**Pintos Project 5: Filesystem**

담당 교수: 박성용

조/조원: 20181687 정지석

개발 기간: 12/07 ~ 12/26:

1. **개발 목표**

File system을 개선하여 하나의 file이 연속적인 space를 차지하지 않도록 하고, file의 크기 확장을 가능하게 한다. 또한 root directory뿐만 아니라 하위 directory 또한 사용할 수 있게 한다. 마지막으로 memory에 cache를 만들어 read와 write를 수행하는 buffer cache 기능을 구현한다.

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

기존에는 file의 size가 고정되어 있어 정해진 block보다 큰 data를 넣는 것이 불가능했는데, file growth를 구현하면 file의 size를 늘리는 것이 가능해져 위의 문제를 해결할 수 있다. 또한 file이 연속적인 block에 저장되지 않아도 문제가 없도록 하여 disk 공간을 좀더 효율적으로 관리할 수 있게 된다.

1. Subdirectory

기존에는 root directory에만 file을 추가하는 것이 가능하였는데, subdirectory 구현을 통하여 root 밑에 여러 개의 하위 directory를 만드는 것이 가능해 졌다.

1. Buffer cache

Memory에 임시 공간을 만들어, 직접 disk에 access하여 page를 가져오는 대신 cache를 통해 page를 가져온다.

* 1. **개발 내용**

Extensible file & file growth

* + Index structure와 management에 대해서 기술

기존의 inode는 file의 시작 address와 size만으로 구성되어 있어 block space를 continuous하게 사용해야만 했다. 이 문제를 해결하기 위해 각 inode가 가리키는 block들을 index로 관리하여 continuous 하지 않은 space에서도 file의 확장이 가능하도록 한다.

* Subdirectory
  + Directory entry 관리 방법

새로운 directory entry를 생성한 뒤 첫번째 entry는 parent directory라 한다. 이는 Child node에서 parent node로의 이동을 가능케하기 위해서이다. File creation을 할 때 dir\_or\_not이라는 bool 변수를 통해 해당 file이 directory인지 아닌지 여부를 확인한다. 위의 방식을 통해 directory를 추가하고 관리하는 것이 가능하다.

* Buffer cache
  + Buffer cache eviction 방식

cache block들이 돌아가면서 valid 여부와 reference 되었는지 아닌지를 확인한다. Valid 하지 않을 경우 해당 block을 victim으로 삼게 된다. Valid할 경우 reference 여부를 확인하는 데, reference bit이 true이면 false로 수정만 해주고 다음으로 넘어간다(second chance). False일 경우 해당 block을 반환한다.

* + Buffer cache flush 방식

Flush를 할 때에는 해당 entry의 dirty bit을 확인해야 한다. True일 경우 write가 있었다는 뜻이므로, disk에 해당 block을 write한 뒤 dirty bit을 false로 바꿔준다.

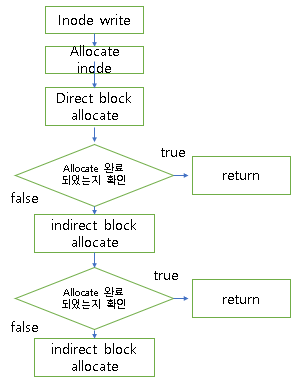
1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* 12/07~12/24: 기능 구현
* 12/25~12/26: 보고서 작성
  1. **개발 방법**

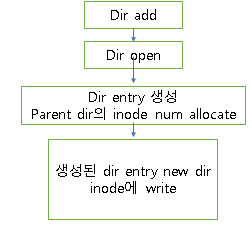
Filesys directory안에 buffer\_cache.c, buffer\_cache.h 파일을 생성하여 바로 disk에 access하여 read와 write를 실행하는 것이 아닌, cache memory에 read, write를 실행하도록 code를 수정한다.

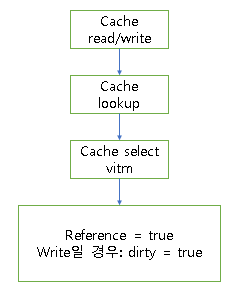
1. **연구 결과**
   1. **Flow Chart**

* Extensible file & file growth



* Subdirectory



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

Inode.c

* off\_t
* inode\_write\_at(struct inode \**inode*, const void \**buffer\_*, off\_t *size*,
* off\_t *offset*)
* {
* const uint8\_t \*buffer = buffer\_;
* off\_t bytes\_written = 0;
* uint8\_t \*bounce = NULL;
* *if* (inode->deny\_write\_cnt)
* *return* 0;
* *if* (byte\_to\_sector(inode, offset + size - 1) == -1) {
* *if*(alloc\_inode(&inode->data, offset + size) == false){
* *return* 0;
* }
* inode->data.length = offset + size;
* cache\_write(inode->sector, &inode->data);
* }

Inode\_write\_at() 함수에 위의 내용을 추가해 준다. Offset + size – 1의 위치를 입력으로 받는 byte\_to\_sector 함수를 실행해준다.

static block\_sector\_t

byte\_to\_sector(const struct inode \**inode*, off\_t *pos*)

{

    ASSERT(inode != NULL);

*if*(pos < 0)

*return* -1;

*else* *if* ( pos < inode->data.length)

*return* get\_sector\_number(&inode->data, pos/BLOCK\_SECTOR\_SIZE);

*else*

*return* -1;

}

File의 size보다 작은 위치에 write를 하려는 경우에는 get\_sector\_number 함수를 return한다.

block\_sector\_t get\_sector\_number(const struct inode\_disk \**disk*, off\_t *n*)

{

    block\_sector\_t ret;

    struct indirect\_block \*tmp\_block;

*if* (n < 123)

*return* disk->direct\_blocks[n];

*else* *if* (n < 123 + 128) {

        tmp\_block = calloc(1, sizeof(struct indirect\_block));

        cache\_read(disk->indirect\_block, tmp\_block);

        ret = tmp\_block->pointers[n - 123];

        free(tmp\_block);

*return* ret;

    }

*else* *if* (n < 123 + 128 + (128 \* 128)) {

        off\_t firstlevel\_block = (n - 123 - 128) / 128;

        off\_t secondlevel\_block = (n - 123 - 128) % 128;

        tmp\_block = calloc(1, sizeof(struct indirect\_block));

        cache\_read(disk->double\_indirect\_block, tmp\_block);

        cache\_read(tmp\_block->pointers[firstlevel\_block], tmp\_block);

        ret = tmp\_block->pointers[secondlevel\_block];

        free(tmp\_block);

*return* ret;

    }

*return* -1;

}

Block의 index를 입력으로 받아서 direct block인지 indirect block인지, 아니면 double indirect block인지를 확인하여 그에 따라서 해당 block의 sector를 return한다.

Inode\_write\_at 함수에서 file의 size를 넘어가는 곳에서 write를 하려 할 때 alloc\_inode() 함수를 사용하여 file size growth를 실행한다.

bool

alloc\_inode(struct inode\_disk \**disk*, off\_t *file\_size*)

{

*if* (file\_size < 0)

*return* false;

    int size = DIV\_ROUND\_UP(file\_size,BLOCK\_SECTOR\_SIZE);

    int temp;

*if*(size < 123)

        temp = size;

*else*

        temp = 123;

*for* (int i = 0; i < temp; ++i) {

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

        {

*if* (free\_map\_allocate(1, &disk->direct\_blocks[i]) == false)

*return* false;

            cache\_write(disk->direct\_blocks[i], empty\_page);

        }

    }

    size -= temp;

*if* (size == 0)

*return* true;

*if*(size < 128)

        temp = size;

*else*

        temp = 128;

*if* (alloc\_inode\_indirect(&disk->indirect\_block, temp) == false)

*return* false;

    size -= temp;

*if* (size == 0)

*return* true;

*if*(size < 128 \* 128)

        temp = size;

*else*

        temp = 128 \* 128;

*if* (alloc\_inode\_double\_indirect(&disk->double\_indirect\_block, temp) == false)

*return* false;

    size -= temp;

*if* (size == 0)

*return* true;

*return* false;

}

Growth 시키려는 file의 size를 입력으로 받아 이를 sector 단위로 나눈 후, 총 몇개의 sector가 존재하는지 확인한다. Sector의 총 개수가 direct block보다 작다면 allocate 되지 않은 block들을 새로 allocate 해준다. allocate해야 할 block이 남는 경우에는 indirect block에 allocate한다. Allocate 해야 할 block이 아직도 더 남는 경우에는 alloc\_inode\_double\_indirect() 함수를 통해 나머지를 allocate 해준다.

bool

alloc\_inode\_double\_indirect(block\_sector\_t\* *block*, size\_t *size*)

{

    struct indirect\_block indirect\_block;

*if* (\*block == 0)

    {

        free\_map\_allocate(1, block);

        cache\_write(\*block, empty\_page);

    }

    cache\_read(\*block, &indirect\_block);

    int length= DIV\_ROUND\_UP(size, 128);

*for* (int i = 0; i < length; i++)

    {

        int alloc\_length;

*if*(size < 128)

            alloc\_length = size;

*else*

            alloc\_length = 128;

*if* (alloc\_inode\_indirect(&indirect\_block.pointers[i], alloc\_length) == false)

*return* false;

        size -= alloc\_length;

    }

    cache\_write(\*block, &indirect\_block);

*return* true;

}

* Subdirectory

Directory.c

bool

dir\_add(struct dir \**dir*, const char \**name*, block\_sector\_t *inode\_sector*, bool *dir\_or\_not*)

{

    struct dir\_entry e;

    off\_t ofs;

    bool success = false;

    ASSERT(dir != NULL);

    ASSERT(name != NULL);

*/\* Check NAME for validity. \*/*

*if* (\*name == '\0' || strlen(name) > NAME\_MAX)

*return* false;

*/\* Check that NAME is not in use. \*/*

*if* (lookup(dir, name, NULL, NULL))

*return* false;

*if* (dir\_or\_not)

    {

        struct dir \*new\_dir = dir\_open(inode\_open(inode\_sector));

*if* (new\_dir == NULL)

*return* 0;

        e.inode\_sector = inode\_get\_inumber(dir\_get\_inode(dir));

*if* (inode\_write\_at(new\_dir->inode, &e, sizeof(e), 0) != sizeof(e)) {

            dir\_close(new\_dir);

*return* 0;

        }

        dir\_close(new\_dir);

    }

*for* (ofs = 0; inode\_read\_at(dir->inode, &e, sizeof e, ofs) == sizeof e;

        ofs += sizeof e)

*if* (!e.in\_use)

*break*;

*/\* Write slot. \*/*

    e.in\_use = true;

    strlcpy(e.name, name, sizeof e.name);

    e.inode\_sector = inode\_sector;

    success = inode\_write\_at(dir->inode, &e, sizeof e, ofs) == sizeof e;

    done:

*return* success;

}

Dir\_add 함수에 dir\_ro\_not이라는 bool 변수를 입력 parameter로 추가하여 입력 받은 entry가 directory인지 아닌지 여부를 확인한다.

* Buffer cache

Buffer\_cache.c

void cache\_read(block\_sector\_t *sector*, void \**buffer*)

{

    lock\_acquire(&c\_lock);

    struct cache\_entry \*tmp = cache\_lookup(sector);

*if* (tmp == NULL) {

        tmp = cache\_select\_victim();

        tmp->valid = true;

        tmp->dirty = false;

        tmp->sector = sector;

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

    }

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

    tmp->referenced = true;

    lock\_release(&c\_lock);

}

void cache\_write(block\_sector\_t *sector*, const void \**buffer*)

{

    lock\_acquire(&c\_lock);

    struct cache\_entry \*tmp = cache\_lookup(sector);

*if* (tmp == NULL) {

        tmp = cache\_select\_victim();

        tmp->valid = true;

        tmp->dirty = false;

        tmp->sector = sector;

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

    }

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

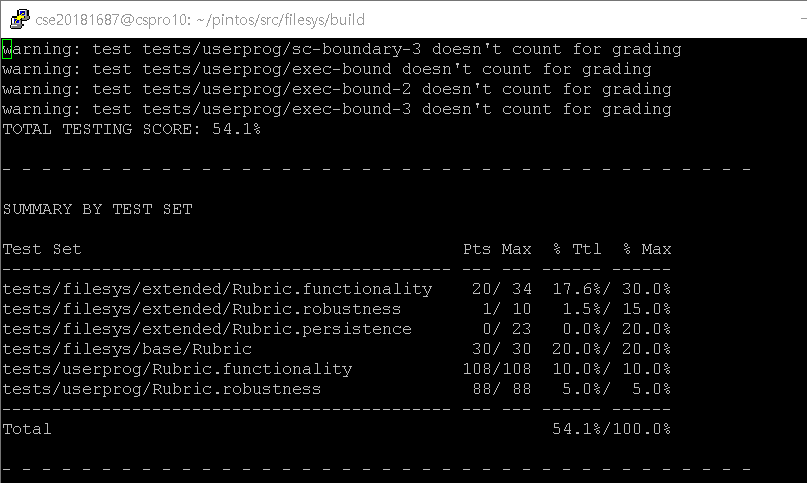
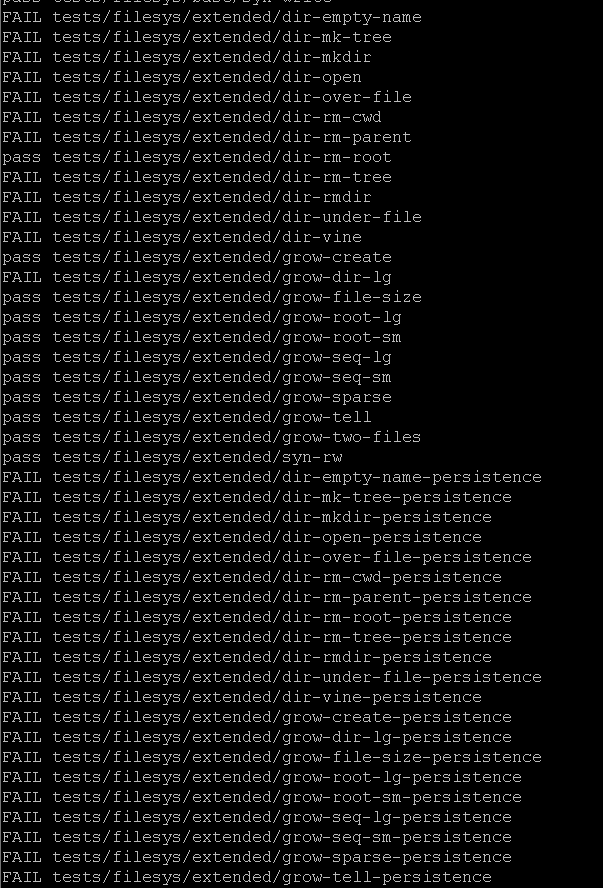
    tmp->referenced = true;

    tmp->dirty = true;

    lock\_release(&c\_lock);

}

cache\_read와 cache\_write 함수를 사용해 disk에서 실행되어야 할 read/write를 cache memory에서 대신 실행한다.

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