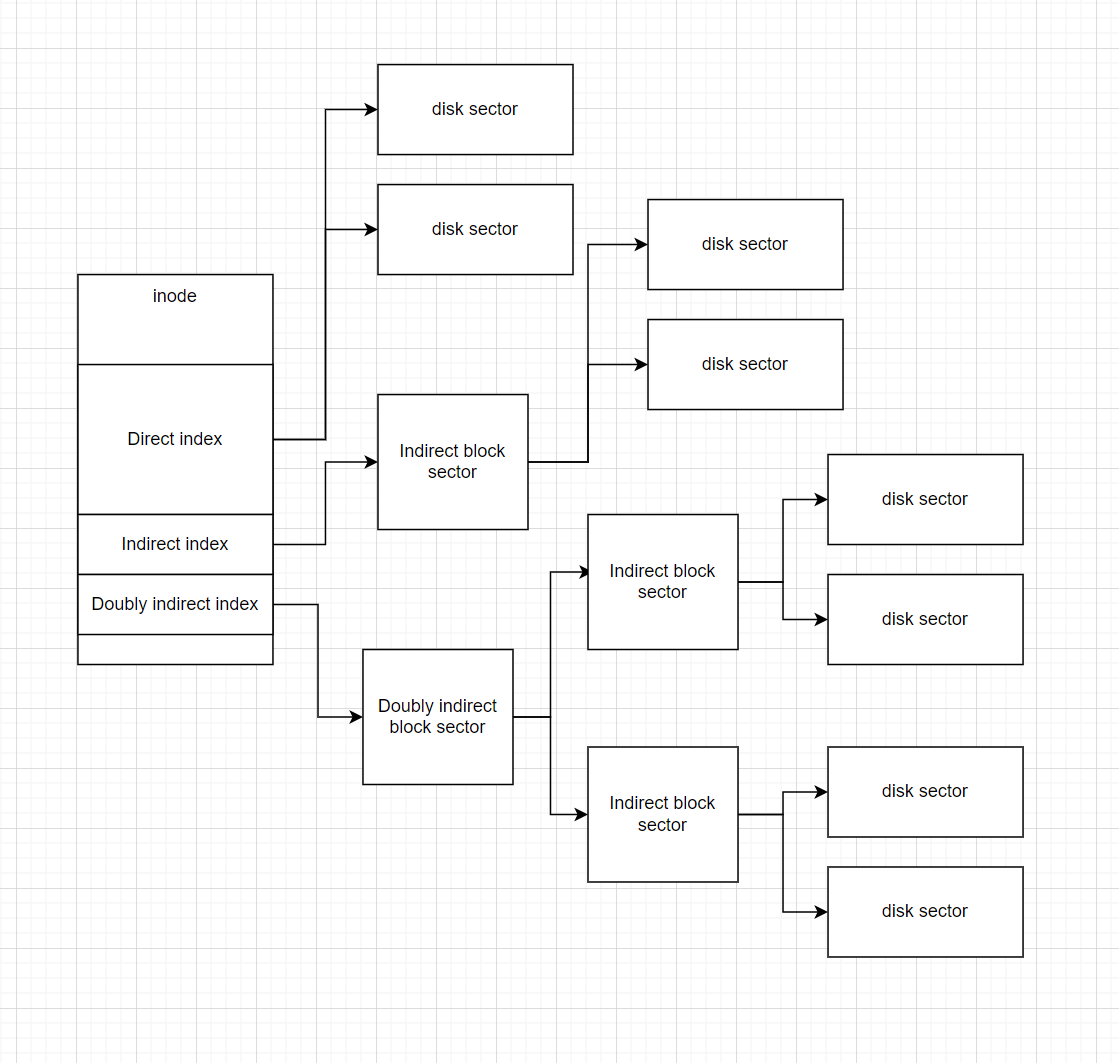
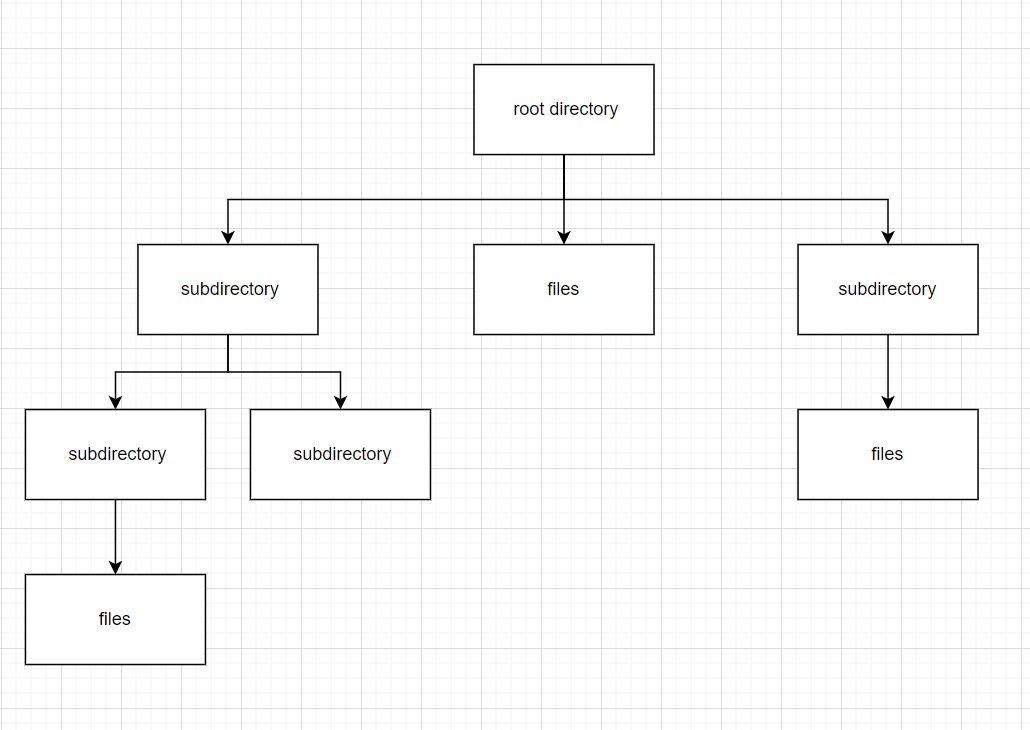
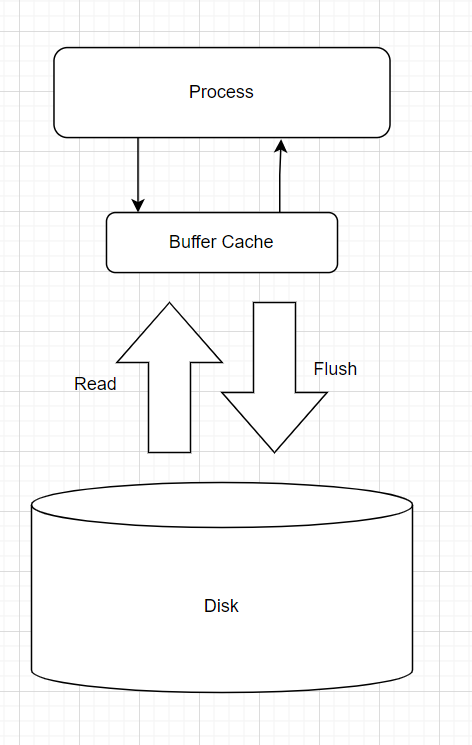
12/1~12/3 : Buffer cache 구현

12/4~12/14 : Extensible file &file growth 구현

12/15~12/22 : Subdirectory 구현

* 1. **개발 방법**
* Extensible file & file growth
  + 현재 구현되어 있는 on-disk structure를 index structure로 수정해야 한다..
  + inode에 index structure를 구현하기 필요한 자료구조를 추가한다.
  + index가 주어지면 disk에서의 위치를 알아내는 함수가 필요하다.
  + inode를 disk에 새로 allocate하는 함수가 필요하다
  + inode를 disk에서 free하는 함수가 필요하다.
* Subdirectory
  + path와 filename을 parsing해주는 함수가 필요하다.
  + directory가 비어 있는지 검사해주는 함수가 필요하다.
  + subdirectory를 open하기 위한 함수가 필요하다.
  + 어떤 inode가 directory인지 아닌지 판별이 필요하다.
  + inode 생성하는 함수를 수정하여 새로운 directory를 만들 수 있도록 한다.
  + 상황이 바뀐 것에 대해서 directory.c와 filesys.c의 함수들을 수정해야한다.
  + 비어 있지 않은 directory에 대해 삭제에 대한 예외처리가 필요하다.
  + system call에 directory 관련한 함수를 수정하거나 추가해야 한다.
  + system call 함수들의 구현을 위해서 file descriptor의 자료구조를 배열에서 리스트로 수정하고 file descriptor를 탐색하는 함수를 새로 구현하였다.
* Buffer cache
  + 새로 buffer\_cache.h와 buffer\_cache.c 파일을 만들어서 buffer cache를 구현하였다. buffer cache를 구현하기 위해 필요한 구조체와 함수들은 다음과 같다.
  + buffer cache entry 구조체 : buffer cache의 entry이다. data를 저장하는 부분과 buffer cache 관리를 위한 flag가 필요하다.
  + buffer cache lock : buffer cache의 concurrency를 보장하기 위해서 필요하다.
  + buffer cache 초기화 함수 : 프로그램 시작 시 buffer cache를 초기화한다.
  + buffer cache read 함수 : buffer cache에서 data를 읽는다. data가 cache에 올라와 있지 않은 경우엔 disk에서 새로 읽는다.
  + buffer cache write 함수 : buffer cache에 있는 data를 수정한다. 이때 buffer cache에 해당하는 data가 없다면 disk에서 새로 읽는다.
  + buffer cache flush 함수 : 앞에서 설명한 flush 기능을 수행한다.
  + buffer cache flush all 함수 : buffer cache의 모든 data에 대해서 flush 함수를 실행한다. 프로그램이 종료되는 경우에 실행된다.
  + buffer cache lookup 함수 : buffer cache에 찾고자 하는 data가 있는지 검사한다.
  + buffer cache select victim 함수 : buffer cache에서 data 하나를 evict한다. second chance 알고리즘을 사용한다.
  + inode.c에서 disk read, write하는 부분을 buffer cache를 사용하도록 수정한다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**
* Extensible file & file growth
  + index structure 개요
* Subdirectory
  + subdirectory 구조 개요
* Buffer cache
  + Buffer cache 개요
  1. **제작 내용**
* Extensible file & file growth
  + index structure를 구현하기 위해서 on-disk inode인 inode-disk 구조체를 수정하여 unused부분을 direct 123, single indirect 1, double indirect 1로 나누었다. direct index 123개, single indirect index 1개, doubly indirect index 1개를 사용할 수 있다. 이때 한 sector안에 indirect block은 128개까지 들어갈 수 있다. 여기에 이후 subdirectory를 구현하기 위해 필요한 is\_dir 변수를 추가했다. 이 변수는 이 inode가 file인지 directory인지를 나타내는 변수이다.
  + indirect block sector을 나타내기 위해서 inode\_indirect\_block 구조체를 정의하였다. 이 구조체는 disk의 sector 중 indirect block들을 담는 sector이다. 한 sector에는 128개의 indirect block이 들어있다.
  + 현재 핀토스에서는 byte\_to\_sector() 함수를 사용해 disk에서의 위치를 알아낸다. 하지만 index structure로 구현하게 되면 byte\_to\_sector() 함수만으로 부족하다. 따라서 index\_to\_sector() 함수를 구현하였다. 이 함수는 pos를 index로 translate하기 위해서 byte\_to\_sector() 함수의 인자로 들어온 pos를 sector의 크기로 나눈 block\_index를 인자로 받아서 disk 상의 위치를 구하는 함수이며 doubly indirect index까지 처리가 필요하다. block\_index는 다음과 같이 translate 된다.

|  |  |
| --- | --- |
| 0<= remain index <DIRECT\_BLOCKS | Direct index |
| Index < DIRECT\_BLOCK + INDIRECT\_BLOCKS\_PER\_SECTOR | Indirect index |
| Index < DIRECT\_BLOCK + INDIRECT\_BLOCKS\_PER\_SECTOR + (INDIRECT\_BLOCKS\_PER\_SECTOR)^2 | Doubly indirect  index |

indirect block 또한 data block에 저장되어 있으므로 disk를 read할 필요가 있는데 disk I/O를 직접 하지 않고 buffer cache를 통하여 하도록 구현했다.

* + bool inode\_allocate() : inode를 disk에 새로 할당하는 함수다. 이 함수는 다음과 같은 순서로 진행된다.
    1. bytes\_to\_sectors() 함수를 호출하여 할당해야 되는 길이를 이용해서 몇 index까지 write가 필요한지 알아내서 remain\_index에 저장한다. 이때 이 remain\_index는 할당이 필요한데 아직 할당을 하지못한 disk sector의 개수이다. remain\_index가 file의 최대 size를 넘어가는 경우를 예외 처리한다. 각 indexing에서 최대로 할당가능한 sector의 개수를 bound로 관리한다.
    2. direct index들에 data block을 1개씩 할당하고 buffer cache에 해당 주소를 올려놓는다.
    3. direct index를 모두 할당하고도 할당이 필요한 block이 남아 있다면 single indirect indexing을 수행한다. indirect 할당을 위한 함수인 inode\_allocate\_indirect() 함수를 level = 1로 호출해 사용한다.
    4. single indirect index를 모두 할당하고도 할당이 필요한 block이 남아 있다면 doubly indirect indexing을 수행한다. inode\_allocate\_indirect() 함수를 level = 2로 호출해 사용한다.
    5. remain\_index가 0인 경우에는 할당이 성공적으로 된 것이므로 true를 return한다. 예외에 걸리거나 block을 할당해주는 함수 free\_map\_allocate()가 실패하는 경우에는 false를 return한다.
  + bool inode\_allocate\_indirect() : inode\_allocate()에서 indirect indexing이 필요한 경우에 호출된다. 함수 인자로 들어오는 level의 값에 따라 처리를 달리한다. 이 값은 현재 direct, indirect, doubly indirect 중 어떤 indexing 중인지를 나타낸다. level이 0이 아닌 경우에는 indirect block을 disk sector에 할당해야 한다. 이 sector도 buffer cache에 올려준다. single이냐 doubly냐에 따라서 필요한 처리가 다르다. 성공적으로 할당하는 경우에 true를 return하고 실패하는 경우엔 false를 return한다.
    1. level = 0인 경우 : direct다. inode\_allocate() data block을 할당한 것처럼 block 1개를 할당한다.
    2. level = 1인 경우 : single indirect다. remain\_index만큼의 sector를 할당해야 한다. 이때 single indirect 이므로 추가 indirect block은 필요하지 않다. 따라서 bound만큼의 sector를 할당하면 된다. 재귀적으로 호출해서 direct에 대해서 처리한다.
    3. level = 2인 경우 : doubly indirect이므로 indirect block이 더 필요하다. indirect block을 저장하기 위해서 bound만큼의 sector를 추가로 할당한다. 함수를 재귀적으로 호출해서 single indirect와 direct에 대해 처리한다.
  + void inode\_free() : disk sector를 free하는 함수다. file의 size가 큰 경우 indirect block sector 들도 모두 free해야하기 때문에 direct와 다른 처리가 필요하다. 함수는 다음과 같이 작동한다. level은 allocate하는 경우와 같다.
    1. bytes\_to\_sector() 함수를 사용해서 free할 sector가 몇 개인지 확인한다. remain\_index로 free할 sector의 개수를 관리한다.
    2. direct sector를 free\_map\_release() 함수를 사용해서 free한다.
    3. direct sector를 free하고도 아직 free할 sector가 남아있는 경우에 single indirect sector를 inode\_free\_indirect() 함수를 level=1로 호출해서 free한다.
    4. single indirect sector를 free하고도 아직 free할 sector가 남아있는 경우에 doubly indirect sector를 inode\_free\_indirect() 함수를 level=2로 호출해서 free한다.
  + void inode\_free\_indirect() : indirect block sector를 free하는 함수다. level이 0이 아닌 경우엔 indirect block sector를 free하는 것이 필요하고 level이 0인 경우엔 data sector를 free해야한다. indirect block sector는 buffer cache를 통해서 읽는다.
* Subdirectory
  + void parse\_path() : path 문자열을 인자로 받아서 directory와 filename으로 split하는 함수다. 마지막 ‘/’이 등장하기 전까지의 문자열을 directory로 저장하고 마지막 ‘/’ 뒤에 있는 문자열을 filename으로 저장한다.
  + bool dir\_is\_empty() : directory를 인자로 받아서 directory entry들을 검사한다. 사용 중인 entry가 있다면 false를 반환한다. 사용 중인 entry가 없다면 true를 반환한다. false가 반환된다면 이 directory는 삭제할 수 없다.
  + struct dir\* dir\_open\_path() : subdirectory를 open하는 함수이다. path를 인자로 받아서 absolute path와 relative path에 대한 처리를 수행한다. path의 ‘/’와 ‘/’ 사이에 있는 문자열이 이름인 directory를 차례로 열고 닫으면서 마지막 ‘/’ 뒤에 있는 directory까지 연다. 삭제된 경우에 대해서 예외처리를 수행한다.
  + bool inode\_create() : 함수 인자에 flag를 추가하여 file뿐만 아니라 directory도 새로 만들 수 있도록 한다. 이때 directory임을 저장하기 위해서 inode-dis에 is\_dir 변수를 추가해서 사용한다. 이때 buffer cache에도 write해서 buffer cache에 올린다.
  + 어떤 inode가 directory인지 판별해주는 함수로 inode.c에 inode\_is\_directory() 함수를 추가하였다. 또한 이 inode가 삭제된 inode인지 판별하기 위해서 inode\_is\_removed() 함수도 추가하였다. 이 함수들은 inode-disk와 inode의 변수를 참조하여 값을 반환한다.
  + root directory 이외의 directory가 생긴 것에 대한 처리를 수행하기 위해서 directory.c의 dir\_create(), dir\_open(), lookup(), dir\_lookup(), dir\_add() 함수들을 수정했다.
  + directory.c의 dir\_remove()함수를 수정하여 비어 있지 않은 directory의 경우 삭제를 수행할 수 없도록 수정하였다.
  + directory에 path가 생겼기 때문에 path를 parsing하고 directory를 open하는 경우에 새로 구현한 parse\_path()와 dir\_open\_path() 함수를 사용하여 처리하였다. 이와 같은 방법으로 수정한 함수는 filesys.c의 filesys\_create(), filesys\_open(), filesys\_remove(), filesys\_chdir() 함수다.
  + inode\_create()와 filesys\_create() 함수의 경우 directory를 생성하기 위해서 함수 인자가 추가되었는데 이를 반영하여 이 함수들이 사용되는 곳의 호출을 수정하였다.
  + syscall.c에 directory 관련한 함수를 추가하거나 수정했다.
    1. chdir : filesys.c에 새로 구현한 filesys\_chdir를 호출한다.
    2. mkdir : is\_dir 인자를 true로 하여 filesys.c의 filesys\_create()를 호출한다.
    3. readdir : 인자로 받은 fd를 file descriptor에서 찾아 해당하는 inode가 directory인지 확인하고 dir\_readdir()함수를 호출한다.
    4. isdir : 인자로 받은 fd를 file descriptor에서 찾아 inode\_is\_directory() 함수를 호출한다.
    5. inumber : 인자로 받은 fd를 file descriptor에서 찾아 inode\_get\_inumber() 함수를 호출한다.
  + 이전에 구현했던 system call들이 상황이 바뀌면서 작동하지 않는 경우가 있어서 이를 수정하여 제대로 작동할 수 있도록 하였다.
    1. 이전 프로젝트에서 lock을 잡아주지 않았었던 함수들에 대해서 lock을 잡아주었다. lock을 잡지 않아도 test case들은 통과하였지만 lock을 잡아주는 것이 더 안전할 것이라고 생각하였다.
    2. int open() : directory를 open하는 경우에 대해서 처리해주었다.
    3. int read(), int write() : 이전 프로젝트에서는 끝 주소가 valid한지 확인하지 않았었는데 이를 확인하도록 수정하였다.
    4. void close() : directory를 close하는 경우에 대해서 처리해주었다.
  + thread 구조체에 배열로 선언되어 있던 file descriptor를 list로 바꿨다. 새로 file descriptor를 구현하기 위해서 file\_desc 구조체를 만들었다. 이 구조체는 list의 entry로 사용되면서 file이나 directory의 주소를 가지고 있다. 또한 이번 프로젝트를 구현하는데 필요한 변수인 struct dir\* cwd와 struct file\* executing\_file을 추가하였다. 이 변수들은 다른 thread 변수들과 함께 초기화한다.
  + struct file\_desc\* find\_file\_desc() : file descriptor list를 탐색하는 함수다. flag로 file을 찾는지 directory를 찾는지 입력 받아서 조건에 해당하면 해당하는 descriptor를 반환한다.
* Buffer cache
  + 텍스트이(가) 표시된 사진

    자동 생성된 설명struct bc\_entry\_t: buffer cache의 entry이다. data를 저장하기 위한 buffer 배열과 disk의 위치를 저장하기 위한 disk\_sector 변수를 만들었다. 또한 buffer cache 관리를 위해서 valid, dirty, access 변수를 만들고 buffer cache 최대 크기만큼의 배열로 cache 변수를 만들었다.
  + struct lock bc\_lock : buffer cache의 concurrency를 보장하기 위해 사용하였다.
  + void bc\_init() : 프로그램 시작 시 buffer cache를 초기화하는 함수이다. buffer cache의 모든 flag 변수를 false로 설정한다. 이 함수는 filesys.c의 filesys\_init()함수에서 호출된다.
  + void bc\_read() : buffer cache에서 data를 읽기 위해 cache에 data가 올라와 있는지 검사한다. data가 cache에 올라와 있지 않은 경우엔 disk에서 새로 읽기 위해서 현재 buffer cache에 있는 data 중 evict할 data를 골라서 그 위치에 data를 저장하도록 한다. disk 상에서의 위치를 저장하고 flag들을 set한다. valid한 data이므로 valid는 true, 수정되지 않았으므로 dirty는 false로 설정한다. 또한 data에 접근되었으므로 access를 true로 설정한다. inode.c에서 file을 관리하기 위해서 사용된다.
  + void bc\_write() : buffer cache에 있는 data를 수정하는 함수이다. bc\_read()함수와 비슷하게 작동한다. 차이점이 있다면 data가 수정되므로 dirty flag를 true로 설정해 추후 flush 함수가 실행될 수 있도록 한다. inode.c에서 file을 관리하기 위해서 사용된다.
  + void bc\_flush() : buffer cache에 올라온 data를 disk에 다시 써주는 역할을 수행한다.
  + void bc\_flush\_all(): buffer cache의 모든 data에 대해서 valid와 dirty flag가 true인 경우에 bc\_flush() 함수를 실행한다. 이 함수는 filesys.c의 filesys\_done() 함수에서 호출된다.
  + struct bc\_entry\* bc\_lookup() : buffer cache에 찾고자 하는 data가 있는지 검사한다. data가 valid한지도 검사해서 valid하지 않은 경우엔 그 주소를 반환하지 않는다.
  + struct bc\_entry\* bc\_select\_victim() : buffer cache에서 data 하나를 evict하기 위해서 second chance 알고리즘을 사용했다. buffer cache에 valid하지 않은 data가 있는 경우 그 주소는 비어 있으므로 즉시 주소를 반환한다. 알고리즘을 구현하기 위해서 access flag를 사용하여 second chance를 부여한다. access bit가 false인 data를 후보로 선정하고 후보의 dirty flag가 true라면 bc\_flush() 함수를 호출하여 disk의 persistency를 유지했다.
  + inode.c의 inode\_read\_at(), inode\_write\_at()에서 disk와 직접 read, write하는 부분을 bc\_read(), bc\_write() 함수를 사용하여서 buffer cache를 통하도록 수정하였다.
  1. **시험 및 평가 내용**
* 텍스트이(가) 표시된 사진

  자동 생성된 설명**Src/filesys make grade 수행결과를 캡처 하여 첨부.**