운영 체제 과제 보고서

project #4

과목명: 운영 체제

담당 교수: 원유집 교수님

담당 조교: 오준택 조교님

제출 일자: 2018/ 06/ 21

2014004411 김시완

2016025714 이동윤

Github address:

https://github.com/gimsiwan/Operating_system.git

목차

1. Buffer Cache

과제 목표:

디스크 블록을 메모리 영역에 둠으로써 파일의 입출력 응답시간을 줄인다.

과제 설명:

Buffer cache 는 디스크 블록을 캐싱하는 메모리 영역이다. 현재 핀토스에는 buffer cache 가 존재 하지 않아, 파일 입출력시 바로 디스크 입출력 동작을 수행하여 파일 입출력 시간이 오래 걸린다. Buffer cache 를 구현하여 파일 입출력 시간을 줄여 성능을 향상 시키고자 한다.

a. buffer_header 구조체 / 전역 변수

```
struct buffer_head{
    bool dirty;     //데이터 변경 체크
    bool check_use;     //사용 유무 체크
    int clock_bit;      //최근 접근 유무 체크
    struct lock buffer_lock;
    block_sector_t sector;      //disk sector 주소
    void* data;     //해당 buffer 주소
};
```

Buffer_head 구조체는 buffer cache entry 를 표현하기 위한 구조체로서, 데이터 변경 유무를 체크하는 dirty bit 와 사용중인지 유무를 판단하는 check_use bit, victim 정책에서 사용할 clock_bit, buffer data 에 접근을 할 때 동시 접근을 막기 위한 buffer_lock, disk 의 sector 주소를 표시하고 있는 sector, 해당 buffer 의 메모리 주소의 포인터 값을 갖고 있는 data 로 이루어져있다.

```
#define BUFFER_CACHE_ENTRY_NB 64

void * p_buffer_cache;  //메모리 공간의 버퍼 주소를 가르킨다.
struct buffer_head buffer_head[BUFFER_CACHE_ENTRY_NB];  //buffer cache의 각 entry를 관리하는 리스트
int clock_hand;     //victim buffer를 선택해주는데 사용
```

우리는 총 64 개의 버퍼를 사용할 것이므로 매크로를 통해 총 버퍼의 개수인 BUFFER_CACHE_ENTRY_NB 를 64 로 잡아준다. 전역 변수로 잡아준 p_buffer_cache 변수는 버퍼의 메모리 주소 공간을 가르키는 변수이며,

buffer_head array 는 각 버퍼마다 entry 의 배열을 나타내며, 마지막으로 clock_hand 는 victim page 를 정할 때 LRU 정책을 사용하기 위해 존재한다.

b. bc_init 함수 / bc_term 함수

```
void bc_init(void){
    p_buffer_cache = malloc(BLOCK_SECTOR_SIZE * BUFFER_CACHE_ENTRY_NB);    //최대 블록 개수만큼 buffer 메모리 할당
    clock_hand = 0;
    int i = 0;
    for(i=0;i<BUFFER_CACHE_ENTRY_NB;i++){        //buffer head entry 초기화
        buffer_head[i].dirty = false;
        buffer_head[i].check_use = false;
        buffer_head[i].clock_bit = 0;
        lock_init(&buffer_head[i].buffer_lock);
        buffer_head[i].data = p_buffer_cache + i*BLOCK_SECTOR_SIZE;
    }
}
```

Bc_init 함수는 선언한 전역 변수들을 초기화 해주는 함수로서, p_buffer_cache 에 64 개의 버퍼 메모리 공간을 할당해준다. (하나의 버퍼당 512(BLOCK_SECTOR_SIZE) 크기) 초기 clock_hand 값은 0 으로 초기화 시켜주어, 0 번째 index 부터 clock_bit 를 check 하게 만들어주며, 총 64 개의 buffer entry 를 모두 초기 값으로 초기화 시켜준다. Buffer_head 에서 data 값은 위에서 할당해준 메모리 공간에서 각자 자기 버퍼 주소 값을 지니고 있는다.

```
void bc_term(void){
    bc_flush_all_entries();
    free(p_buffer_cache);        //buffer 메모리 할당 해제
}
```

Bc_term 은 사용중인 모든 buffer 를 flush 함과 동시에 할당 해준 버퍼 메모리 공간을 해제 시켜주는 함수이다. 모든 buffer 를 flush 해주는 데에는 bc_flush_all_entries 함수를 호출한다.

c. bc_flush_entry 함수 / bc_flush_all_entries 함수

```
void bc_flush_entry(struct buffer_head *p_flush_entry){
    if(p_flush_entry->check_use){
        if(p_flush_entry->dirty){        //buffer의 데이터가 변경이 있을 경우 disk에 다시 써준다.
            block_write(fs_device, p_flush_entry->sector, p_flush_entry->data);
            p_flush_entry->dirty = false;
        }
        p_flush_entry->check_use = false;
    }
}
```

Bc_flush_entry 함수는 해당 하는 주소의 버퍼 메모리가 dirty 상태인 경우 디스크에 flush 해주는 함수로서, 먼저 해당 메모리가 dirty 상태인지 아닌지를 판단하기 위해 buffer entry 의 check_use bit 와 dirty bit 를 체크한다. 만약 해당 buffer 가 사용중이고, dirty 상태이라면 block_write 함수를 호출하여 해당하는 sector 주소에 해당 버퍼 메모리를 write 해주고, dirty bit 와 check_use bit 를 false 로 바꿔준다.

```
void bc_flush_all_entries(void){
    int i = 0;
    for(i=0;i<BUFFER_CACHE_ENTRY_NB;i++)
        bc_flush_entry(&buffer_head[i]);
}
```

Bc_flush_all_entries 는 64 개의 버퍼 중 dirty 상태인 버퍼를 체크하여 디스크에 flush 해주는 함수로서 반복문을 돌려 bc_flush_entry 함수를 총 64 번 호출 해준다.

d. bc_select_victim 함수 / bc_lookup 함수

```
struct buffer_head * bc_select_victim(void){    //버퍼에 빈 블록이 없는 경우에서 버퍼 블록 요청시 실행

    while(1){
        if(clock_hand == BUFFER_CACHE_ENTRY_NB)
            clock_hand = 0;
        if(buffer_head[clock_hand].clock_bit == 1){    //clock_bit가 1이면 0으로 변경 후 다음 버퍼 탐색
            buffer_head[clock_hand].clock_bit = 0;
            clock_hand ++;
        }
        else{    //clock_bit가 0인경우
            if(buffer_head[clock_hand].dirty)    //해당 버퍼가 더티인 경우
                bc_flush_entry(&buffer_head[clock_hand]);    //디스크에 버퍼 내용 저장
            clock_hand ++;
            return &buffer_head[clock_hand - 1];
        }
    }
}
```

Bc_select_victim 함수는 버퍼에 모든 공간이 사용 중일 경우 버퍼 메모리 공간에 대한 요청이 있을시 호출하는 함수이다. 모든 버퍼 메모리 공간이 사용 중이기 때문에, 우리는 하나의 버퍼를 디스크로 내려주어 메모리 공간을 만들어주어야 한다. 이때 디스크로 내려줄 메모리를 결정하는데 있어 전역 변수 clock_hand 와 buffer entry 의 clock_bit 를 사용한다. LRU 방식을 사용하였는데, clock_hand 의 index 값 부터 순차대로 탐색을 하면서, buffer entry 의 clock_bit 값이 1 인 경우 0 으로 바꿔주고 다음 번 버퍼의 탐색으로 넘어간다.

만약 0 일 경우는 해당 buffer 메모리가 dirty 인 경우 디스크에 해당 버퍼를 write 해주고, 해당 버퍼를 victim page 로 선택하여 반환해준다.

```
struct buffer_head * bc_lookup(block_sector_t sector){
    int i = 0;
    for(i=0;i<BUFFER_CACHE_ENTRY_NB;i++){    //버퍼를 탐색하여 sector 주소가 일치 할경우 해당 버퍼 리턴
        if(buffer_head[i].check_use == true && buffer_head[i].sector == sector)
            return &buffer_head[i];
    }
    return NULL;
}
```

Bc_lookup 함수는 인자로 주어진 sector 주소 값과 동일한 sector 주소값을 갖고 있는 buffer 가 존재하는지 검사하는 함수이다. 총 64 개의 버퍼를 모두 탐색하여 동일 값이 존재하면 해당 버퍼 entry 를 리턴하고, 존재 하지 않는다면 NULL 값을 리턴 해준다.

e. bc_read 함수

```
bool bc_read(block_sector_t sector_idx, void *buffer, off_t bytes_read, int chunck_size, int sector_ofs){
    struct buffer_head * bh;

    if((bh = bc_lookup(sector_idx))){    //해당 sector 주소 데이터를 갖고 있는 버퍼가 존재할 시
        bh->clock_bit = 1;    //접근 비트 1로 변경
        lock_acquire(&bh->buffer_lock);
        memcpy(buffer + bytes_read, bh->data + sector_ofs, chunck_size);    //버퍼에서 데이터를 읽어 유저 buffer에 저장
        lock_release(&bh->buffer_lock);
    }
    else{    //현재 버퍼에 해당 sector주소 데이터를 갖고 있는 버퍼가 없을시
        int i = 0;
        for(i=0;i<BUFFER_CACHE_ENTRY_NB;i++){
            if(buffer_head[i].check_use == false){    //사용하지 않는 버퍼가 있을시 해당 버퍼 사용
                buffer_head[i].check_use = true;
                bh = &buffer_head[i];
                break;
            }
        }
        if(i==BUFFER_CACHE_ENTRY_NB)
            bh = bc_select_victim();    //모든 버퍼 사용하고 있을시 victim buffer 지정
        bh->sector = sector_idx;
        bh->clock_bit = 1;
        block_read(fs_device, sector_idx, bh->data);    //버퍼에 디스크 내용 올려준다음
        lock_acquire(&bh->buffer_lock);
        memcpy(buffer + bytes_read, bh->data + sector_ofs, chunck_size);    //버퍼에서 데이터를 읽어 유저 buffer에 저장
        lock_release(&bh->buffer_lock);
    }

    return true;
}
```

Bc_read 함수는 buffer 메모리에서 데이터를 읽어 인자 buffer 에 저장해주는 함수로서, 먼저 인자로 주어진 sector 주소 값과 동일한 값을 가지고 있는 buffer 가 존재하는지 bc_lookup 함수를 통해 탐색해준다. 만약 존재한다면 memcpy 함수를 통해 해당 버퍼에서 바로 유저 buffer 로 데이터를 복사해주면

된다. 만약 존재하지 않고, 모든 버퍼 공간이 사용 중이라면 bc_select_victim 함수를 통해 buffer 공간을 만들어 준 다음, block_read 함수를 통해 디스크 내용을 버퍼로 읽어 온 다음 memcpy 처리를 해주었고, 사용 하지 않는 버퍼가 존재 할 시에는 해당 버퍼를 사용해주었다.

```
/* Number of bytes to actually copy out of this sector. */
int chunk_size = size < min_left ? size : min_left;
if (chunk_size <= 0)
  break;

bc_read(sector_idx, (void*)buffer, bytes_read, chunk_size, sector_ofs);
```

이제 inode_read_at 함수에서 파일 read 시 기존 함수를 지워주고 bc_read 함수를 통해 buffer 에서 읽어 올 수 있도록 변경시켜주었다.

f. bc_write 함수

```
bool bc_write(block_sector_t sector_idx, void *buffer, off_t bytes_written, int chunck_size, int sector_ofs){

  struct buffer_head * bh;

  if((bh = bc_lookup(sector_idx))){  //해당 sector 주소 데이터를 갖고 있는 버퍼가 존재할 시
    bh->clock_bit = 1;
    lock_acquire(&bh->buffer_lock);
    memcpy(bh->data + sector_ofs, buffer + bytes_written, chunck_size); //buffer의 데이터를 buffer cache에 기록
    bh->dirty = true;
    lock_release(&bh->buffer_lock);
  }
  else{
    int i = 0;
    for(i=0;i<BUFFER_CACHE_ENTRY_NB;i++){
      if(buffer_head[i].check_use == false){  //사용하지 않는 버퍼가 있을시 해당 버퍼 사용
        buffer_head[i].check_use = true;
        bh = &buffer_head[i];
        break;
      }
    }
    if(i==BUFFER_CACHE_ENTRY_NB)
      bh = bc_select_victim();  //모든 버퍼 사용하고 있을시 victim buffer 지정
    bh->sector = sector_idx;
    bh->clock_bit = 1;
    bh->check_use = true;
    block_read(fs_device, sector_idx, bh->data);  //버퍼에 디스크 내용 올려준다음
    lock_acquire(&bh->buffer_lock);
    memcpy(bh->data + sector_ofs, buffer + bytes_written, chunck_size);  //buffer의 데이터를 buffer cache에 기록
    bh->dirty = true;  //버퍼의 데이터가 변경이 되었으므로 dirty bit check
    lock_release(&bh->buffer_lock);
  }
  return true;
}
```

Bc_write 함수는 전반적으로 bc_read 함수와 동일하나, 유저 buffer 에 데이터를 써주는 대신 buffer 의 내용을 buffer 메모리에 저장해주고, 이로 인해 버퍼의 메모리 공간 데이터가 변경되었으므로 dirty bit 를 true 로 체크해준다.

```
···/* Number of bytes to actually write into this sector. */
···int chunk_size = size < min_left ? size : min_left;
···if (chunk_size <= 0)
·····break;
···bc_write(sector_idx, (void *)buffer, bytes_written, chunk_size, sector_ofs);
```

여기서도 마찬가지로 inode_write_at 함수에서 파일을 write 해줄 때 disk 로 바로 write 해주는 기존 함수를 지워주고, buffer 로 write 할 수 있도록 bc_write 함수를 이용해주었다.

2. Extensible File

과제 목표:

생성된 파일 크기가 고정되지 않고, 확장 가능하도록 변경

과제 설명:

현재 핀토스에서는 파일 생성 시 파일이 크기가 결정되고, 추후 변경이 불가능하다. 파일 생성 이후, 파일에 쓰기 동작을 하였을 때 디스크 블록을 할당받아 파일의 크기가 변경될 수 있도록 변경

a. Inode 구조체 / inode_disk 구조체 / 전역변수

```
struct inode
  {
    struct list_elem elem;              /* Element in inode list. */
    block_sector_t sector;              /* Sector number of disk location. */
    int open_cnt;                       /* Number of openers. */
    bool removed;                       /* True if deleted, false otherwise. */
    int deny_write_cnt;                 /* 0: writes ok, >0: deny writes. */
    struct lock extend_lock;
    // struct inode_disk data;          /* Inode content. */
  };
```

Inode 구조체에 inode 접근시 획득하는 세마포어 락을 추가시켜주고, data 를 inode_disk 를 통해 접근하도록 변경 해주기 때문에 data 변수를 삭제 해준다.

```
struct inode_disk
  {
    // block_sector_t start;            /* First data sector. */
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    uint32_t is_dir;
    block_sector_t direct_map_table[DIRECT_BLOCK_ENTRIES];//다이렉트로 접근할 디스크 블록의 번호 저장
    block_sector_t indirect_block_sec;//indirect로 접근할 디스크 블록의 번호 저장
    block_sector_t double_indirect_block_sec;//double indirect로 접근할 경우, 1차 index block 번호 저장
    // uint32_t unused[125];            /* Not used. */
  };
```

Inode_disk 구조체에 블락 주소를 모두 저장하도록 배열을 추가시켜준다. 블록 위치는 direct, indirect, double_indirect 세가지 방식으로 표현해주며, direct 방식으로 표현할 블록의 개수는 inode_disk 의 자료구조 크기가 512byte 가 되도록 결정해준다.

```
#define DIRECT_BLOCK_ENTRIES 123
#define INDIRECT_BLOCK_ENTRIES BLOCK_SECTOR_SIZE/(sizeof(block_sector_t))
```

Direct_block_entries 전역 변수는 direct block 의 개수를 의미하며, 이 개수는 inode_disk 구조체 크기가 block 의 크기가 되도록 결정된다. 여기서는 3 번째 과제까지 구현하여 is_dir 변수가 추가되어 총 123 개를 갖게 되었다. Indirect_block_entries 는 하나의 인덱스 블록이 저장할 수 잇는 다음 인덱스 블록의 개수를 의미한다.

b. Enum direct_t 변수 / setor_location 구조체 / inode_indirect_block 구조체

```
enum direct_t{
  NORMAL_DIRECT,  //inode에 디스크 블록 번호 저장
  INDIRECT,   //1개의 인덱스 블록을 통해 디스크 블록 번호 접근
  DOUBLE_INDIRECT,  //2개의 인덱스 블록을 통해 디스크 블록 번호 접근
  OUT_LIMIT //잘못된 파일 오프셋
};
```

Inode 가 디스크 블록의 번호를 가리키는 방식을 의미하며, direct 로 접근하는 방법, 1 개의 인덱스 블록을 통해 접근하는 방법, 2 개의 디스크 블록을 통해 접근하는 방법이 있다.

```
struct sector_location{
  enum direct_t directness; //디스크 블록 접근 방법
  off_t index1; //첫번째 접근할 디스크 블록 index
  off_t index2; //두번째 접근할 디스크 블록 index
};
```

블록 주소 접근 방식과, 인덱스 블록내의 오프셋 값을 저장하는 구조체로 뒤의 함수들에서 어느 위치의 디스크 블록에 데이터가 저장되어 있는지 찾을 때 사용한다.

```
struct inode_indirect_block{
block_sector_t map_table[INDIRECT_BLOCK_ENTRIES];···//인덱스 블록 배열
};
```

인덱스 블록을 표현하는 자료구조로, block_sector_t 의 배열로 구성되어 있다. 배열의 index 개수는 INDIRECT_BLOCK_ENTRIES 이다.

c. Get_disk_inode 함수 / locate_byte 함수

```
static bool get_disk_inode(const struct inode *inode, struct inode_disk *inode_disk){
  return bc_read(inode->sector, (void*)inode_disk, 0, sizeof(struct inode_disk), 0);
  ····//inode의 sector에 해당하는 데이터를 buffer cache에서 읽어 inode_disk에 저장하는 함수이다.
}
```

Inode 에 data 변수를 없애 줬으므로 inode 를 통해 data 를 접근하기 위해서는 먼저 inode_disk 구조체가 필요하다. Bc_Read 함수를 통해 inode 의 sector 에 해당하는 데이터를 inode_disk 에 저장해주어 inode_disk 구조체를 통해 데이터에 대해 접근이 가능하도록 해준다.

```
static void locate_byte (off_t pos, struct sector_location *sec_loc){····//디스크 블록 접근 방법을 sec_loc에 저장
  off_t pos_sector = pos / BLOCK_SECTOR_SIZE;···//해당 offset의 데이터의 block index 값

  if(pos_sector < DIRECT_BLOCK_ENTRIES){··//direct 방식일 경우
    sec_loc -> directness = NORMAL_DIRECT;
    sec_loc -> index1 = pos_sector;
  }
  else if(pos_sector < (off_t)(DIRECT_BLOCK_ENTRIES + INDIRECT_BLOCK_ENTRIES)){···//indirect 방식일 경우
    sec_loc -> directness = INDIRECT;
    sec_loc -> index1 = pos_sector - DIRECT_BLOCK_ENTRIES;
  }
  else if(pos_sector < (off_t)(DIRECT_BLOCK_ENTRIES + (INDIRECT_BLOCK_ENTRIES * (INDIRECT_BLOCK_ENTRIES + 1)))){··//double indirect
    방식일 경우
    sec_loc -> directness = DOUBLE_INDIRECT;
    sec_loc -> index1 = (pos_sector - DIRECT_BLOCK_ENTRIES - INDIRECT_BLOCK_ENTRIES) / INDIRECT_BLOCK_ENTRIES;
    sec_loc -> index2 = (pos_sector - DIRECT_BLOCK_ENTRIES - INDIRECT_BLOCK_ENTRIES) % INDIRECT_BLOCK_ENTRIES;
  }
  else
    sec_loc -> directness = OUT_LIMIT;
}
```

디스크 블록의 접근 방법에는 direct, indirect, double_indirect 3 가지로, 해당 함수를 통해 인자로 주어진 offset 의 블록이 어떠한 방법으로 접근이 되었는지 sec_loc 구조체에 저장해주는 함수이다. Sec_loc 구조체의 directeness 변수에는 접근 방법을 저장해주고, index1, index2 에는 각각 배열의 index 값을 저장해준다.

d. Register_sector 함수

```
static bool register_sector(struct inode_disk **inode_disk, block_sector_t new_sector, struct sector_location sec_loc){//새로 할당 받은 디스크 블록의 번호를 inode_disk 인자에 업데이트

  struct inode_indirect_block *indirect, *double_indirect;

  if(sec_loc.directness == NORMAL_DIRECT){//새로 할당 받은 디스크 블록이 direct접근일 경우
    inode_disk->direct_map_table[sec_loc.index1] = new_sector;
  }
  else if(sec_loc.directness == INDIRECT){//indirect 접근일 경우
    indirect = (struct inode_indirect_block *)malloc(sizeof(struct inode_indirect_block));
    if(sec_loc.index1 == 0){//indirect 접근이 처음일 경우 free_map_allocate 함수를 통해 block 할당
      if(free_map_allocate(1, &inode_disk->indirect_block_sec) == false){
        free(indirect);
        return false;
      }
    }
    bc_read(inode_disk->indirect_block_sec,(void *)indirect, 0, sizeof(struct inode_indirect_block), 0);
    indirect->map_table[sec_loc.index1] = new_sector;
    //bc_read함수를 통해 인덱스 블록을 buffer_cache로 읽은다음 해당 entry 업데이트
    bc_write(inode_disk->indirect_block_sec, (void *)indirect, 0, sizeof(struct inode_indirect_block),0);
    //업데이트 후 다시 블록에 업데이트 된 내용 write
    free(indirect);
  }
  else if(sec_loc.directness == DOUBLE_INDIRECT){//double_indirect 접근인 경우
    double_indirect = (struct inode_indirect_block *)malloc(sizeof(struct inode_indirect_block));
    if(sec_loc.index1 == 0){//첫번째 indirect block array에 처음 접근한 경우 free_map_allocate 함수를 통해 block 할당
      if(free_map_allocate(1, &inode_disk->double_indirect_block_sec) == false){
        free(double_indirect);
        return false;
      }
    }
```

```
    bc_read(inode_disk->double_indirect_block_sec,(void *)double_indirect, 0, sizeof(struct inode_indirect_block), 0);//double_indirect에 해당 블록 인덱스 read
    indirect = (struct inode_indirect_block *)malloc(sizeof(struct inode_indirect_block));
    if(sec_loc.index2 == 0){//두번째 indirect block array에 처음 접근한 경우 free_map_allocate 함수를 통해 block 할당
      if(free_map_allocate(1, &double_indirect->map_table[sec_loc.index1])==false){
        free(double_indirect);
        free(indirect);
        return false;
      }
    }
    bc_read(double_indirect->map_table[sec_loc.index1],(void *)indirect,0,sizeof(struct inode_indirect_block), 0);//bc_read함수를 통해 인덱스 블록을 buffer_cache로 읽은다음 해당 entry 업데이트

    indirect->map_table[sec_loc.index2] = new_sector;

    //업데이트 후 다시 블록에 업데이트 된 내용 write
    bc_write(double_indirect->map_table[sec_loc.index1],(void *)indirect,0,sizeof(struct inode_indirect_block), 0);
    bc_write(inode_disk->double_indirect_block_sec,(void *)double_indirect,0, sizeof(struct inode_indirect_block), 0);
    free(double_indirect);
    free(indirect);
  }
  else return false;
  return true;
}
```

새로 할당 받은 디스크 블록 번호를 inode_disk 구조체에 저장해주는 함수로서 해당 블록의 접근 방법에 따라 인덱스 블록을 구하여 해당 인덱스 블록에 디스크

블록 번호를 저장시켜준다. Direct 접근인 경우 바로 inode_disk 의 direct 배열에 저장시켜주면 되고, indirect 인 경우 indirect index block 을 한 번 읽은 다음, double_indirect 인 경우 두번의 index block 을 읽은 후 디스크 블록 번호를 저장시켜준다. 접근 방법에 대해서는 위에서 구현해 준 locate_byte 함수를 통해 알아낸다.

e. Inode_update_file_length 함수

```
bool inode_update_file_length(struct inode_disk *inode_disk, off_t start_pos, off_t end_pos){
  off_t offset = start_pos;
  off_t size = end_pos - start_pos;//update할 크기
  off_t chunk_size;
  struct inode_indirect_block zeroes;
  struct sector_location sec_loc;
  block_sector_t sector_idx;

  memset(&zeroes, 0, BLOCK_SECTOR_SIZE);

  while(size > 0){
    int sector_ofs = offset % BLOCK_SECTOR_SIZE;//block내에서 업데이트 전 파일의 끝 offset

    if(sector_ofs + size > BLOCK_SECTOR_SIZE)//다른 block이 더 필요할 경우
      chunk_size = BLOCK_SECTOR_SIZE - sector_ofs;
    else
      chunk_size = size;

    if(sector_ofs == 0){  //새로운 block 주소에 대한 접근 일 경우
      locate_byte(offset, &sec_loc);    //해당 주소 block의 접근 방법 sec_loc에 저장

      if(free_map_allocate(1, &sector_idx) == false)  //새로운 disk block 할당
        return false;
      register_sector(inode_disk, sector_idx, sec_loc);   //새로 할당 받은 block 번호를 inode_disk에 업데이트
      bc_write(sector_idx, &zeroes, 0, BLOCK_SECTOR_SIZE, 0);   //새로 할당 받은 block 데이터 초기화
    }

    size -= chunk_size;
    offset += chunk_size;
  }
  return true;
}
```

새로 파일이 업데이트 되어 새로운 디스크 블록이 필요할 경우 호출되는 함수로서, 만약 파일의 내용이 하나의 디스크 블록을 넘어가서 새로 필요하게 된다면 free_map_allocate 함수를 통해 할당 해 준 다음 register_sector 함수를 통해 inode_disk 에 업데이트 해준다. 새로 할당 해준 블록은 memset 함수와 bc_write 함수를 통해 0 값으로 초기화 시켜준다.

f. Free_inode_sectors 함수

```
static void free_inode_sectors(struct inode_disk *inode_disk){
  unsigned int i = 0,j = 0;
  struct inode_indirect_block  *indirect_block, *double_indirect_block;

  for(i=0;i<DIRECT_BLOCK_ENTRIES;i++){
    if(inode_disk->direct_map_table[i] > 0)  //direct 접근이 존재할 경우
      free_map_release (inode_disk->direct_map_table[i], 1);
  }

  if(inode_disk->indirect_block_sec > 0){  //indirect 접근이 존재할 경우
    indirect_block = (struct inode_indirect_block *)malloc(sizeof(struct inode_indirect_block));
    bc_read(inode_disk->indirect_block_sec, indirect_block, 0, sizeof(struct inode_indirect_block), 0);  //해당 index block을 indirect_block
    변수에 읽어온다음
    for(i=0;i<INDIRECT_BLOCK_ENTRIES;i++){
      if(indirect_block->map_table[i] > 0)    //해당 block에 할당이 존재할 경우
        free_map_release (indirect_block->map_table[i], 1);
    }
    free_map_release (inode_disk->indirect_block_sec, 1);
    free(indirect_block);
  }

  if(inode_disk->double_indirect_block_sec > 0){  //double_indirect 접근이 존재할 경우
    double_indirect_block= (struct inode_indirect_block *)malloc(sizeof(struct inode_indirect_block));
    indirect_block = (struct inode_indirect_block *)malloc(sizeof(struct inode_indirect_block));

    bc_read(inode_disk->double_indirect_block_sec, indirect_block,0,sizeof(struct inode_indirect_block),0);
    //해당 double index block을 indirect_block 변수에 읽어온다음
    for(i=0; i<INDIRECT_BLOCK_ENTRIES;i++){
      if (indirect_block->map_table[i] > 0){  //해당 block에 할당이 존재할 경우
        bc_read(indirect_block->map_table[i],double_indirect_block,0,sizeof(struct inode_indirect_block),0);
        //double_indirect_block 변수에 해당 인덱스 block 읽은 다음
        for(j=0;j<INDIRECT_BLOCK_ENTRIES;j++){
          if(double_indirect_block->map_table[j] > 0)  //인덱스 블록 탐색하면서 블록에 대한 접근이 존재할 경우 해당 블록 할당 해지
            free_map_release(double_indirect_block->map_table[j], 1);
        }
        free_map_release (indirect_block->map_table[i], 1); //두번째 인덱스 블록 할당 해지
      }
    }
    free_map_release(inode_disk->double_indirect_block_sec, 1);  //첫번째 인덱스 블록 할당 해지
    free(double_indirect_block);
    free(indirect_block);
  }
}
```

Inode_disk 에 존재하는 모든 디스크 블록의 할당을 해지하는 함수로서 direct 배열과 indirect 배열, double_indirect 배열 모두 탐색 하여 해당 블록에 대해 디스크 할당이 존재할 경우 free_map_release 함수를 통해 해지 시켜준다.

g. Byte_to_sector 함수

```
static block_sector_t
byte_to_sector (const struct inode_disk *inode_disk, off_t pos)
{
  // ASSERT (inode != NULL);
  // if (pos < inode->data.length)
  //   return inode->data.start + pos / BLOCK_SECTOR_SIZE;
  // else
  //   return -1;

  block_sector_t result_sec;
  struct inode_indirect_block *ind_block, *double_ind_block;
  struct sector_location sec_loc;

  if(pos < inode_disk->length){
    locate_byte(pos, &sec_loc);    //locate_byte 함수를 통해 해당 주소의 블록 접근 방법 확인

    if(sec_loc.directness == NORMAL_DIRECT){  //direct 접근일 경우
      result_sec = inode_disk->direct_map_table[sec_loc.index1];
    }
    else if(sec_loc.directness == INDIRECT){    //indirect 접근일 경우
      ind_block = (struct inode_indirect_block *)malloc(sizeof(struct inode_indirect_block));
      if(ind_block){
        bc_read(inode_disk->indirect_block_sec,(void*)ind_block,0,sizeof(struct inode_indirect_block), 0);
          //block index 읽은 다음 block index에서 해당 index의 블록 탐색 후 번호 반환
        result_sec = ind_block->map_table[sec_loc.index1];
        free(ind_block);
      }
      else result_sec = 0;
    }
    else if(sec_loc.directness == DOUBLE_INDIRECT){  //double_indirect 접근일 경우
      ind_block = (struct inode_indirect_block *)malloc(sizeof(struct inode_indirect_block));
      double_ind_block = (struct inode_indirect_block *)malloc(sizeof(struct inode_indirect_block));

        if(ind_block && double_ind_block){
          bc_read(inode_disk->double_indirect_block_sec,(void*)ind_block,0,sizeof(struct inode_indirect_block),0);    //첫번째 인덱스 블록 읽은후
          bc_read(ind_block->map_table[sec_loc.index1],(void*)double_ind_block, 0,sizeof(struct inode_indirect_block), 0);  //두번째 인덱스 블록 읽은후
          result_sec = double_ind_block->map_table[sec_loc.index2];    // block index에서 해당 index의 블록 탐색 후 번호 반환
          free(ind_block);
          free(double_ind_block);
        }
        else result_sec = 0;
      }
    else result_sec = 0;
  }
  else result_sec = 0;

  return result_sec;
}
```

파일 오프셋을 통해 inode_disk에서 해당 블록을 찾아 낸 다음 디스크 블록 번호를 리턴해주는 함수로서, 해당 블록 접근 방법이 direct인 경우 inode_disk의 direct 배열을 통해 바로 번호를 리턴해주고, indirect인 경우 indirect 배열을 읽은 다음, 해당 배열에서 오프셋 주소에 해당하는 블록의 번호를 리턴해준다. Double_indirect 인 경우 indirect와 동일하나 index block을 두 번 읽어주어 접근하게 된다.

h. Inode_create 함수 / inode_open 함수 / inode_read_at 함수 / inode_write_at 함수/ inode_close 함수

```
if(length > 0){
    if(inode_update_file_length(disk_inode, 0, length) == false){//inode_update_file_length로 length만큼의
    디스크블록을 호출하여 할당
        free(disk_inode);
        return false;
    }
}
```

Inode_create 함수에서 디스크 블록을 할당 해줄 때 inode_update_file_length 함수를 이용하여 length 크기만큼 블록 할당

```
  inode->deny_write_cnt = 0;
  inode->removed = false;
  lock_init(&inode->extend_lock);
  // block_read (fs_device, inode->sector, &inode->data);
  return inode;
}
```

Inode_open 함수에서 inode 구조체에 새로 추가해준 lock 의 초기화를 추가 시켜주고 없애 준 data 변수에 대한 read 를 없애 준다.

```
inode_disk = (struct inode_disk *)malloc(BLOCK_SECTOR_SIZE);
get_disk_inode(inode, inode_disk);
while (size > 0)
{
    /* Disk sector to read, starting byte offset within sector. */
    block_sector_t sector_idx = byte_to_sector (inode_disk, offset);
    int sector_ofs = offset % BLOCK_SECTOR_SIZE;
```

Inode_read_at 함수에서 data 에 대한 접근이 inode_disk 에서 이루어지므로 get_disk_inode 함수를 통해 inode 를 이용하여 inode_disk 를 구해주고, byte_to_sector 의 인자를 inode_disk 로 변경 해준다.

```
  get_disk_inode(inode,inode_disk);
  lock_acquire(&inode->extend_lock);

  int old_length = inode_disk -> length;
  int write_end = offset + size - 1;
  if(write_end > old_length - 1){
    inode_update_file_length(inode_disk, old_length, write_end);    //파일 길이가 증가하였을 경우 새로운 디스크 블록 할당
    및 inode_disk 업데이트
    inode_disk->length = write_end + 1;
  }
  lock_release(&inode->extend_lock);
  while (size > 0)
    {
      /* Sector to write, starting byte offset within sector. */
      block_sector_t sector_idx = byte_to_sector (inode_disk, offset);
      int sector_ofs = offset % BLOCK_SECTOR_SIZE;
```

Inode_write-at 함수에서도 inode_read_at 함수와 마찬가지로 data 에 대한 접근이 이제 inode_disk 구조체를 통해 이루어지므로 byte_to_sector 의 인자를 변경해주고, 만약 파일이 write 를 하였을 때 새로운 디스크 블락이 필요할 경우 inode_update_file_length 함수를 통해 할당해준다.

```
      if (inode->removed)
        {
          struct inode_disk * inode_disk = (struct inode_disk *)malloc(BLOCK_SECTOR_SIZE);
          get_disk_inode(inode,inode_disk);
          free_inode_sectors(inode_disk);
          free(inode_disk);
          free_map_release (inode->sector, 1);
          // free_map_release (inode->data.start, bytes_to_sectors (inode->data.length));
```

Inode_close 함수에서 inode 를 제거 시켜줄 때 free_inode_sectors 함수를 호출하여 할당 해준 모든 디스크 블록을 해지시켜준다.

3. Subdirectory

과제 목표:

Root 디렉터리 내 다른 디렉터리를 생성 할 수 있도록 계층 구조 구현

과제 설명:

현재 핀토스는 root 디렉터리만 존재하는 단일 계층으로 root 디렉터리에만 파일을 생성할 수 있다. Root 디렉터리가 아닌 다른 디렉터리를 생성할 수 있게끔 만들어주어 단일 구조가 아닌 여러 계층으로 디렉터리 생성을 가능하게끔 만들도록 한다.

a. Inode_disk 구조체 / thread 구조체 / inode_create 함수

```
struct inode_disk
  {
    // block_sector_t start;               /* First data sector. */
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    uint32_t is_dir;
    block_sector_t direct_map_table[DIRECT_BLOCK_ENTRIES];  //다이렉트로 접근할 디스크 블록의 번호 저장
    block_sector_t indirect_block_sec;  //indirect로 접근할 디스크 블록의 번호 저장
    block_sector_t double_indirect_block_sec;  //double indirect로 접근할 경우, 1차 index block 번호 저장
    // uint32_t unused[125];               /* Not used. */
  };
```

Inode_Disk 에 해당 inode_disk 의 데이터가 파일인지, 디렉터리인지 구별하기 위한 변수 is_dir 을 추가시켜준다.

```
  int next_fd;                /* number of next empty space in file descripter table */
  //this extra value on structure is needed on project5
  struct file *executing_file;      // executing file on the thread

  struct dir *cur_dir;      //작업중인 디렉터리 정보
}
```

Thread 구조체에 현재 스레드에서 작업 중인 디렉토리 정보를 저장하고 있는 구조체 포인터 cur_dir 를 추가 시켜준다.

```
bool
inode_create (block_sector_t sector, off_t length, uint32_t is_dir)////디렉토리 구분하는 인자 추가
{
  struct inode_disk *disk_inode = NULL;
  bool success = false;

  ASSERT (length >= 0);

  /* If this assertion fails, the inode structure is not exactly
     one sector in size, and you should fix that. */
  ASSERT (sizeof *disk_inode == BLOCK_SECTOR_SIZE);

  disk_inode = calloc (1, sizeof *disk_inode);
  if (disk_inode != NULL)
    {
      // size_t sectors = bytes_to_sectors (length);
      disk_inode->length = length;
      disk_inode->magic = INODE_MAGIC;
      disk_inode->is_dir = is_dir;////파일인지 디렉터리인지 구분
```

Inode_creat 함수 인자에 디렉터리인지, 파일인지를 나타내는 is_dir 인자를 추가시켜준다. Inode 생성시 해당 파일이 디렉터리인지, 파일인지 inode_disk 구조체 변수에 저장시켜주어 파악한다.

b. Dir_create 함수 / free_map_create 함수

```
bool
dir_create (block_sector_t sector, size_t entry_cnt)
{
  return inode_create (sector, entry_cnt * sizeof (struct dir_entry), 1);//디렉터리 정보를 디렉터리로 표시
}
```

디렉터리를 생성시 inode_create 함수 마지막 인자에 1 값을 넣어주어 디렉터리임을 나타내준다.

```
void
free_map_create (void)
{
  /* Create inode. */
  if (!inode_create (FREE_MAP_SECTOR, bitmap_file_size (free_map), 0))//디렉터리 정보를 파일로 표시
    PANIC ("free map creation failed");

  /* Write bitmap to file. */
  free_map_file = file_open (inode_open (FREE_MAP_SECTOR));
  if (free_map_file == NULL)
    PANIC ("can't open free map");
  if (!bitmap_write (free_map, free_map_file))
    PANIC ("can't write free map");
}
```

Free_map 파일 생성시 inode_create 함수 마지막 인자에 0 값을 넣어주어 파일임을 나타내준다.

c. Thread_init 함수 / filesys_init 함수 / thread_create 함수 / process_exit 함수

```
void
thread_init (void)
{
  ASSERT (intr_get_level () == INTR_OFF);

  lock_init (&tid_lock);
  list_init (&ready_list);
  list_init (&all_list);

  /* Set up a thread structure for the running thread. */
  initial_thread = running_thread ();
  init_thread (initial_thread, "main", PRI_DEFAULT);
  initial_thread->status = THREAD_RUNNING;
  initial_thread->tid = allocate_tid ();
  initial_thread->cur_dir = NULL;    //작업중인 디렉터리 표기 변수 초기화
}
```

스레드를 초기화 시켜줄 때, 새로 추가해준 cur_dir 변수를 NULL 값으로 초기화 시켜준다.

```
void
filesys_init (bool format)
{
  fs_device = block_get_role (BLOCK_FILESYS);
  if (fs_device == NULL)
    PANIC ("No file system device found, can't initialize file system.");

  inode_init ();
  bc_init ();
  free_map_init ();

  if (format)
    do_format ();
  free_map_open ();
  thread_current()->cur_dir = dir_open_root();    //현재 작업중인 디렉터리를 루트 디렉터리로 설정
}
```

Filesystem 을 초기화 시켜줄 때 현재 스레드의 실행 중인 디렉터리를 루트 디렉터리로 설정 시켜준다.

```
init_thread (t, name, priority);
tid = t->tid = allocate_tid ();

if(thread_current()->cur_dir != NULL)
    t->cur_dir = dir_reopen(thread_current()->cur_dir);    //자식 스레드의 작업 디렉터리를 부모 스레드의 작업 디렉터리로 설정

/* Prepare thread for first run by initializing its stack.
   Do this atomically so intermediate values for the 'stack'
   member cannot be observed. */
old_level = intr_disable ();
```

Thread_create 함수를 이용해 자식 스레드를 생성시, 자식 스레드의 실행중인 디렉터리를 현재 스레드의 실행중인 디렉터리로 설정 시켜준다.

```
  for(i = MAX_FILE-1; cur->next_fd>2;i--){//close the all open file
    process_close_file(i);
    if(i==2) i = MAX_FILE - 1;
  }

  free(cur->fdt);//and free the file descripter table memory

  dir_close(cur->cur_dir);    //프로세스 종료시 스레드가 작업하고 있던 디렉터리를 닫아준다.
```

프로세스 종료시 dir_close 함수를 이용하여 현재 작업중인 디렉터리를 먼저 닫아주고 프로세스를 종료 시킨다.

d. Inode_is_dir 함수

```
bool inode_is_dir (const struct inode * inode){
  bool result = false;
  struct inode_disk * inode_disk = (struct inode_disk *)malloc(sizeof(struct inode_disk));
  get_disk_inode(inode,inode_disk);    //inode의 on-disk inode를 읽어 inode_disk에 저장
  if(inode_disk->is_dir == 1)//inode_disk에 저장되어 있는 변수를 기반으로 해당 inode가 디렉터리를 가르키면 true값을
  리턴한다.
    result = true;
  free(inode_disk);
  return result;
}
```

Get_disk_inode 함수를 통해 in-memory inode 의 on-disk inode 를 읽어 inode_disk 에 저장한다. Inode_disk 구조체에 존재하는 is_dir 변수 데이터를 읽어 해당 inode 가 파일의 inode 인지 디렉터리의 inode 인지 반환해준다.

e. Parse_path 함수

```
struct dir* parse_path (char *path_name, char* file_name){

  if(path_name == NULL || file_name == NULL)
    return NULL;
  if(strlen(path_name) == 0)
    return NULL;

  struct dir * dir;
  if(path_name[0] == '/')    ////절대 주소 접근시
    dir = dir_open_root();
  else    //상대 주소 접근시
    dir = dir_reopen(thread_current()->cur_dir);

  char * token, *nextToken, *savePtr;
  token = strtok_r(path_name, "/",&savePtr);    //'/' 기준으로 파일 경로 파싱
  nextToken = strtok_r(NULL, "/", &savePtr);

  while(token != NULL && nextToken != NULL){    //경로 끝에 있는 파일이름 분리
    struct inode *inode;
    bool success = dir_lookup(dir, token, &inode);    //dir에서 token이름의 파일을 검색하여 inode의 정보를 저장
    if(!success || !inode_is_dir(inode))    //해당 디렉터리가 존재 하지 않거나 파일 일시
      return NULL;

    struct dir * nextDir = dir_open(inode);    //파싱한 디렉터리 오픈
    dir_close(dir);
    dir = nextDir;
    token = nextToken;
    nextToken = strtok_r (NULL, "/", &savePtr);
  }
  if(token == NULL)
    token = ".";
  size_t size = strlcpy(file_name, token, NAME_MAX+1);    //file_name 변수에 파일 이름 저장
  if(size > NAME_MAX + 1)
    return NULL;
  return dir;    //디렉터리 이름 반환
}
```

전체 경로로 주어진 path_name 에서 맨 마지막 위치에 존재하는 파일 이름과 그 사이의 파일 경로를 파싱 하여 따로 저장하여 리턴 하는 함수이다. 파일 경로 문자열을 '/' 문자를 기준으로 파싱하여 while 문을 통해 하나 하나 디렉터리를 열어보며 파일 끝의 디렉터리까지 도달하게 되면 반복문을 빠져나와 끝의 파일 이름을 가르키는 주소를 file_name 포인터 변수에 저장 시켜 주고 해당 경로까지 도달했던 파일 경로를 리턴 값으로 반환 시켜 준다. 단, 인자로 주어진 경로가

문자열 '/'로 시작하지 않을 시 현재 스레드의 작업 디렉터리의 주소인 상대주소로부터 경로를 생각해준다.

f. Filesys_create 함수 / filesys_open 함수

```
bool
filesys_create (const char *name, off_t initial_size)
{
  block_sector_t inode_sector = 0;
  // struct dir *dir = dir_open_root ();
  char * cp_name = (char*)malloc(strlen(name) + 1);
  char * name_ = (char*)malloc(strlen(name)+1);
  strlcpy(name_,name,strlen(name)+1);
  struct dir * dir = parse_path(name_, cp_name);//경로 파싱하여 디렉터리 경로와 파일 이름 저장
  if(dir == NULL){
    free(cp_name);
    free(name_);
    return false;
  }
  lock_acquire(&filesys_lock);

  bool success = (dir != NULL
                  && free_map_allocate (1, &inode_sector)
                  && inode_create (inode_sector, initial_size, 0)...//디렉터리 정보를 파일로 표시
                  && dir_add (dir,cp_name, inode_sector));....//파싱한 디렉터리 경로에 파일 저장
  if (!success && inode_sector != 0)
    free_map_release (inode_sector, 1);
  dir_close (dir);
  free(name_);
  free(cp_name);
  lock_release(&filesys_lock);
  return success;
}
```

파일 생성시 parse_path 함수를 이용하여 파일을 생성시키고 싶은 디렉터리 경로와, 파일 이름을 파싱 하여 해당 디렉터리에 파일을 생성시킨다. Dir_add 함수를 통해 생성시킨 파일을 해당 디렉터리에 저장시켜준다.

```
struct file *
filesys_open (const char *name)
{
  char *cp_name = (char*)malloc(strlen(name)+1);
  char *name_ = (char*)malloc(strlen(name)+1);
  memcpy(name_,name,strlen(name)+1);
  struct dir *dir = parse_path(name_, cp_name);   //경로 파싱하여 디렉터리 경로와 파일 이름 저장
  struct inode *inode = NULL;

  if (dir != NULL)
    dir_lookup (dir, cp_name, &inode);    //디렉터리 엔트리를 검색하여, 파일의 inode를 저장
  dir_close (dir);
  free(cp_name);
  free(name_);
  return file_open (inode);   //메모리에 file 자료구조 할당
}
```

파일을 오픈 할 때 마찬 가지로 parse_path 함수를 이용하여 경로와 파일 이름을 파싱해 준 다음 해당 경로에서 파일을 찾아 open 해준다.

g. Filesys_remove 함수 / do_format 함수

```
bool
filesys_remove (const char *name)
{
  char *cp_name = (char *)malloc(strlen(name)+1);
  char *name_ = (char *)malloc(strlen(name)+1);
  strlcpy(name_,name,strlen(name)+1);
  struct dir *dir = parse_path(name_, cp_name);   //디렉터리 경로와 파일 이름 파싱
  struct inode *inode;
  bool success = true;
  bool is_deletable = true;
  dir_lookup(dir, cp_name, &inode);

  if(inode != NULL && inode_is_dir(inode) == true){   //지우려는 파일이 디렉터리인 경우
    char *dir_name = (char*)malloc(NAME_MAX + 1);
    struct dir * child_dir = dir_open(inode);

    while(dir_readdir(child_dir,dir_name)){   //해당 디렉터리를 읽으면서 존재하는 디렉터리 탐색
      if(strcmp(dir_name,".") == 0 || strcmp(dir_name,"..") == 0)
        continue;
      else{   //만약 본인과 부모 디렉터리를 제외한 다른 파일 존재시 삭제 불가
        is_deletable = false;
        break;
      }
    }
    dir_close(child_dir);
    free(dir_name);
    success = is_deletable && dir_remove(dir, cp_name);   //디렉터리에 '.','..'을 제외한 다른 파일이 없을 경우
  }
  else{   //지우려는 파일이 파일인 경우 바로 삭제
    success = dir != NULL && dir_remove(dir, cp_name);
  }
  dir_close(dir);
  free(cp_name);
  free(name_);

  return success;
}
```

파일을 삭제해줄 때 parse_path 함수를 이용하여 삭제 하고 싶은 파일이 존재하는 경로와 파일이름을 파싱해준다. 만약 파일이 디렉터리가 아닌 파일일 경우 바로 삭제해주며, 디렉터리일 경우 해당 디렉터리를 읽어 '.', '..' 이름의 파일이 아닌 다른 파일이 존재 하는지 먼저 탐색해준다. 만약 존재할 경우 해당 디렉터리는 삭제 하지 못하게 된다. 만약 탐색했는데 존재 하지 않을 경우 일반 파일을 삭제 해줄 때와 마찬가지로 dir_remove 함수를 통해 삭제 해준다.

```
static void
do_format (void)
{
  bool success = true;
  struct dir * root;
  struct inode * inode;

  printf ("Formatting file system...");
  free_map_create ();
  if (!dir_create (ROOT_DIR_SECTOR, 16))
    PANIC ("root directory creation failed");
  root = dir_open_root();
  inode =dir_get_inode(root);
  success = dir_add(root,".",inode_get_inumber(inode)) && dir_add(root,"..",inode_get_inumber(inode));    //루트 디렉터리 엔트리에 '.','..' 파일 추가
  if(!success)
    PANIC ("root directory creation failed");
  free_map_close ();
  printf ("done.\n");
}
```

처음 파일 시스템 포맷시 root 디렉터리에 자기 파일 자신을 가르키는 '.' 파일, '..' 파일(부모 디렉터리가 존재 하지 않으므로)을 dir_add 함수를 통해 추가 시켜준다.

h. Filesys_create_dir 함수

```
bool filesys_create_dir(const char *name){

  block_sector_t inode_sector = 0;
  struct inode *inode;
  struct inode *new_inode;
  char * cp_name = (char*)malloc(strlen(name) + 1);
  char *name_ = (char*)malloc(strlen(name)+1);
  strlcpy(name_,name,strlen(name)+1);
  struct dir * dir = parse_path(name_, cp_name);    //위치시킬 디렉터리 경로와 디렉터리 이름 파싱
  if(dir_lookup(dir, cp_name, &new_inode)){   // 이미 같은 이름 파일 존재시
    free(cp_name);
    free(name_);
    return false;
  }
  struct dir * new_dir;
  bool success = (dir != NULL
                  && free_map_allocate (1, &inode_sector)
                  && dir_create (inode_sector, 16)        //디렉터리 생성 및 경로 디렉터리에 저장
                  && dir_add (dir,cp_name, inode_sector));
  if(success){
    new_dir = dir_open(inode_open(inode_sector));   //생성한 디렉터리 오픈
    inode = dir_get_inode(dir);
    success = success && dir_add(new_dir,".",inode_sector) && dir_add(new_dir,"..",inode_get_inumber(inode));//
    생성한 디렉터리에 '.','..' 파일 추가
    dir_close (new_dir);
  }
  if (!success && inode_sector != 0)
    free_map_release (inode_sector, 1);
  dir_close (dir);
  free(cp_name);
  free(name_);
  return success;
}
```

디렉터리를 생성하는 함수로서 Filesys_create 함수와 유사하다. 마찬가지로 parse_path 함수를 이용하여 파일 경로와 파일 이름을 파싱해주고, dir_create 함수와 dir_add 함수를 통해 디렉터리를 생성하여 부모 디렉터리에 추가시켜준다. 생성 시킨 디렉터리 파일을 open하여 해당 디렉터리에 자기 파일 자신을 가르키는 '.' 파일과 부모 디렉터리를 가르키는 '..' 파일을 dir_add 함수를 통해 추가시켜준다.

i. Dir_remove 함수

```
//* Find directory entry. */
if(strcmp(name,".")==0||strcmp(name,"..")==0)…//디렉터리 이름이 '.','..'인 경우 삭제 불가능
    goto done;
if (!lookup (dir, name, &e, &ofs))
    goto done;
```

디렉터리를 삭제 시켜줄 때 삭제 시켜주는 디렉터리의 이름이 '.' 이거나 '..' 인 경우 삭제가 불가능하도록 예외조건을 달아준다.

j. Sys_chdir 함수 / sys_mkdir 함수 / sys_readdir 함수 / sys_isdir 함수 / sys_inumber 함수

```
bool sys_chdir(const char *dir){
    struct file **file = filesys_open(dir);

    if(file){
        dir_close(thread_current()->cur_dir);……//스레드의 현재 작업중인 디렉터리를 닫아주고
        thread_current()->cur_dir = dir_open(file_get_inode(file));……//인자로 주어진 디렉터리로 변경시켜준다.
        return true;
    }
    return false;
}
```

현재 스레드의 작업 중인 디렉터리를 인자로 주어진 디렉터리로 변경 시켜주는 시스템 콜이다. Dir_close 로 실행 중이던 디렉터리를 닫아주고, dir_open 함수를 통해 새로운 디렉터리를 열어 thread 구조체의 cur_dir 변수에 저장시켜준다.

```
bool sys_mkdir(const char *dir){

    if(dir && strlen(dir) != 0)
        return filesys_create_dir(dir);……//해당 이름을 가진 디렉터리를 생성 시켜준다.
    else return false;
}
```

인자로 주어진 이름의 디렉터리를 생성시켜주는 시스템 콜로, filesys_create_dir 함수를 이용하여 디렉터리를 생성시켜준다.

```
bool sys_readdir(int fd, char *name){
    struct file *file = process_get_file(fd);
    struct inode *inode;
    off_t offset = 0;

    if(!file){
        inode = file_get_inode(file);
        if(!(inode && inode_is_dir(inode)))    //해당 파일이 존재하지 않거나 디렉터리가 아닌경우
            return false;
        struct dir *dir = dir_open(inode);
        char *cp_name = (char*)malloc(NAME_MAX+1);
        while(dir_readdir(dir, cp_name)){    //디렉터리에 존재하는 파일을 읽는다
            if(strcmp(cp_name,".")==0||strcmp(cp_name,"..")==0)
                continue;
            else{    //존재하는 파일 이름을 name 인자에 저장
                strlcpy(&name[offset], cp_name, strlen(cp_name)+1);
                offset = strlen(cp_name) + 1;
            }
        }
        free(cp_name);
        dir_close(dir);
        return true;
    }
    return false;
}
```

해당 파일 디스크립터의 디렉터리를 읽어 해당 디렉터리에 존재하는 파일명을 인자로 주어진 문자열 포인터에 저장시켜주는 시스템 콜이다. File_get_inode 함수를 이용하여 해당 파일의 inode 를 가져온다음 inode_is_dir 함수를 통해 해당 파일이 디렉터리인지 확인해준다. 디렉터리 인경우 dir_readdir 함수를 통해 디렉터리에 존재하는 파일을 모두 읽고, '.' 디렉터리와 '..' 디렉터리를 제외한 모든 파일의 이름을 name 변수 포인터에 저장시켜준다.

```
bool sys_isdir(int fd){
    struct file *file = process_get_file(fd);
    bool result = false;

    if(!file)
        result = inode_is_dir(file_get_inode(file));    //해당 디스크립터의 파일이 디렉터리인지 확인한다.

    return result;
}
```

해당 파일 디스크립터의 파일이 디렉터리인지 확인하는 시스템콜이다.

```
int sys_inumber(int fd){
    struct file * file = process_get_file(fd);
    if(!file)
        return inode_get_inumber(file_get_inode(file));    //해당 디스크립터의 파일의 inode number를 반환한다.
    return -1;
}
```

해당 파일 디스크립터의 파일의 inode_number 를 반환하는 시스템 콜로, file_get_inode 함수를 통해 파일의 inode 를 가져온다음, inode_get_inumber 함수를 통해 inode_number 를 구하여 리턴해주었다.

k. Syscall_handler 함수

```
        case SYS_CHDIR:
            get_argument(f->esp, arg, 1);
            check_address((void *)arg[0]);
            f->eax = sys_chdir((const char *)arg[0]);
            break;
        case SYS_MKDIR:
            get_argument(f->esp, arg, 1);
            check_address((void *)arg[0]);
            f->eax = sys_mkdir((const char *)arg[0]);
            break;
        case SYS_READDIR:
            get_argument(f->esp, arg, 2);
            check_address((char *)arg[1]);
            f->eax = sys_readdir(arg[0], (char *)arg[1]);
            break;
        case SYS_ISDIR:
            get_argument(f->esp, arg, 1);
            f->eax = sys_isdir(arg[0]);
            break;
        case SYS_INUMBER:
            get_argument(f->esp, arg, 1);
            f->eax = sys_inumber(arg[0]);
            break;
        default:
            thread_exit();
```

위에서 구현한 5 가지의 시스템콜이 제대로 핸들링 되게 하기 위해 시스템콜 핸들링 함수에 위와 같이 처리 코드를 추가시켜주었다.

실행 결과

```
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
pass tests/filesys/extended/dir-empty-name
pass tests/filesys/extended/dir-mk-tree
pass tests/filesys/extended/dir-mkdir
FAIL tests/filesys/extended/dir-open
pass tests/filesys/extended/dir-over-file
FAIL tests/filesys/extended/dir-rm-cwd
pass tests/filesys/extended/dir-rm-parent
pass tests/filesys/extended/dir-rm-root
pass tests/filesys/extended/dir-rm-tree
pass tests/filesys/extended/dir-rmdir
pass tests/filesys/extended/dir-under-file
FAIL tests/filesys/extended/dir-vine
pass tests/filesys/extended/grow-create
pass tests/filesys/extended/grow-dir-lg
pass tests/filesys/extended/grow-file-size
pass tests/filesys/extended/grow-root-lg
pass tests/filesys/extended/grow-root-sm
pass tests/filesys/extended/grow-seq-lg
pass tests/filesys/extended/grow-seq-sm
pass tests/filesys/extended/grow-sparse
pass tests/filesys/extended/grow-tell
pass tests/filesys/extended/grow-two-files
pass tests/filesys/extended/syn-rw
```

Make check 결과 총 36 개의 test 에서 3 개 영역에서 fail 발생하였으며 subdirectory 와 관련된 테스트들이었다. 이부분에 있어서는 해결하지 못하였다.