**Pintos Project 5: Filesystem**

담당 교수 : 김영재

조 / 조원 : 강주언

개발 기간 : 2021. 12. 1 (수) ~ 2021. 12. 22. (수)

1. **개발 목표**

이번 프로젝트의 개발 목표는 Pintos의 파일 시스템을 구현하는 것이다. 기존의 basic file system에서 확장시켜 extensible file system을 구현해야한다. 또한, 파일 시스템이 create, open, close 등과 같은 호출을 disk operation으로 바꿔준다.

다음 세가지 구현에 초점을 맞추어 진행한다.

1. Extensible file & file growth

2. Subdirectory

3. Buffer cache (추가 구현)

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

basic file system은 contiguous Allocation strategy를 따른다. 이에 따른 문제점을 해결하고자, Extensible file & file growth가 가능한 strategy 인 multi-level indexed strategy를 구현하고자 한다.

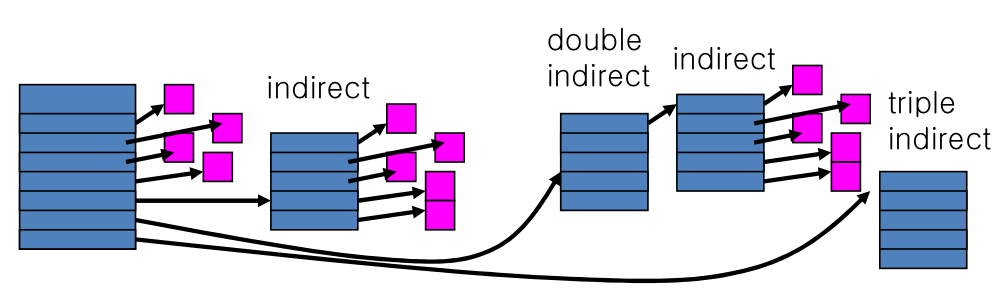
1. Subdirectory

basic file system은 단 하나의 root directory에 모든 파일이 존재하였다. 이러한 단일 디렉토리 구조에서 여러 sub directory가 가능한 환경을 구축하는 것이 이번 범위이다. 절대경로와 상대경로 두가지 경우를 모두 고려해야 하며, current directory(”.”), parent directory(“..”)도 신경써야 한다. filesystem 관련 system call도 구현하여야 한다.

1. Buffer cache  
   cache.c와 cache.h를 구현하여 파일시스템 기능들에서 block 단위의 read와 write 대신 buffer\_cache 단위의 read와 write이 쓰이도록 구현해야 한다.
   1. **개발 내용**

* Extensible file & file growth

Extensible file & file growth 를 만족시키고자, multi-level indexing을 구현해야 한다. 해당 Index struture는 다음 그림과 같다.



개발하여 구현된 파일의 inode 블록은 위의 그림과 비슷하게 되어 있다.

direct block(123개), single indirect block (1개), double indirect block(1개)

최대 manage 가능한 sector는 123 + 128 + 128\*128 이고, 이는 8MB 이상이다.

이처럼 Multi-Level Indexing 방법을 사용하면 Extensible file & file growth가 보장되지만, 주소를 계산하기 위해 포인터의 indirect block을 읽어야 하고 indirect block들을 main memory에 cached 해야한다.

* Subdirectory

directory entry의 첫 번째에는 parent directory 관련 정보를 저장한다. Basic Filesystem 은 root directory를 저장하는 disk block 이 있는데, 여기엔 1개 이상의 directory entry 를 저장할 수 있다. 그리고 이 directory entry들이 의미하는 것은 root directory 밑에 있는 디렉토리 또는 파일이다. Sector number로 directory를 구분하는데, 첫 번째 entry에 들어있는 sector number가 parent directory의 sector number가 되고, parent directory path segment 를 할 때에는 lookup 함수를 활용하여 parent directory로 연결시켜준다.

thread 구조체에 cwd를 추가하여, 쓰레드의 CWD를 기록할 수 있게 구현한다.

파일인지 디렉토리인지 구별이 필요하므로 bool is\_dir를 적절히 추가한다.

* Buffer cache

Buffe cache는 명세서를 참고하여, cache.c와 cache.h를 구현한다.

Buffer cache eviction은 buffer cache 공간이 꽉 찼을때나 OS가 종료될 때이다.

Buffer cache eviction 될 때 dirty bit가 set되어 있다면 flush 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

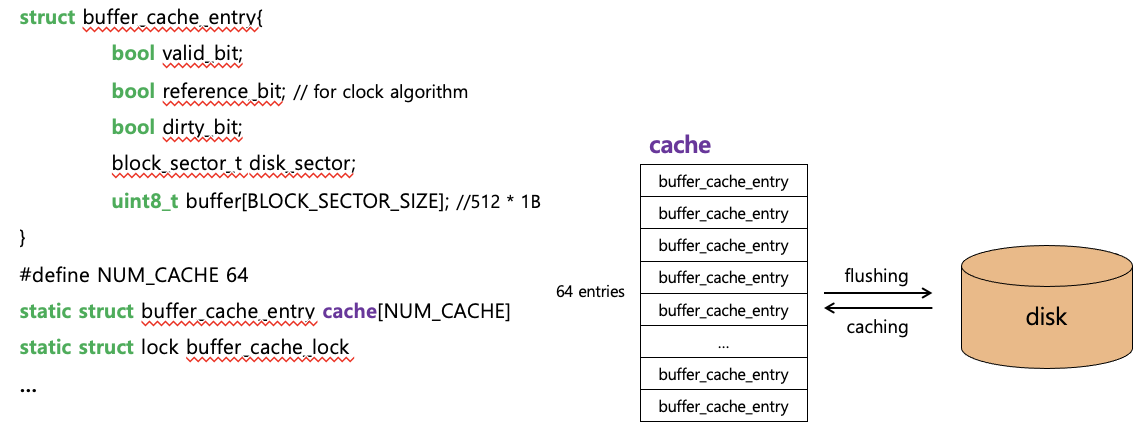
|  |  |
| --- | --- |
| 기간 | 일정 |
| 12. 1 | 전체 강의를 통한 프로젝트의 전반적인 내용 이해 |
| 12. 2 ~ 12. 7 | 프로젝트 매뉴얼 정독 및 파일 시스템 수업 복습 |
| 12. 8 ~ 12. 13 | Pintos Filesystem 관련 search |
| 12. 17 ~ 12. 18 | Cache.c Cache.h 구현 및 buffer cache 토대 구축 |
| 12. 19 ~ 12. 21 | Extensible file & growth / Subdirectory 구축 |
| 12. 21 ~ 12. 22 | 보고서 작성 |

* 1. **개발 방법**

1. Buffer Cache 구현

**- filesys/cache.h & filesys/cache.c**

구현에 필요한 여러 헤더파일들과 함수들을 선언하고



buffer\_cache\_entry 구조체를 위와 같이 설정하고, 다음 함수들을 정의한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**- filesys/filesys.c & filesys/filesys.h**

filesys\_init 과 filesys\_done 에서 buffer\_cache 부분으로 수정한다.

- filesys/inode.c & filesys/inode.h

inode open/remove/write\_at 등에서 buffer\_cache 부분으로 수정한다.

2. Extensible files & growth

**- filesys/inode.c & filesys/inode.h**

struct inode\_disk 구조체에서 기존 in memory에서 on-disk 형태로 수정한다.

(이후 inode\_disk 구조체 사용하는 코드 및 함수도 추가적으로 수정)

indexed inode 형태로 바꾸어 자유롭게 file grow 할 수 있게 수정

inode create, write, read 함수 등을 수정하여 External Fragmentation 해결

3. Subdirectory

**- threads/thread.h**

thread 구조체에 cwd 변수를 추가해준다. (current work directory)

**- filesys/directory.c & filesys/directory.h**

기존에 있던 헤더파일에 struct dir \*dir\_open\_path를 추가하여, root이외의 path들에 대해서도 구현한다.

subdirectory를 구현하면 빈 디렉토리도 발생할 수 있으므로, bool dir\_is\_empty를 추가하여, 해당 부분에 대한 처리를 구현한다.

dir\_lookup에서 parent (“..”)와 current(“.”)에 대한 directory를 처리한다.

dir\_add에서 child directory update에 대한 구현을 한다.

dir\_remove에서 비어있지 않은 directory는 지울 수 없게 구현한다.

**- filesys/filesys.c & filesys/filesys.h**

filesys\_create/open/remove 시에, subdirectory를 고려하여 수정한다.

4. Synchronization

**- userprog/syscall.c, userprog/process.c …**

우선 FILESYS에 해당하는 syscall 들을 선언 및 정의해준다.

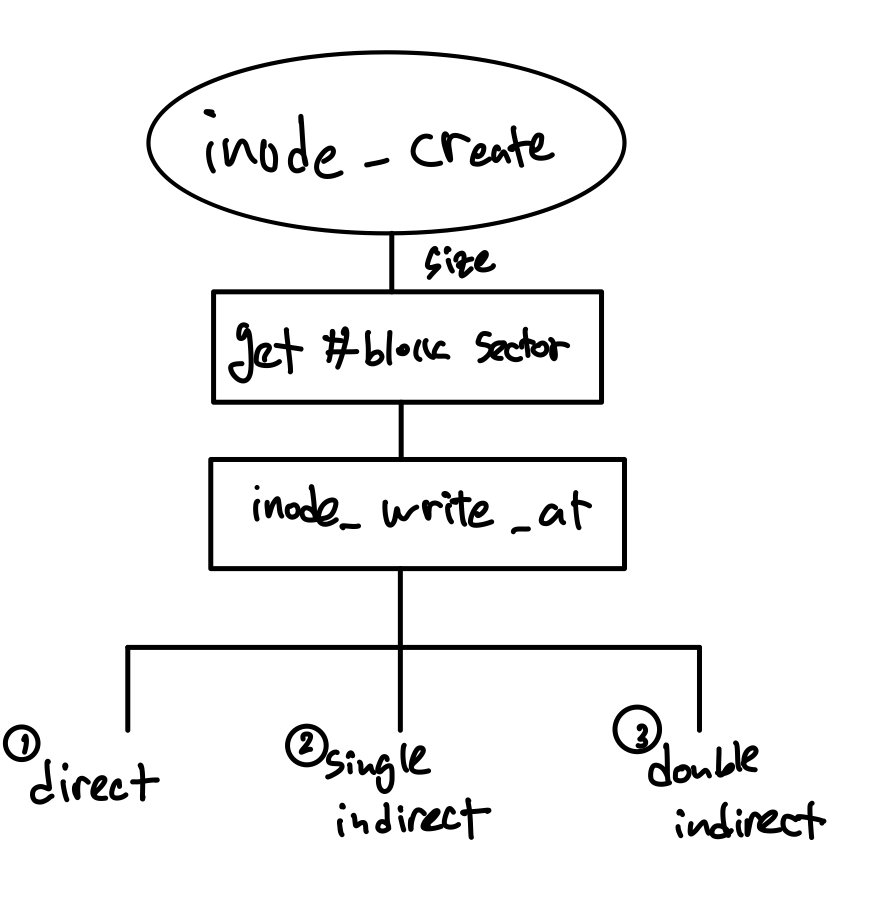
텍스트이(가) 표시된 사진

자동 생성된 설명

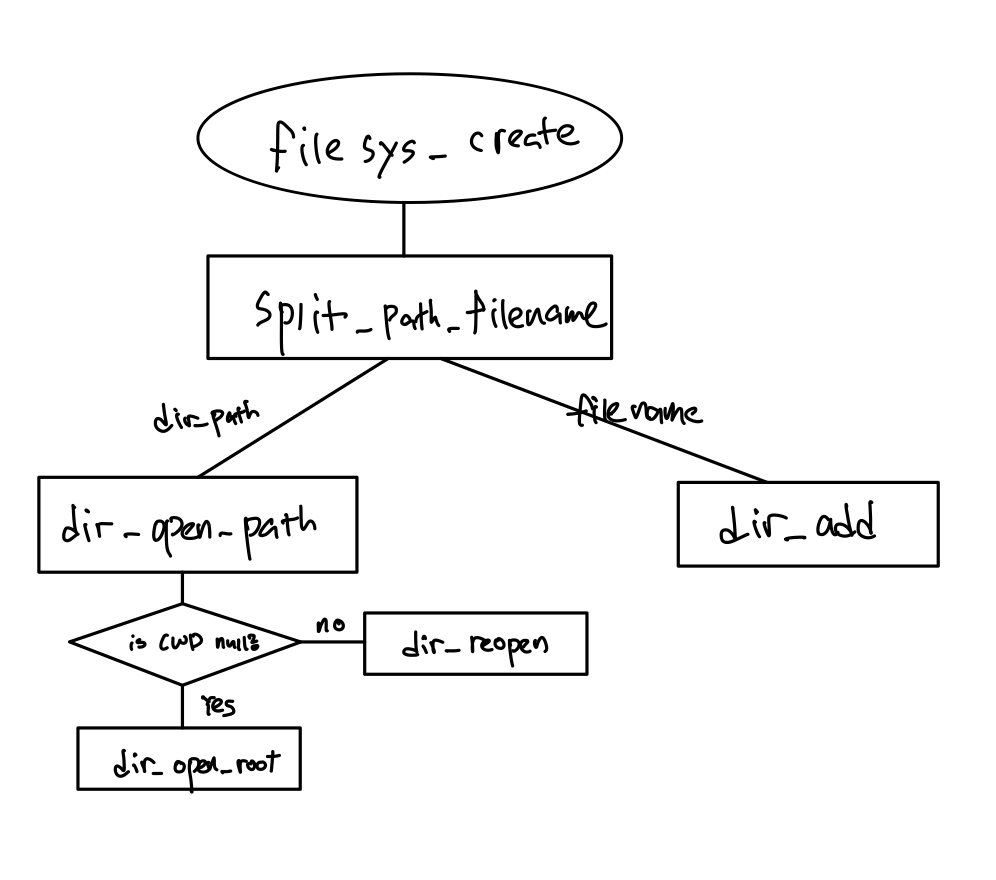
shared data 에 엑세스 하는 critical section에 해당하는 부분에 대하여, atomicity 유지하기 위한 lock & unlock을 해준다. 새롭게 추가 되는 함수 및 코드에 맞추어 synchronization을 해준다. Reader writer policy도 고려한다.

1. **연구 결과**
   1. **Flow Chart**

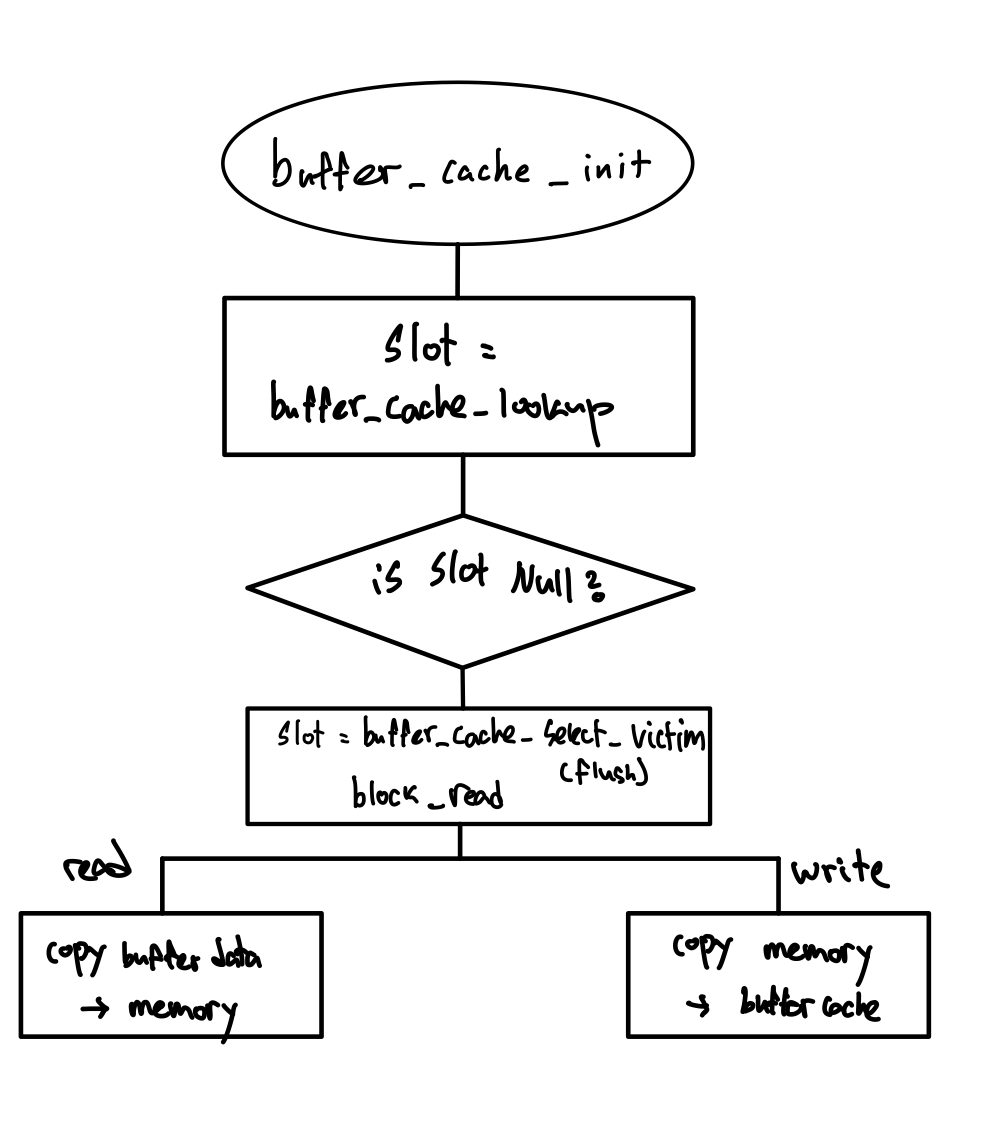
* Extensible file & file growth



* Subdirectory



* Buffer cache



* 1. **제작 내용**

**\* 제작 대하여 내용이 부족한 부분은 “3-B 개발 방법” 을 참고**

**\* 핵심 코드 위주로 설명**

* Extensible file & file growth

아래의 조건을 만족시키기 위한 double indirect 구현한다.

텍스트이(가) 표시된 사진

자동 생성된 설명(123 + 128 + 128\*128 sectors)

struct inode\_disk

{

/\*\* Data sectors \*/

block\_sector\_t direct\_blocks[123];

block\_sector\_t indirect\_block;

block\_sector\_t doubly\_indirect\_block;

…

…

};

indexed inode 형태로 바꾸어 자유롭게 file grow 할 수 있게 수정하는 것은 index\_to\_sector() 함수로 구현하였다.

static block\_sector\_t

index\_to\_sector (const struct inode\_disk \*idisk, off\_t index)

{

off\_t index\_base = 0, index\_limit = 0; // base, limit for sector index

block\_sector\_t ret;

// (1) direct blocks

index\_limit += 123 \* 1;

if (index < index\_limit) {

return idisk->direct\_blocks[index];

}

index\_base = index\_limit;

// (2) a single indirect block

index\_limit += 128;

if (index < index\_limit) {

struct inode\_indirect\_block\_sector \*indirect\_idisk;

indirect\_idisk = calloc(1, sizeof(struct inode\_indirect\_block\_sector));

buffer\_cache\_read (idisk->indirect\_block, indirect\_idisk);

ret = indirect\_idisk->blocks[ index - index\_base ];

free(indirect\_idisk);

return ret;

}

index\_base = index\_limit;

// (3) a single doubly indirect block

index\_limit += 128 \* 128;

if (index < index\_limit) {

// first and second level block index, respecitvely

off\_t index\_first = (index - index\_base) / 128;

off\_t index\_second = (index - index\_base) % 128;

// fetch two indirect block sectors

struct inode\_indirect\_block\_sector \*indirect\_idisk;

indirect\_idisk = calloc(1, sizeof(struct inode\_indirect\_block\_sector));

buffer\_cache\_read (idisk->doubly\_indirect\_block, indirect\_idisk);

buffer\_cache\_read (indirect\_idisk->blocks[index\_first], indirect\_idisk);

ret = indirect\_idisk->blocks[index\_second];

free(indirect\_idisk);

return ret;

}

// (4) what up?

return -1;

}

이후 indirect 를 구현하기 위하여 아래의 두 함수를 정의하여 사용하였다.

static bool inode\_reserve (struct inode\_disk \*disk\_inode, off\_t length);

static bool inode\_deallocate (struct inode \*inode);

해당 함수의 자세한 사항은 코드에서 직접 확인하면 된다.

* Subdirectory

inode.c에서 inode\_disk 구조체에 bool is\_dir를 추가한 뒤 아래와 같이 여러 함수에서 활용을 시켜준다.

filesys.h / directory.h / inode.h

bool filesys\_create (const char \*name, off\_t initial\_size, bool is\_dir);

bool dir\_add (struct dir \*, const char \*name, block\_sector\_t, bool is\_dir);

bool inode\_create (block\_sector\_t, off\_t, bool is\_dir);

struct dir \*dir\_open\_path를 추가하여, root이외의 path들에 대해서도 구현한다.

struct dir \*dir\_open\_path (const char \*path)

{

// copy of path, to tokenize

int l = strlen(path);

char s[l + 1];

strlcpy(s, path, l + 1);

// relative path handling

struct dir \*curr;

if(path[0] == '/') { // absolute path

curr = dir\_open\_root();

}

else { // relative path

struct thread \*t = thread\_current();

if (t->cwd == NULL) // may happen for non-process threads (e.g. main)

curr = dir\_open\_root();

else {

curr = dir\_reopen( t->cwd );

}

}

// tokenize, and traverse the tree

char \*token, \*p;

for (token = strtok\_r(s, "/", &p); token != NULL;

token = strtok\_r(NULL, "/", &p))

{

struct inode \*inode = NULL;

if(! dir\_lookup(curr, token, &inode)) {

dir\_close(curr);

return NULL; // such directory not exist

}

struct dir \*next = dir\_open(inode);

if(next == NULL) {

dir\_close(curr);

return NULL;

}

dir\_close(curr);

curr = next;

}

// prevent from opening removed directories

if (inode\_is\_removed (dir\_get\_inode(curr))) {

dir\_close(curr);

return NULL;

}

return curr;

}

이외의 subdirectory 에 맞게 개발 방법에서 설명한대로 구현한다. 자세한 내용은 코드를 열람하면 된다.

* Buffer cache

**-** 개발 방법에 대해서 기재한 대로 구현하였다.

filesys/cache.h은 생략하고, filesys/cache.c에 대한 코드를 살펴보면 다음과 같다.

1) buffer cache init

void buffer\_cache\_init (void){

lock\_init (&buffer\_cache\_lock);

size\_t i;

for (i = 0; i < NUM\_CACHE; ++ i){

cache[i].valid\_bit = false;

}

}

2) buffer cache terminate

void buffer\_cache\_terminate (void){

lock\_acquire (&buffer\_cache\_lock);

size\_t i;

for (i = 0; i < NUM\_CACHE; ++ i){

if (cache[i].valid\_bit == false) continue;

buffer\_cache\_flush( &(cache[i]) );

}

lock\_release (&buffer\_cache\_lock);

}

3) buffer cache read

void buffer\_cache\_read (block\_sector\_t sector, void \*target){

lock\_acquire (&buffer\_cache\_lock);

struct buffer\_cache\_entry \*slot = buffer\_cache\_lookup (sector);

if (slot == NULL) {

// cache miss: need select\_victimion.

slot = buffer\_cache\_select\_victim ();

// fill in the cache entry.

slot->valid\_bit = true;

slot->disk\_sector = sector;

slot->dirty = false;

block\_read (fs\_device, sector, slot->buffer);

}

// copy the buffer data into memory.

slot->reference\_bit = true;

memcpy (target, slot->buffer, BLOCK\_SECTOR\_SIZE);

lock\_release (&buffer\_cache\_lock);

}4) buffer cache write

void buffer\_cache\_write (block\_sector\_t sector, const void \*source){

lock\_acquire (&buffer\_cache\_lock);

struct buffer\_cache\_entry \*slot = buffer\_cache\_lookup (sector);

if (slot == NULL) {

// cache miss: need select\_victimion.

slot = buffer\_cache\_select\_victim ();

ASSERT (slot != NULL && slot->valid\_bit == false);

// fill in the cache entry.

slot->valid\_bit = true;

slot->disk\_sector = sector;

slot->dirty = false;

block\_read (fs\_device, sector, slot->buffer);

}

// copy the data form memory into the buffer cache.

slot->reference\_bit = true;

slot->dirty = true;

memcpy (slot->buffer, source, BLOCK\_SECTOR\_SIZE);

lock\_release (&buffer\_cache\_lock);

}

5) buffer cache lookup

struct buffer\_cache\_entry\* buffer\_cache\_lookup (block\_sector\_t sector){

size\_t i;

for (i = 0; i < NUM\_CACHE; ++ i){

if (cache[i].valid\_bit == false) continue;

if (cache[i].disk\_sector == sector) { // cache hit

return &(cache[i]);

}

}

return NULL; // cache miss

}

6) buffer cache select victim

struct buffer\_cache\_entry\* buffer\_cache\_select\_victim (void){

static size\_t clock = 0;

while (true) {

if (cache[clock].valid\_bit == false) { // found an empty slot -- use it

return &(cache[clock]);

}

if (cache[clock].reference\_bit) { // give a second chance

cache[clock].reference\_bit = false;

}

else break;

clock ++;

clock %= NUM\_CACHE;

}

// select\_victim cache[clock]

struct buffer\_cache\_entry \*slot = &cache[clock];

if (slot->dirty) { // write back into disk

buffer\_cache\_flush (slot);

}

slot->valid\_bit = false;

return slot;

}

2) buffer cache flush

void buffer\_cache\_flush (struct buffer\_cache\_entry \*entry)

{

if (entry->dirty) {

block\_write (fs\_device, entry->disk\_sector, entry->buffer);

entry->dirty = false;

}

}

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

텍스트이(가) 표시된 사진

자동 생성된 설명

Total Testing Score가 99.2% 나왔지만,텍스트이(가) 표시된 사진

자동 생성된 설명

이번 프로젝트 35개

채점에 대해서는

모두 pass 를 하였다.