**Pintos Project 5: Filesystem**

담당 교수 : 김영재

조 / 조원 : 4반

개발 기간 : 11.29 ~ 12.22

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

기존의 파일시스템은 파일을 contiguous하게 밖에 관리를 하지 못하기 때문에, 이를 sector단위로 나누어서 external fragmentation을 해결해야 한다. 또한 기존에는 파일의 사이즈가 고정된 채로 growth하는 경우를 처리하지 못했는데, 이를 해결하기 위해 file growth를 해결한다. 또한 directory안에 directory가 존재하는 subdirectory를 구현한다. 새로운 시스템 콜 chdir, mkdir, readdir, isdir, inumber를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Extensible file & file growth

각 data block을 파일의 inode에서 접근할 수 있도록 코드를 작성해 Extensible file을 구현한다. 이를 통해 contiguous하게 파일이 저장되어 외부단편화가 일어나는 문제를 해결할 수 있다. 또한 기존의 file system은 파일의 사이즈가 늘어날 때 이를 handling하지 못했는데, 이를 block단위로 쪼개진 메모리에 grow하고자 하는 내용을 비어 있는 block에 작성하거나 채워져 있는 block을 disk로 evict하여 메모리에 넣어주는 동작을 구현하여 해결한다.

1. Subdirectory

디렉토리 안에 추가적인 디렉토리가 존재할 수 있도록 구현을 한다. 또한 directory관련 시스템 콜을 구현한다. 이는 chdir, mkdir, readdir, isdir, inumber가 존재한다. 그리고 Subdirectory구현을 위해 기존에 구현되어 있던 시스템 콜들 중 open과 remove를 수정한다. 이를 통해 open시스템 콜을 사용하여 디렉토리를 오픈하거나 remove를 통해 디렉토리를 삭제하는 동작을 수행할 수 있을 것이다.

1. Buffer cache

매번 Input Output마다 디스크에 접근하여 데이터를 불러오고 쓰는 작업은 매우 비효율적이기 때문에 disk보다 상위 계층에 존재하는 메모리인 Buffer cache를 구현한다. 이를 통해 효율적인 IO를 수행할 수 있을 것이며 이를 위해 cache.c, cache.h를 filesys디렉토리에 추가한다. 캐시는 효율적이지만 용량이 작은 메모리이기 때문에 타당성이 있는 eviction알고리즘이 필요하고 이는 LRU알고리즘을 사용하여 구현한다.

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

index structure는 inode에서의 구조를 의미하는데 inode자체가 block을 가리키는 direct block과 inode가 또 다른 table을 가리켜 이 table이 block을 가리키는 indirect block으로 나뉜다. 이는 128개의 direct\_blocks index로 16384의 sector를 가리킬 수 있어야 이번 프로젝트를 정상적으로 진행할 수 있기 때문이다. 이를 통해 direct\_blocks[128]크기의 배열로 16384개의 sector를 담을 수 있다.

* Subdirectory
  + Directory entry 관리 방법

기존의 filesystem은 단 하나의 디렉토리(root directory)에 모든 파일이 존재하는 구조인데, 한 디렉토리에 또 다른 디렉토리가 존재할 수 있는 subdirectory를 구현한다. 쓰레드 별로 가지는 list인 file\_desc에 디렉토리를 표현할 수 있는 dir을 추가하였고, 이를 통해 open이나 close등의 시스템 콜에서 디렉토리를 열고 닫을 수 있는 동작을 할 수 있도록 하였다. 또한 디렉토리 관련 시스템콜을 구현하였다. 디렉토리를 생성하는 mkdir, 주어진 경로의 directory를 열고 이를 현재 thread의 cwd로 바꿔주는 chdir, 해당 디렉토리에 해당하는 inode를 찾는 readdir, 디렉토리인지 단순 파일인지를 구분하는 isdir, 디렉토리 별 유니크한 식별 번호를 반환하는 inumber를 구현하였다.

* Buffer cache
  + Buffer cache eviction 방식
  + Buffer cache flush 방식

기존에 disk에 IO작업을 접근하는 block\_write, block\_read대신 buffer\_cache\_write, buffer\_cache\_read로 IO작업을 빠르게 수행할 수 있는 상위 메모리인 buffer cache에 읽고 쓰는 작업을 하는 API를 구현해 효율성을 높였다. buffer cache구조체에 reference bit을 두어 접근 여부를 판단하였고 이를 통해 LRU알고리즘으로 cache eviction을 진행하였다. 또한 flush작업은 write back방식에서 캐시의 해당 블록이 수정이 된 경우 dirty bit가 세팅이 되고 만약 인자로 받은 entry가 dirty bit가 세팅이 된, 캐시 내에서 수정이 되었고 아직 disk에 이 작업이 반영이 되지 않은 경우 disk에 직접 block\_write를 해주었고 이 때 update를 하였으므로 다시 해당 entry의 dirty bit를 false로 세팅하는 방식으로 구현하였다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**

11.29 ~ 12.15 : filesys디렉토리 파일 수정, cache구현

12.15 ~ 12.20 : syscall수정

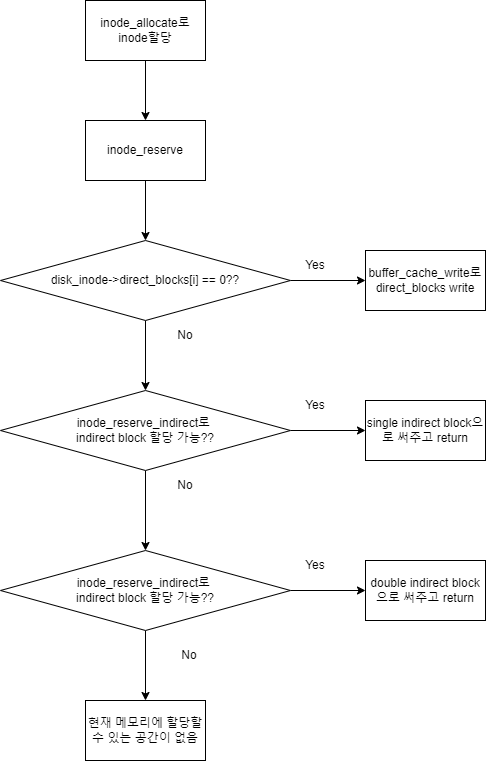
12.22 : 보고서 작성

* 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)**

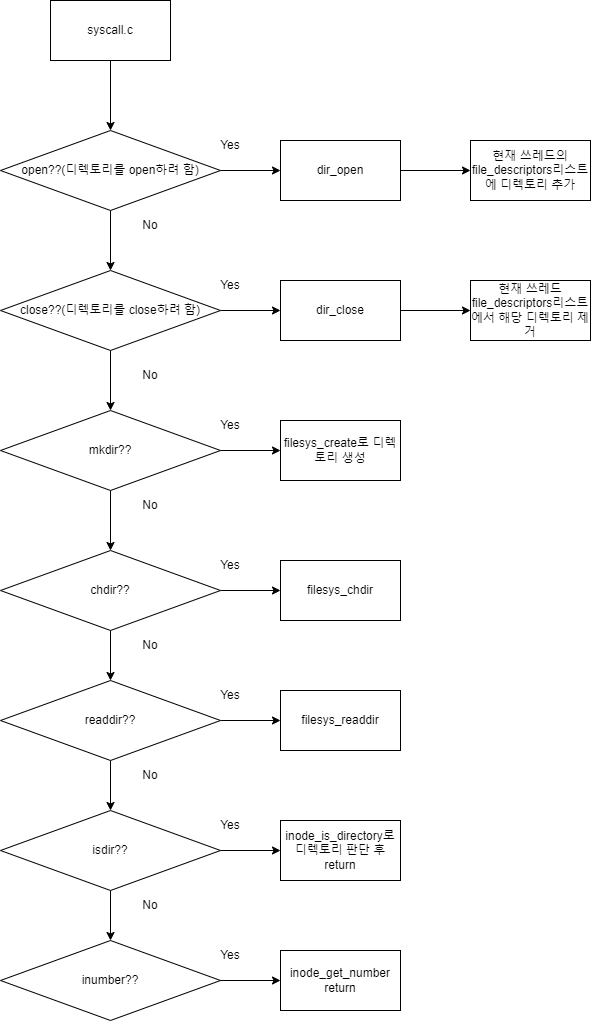
filesys디렉토리에 buffer cache와 이에 대한 API를 담고 있는 cache.c, cache.h를 만든다. buffer cache의 엔트리를 나타내는 bc\_entry에는 해당 엔트리가 유효한지를 나타내는 valid\_bit, 최근에 접근이 되었는지를 보여주는 reference\_bit, 캐시 내에서 수정이 되었는지를 나타내는 dirty\_bit, 어느 섹터를 가리키고 있는지를 나타내는 disk\_sector, 내용을 나타내는 buffer배열이 존재한다. 이 구조체를 가지고 bc\_entry cache[NUM\_CACHE]를 하여 캐시를 만들었다. 또한 기존의 쓰레드 구조체에 할당되어 있는 파일 디스크립터 테이블 fd는 디렉토리 관련한 핸들링을 처리할 수 없었기에 process.h에 file\_desc라는 구조체를 따로 선언하였고 file\_desc리스트를 만들어 이를 통해 디렉토리 관련 시스템 콜과 기존에 구현이 되어 있던 open, close에서 디렉토리를 열고 닫을 수 있도록 처리를 해주었다. 그리고 directory.c에 경로 이름을 받으면 디렉토리 별로 split을 해줄 필요성이 있기에 이를 동작시켜주는 split\_file\_name함수도 추가하였다. 이외에 버퍼 캐시를 구현하였기 때문에 IO작업을 직접 disk에 수행하는 함수인 block\_write, block\_read대신 캐시에 읽고 쓰는 buffer\_cache\_write, buffer\_cache\_read로 대체하여 효율적인 IO작업을 수행할 수 있도록 구현하였다.

1. **연구 결과**
   1. **Flow Chart**

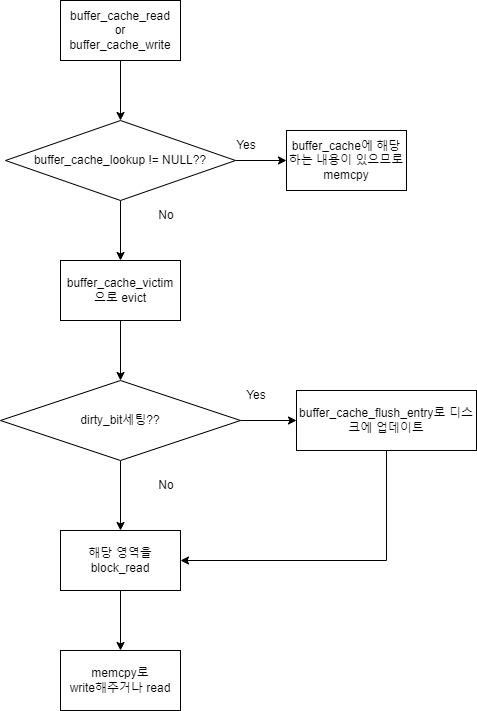
* **II. B. 개발 내용에 대한 Flow Chart를 작성**
* Extensible file & file growth



* Subdirectory



* Buffer cache  
  **구현한 경우만 작성**



* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**
* Extensible file & file growth

inode구조체를 다음과 같이 변경하였다.

Must be exactly BLOCK\_SECTOR\_SIZE bytes long. \*/

struct inode\_disk

{

block\_sector\_t direct\_blocks[DIRECT\_BLOCKS\_COUNT];

block\_sector\_t indirect\_block;

block\_sector\_t doubly\_indirect\_block;

bool is\_dir;

off\_t length; /\* File size in bytes. \*/

unsigned magic; /\* Magic number. \*/

};

direct\_block, indirect\_block, doubly\_indirect\_block element를 두어 extensible file에 대한 동작을 정상적으로 수행할 수 있도록 구현하였다.

static

bool inode\_allocate (struct inode\_disk \*disk\_inode)

{

return inode\_reserve (disk\_inode, disk\_inode->length);

}

inode를 할당하는 inode\_allocacte함수에서는 다음과 같이 inode\_reserve를 호출할 수 있도록 구현하였고

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

{

if(length < 0)

return false;

static char zeros[BLOCK\_SECTOR\_SIZE];

//if (length < 0) return false;

size\_t num\_sectors = bytes\_to\_sectors(length);

size\_t i, l;

l = min(num\_sectors, DIRECT\_BLOCKS\_COUNT \* 1);

for (i = 0; i < l; ++ i) {

if (disk\_inode->direct\_blocks[i] == 0) {

if(! free\_map\_allocate (1, &disk\_inode->direct\_blocks[i]))

return false;

buffer\_cache\_write (disk\_inode->direct\_blocks[i], zeros);

}

}

num\_sectors -= l;

if(num\_sectors == 0) return true;

l = min(num\_sectors, 1 \* INDIRECT\_BLOCKS\_PER\_SECTOR);

if(! inode\_reserve\_indirect (& disk\_inode->indirect\_block, l, 1))

return false;

num\_sectors -= l;

if(num\_sectors == 0) return true;

l = min(num\_sectors, 1 \* INDIRECT\_BLOCKS\_PER\_SECTOR \* INDIRECT\_BLOCKS\_PER\_SECTOR);

if(! inode\_reserve\_indirect (& disk\_inode->doubly\_indirect\_block, l, 2))

return false;

num\_sectors -= l;

if(num\_sectors == 0) return true;

ASSERT (num\_sectors == 0);

return false;

}

이 함수에서 해당하는 부분이 direct block인지 indirect block인지를 구분하여 buffer cache에 write하는 동작을 수행하였다. indirect block에 대한 함수는 다음과 같이 작성하였다.

static bool

inode\_reserve\_indirect (block\_sector\_t\* p\_entry, size\_t num\_sectors, int level)

{

static char zeros[BLOCK\_SECTOR\_SIZE];

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

// only supports 2-level indirect block scheme as of now

//ASSERT (level <= 2);

if (level == 0) {

if (\*p\_entry == 0) {

if(free\_map\_allocate (1, p\_entry) == false)

return false;

buffer\_cache\_write (\*p\_entry, zeros);

}

return true;

}

//struct inode\_indirect\_block\_sector indirect\_block;

if(\*p\_entry == 0) {

free\_map\_allocate (1, p\_entry);

buffer\_cache\_write (\*p\_entry, zeros);

}

buffer\_cache\_read(\*p\_entry, &indirect\_block);

//size\_t unit = (level == 1 ? 1 : INDIRECT\_BLOCKS\_PER\_SECTOR);

size\_t unit = INDIRECT\_BLOCKS\_PER\_SECTOR;

if(level == 1)

unit = 1;

//else

//unit = INDIRECT\_BLOCKS\_PER\_SECTOR;

size\_t i, l = DIV\_ROUND\_UP (num\_sectors, unit);

for (i = 0; i < l; ++ i) {

size\_t subsize = min(num\_sectors, unit);

if(! inode\_reserve\_indirect (& indirect\_block.blocks[i], subsize, level - 1))

return false;

num\_sectors -= subsize;

}

//ASSERT (num\_sectors == 0);

buffer\_cache\_write (\*p\_entry, &indirect\_block);

return true;

}

* Subdirectory

디렉토리 경로를 split해주는 함수는 다음과 같이 작성하였다.

void split\_file\_name(const char \*path, char \*directory, char \*filename){

int len = strlen(path);

len++;

char \*arr = (char \*)calloc(len, sizeof(char));

//char \*arr = (char \*)malloc(sizeof(char) \* len);

memcpy (arr, path, sizeof(char) \* len);

char \*dir = directory;

if(len > 1 && path[0] == '/' && dir)

\*dir++ = '/';

char \*tmp, \*ptr, \*last = "";

for (tmp = strtok\_r(arr, "/", &ptr); tmp != NULL; tmp = strtok\_r(NULL, "/", &ptr)){

int tmplen = strlen (last);

if (dir && tmplen > 0) {

memcpy (dir, last, sizeof(char) \* tmplen);

dir[tmplen] = '/';

dir += tmplen;

dir++;

}

last = tmp;

}

if(dir) \*dir = '\0';

memcpy (filename, last, sizeof(char) \* (strlen(last) + 1));

free (arr);

}

디렉토리를 나타내는 ‘/’으로 split하여 memcpy를 해주는 코드를 추가하였다.

subdirectory동작을 위해 수정해주어야 하는 시스템 콜 open, close와 디렉토리 관련 시스템 콜 mkdir, chdir, readdir, isdir, inumber은 다음과 같이 구현하였다.

int open(const char \*file){

int i;

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

struct file\_desc\* fd = palloc\_get\_page(0);

if(!fd)

return -1;

check\_user((const uint8\_t\*)file);

if(!is\_user\_vaddr(file) || file == NULL){

palloc\_free\_page(fd);

lock\_release(&sys\_lock);

exit(-1);

}

struct file\* fp = filesys\_open(file);

if(!fp){

palloc\_free\_page(fd);

return -1;

}

fd->file = fp;

struct inode \*inode = file\_get\_inode(fd->file);

if(inode != NULL && inode\_is\_directory(inode)) {

fd->dir = dir\_open( inode\_reopen(inode) );

}

else fd->dir = NULL;

if(fp == NULL)

return -1;

else{

for(i = 3; i < 128; i++){

if(thread\_current()->file\_des[i] == NULL){

if(strcmp(thread\_current()->name, file) == 0)

file\_deny\_write(fp);

else

file\_allow\_write(fp);

fd->id = i;

cur->file\_des[i] = fp;

list\_push\_back(&cur->file\_descriptors, &(fd->elem));

return i;

}

}

}

return -1;

}

open에서는 inode\_is\_directory를 사용하여 디렉토리인지를 따로 구분해주었고

void close(int fd){

struct file\* fp;

struct file\_desc\* file\_d = find\_file\_desc(thread\_current(), fd, FD\_FILE | FD\_DIRECTORY);

if(thread\_current()->file\_des[fd] == NULL)

exit(-1);

lock\_acquire(&sys\_lock);

if(file\_d && file\_d->file){

fp = thread\_current()->file\_des[fd];

thread\_current()->file\_des[fd] = NULL;

file\_close(fp);

if(file\_d->dir) dir\_close(file\_d->dir);

list\_remove(&(file\_d->elem));

palloc\_free\_page(file\_d);

}

lock\_release(&sys\_lock);

}

close에서는 현재 가리키고 있는 file descriptor가 디렉토리인지 file\_d->dir로 구분해주었으며

bool mkdir(const char \*filename)

{

check\_user((const uint8\_t\*) filename);

lock\_acquire (&sys\_lock);

bool res = filesys\_create(filename, 0, true);

lock\_release (&sys\_lock);

return res;

}

bool chdir(const char \*filename)

{

check\_user((const uint8\_t\*) filename);

lock\_acquire (&sys\_lock);

bool res = filesys\_chdir(filename);

lock\_release (&sys\_lock);

return res;

}

bool readdir(int fd, char \*name)

{

struct file\_desc\* file\_d;

bool res = false;

lock\_acquire (&sys\_lock);

file\_d = find\_file\_desc(thread\_current(), fd, FD\_DIRECTORY);

if (file\_d == NULL){

lock\_release(&sys\_lock);

return false;

}

struct inode \*inode;

inode = file\_get\_inode(file\_d->file);

if(inode == NULL || inode\_is\_directory(inode) == false) {

lock\_release(&sys\_lock);

return false;

}

res = dir\_readdir (file\_d->dir, name);

lock\_release(&sys\_lock);

return res;

}

bool isdir(int fd)

{

lock\_acquire (&sys\_lock);

struct file\_desc\* file\_d = find\_file\_desc(thread\_current(), fd, FD\_FILE | FD\_DIRECTORY);

bool res = inode\_is\_directory(file\_get\_inode(file\_d->file));

lock\_release (&sys\_lock);

return res;

}

int inumber(int fd)

{

lock\_acquire (&sys\_lock);

struct file\_desc\* file\_d = find\_file\_desc(thread\_current(), fd, FD\_FILE | FD\_DIRECTORY);

int res = (int)inode\_get\_inumber(file\_get\_inode(file\_d->file));

lock\_release (&sys\_lock);

return res;

}

디렉토리 관련한 시스템 콜은 다음과 같이 작성하였다.

* Buffer cache  
  **구현한 경우만 작성**

buffer cache에 대한 엔트리는 다음과 같이 구현하였다.

typedef struct BUFFER\_CACHE\_ENTRY{

bool valid\_bit;

bool reference\_bit;

bool dirty\_bit;

block\_sector\_t disk\_sector;

uint8\_t buffer[BLOCK\_SECTOR\_SIZE];

} bc\_entry;

유효한 지를 나타내는 valid\_bit, 참조 여부를 나타내는 reference\_bit, 수정 여부를 나타내는 dirty\_bit, 어느 섹터를 가리키고 있는지를 나타내는 disk\_sector, 내용을 담고 있는 buffer배열을 element로 하였다.

static bc\_entry cache[NUM\_CACHE];

cache.c에 다음과 같이 캐시를 선언하였다.

eviction을 할 때 LRU알고리즘으로 어떤 블락을 내쫓을지 선택하는 함수는 다음과 같이 작성하였다.

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

unsigned int clock = 0;

for(clock = 0;;clock++){

if(cache[clock].reference\_bit)

cache[clock].reference\_bit = false;

else

break;

if(!cache[clock].valid\_bit)

return &(cache[clock]);

if(clock == NUM\_CACHE)

clock = 0;

}

if(cache[clock].dirty\_bit)

buffer\_cache\_flush\_entry(&(cache[clock]));

cache[clock].valid\_bit = false;

return &(cache[clock]);

}

clock이라는 인덱스가 순차적으로 돌며 reference\_bit과 valid\_bit을 통해 어떤 블록을 내쫓을 지를 선택한 후 리턴하였다.

캐시 안에서 수정이 되어 있는 블록이라 만약 read write를 하는 경우 이를 disk에 업데이트를 해줘야 하고, 이를 판별해주는 flush함수는 다음과 같이 작성하였다.

void buffer\_cache\_flush\_entry(bc\_entry \*entry){

if(entry->dirty\_bit && entry != NULL){

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

entry->dirty\_bit = false;

}

}

void buffer\_cache\_flush\_all(void){

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

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

if(cache[i].valid\_bit == true)

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

}

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

}

캐시 안에서 수정 여부를 나타내는 dirty\_bit를 가지고 이를 수행하였다.

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

**텍스트, 명판이(가) 표시된 사진

자동 생성된 설명텍스트, 명판이(가) 표시된 사진

자동 생성된 설명**