FileSystem 과제 보고서

운영체제 (ELE-3021 12781)

교수명 : 원유집

조교명 : 오준택

학생명 :

서창범 2014004648, 서문은지 2016025487

제출일자 : 2018 년 6 월 22 일

FileSystem

- Buffer Cache

1. 과제 목표

Buffer cache는 디스크 블록을 캐싱하는 메모리 영역이다. 디스크블록을 메모리영역에 둠으로써 파일의 입출력 응답시간을 줄일 수 있다. 현재 핀토스에는, buffer cache가 존재하지 않으므로 사용자의 읽기/쓰기 요청 시(파일 입출력 요청시) 바로 디스크 입출력 동작을 수행한다. 그래서, 본 과제에서 buffer cache를 구현하고, 성능향상을 살펴보려고 한다.

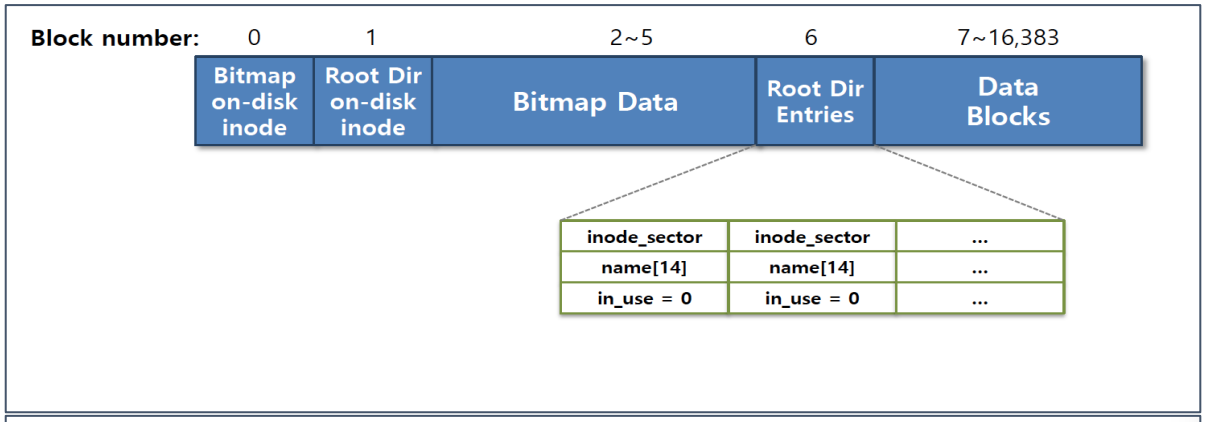
2. 과제 설명

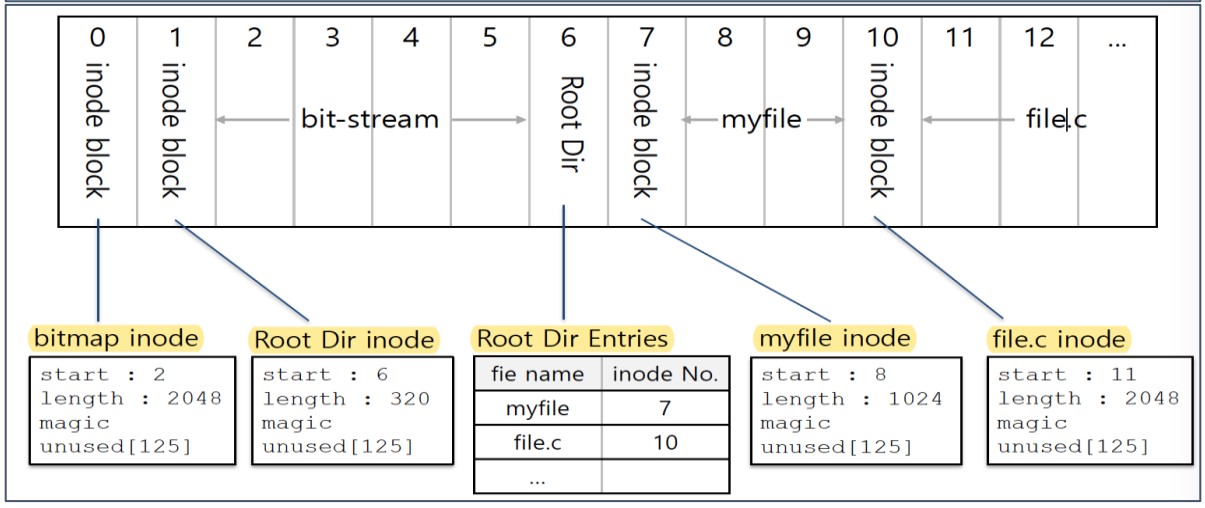
<Filesystem>

- OS가 디렉토리나 파일을 생성, 접근 및 보관할 수 있도록 하는 계층이다.
- 블록 디바이스를 일정한 크기의 블록(sector)들로 관리한다.
- 블록의 사용 여부를 bitmap을 통해 관리한다.

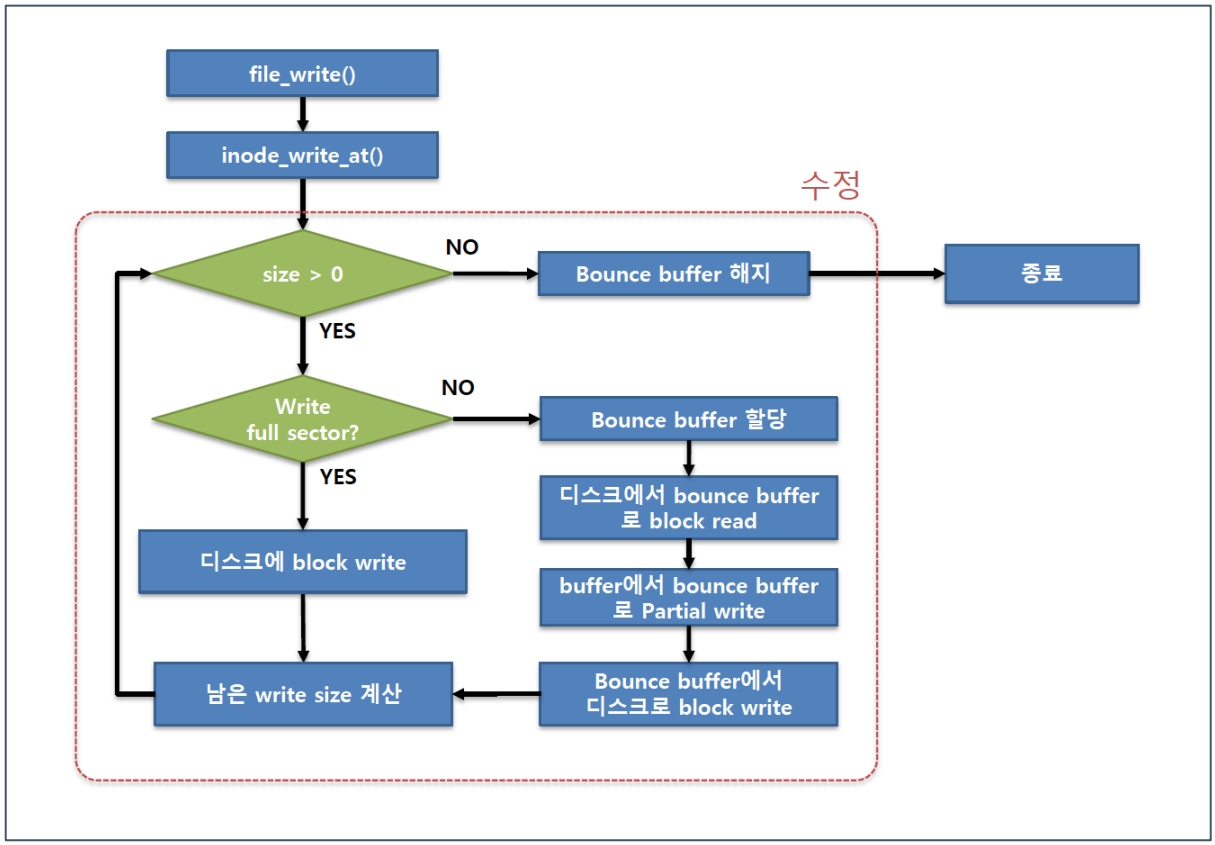
<핀토스의 8MB FileSystem의 예시>

- 8MB/512byte = 16,384개의 블록을 사용하고있다.

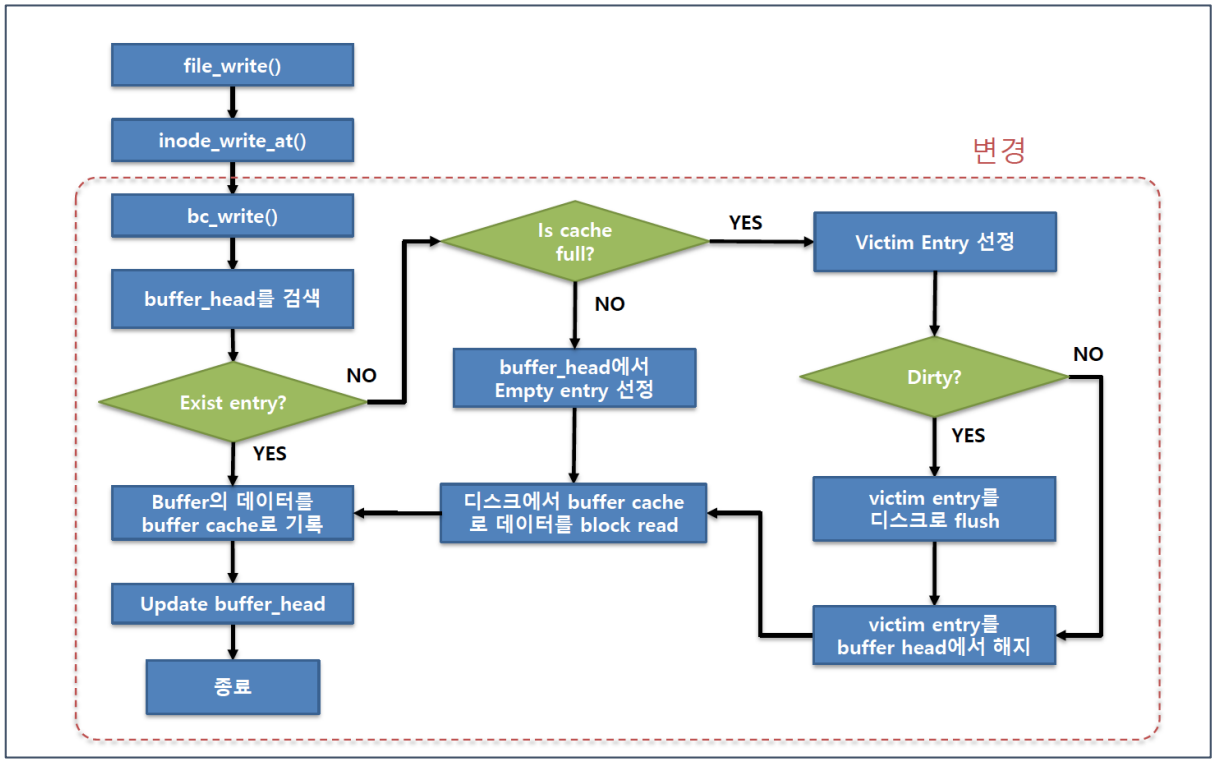




현재의 file_write는 아래와 같다.



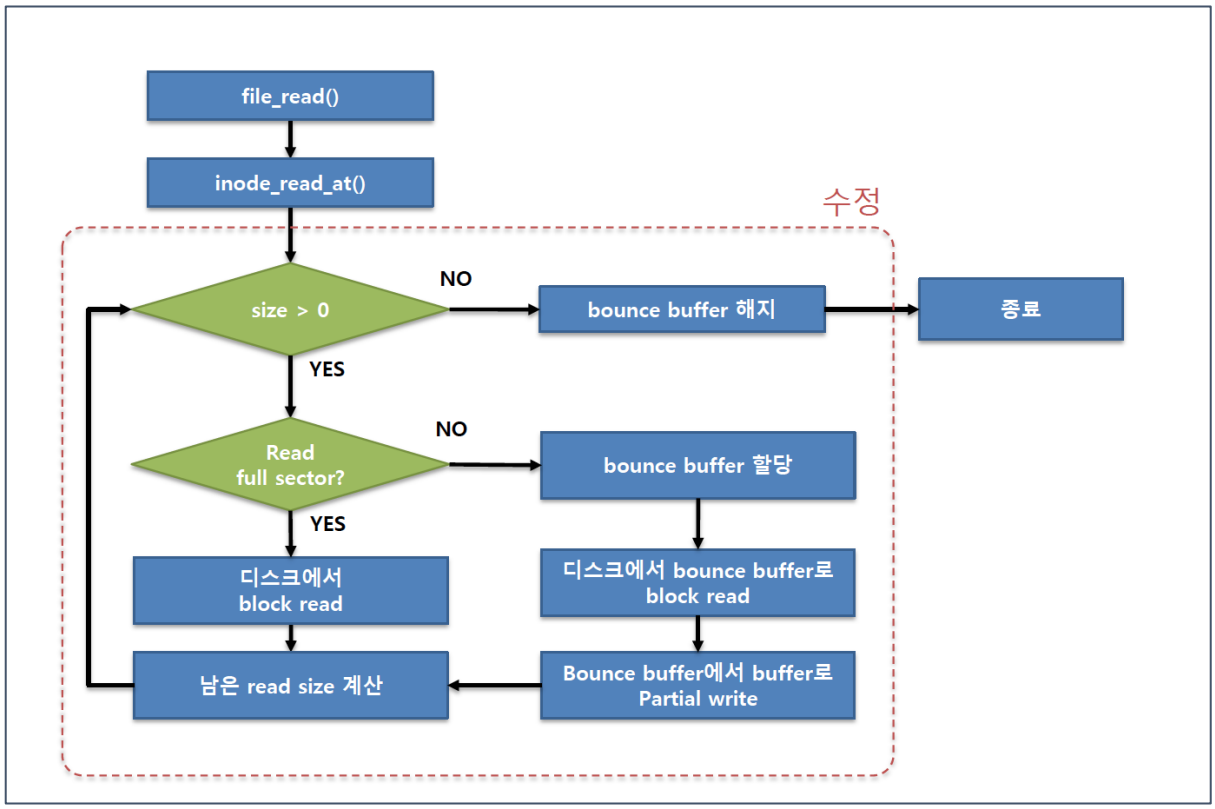
이를 buffer cache를 이용하여 아래와 같이 변경 해주어야한다.



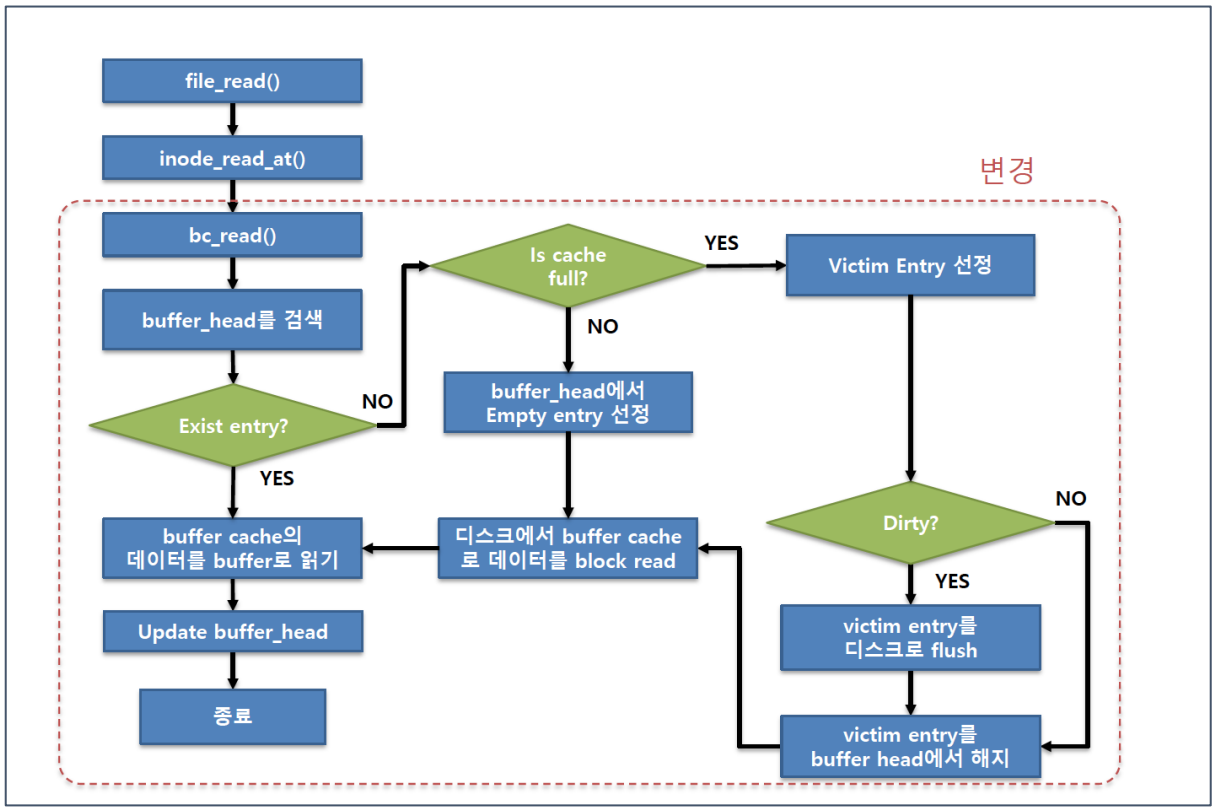
파일 "A" 에 내용을 쓴다고 해보자. 우리가 수정해야 할 부분을 파악하기 위해 간략하게 흐름을 작성하였다.

1. root directory의 inode(=1's block)를 읽는다.
2. root directory on-disk-inode에 있는 start 필드 값을 참조하여, root dir의 내용을 읽는다. 즉, dir entries를 읽는다.
3. directory entry들을 순회하면서, file name "A"를 탐색한다.
4. 찾았으면, 파일 이름 "A" 를 entry로 갖고있는 dir entry에서 inode_sector를 읽어 파일 "A"의 on-disk-inode를 읽는다.
5. inode를 읽으면 open이 가능하다. open을 하여 in-memory-inode에 로드한다.
6. 파일에 내용을 쓰기 위해, offset을 찾아야한다. 내가 쓰려고 하는 내용이 몇 번 블록에 저장되는지를 명시해야 하기 때문이다.
7. offset을 계산해서 데이터 블록에 내용을 쓴다.

현재의 read는 아래와 같다.



이를 buffer cache를 이용하여 아래와 같이 변경 해주어야한다.



디스크 블록 단위로 loop를 돌며 디스크에서 데이터를 읽었었는데, 이것을 버퍼 캐시에서 검색을 하여 읽어올 수 있도록 한다.(bc_lookup()함수 이용) 만약 버퍼 캐시에 검색 결과가 없을 경우, 디스크 블록을 캐싱 할 buffer entry의 buffer_head를 구하여, 디스크 블록 데이터를 버퍼 캐시로 read할 수 있도록 한다.

3. 함수 구현 및 수정

buffer_head 자료구조 추가

filesys/buffer_cache.h

```
 9 /* buffer cache의 각 entry를 관리하기 위한 구조체 */
10 struct buffer_head {
11   /* 해당 entry가 dirty인지를 나타내는 flag */
12   bool dirty;
13   /* 해당 entry의 사용 여부를 나타내는 flag */
14   bool in_use;
15   /* 해당 entry의 disk sector 주소 */
16   block_sector_t sector;
17   /* clock algorithm을 위한 clock bit */
18   bool clock_bit;
19   /* lock 변수 (struct lock) */
20   struct lock lock;
21   /* buffer cache entry를 가리키기 위한 데이터 포인터 */
22   void *bc_entry;
23 };
```

buffer_head 자료구조를 추가한다. 각 버퍼헤드는 각각 512 바이트 크기의 버퍼 캐시의 속성을 나타내며, 내용이 변경되었는지(dirty), 유효한 캐시인지(in_use), 버퍼 캐시에 저장되는 블록이 디스크의 몇 번째 블록인지(sector), lru 알고리즘에 의해 lru 를 근사하기 위한 flag 와 버퍼 캐시의 접근할 때 동기식으로 접근이 이뤄질 수 있도록 하기 위한 lock 과, 그리고 실제 512 바이트 크기의 버퍼를 가리키는 bc_entry 를 멤버로 하고 있다.

buffer_cache 전역변수 추가

filesys/buffer_cache.c

```
11 /* buffer cache entry의 개수 (32kb) */
12 #define BUFFER_CACHE_SIZE 32*1024
13 #define BUFFER_CACHE_ENTRY_NB (BUFFER_CACHE_SIZE / BLOCK_SECTOR_SIZE)  //64
14
15 /* buffer cache 메모리 영역을 가리킴 */
16 void *p_buffer_cache = NULL;
17 struct lock bc_lock;
18
19 /* buffer head 배열 */
20 struct buffer_head buffer_head_table[BUFFER_CACHE_ENTRY_NB];
21
22 /* victim entry 선정 시 clock 알고리즘을 위한 변수 */
23 int clock_hand = 0;
```

p_buffer_cache 는 버퍼캐시의 시작 주소를 나타낸다. 버퍼캐시의 총 크기는 32kb 이며, 이는 512 바이트 단위로 64 개로 분할하여 디스크 블록의 버퍼로 사용된다.
그리고 또한 buffer_head entry 들에 대한 연산 수행이나 접근 시 lock 을 사용하여 동기적으로 수행될 수 있게 bc_lock 을 선언하였다. buffer_head entry 들은 배열로서 전역변수로 선언하여 접근한다. clock_hand 는 buffer_head 의 배열에서 clock 알고리즘으로 victim 을 찾을 때 시계바늘 역할을 하는 배열의 인덱스를 가리키는 변수이다.

buffer cache read 구현

filesys/buffer_cache.c

```
144 bool bc_read (block_sector_t sector_idx, void *buffer,
145               off_t bytes_read, int chunk_size, int sector_ofs)
146 {
147   /*  sector_idx를 buffer_head에서 검색 (bc_lookup 함수 이용) */
148   struct buffer_head *bch = NULL;
149   bch = bc_lookup (sector_idx);
150
151   /*  검색 결과가 없을 경우, 디스크 블록을 캐싱 할 buffer entry의
152       buffer_head를 구함 (bc_select_victim 함수 이용) */
153   if (bch == NULL) {
154     ASSERT (lock_held_by_current_thread (&bc_lock));
155     bch = bc_select_victim ();
156     ASSERT (lock_held_by_current_thread (&bch->lock));
157     bch->sector = sector_idx;
158     lock_release (&bc_lock);
159
160     /*  block_read 함수를 이용해, 디스크 블록 데이터를 buffer cache로 read */
161     block_read (fs_device, sector_idx, bch->bc_entry);
162     bch->in_use = true;
163     bch->dirty = false;
164   }
165   ASSERT (lock_held_by_current_thread (&bch->lock));
166   ASSERT (bch->in_use == true);
167
168   /*  memcpy 함수를 통해, buffer에 디스크 블록 데이터를 복사 */
169   memcpy (buffer + bytes_read, bch->bc_entry + sector_ofs, chunk_size);
170
171   /*  buffer_head의 clock bit을 setting */
172   bch->clock_bit = false;
173
174   lock_release (&bch->lock);
175   return true;
176 }
```

기존 하드디스크 읽기 수행 시 block_read() 함수를 이용했던 것을 대체할 함수이다. 이 함수의 핵심은 버퍼캐시에 인자로 넘어온 sector 가 캐시된 버퍼가 있는지 검사하여 만약 버퍼가 존재하는 경우 해당 버퍼에서 읽어오고 끝내게 된다는 것이다. 캐시 미스 상황 시에는 기존 버퍼 캐시 중 하나를 victim 으로 선정해 flush 시킨 후, 해당 버퍼 캐시에 해당 디스크 섹터의 내용을 로드하여 버퍼캐시에 memcpy()로 접근하여 필요한 데이터를 읽어올 수 있다. 캐시 미스 상황에서도 결국엔 buffer cache 상에 있는 데이터를 읽어와 buffer 에 복사를 해주게 된다.

디스크 read 를 buffer cache read 로 수정

filesys/inode.c

```
504 off_t
505 inode_read_at (struct inode *inode, void *buffer_, off_t size, off_t offset)
506 {
507   uint8_t *buffer = buffer_;
508   off_t bytes_read = 0;
509   struct inode_disk *disk_inode = (struct inode_disk*)malloc (sizeof (struct inode_disk));
510   ASSERT (disk_inode);
511   get_disk_inode (inode, disk_inode);
512
513   while (size > 0)
514   {
515     /* Disk sector to read, starting byte offset within sector. */
516     block_sector_t sector_idx = byte_to_sector (disk_inode, offset);
517     int sector_ofs = offset % BLOCK_SECTOR_SIZE;
518
519     /* Bytes left in inode, bytes left in sector, lesser of the two. */
520     off_t inode_left = inode_length (inode) - offset;
521     int sector_left = BLOCK_SECTOR_SIZE - sector_ofs;
522     int min_left = inode_left < sector_left ? inode_left : sector_left;
523
524     /* Number of bytes to actually copy out of this sector. */
525     int chunk_size = size < min_left ? size : min_left;
526     if (chunk_size <= 0)
527       break;
528
529     bc_read (sector_idx, buffer, bytes_read, chunk_size, sector_ofs);
530
531     /* Advance. */
532     size -= chunk_size;
533     offset += chunk_size;
534     bytes_read += chunk_size;
535   }
536   free (disk_inode);
537
538   return bytes_read;
539 }
```

기존에 있던 block_read()를 bc_read()로 대체하였다.

buffer cache write 함수 구현

filesys/buffer_cache.c

```
117 bool bc_write (block_sector_t sector_idx, void *buffer,
118     off_t bytes_written, int chunk_size, int sector_ofs)
119 {
120   bool success = false;
121   struct buffer_head *bch = NULL;
122
123   /*  sector_idx를 buffer_head에서 검색하여 buffer에 복사(구현)*/
124   bch = bc_lookup (sector_idx);
125   if (bch == NULL) {
126     bch = bc_select_victim ();
127     ASSERT (lock_held_by_current_thread (&bch->lock));
128
129     bch->sector = sector_idx;
130     bch->in_use = true;
131     block_read (fs_device, sector_idx, bch->bc_entry);
132     lock_release (&bc_lock);
133   }
134   ASSERT (lock_held_by_current_thread (&bch->lock));
135
136   /*  update buffer head (구현) */
137   memcpy (bch->bc_entry + sector_ofs, buffer + bytes_written, chunk_size);
138   bch->dirty = true;
139   bch->clock_bit = false;
140   lock_release (&bch->lock);
141   success = true;
142   return success;
143 }
```

이 함수도 bc_read()와 마찬가지로 buffer cache 에 해당 섹터가 캐시된 것이 있는지 없는지 검사한 후 캐시된 것이 있으면 바로 캐시에 write 후 dirty 플래그를 표시하고, 캐시된 것이 없다면, 디스크에서 해당 섹터의 내용을 읽어와 victim 으로 선정된 버퍼 캐시에 로드하여 버퍼 캐시에 write 를 하고 dirty 플래그를 표시한다.

디스크 write 를 buffer cache write 로 수정

filesys/inode.c :: inode_write_at()

```
577
578   while (size > 0)
579   {
580     /* Sector to write, starting byte offset within sector. */
581     block_sector_t sector_idx = byte_to_sector (disk_inode, offset);
582     int sector_ofs = offset % BLOCK_SECTOR_SIZE;
583
584     ASSERT (sector_idx != -1);
585
586     /* Bytes left in inode, bytes left in sector, lesser of the two. */
587     off_t inode_left = inode_length (inode) - offset;
588     int sector_left = BLOCK_SECTOR_SIZE - sector_ofs;
589     int min_left = inode_left < sector_left ? inode_left : sector_left;
590
591     /* Number of bytes to actually write into this sector. */
592     int chunk_size = size < min_left ? size : min_left;
593     if (chunk_size <= 0)
594       break;
595
596     bc_write (sector_idx, buffer, bytes_written, chunk_size, sector_ofs);
597
598     /* Advance. */
599     size -= chunk_size;
600     offset += chunk_size;
601     bytes_written += chunk_size;
602   }
603
604   free (disk_inode);
605
606   return bytes_written;
607 }
```

기존 inode_write_at 함수 내에서 block_write() 함수를 통해 매번 io 트래픽을 발생시켜 디스크에 접근했던 것을 버퍼 캐시를 사용하여 성능을 올릴 수 있다. 기존의 block_write() 함수를 bc_write()로 바꾸었다.

buffer cache 초기화 함수 구현

filesys/buffer_cache.c

```
25 void bc_init (void) {
26   int i = 0;
27
28   /*  Allocation buffer cache in Memory */
29   /*  p_buffer_cache가 buffer cache 영역 포인팅 */
30   p_buffer_cache = palloc_get_multiple (PAL_ZERO, DIV_ROUND_UP (BUFFER_CACHE_SIZE, PGSIZE));
31
32   ASSERT (p_buffer_cache != NULL);
33   lock_init (&bc_lock);
34
35   /*  전역변수 buffer_head 자료구조 초기화 */
36   memset (buffer_head_table, 0x00, sizeof (struct buffer_head)*BUFFER_CACHE_ENTRY_NB);
37   for (i = 0; i < BUFFER_CACHE_ENTRY_NB; i++) {
38     lock_init (&buffer_head_table[i].lock);
39     buffer_head_table[i].bc_entry = p_buffer_cache + i*BLOCK_SECTOR_SIZE;
40   }
41 }
42
```

p_buffer_cache 에 캐시 메모리로 사용할 공간을 할당하여 포인터를 저장한다. 또한 전역변수 bc_lock 에 대한 초기화를 수행하고, 각 buffer_head entry 를 for 문을 돌면서 초기화를 수행한다. 그 과정에서 buffer cache 메모리를 512 바이트 단위로 잘라 포인터를 저장한다.

buffer cache 종료 함수 구현

filesys/buffer_cache.c

```
67 void bc_term (void) {
68   /*  bc_flush_all_entries 함수를 호출하여 모든 buffer cache entry를 디스크로 flush */
69   bc_flush_all_entries ();
70   /*  buffer cache 영역 할당 해제 */
71   palloc_free_multiple (p_buffer_cache, DIV_ROUND_UP (BUFFER_CACHE_SIZE, PGSIZE));
72 }
73
```

모든 buffer_head 엔트리가 속성을 나타내고 있는 모든 유효한 캐시 중 dirty 한 것을 디스크로 write-back 을 수행하고 버퍼 캐시로 사용했던 메모리 공간을 해제한다. 이는 핀토스가 종료되면서 파일 시스템을 해제 하면서 작업이 이뤄져야 하므로 filesys_done() 함수에서 호출되게 하였다.

```
87 void
88 shutdown_power_off (void)
89 {
90   const char s[] = "Shutdown";
91   const char *p;
92
93 #ifdef FILESYS
94   filesys_done ();
95 #endif
96
97   print_stats ();
98
```

```
42 void
43 filesys_done (void)
44 {
45   free_map_close ();
46   bc_term();
47 }
```

victim selection 함수 구현

filesys/buffer_cache.c

```
 94 struct buffer_head* bc_select_victim (void) {
 95   struct buffer_head *victim = NULL;
 96   ASSERT (lock_held_by_current_thread (&bc_lock));
 97
 98   /*  clock 알고리즘을 사용하여 victim entry를 선택 */
 99   /*  buffer_head 전역변수를 순회하며 clock_bit 변수를 검사 */
100   /*  victim entry에 해당하는 buffer_head 값 update */
101   for (; true; clock_hand = (clock_hand + 1) % BUFFER_CACHE_ENTRY_NB) {
102     struct buffer_head *bhe = &buffer_head_table[clock_hand];
103     if (bhe->in_use == false || bhe->clock_bit == true) {
104       victim = bhe;
105       clock_hand = (clock_hand + 1) % BUFFER_CACHE_ENTRY_NB;
106       break;
107     }
108     bhe->clock_bit = true;
109     continue;
110   }
111
112   /* 해당 cache buffer에 read, write가 완료될 때 까지 lock유지 */
113   lock_acquire (&victim->lock);
114
115   /*  선택된 victim entry가 dirty일 경우, 디스크로 flush */
116   if (victim->in_use == true && victim->dirty == true) {
117     bc_flush_entry (victim);
118   }
119
120   /*  victim entry를 return */
121   return victim;
122 }
```

clock 알고리즘에 의해 victim buffer head entry 를 선정하는 함수이다. 각 buffer head entry 의 clock_bit 는 초기에 false 로 초기화 되고, bc_select_victim() 함수 내의 for 문에 의해 순회를 돌면서 clock_bit 를 true 로 바꾼다. 순회가 충분히 돌아 clock_bit 가 true 인 buffer head 를 만나면 해당 버퍼가 사용 중인 buffer 인지 검사하고 사용 중이었을 경우 디스크로 write-back 수행 후 해당 buffer head 를 victim 으로서 반환한다. for 문이 순회를 돌면서 사용 중이지 않은 buffer cache 를 만나면 바로 반환하여 cache 로서 사용될 수 있게 하였다.

이 과정에서 lock 으로 동기적으로 수행될 수 있게 하였다. bc_lock 은 buffer_head entry 들을 순회하는 동안 다른 스레드에 의해 buffer_head 엔트리들이 변경되지 않는 것을 보장하기 위해 사용되어졌고, victim buffer head 가 선정되고 나면 해당 버퍼 캐시에 변경이 이뤄지지 않도록 victim->lock 을 잡아주었다.

Entry lookup 함수 구현

filesys/buffer_cache.c

```
74 struct buffer_head* bc_lookup (block_sector_t sector) {
75
76   /*  buffe_head를 순회하며, 전달받은 sector 값과 동일한 sector 값을 갖는 buffer cache entry가 있는지 확인 */
77   struct buffer_head *bch = NULL;
78   int i = 0;
79
80   /* bc_lock은 캐시될 버퍼가 결정되는 과정이 atomic하게 수행되어질 수 있도록 유지되어야함. */
81   lock_acquire (&bc_lock);
82   for (i = 0; i < BUFFER_CACHE_ENTRY_NB; i++) {
83     if (buffer_head_table[i].sector == sector && buffer_head_table[i].in_use == true) {
84       bch = &buffer_head_table[i];
85
86       /* 캐시될 버퍼가 정해지고 해당 캐피 버퍼에 read, write가 완료될 때까지 buffer head lock유지 */
87       lock_acquire (&bch->lock);
88
89       /* 캐시될 버퍼가 결정되었으므로 bc_lock을 해제함 */
90       lock_release (&bc_lock);
91       break;
92     }
93   }
94
95   /*  성공 : 찾은 buffer_head 반환, 실패 : NULL */
96   return bch;
97 }
```

buffer cache 에 주어진 디스크 섹터가 cache 된 버퍼가 있는지 탐색하는 함수이다. 여기서 찾아지지 않는다는 것은 cache miss 를 의미하며, 후에 bc_select_victim()에 의해 하나의 buffer cache 를 flush 한 후 주어진 섹터를 해당 버퍼에 캐싱하게 된다. 이곳에서 역시 buffer head 를 순회하는 것이 동기적으로 이뤄질 수 있게 lock 으로 보호하였다.

Entry flush 함수 구현

filesys/buffer_cache.c

```
43 void bc_flush_entry (struct buffer_head *p_flush_entry) {
44   ASSERT (p_flush_entry->dirty == true);
45   ASSERT (p_flush_entry->in_use == true);
46   ASSERT (lock_held_by_current_thread (&p_flush_entry->lock));
47
48   /*  block_write을 호출하여, 인자로 전달받은 buffer cache entry의 데이터를 디스크로 flush */
49   block_write (fs_device, p_flush_entry->sector, p_flush_entry->bc_entry);
50
51   /*  buffer_head의 dirty 값 update */
52   p_flush_entry->dirty = false;
53 }
```

주어진 buffer_head 가 속성을 나타내고 있는 buffer cache 를 디스크에 write-back 하고 dirty 플래그를 false 로 체크하여 해당 캐시의 내용은 디스크와 내용이 같다는 것을 나타내게 한다. victim buffer head 로 선정되어질 때, 파일 시스템이 해제될 때 등 캐싱된 섹터들의 변경내역이 원래 디스크로 동기화될 필요가 있을 때 수행된다.

모든 entry flush 함수 구현

filesys/buffer_cache.c

```
55 void bc_flush_all_entries (void) {
56   int i = 0;
57   /*  전역변수 buffer_head를 순회하며, dirty인 entry는 block_write 함수를 호출하여 디스크로 flush */
58   /*  디스크로 flush한 후, buffer_head의 dirty 값 update */
59   for (i = 0; i < BUFFER_CACHE_ENTRY_NB; i++) {
60     if (buffer_head_table[i].dirty == true && buffer_head_table[i].in_use == true) {
61       lock_acquire (&buffer_head_table[i].lock);
62       bc_flush_entry (&buffer_head_table[i]);
63       lock_release (&buffer_head_table[i].lock);
64     }
65   }
66 }
```

모든 버퍼 캐시들의 내용이 디스크로 flush 되게 한다. 실제 운영체제에서는 이러한 작업을 주기적으로 수행하는 반면, 핀토스에서는 운영체제가 꺼질 때 파일시스템을 해제하면서 수행되어진다.

FileSystem

- Extensible File

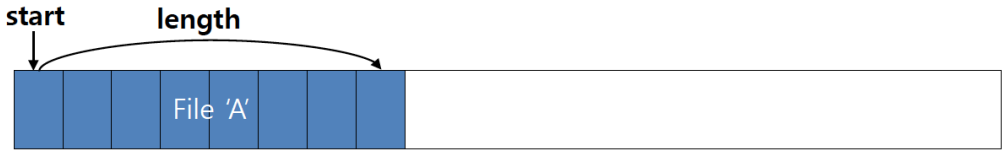
1. 과제 목표

현재 핀토스에서는 파일 생성 시 파일의 크기가 결정되고, 추후 변경이 불가능하다. 이번 과제를 통해 파일에 쓰기 동작을 수행할 때에 디스크 블록을 할당 받아 사용하도록 구현한다. 이를 통해 파일 크기가 생성 시에 고정되지 않고, 확장 가능하도록 한다.

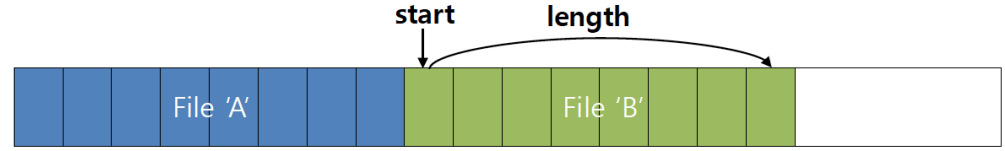
2. 과제 설명

아래의 그림이 현재 핀토스의 파일 생성 시 블록 할당 방식을 나타낸 그림이다.

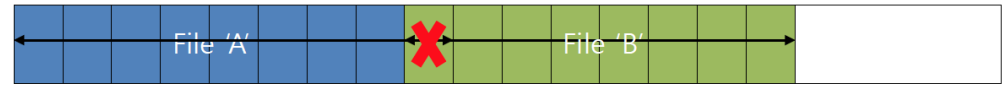
1. 파일 A 생성 (inode에 시작 블록 번호와 크기 저장)



2. 파일 B 생성 (inode에 시작 블록 번호와 크기 저장)



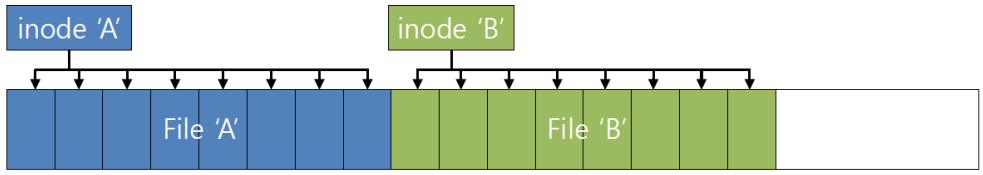
3. 파일 A는 파일 B 때문에 크기 증가 불가



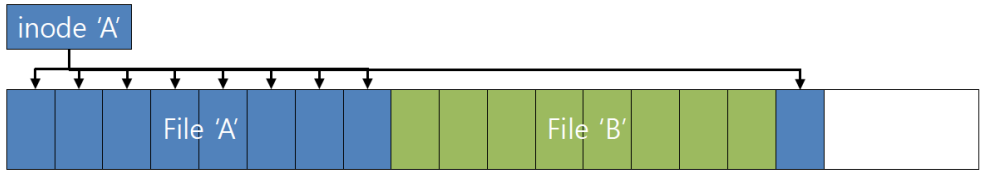
과제를 구현 완료 한 후의 핀토스의 파일 생성시 블록 할당 방식은 다음과 같이 되어야 한다.

1. 파일 A 생성 (inode에 할당된 블록 번호들을 저장)

2. 파일 B 생성 (inode에 할당된 블록 번호들을 저장)



3. 파일 A는 불연속 위치에 할당받은 블록 번호를 inode에 저장 가능



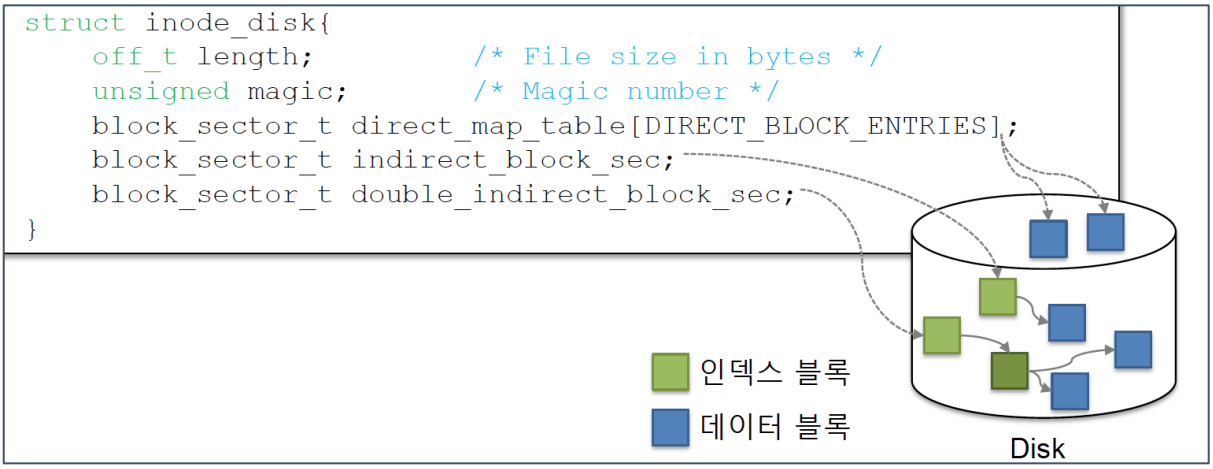
일단 디스크 블록 주소의 표현 방식에 대한 이해가 필요하다.

기존 on disk 아이노드의 구조체는 아래와 같았다.

```
struct inode_disk
{
    block_sector_t start;   /* First data sector */
    off_t length;           /* File size in bytes */
    unsigned magic;         /* Magic number */
    uint32_t unused[125];   /* Not Used */
}
```

start는 시작 블록 번호이고, length는 할당 된 블록 길이(byte)를 나타난다. 파일 생성 시에 디스크 상에 연속된 블록을 할당 받는다. 이렇기 때문에 위의 그림을 참고하면 파일 A의 크기를 늘리지 못했던 것이다. 아래와 같이 inode가 갖고 있는 파일을 배열을 통해 연속으로 할당되어 있지 않더라도 블록들의 주소를 가리키도록 수정 할 것이다.

```
struct inode_disk{
    off_t length;           /* File size in bytes */
    unsigned magic;         /* Magic number */
    block_sector_t direct_map_table[DIRECT_BLOCK_ENTRIES];
    block_sector_t indirect_block_sec;
    block_sector_t double_indirect_block_sec;
}
```



블록 위치를 direct, indirect, double indirect 방식으로 표현하여 inode_disk 자료구조의 크기가 1블록 크기가 되도록, direct 방식으로 표현할 블록의 수를 정한다.
데이터 블록들로만 모두 채우게 되면 아이노드가 표현할 수 있는 파일 크기가 작아진다. 그러하기 때문에 위와 같이 파일 구조를 변경해주어 파일의 크기를 자유롭게 조절할 수 있도록 한다.

3. 함수 구현 및 수정

블록 번호 표현 방법 변경

filesys/inode.c

```
/* direct_map_table의 엔트리 개수 */
#define DIRECT_BLOCK_ENTRIES 123
/* indirect_map_table의 엔트리 개수 */
#define INDIRECT_BLOCK_ENTRIES (BLOCK_SECTOR_SIZE / sizeof (block_sector_t))

#define PATH_MAX 256

/* On-disk inode.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct inode_disk {
  off_t length;                       /* File size in bytes. */
  uint32_t is_dir;
  block_sector_t direct_map_table[DIRECT_BLOCK_ENTRIES];
  block_sector_t indirect_block_sec;
  block_sector_t double_indirect_block_sec;
  unsigned magic;                     /* Magic number. */
};

```

기존엔 디스크에 데이터가 저장된 블록이 연속적으로 할당되어있었고, 동적으로 유동적으로 파일의 크기가 변경 불가능하여 파일 생성 초기에 연속된 블록의 개수가 결정되었었다. 하지만 지금은 데이터가 저장된 블록을 간접 참조로 유동적으로 가리킬 수 있어 동적으로 파일의 크기를 변경할 수 있다.

inode_disk 는 핀토스에서는 512 바이트이며, 한 블록 전체를 차지한다. DIRECT_BLOCK_ENTRIES 매크로 상수로 direct 방식의 맵 테이블의 배열 크기를 결정하여 inode_disk 구조체의 크기가 512 바이트가 되게한다.

indirect_block_sec 은 indirect block map table 이 저장되어있는 디스크 섹터 번호를 나타내며, double_indirect_block_sec 마찬가지로 이중 참조로 indirect block map table 를 거쳐 데이터가 저장된 sector 를 유지할 수 있도록 indirect block map table 가 저장된 섹터 번호를 배열에 담아 저장되어 있는 섹터 번호를 저장한다. 어떤 방식을 이용하여 섹터 번호를 저장할지는 파일의 오프셋에 따라 결정된다.

enum direct_t 변수 추가

filesys/inode.c

```
33 enum direct_t {
34   NORMAL_DIRECT,
35   INDIRECT,
36   DOUBLE_INDIRECT,
37   OUT_LIMIT
38 };
```

sector_location 구조체가 데이터 블록이 저장되어있는 섹터 번호를 어떤 방식으로 inode_disk 에 저장했는지를 나타내고, 해당 map_table 의 index 를 저장하는데 그 때 파일의 해당 오프셋이 direct 인지 indirect 인지 double indirect 인지 구분할 수 있게 enum 을 정의하였다.

sector_location 자료구조 추가

filesys/inode.c

```
40 struct sector_location {
41   int directness;
42   int index1;
43   int index2;
44 };
```

파일의 데이터가 저장된 sector 번호가 inode_disk 구조체의 어디에 저장되어있는지 나타낼 때 사용하는 구조체이다.

inode_indirect_block 자료구조 추가

filesys/inode.c

```
46 struct inode_indirect_block {
47   block_sector_t map_table[INDIRECT_BLOCK_ENTRIES];
48 };
```

각 inode_indirect_block 은 내부에 block_sector_t (섹터 번호)를 배열로 하는 map_table 을 저장하고 있다.

struct inode 자료구조 수정

filesys/inode.c

```
58 /* In-memory inode. */
59 struct inode {
60     struct list_elem elem;              /* Element in inode list. */
61     block_sector_t sector;              /* Sector number of disk location. */
62     int open_cnt;                       /* Number of openers. */
63     bool removed;                       /* True if deleted, false otherwise. */
64     int deny_write_cnt;                 /* 0: writes ok, >0: deny writes. */
65     struct lock extend_lock;
66 };
67
```

기존엔 on-disk-inode 를 in-memory-inode 에 저장해놓고 참조를 했었던 반면, 지금은 필요에 따라 sector 번호를 참조하여 디스크에서 읽어온다.

on-disk inode 획득 함수 구현

filesys/inode.c

```
68 static bool get_disk_inode (const struct inode *inode, struct inode_disk *inode_disk) {
69   /*  inode->sector에 해당하는 on-disk inode를 buffer cache에서
70       읽어 inode_disk에 저장 (bc_read() 함수 사용) */
71   bc_read (inode->sector, inode_disk, 0, sizeof (struct inode_disk), 0);
72   /*  true 반환 */
73   return true;
74 }
75
```

기존 in-memory-inode 에서 on-disk-inode 멤버를 지우고, 필요에 따라 디스크에서 읽어올 수 있도록 함수를 만들어 사용한다.

인덱스 블록 내 오프셋 계산함수 구현

filesys/inode.c

```
76 static void locate_byte (off_t pos, struct sector_location *sec_loc)
77 {
78   off_t pos_sector = pos / BLOCK_SECTOR_SIZE;
79
80   /*  Direct 방식일 경우 */
81   if (pos_sector < DIRECT_BLOCK_ENTRIES) {
82
83     //sec_loc 자료구조의 변수 값 업데이트(구현)
84     sec_loc->directness = NORMAL_DIRECT;
85     sec_loc->index1 = pos_sector;
86
87     /* indirect 방식일 경우 */
88   } else if (pos_sector < (off_t)(INDIRECT_BLOCK_ENTRIES + DIRECT_BLOCK_ENTRIES)) {
89     sec_loc->directness = INDIRECT;
90     sec_loc->index1 = (pos_sector - DIRECT_BLOCK_ENTRIES);
91
92     /* double indirect 방식일 경우 */
93   } else if (pos_sector < (off_t)(INDIRECT_BLOCK_ENTRIES * (INDIRECT_BLOCK_ENTRIES + 1) + DIRECT_BLOCK_ENTRIES)) {
94     sec_loc->directness = DOUBLE_INDIRECT;
95     sec_loc->index1 = (pos_sector - (DIRECT_BLOCK_ENTRIES + INDIRECT_BLOCK_ENTRIES)) / INDIRECT_BLOCK_ENTRIES;
96     sec_loc->index2 = (pos_sector - (DIRECT_BLOCK_ENTRIES + INDIRECT_BLOCK_ENTRIES)) % INDIRECT_BLOCK_ENTRIES;
97
98     /* 파일 최대 표현 크기 초과 */
99   } else {
100    sec_loc->directness = OUT_LIMIT;
101  }
102
103  return;
104 }
105
```

주어진 파일 오프셋에 따라 inode_disk 구조체에 어떤 방식으로 어떤 인덱스에 섹터번호를 저장해야 할지 결정해주는 함수이다.

inode 업데이트 함수 구현

filesys/inode.c

```
106 static bool register_sector (struct inode_disk *inode_disk,
107     block_sector_t new_sector, struct sector_location sec_loc)
108 {
109   struct inode_indirect_block *new_block = NULL, *double_indirect_directory_block = NULL;
110   bool to_write = false;
111   switch (sec_loc.directness)
112   {
113     case NORMAL_DIRECT:
114       /*  inode_disk에 새로 할당받은 디스크 번호 업데이트 */
115       inode_disk->direct_map_table[sec_loc.index1] = new_sector;
116       break;
117
118     case INDIRECT:
119       new_block = (struct inode_indirect_block*)malloc (BLOCK_SECTOR_SIZE);
120       if (new_block == NULL)
121         return false;
122       /*  인덱스 블록에 새로 할당 받은 블록 번호 저장 */
123       if (inode_disk->indirect_block_sec == (block_sector_t)-1) {
124         if (!free_map_allocate (1, &inode_disk->indirect_block_sec))
125           return false;
126         memset (new_block, 0xFF, sizeof (struct inode_indirect_block));
127       } else {
128         bc_read (inode_disk->indirect_block_sec, new_block, 0, BLOCK_SECTOR_SIZE, 0);
129       }
130
131       new_block->map_table[sec_loc.index1] = new_sector;
132
133       /*  인덱스 블록을 buffer cache에 기록 */
134       bc_write (inode_disk->indirect_block_sec, new_block, 0, BLOCK_SECTOR_SIZE, 0);
135       break;
136
```

새로 할당받은 섹터번호와 파일 오프셋으로 결정된 sector_location 으로 실제로 inode_disk 에 해당 섹터번호를 주어진 인덱스에 맞춰서 저장하는 함수이다. sec_loc.directness 로 어떤 방식을 사용하여 섹터번호를 저장할 지 결정한다.

```
137     case DOUBLE_INDIRECT:
138       double_indirect_directory_block = (struct inode_indirect_block*)malloc (BLOCK_SECTOR_SIZE);
139       if (new_block == NULL)
140         return false;
141       /*  2차 인덱스 블록에 새로 할당 받은 블록 주소 저장 후,
142        각 인덱스 블록을 buffer cache에 기록 */
143       if (inode_disk->double_indirect_block_sec == (block_sector_t)-1) {
144         if (!free_map_allocate (1, &inode_disk->double_indirect_block_sec))
145           return false;
146         memset (double_indirect_directory_block, 0xFF, sizeof (struct inode_indirect_block));
147       } else {
148         bc_read (inode_disk->double_indirect_block_sec, double_indirect_directory_block, 0, sizeof (struct inode_indirect_block), 0);
149       }
150
```

double indirect 방식으로는 처음에는 double_indirect_block_sec 섹터 번호에 저장된 map_table 을 로드하여 다시 해당 map_table 에서 주어진 인덱스를 참조하여 2 차 map_table 를 로드한다.

```
155       if (double_indirect_directory_block->map_table[sec_loc.index1] == (block_sector_t)-1) {
156         if (!free_map_allocate (1, &double_indirect_directory_block->map_table[sec_loc.index1]))
157           return false;
158         memset (new_block, 0xFF, sizeof (struct inode_indirect_block));
159         to_write = true;
160       } else {
161         bc_read (double_indirect_directory_block->map_table[sec_loc.index1], new_block, 0, BLOCK_SECTOR_SIZE, 0);
162       }
163
164       ASSERT (new_block->map_table[sec_loc.index2] == (block_sector_t)-1);
165       new_block->map_table[sec_loc.index2] = new_sector;
166
167       if (to_write == true) {
168         bc_write (inode_disk->double_indirect_block_sec, double_indirect_directory_block, 0, BLOCK_SECTOR_SIZE, 0);
169         free (double_indirect_directory_block);
170       }
171
172       /*  인덱스 블록을 buffer cache에 기록 */
173       bc_write (inode_disk->indirect_block_sec, new_block, 0, BLOCK_SECTOR_SIZE, 0);
174       break;
175
176     default:
177       return false;
178   }
179   free(new_block);
180   return true;
181 }
```

그 과정에서 기존에 할당받은 적 없는 경우에는 메모리를 새로 할당받아 -1 로 초기화를 하고 이 함수에서 주어진 인덱스에 섹터번호를 저장하고 새로 생성된 내역을 해당 섹터에 저장한다.

파일 오프셋으로 블록 번호 검색 구현

filesys/inode.c

```
187 static block_sector_t
188 byte_to_sector (const struct inode_disk *inode_disk, off_t pos)
189 {
190   block_sector_t result_sec = -1;
191
192   if (pos < inode_disk->length) {
193     struct inode_indirect_block *ind_block;
194     struct sector_location sec_loc;
195     locate_byte(pos, &sec_loc); // 인덱스 블록 offset 계산
196
197     switch (sec_loc.directness) {
198
199       /*  Direct 방식일 경우 */
200       case NORMAL_DIRECT :
201         /*  on-disk inode의 direct_map_table에서 디스크 블록 번호를 얻음 */
202         result_sec = inode_disk->direct_map_table[sec_loc.index1];
203         break;
204
```

주어진 파일 오프셋이 어떤 섹터번호에 저장되어 있는지 찾아주는 함수이다.

```
205       /*  Indirect 방식일 경우 */
206      case INDIRECT :
207       ind_block = (struct inode_indirect_block*)malloc (BLOCK_SECTOR_SIZE);
208       if (ind_block) {
209         /*  buffer cache에서 인덱스 블록을 읽어 옴 */
210         bc_read (inode_disk->indirect_block_sec, ind_block, 0, BLOCK_SECTOR_SIZE, 0);
211         ASSERT (ind_block->map_table[sec_loc.index1] != -1);
212
213         /*  인덱스 블록에서 디스크 블록 번호 확인 */
214         result_sec = ind_block->map_table[sec_loc.index1];
215       } else {
216         NOT_REACHED ();
217       }
218       free (ind_block);
219       break;
220
```

파일 오프셋의 크기에 따라 어떤 방식으로 inode_disk 에 섹터번호가 저장되어있는지를 결정하고 switch 문에서 분기하여 해당 방식에 맞는 처리 루틴으로 보내준다.

```
221       /*  Double indirect 방식일 경우 */
222      case DOUBLE_INDIRECT :
223       ind_block = (struct inode_indirect_block *)malloc (BLOCK_SECTOR_SIZE);
224       if (ind_block){
225         /*  1차 인덱스 블록을 buffer cache에서 읽음 */
226         bc_read (inode_disk->double_indirect_block_sec, ind_block, 0, BLOCK_SECTOR_SIZE, 0);
227
228         /*  2차 인덱스 블록을 buffer cache에서 읽음 */
229         ASSERT (ind_block->map_table[sec_loc.index1] != -1);
230         bc_read (ind_block->map_table[sec_loc.index1], ind_block, 0, BLOCK_SECTOR_SIZE, 0);
231
232         /*  2차 인덱스 블록에서 디스크 블록 번호 확인 */
233         result_sec = ind_block->map_table[sec_loc.index2];
234       } else {
235         NOT_REACHED ();
236       }
237       free (ind_block);
238       break;
239
240      default :
241       NOT_REACHED ();
242
243     }
244   }
245
246   return result_sec;
247 }
248
```

파일 크기 업데이트 함수 구현

filesys/inode.c

```
249 bool inode_update_file_length (struct inode_disk* inode_disk, off_t start_pos, off_t end_pos) {
250
251   off_t size = end_pos - (start_pos - 1);
252   off_t offset = start_pos;
253   void *zeroes = NULL;
254   struct sector_location sec_loc;
255   memset (&sec_loc, 0x00, sizeof (struct sector_location));
256   zeroes = calloc (sizeof (char), BLOCK_SECTOR_SIZE);
257
```

이번 과제를 통해 기존 파일의 크기를 변경하지 못했던 것을 동적으로 파일 크기를 새로 할당할 수 있게 되었다. 이 함수로 시작 오프셋과 마지막 오프셋을 인자로 넘겨주면 해당 범위 내에 섹터를 할당받지 못한 오프셋 범위가 있다면 새로 섹터번호를 할당해주어 파일 크기를 조정할 수 있게 된다.

```
258   /*  블록 단위로 loop을 수행하며 새로운 디스크 블록 할당 */
259   while (size > 0) {
260
261     /*  디스크 블록 내 오프셋 계산 */
262     int sector_ofs = offset % BLOCK_SECTOR_SIZE;
263     int chunk_size = BLOCK_SECTOR_SIZE - sector_ofs;
264     off_t sector_idx = 0;
265
266     if (sector_ofs > 0) {
267       /*  블록 오프셋이 0보다 클 경우, 이미 할당된 블록 */
268     } else {
269       /*  새로운 디스크 블록을 할당 */
270       if (free_map_allocate (1, &sector_idx)) {
271         locate_byte (offset, &sec_loc);
272
273         /*  inode_disk에 새로 할당 받은 디스크 블록 번호 업데이트 */
274         if (!register_sector (inode_disk, sector_idx, sec_loc)) {
275           free (zeroes);
276           return false;
277         }
278
279       } else {
280         free (zeroes);
281         return false;
282       }
283       /*  새로운 디스크 블록을 0으로 초기화 */
284       bc_write (sector_idx, zeroes, 0, BLOCK_SECTOR_SIZE, 0);
285     }
286     /*  Advance. */
287     size -= chunk_size;
288     offset += chunk_size;
289
290   }
291   free (zeroes);
292   return true;
293 }
294
```

디스크 블록 할당 해제 함수 구현

filesys/inode.c

```
295 static void free_inode_sectors (struct inode_disk *inode_disk) {
296   struct inode_indirect_block *ind_block_1 = NULL;
297   struct inode_indirect_block *ind_block_2 = NULL;
298   int i = 0, j = 0;
299
300   ind_block_1 = (struct indirect_block_sec*)malloc (BLOCK_SECTOR_SIZE);
301   ind_block_2 = (struct indirect_block_sec*)malloc (BLOCK_SECTOR_SIZE);
302
303   /*  Double indirect 방식으로 할당된 블록 해지 */
304   if ((int)inode_disk->double_indirect_block_sec > (int)0) {
305     /*  1차 인덱스 블록을 buffer cache에서 읽음 */
306     bc_read (inode_disk->double_indirect_block_sec, ind_block_1, 0, BLOCK_SECTOR_SIZE, 0);
307     i = 0;
308     /*  1차 인덱스 블록을 통해 2차 인덱스 블록을 차례로 접근 */
309     while ((int)ind_block_1->map_table[i] > 0) {
310       /*  2차 인덱스 블록을 buffer cache에서 읽음 */
311       j = 0;
312       bc_read (ind_block_2->map_table[j], ind_block_2, 0, BLOCK_SECTOR_SIZE, 0);
313       /*  2차 인덱스 블록에 저장된 디스크 블록 번호를 접근 */
314       while ((int)ind_block_2->map_table[j] > 0) {
315         /*  free_map 업데이틀 통해 디스크 블록 할당 해지 */
316         free_map_release (ind_block_2->map_table[j], 1);
317         j++;
318       }
319       /*  2차 인덱스 블록 할당 해지 */
320       free_map_release (ind_block_1->map_table[i], 1);
321       i++;
322     }
323   }
324
```

파일을 삭제할 경우 디스크의 어떤 sector 가 할당되어 사용중인지 아닌지를 bitmap 으로 관리하는 free_map 에 해당 섹터들의 bit 를 0 으로 표시하여 할당 가능하도록 만들어야한다. 이 과정은 on-disk-inode 의 map_table 를 순회하여 유효한 섹터번호들에 대해 free_map_release() 함수를 호출하여 free_map 을 갱신한다.

```
325   /*  1차 인덱스 블록 할당 해지 */
326   /*  Indirect 방식으로 할당된 디스크 블록 해지*/
327   if ((int)inode_disk->indirect_block_sec > (int)0) {
328     /*  인덱스 블록을 buffer cache에서 읽음 */
329     bc_read (inode_disk->indirect_block_sec, ind_block_1, 0, BLOCK_SECTOR_SIZE, 0);
330     i = 0;
331     /*  인덱스 블록에 저장된 디스크 블록 번호를 접근 */
332     while((int)ind_block_1->map_table[i] > 0) {
333       /*  free_map 업데이트를 통해 디스크 블록 할당 해지 */
334       free_map_release (ind_block_1->map_table[i], 1);
335       i++;
336     }
337   }
338
339   /*  Direct 방식으로 할당된 디스크 블록 해지*/
340   i = 0;
341   while ((int)inode_disk->direct_map_table[i] > 0) {
342     /*  free_map 업데이트를 통해 디스크 블록 할당 해지 */
343     free_map_release (inode_disk->direct_map_table[i], 1);
344     i++;
345   }
346
347   free (ind_block_1);
348   free (ind_block_2);
349
350 }
```

double_indirect, indirect, direct 세가지 방식의 map_table 를 전부 검사한다. 하지만 저장된 섹터번호가 -1(혹은 unsigned int 의 최대값 2^32-1)인지 아닌지 검사하여 전부 순회할 필요는 없게 구현하였다.

inode_create() 함수 수정

filesys/inode.c :: inode_create()

```
393
394     if (length > 0) {
395       /*  length 만큼의 디스크 블록을 inode_updafe_file_length()를 호출하여 할당 */
396       inode_update_file_length (disk_inode, 0, length - 1);
397     }
398
399     /*  on-disk inode를 bc_write()를 통해buffer cache에 기록 */
400     bc_write (sector, disk_inode, 0, sizeof (struct inode_disk), 0);
401
402     /*  할당받은 disk_inode 변수 해제 */
403     free (disk_inode);
404
405     /*  success 변수 update */
406     success = true;
407   }
408   return success;
409 }
410
```

inode_create() 함수를 통해 새로 파일을 생성하면서 파일의 크기를 부여하고, 새로 생성한 on-disk-inode 를 디스크에 저장한다.

inode_open() 함수 수정

filesys/inode.c

```
414 struct inode *inode_open (block_sector_t sector) {
415
416   struct list_elem *e;
417   struct inode *inode;
418
419   /* Check whether this inode is already open. */
420   for (e = list_begin (&open_inodes); e != list_end (&open_inodes);
421        e = list_next (e))
422   {
423     inode = list_entry (e, struct inode, elem);
424     if (inode->sector == sector)
425     {
426       inode_reopen (inode);
427       return inode;
428     }
429   }
430
431   /* Allocate memory. */
432   inode = malloc (sizeof *inode);
433   if (inode == NULL)
434     return NULL;
435
436   /* Initialize. */
437   list_push_front (&open_inodes, &inode->elem);
438   lock_init (&inode->extend_lock);
439   inode->sector = sector;
440   inode->open_cnt = 1;
441   inode->deny_write_cnt = 0;
442   inode->removed = false;
443   return inode;
444 }
445
```

파일을 열면서 on-disk-inode 를 in-memory-inode 로 load(실제로는 섹터 번호만)하는데, inode 를 초기화 하면서 extend_lock 도 같이 초기화 할 수 있도록 한다. 이 extend_lock 은 on_disk_inode 에 대한 접근을 동기적으로 수행하기 위해 사용한다.

inode_read_at() 함수 수정

filesys/inode.c

```
510 off_t
511 inode_read_at (struct inode *inode, void *buffer_, off_t size, off_t offset)
512 {
513   uint8_t *buffer = buffer_;
514   off_t bytes_read = 0;
515   struct inode_disk *disk_inode = (struct inode_disk*)malloc (sizeof (struct inode_disk));
516   ASSERT (disk_inode);
517   get_disk_inode (inode, disk_inode);
518
519   while (size > 0)
520   {
521     /* Disk sector to read, starting byte offset within sector. */
522     block_sector_t sector_idx = byte_to_sector (disk_inode, offset);
523     int sector_ofs = offset % BLOCK_SECTOR_SIZE;
524
525     /* Bytes left in inode, bytes left in sector, lesser of the two. */
526     off_t inode_left = inode_length (inode) - offset;
527     int sector_left = BLOCK_SECTOR_SIZE - sector_ofs;
528     int min_left = inode_left < sector_left ? inode_left : sector_left;
529
530     /* Number of bytes to actually copy out of this sector. */
531     int chunk_size = size < min_left ? size : min_left;
532     if (chunk_size <= 0)
533       break;
534
535     bc_read (sector_idx, buffer, bytes_read, chunk_size, sector_ofs);
536
537     /* Advance. */
538     size -= chunk_size;
539     offset += chunk_size;
540     bytes_read += chunk_size;
541   }
542   free (disk_inode);
543
544   return bytes_read;
545 }
546
```

on-disk-inode 를 디스크에서 읽어와 사용할 수 있도록 수정한다.

inode_write_at() 함수 수정

filesys/inode.c :: inode_write_at()

```
569   lock_acquire (&inode->extend_lock);
570
571   old_length = disk_inode->length;
572   write_end = offset + (size - 1);
573   if (write_end > (old_length - 1)) {
574     /*  파일길이가 증가하였을 경우, on-disk inode 업데이트 */
575     if(!inode_update_file_length (disk_inode, old_length, write_end)) {
576       NOT_REACHED ();
577     }
578     disk_inode->length = write_end + 1;
579     bc_write (inode->sector, disk_inode, 0, BLOCK_SECTOR_SIZE, 0);
580   }
581
582   lock_release (&inode->extend_lock);
583
584   while (size > 0)
585   {
586     /* Sector to write, starting byte offset within sector. */
587     block_sector_t sector_idx = byte_to_sector (disk_inode, offset);
588     int sector_ofs = offset % BLOCK_SECTOR_SIZE;
589
590     ASSERT (sector_idx != -1);
591
592     /* Bytes left in inode, bytes left in sector, lesser of the two. */
593     off_t inode_left = inode_length (inode) - offset;
594     int sector_left = BLOCK_SECTOR_SIZE - sector_ofs;
595     int min_left = inode_left < sector_left ? inode_left : sector_left;
596
```

파일에 쓰기를 하려는데 오프셋이 파일의 크기를 벗어난다면 파일의 크기를 늘려주어야한다. 그 과정에서 inode_disk 의 변경이 일어나므로 extend_lock 을 사용하여 동기적으로 수행되게 한다.

inode_close() 함수 수정

```
465 void
466 inode_close (struct inode *inode)
467 {
468   struct inode_disk *disk_inode = (struct inode_disk*)malloc (sizeof (struct inode_disk));
469   ASSERT (disk_inode);
470   /* Ignore null pointer. */
471   if (inode == NULL)
472     return;
473
474   /* Release resources if this was the last opener. */
475   if (--inode->open_cnt == 0)
476   {
477     /* Remove from inode list and release lock. */
478     list_remove (&inode->elem);
479
480     /* Deallocate blocks if removed. */
481     if (inode->removed)
482     {
483       /*  inode의 on-disk inode 획득(get_disk_inode() 이용) */
484       get_disk_inode (inode, disk_inode);
485       ASSERT (disk_inode);
486       /*  디스크 블록 반환(free_inode_sectors() 이용) */
487       free_inode_sectors (disk_inode);
488       /*  on-disk inode 반환 (free_map_release() 이용) */
489       free_map_release (inode->sector, 1);
490     }
491
492     /*  disk inode 변수 할당 해제 (free() 이용) */
493     free (disk_inode);
494     free (inode);
495   }
496 }
```

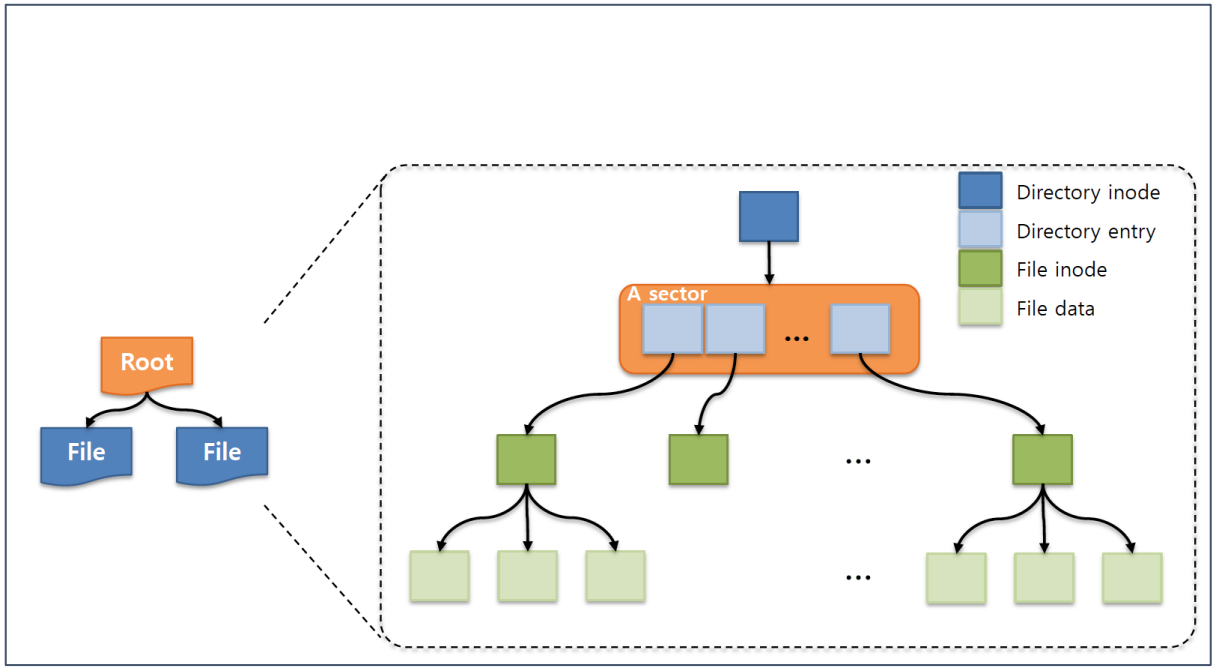
기존 방식으로는 연속된 블록 섹터번호를 free_map 에서 해제해주면 되었는데, 지금은 indirect 방식으로 재구현 되어 각 map_table 을 순하여 유효한 섹터번호를 free_map 에서 해제해주어야한다. 하여 free_inode_sectors() 함수를 이용하여 파일의 데이터가 저장된 섹터번호를 해제해주고, on-disk-inode 가 저장된 섹터번호도 해제해준다.

FileSystem

- Subdirectory

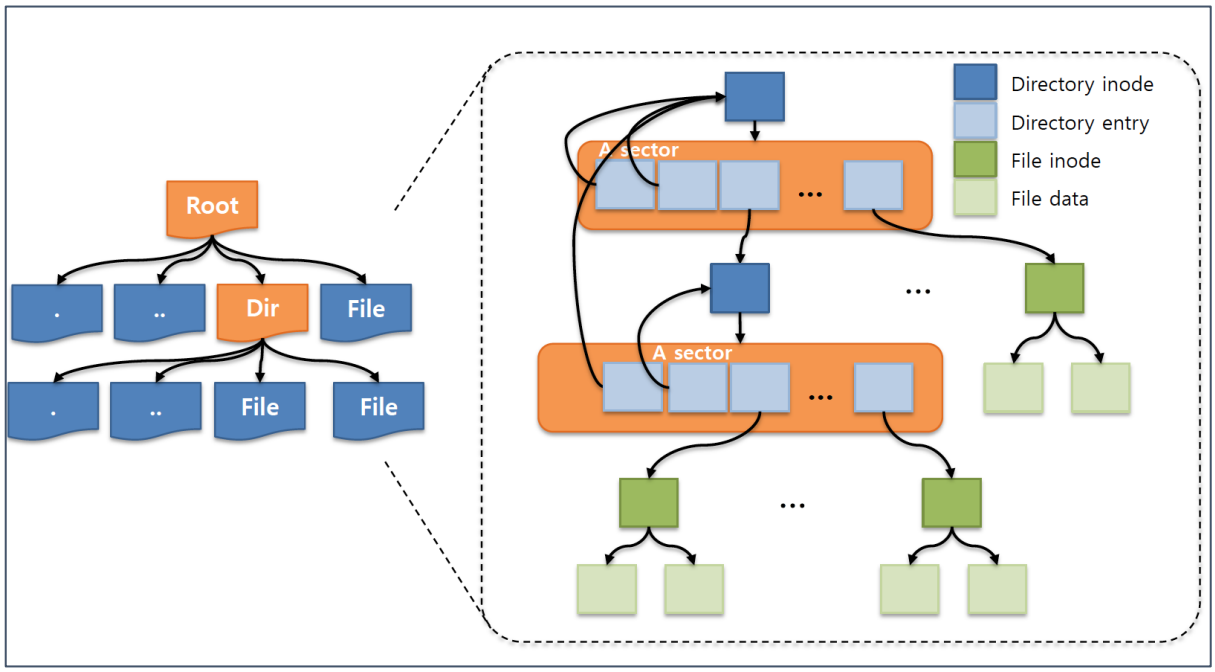
1. 과제 목표

현재 핀토스에서는 아래의 그림과 같이 root 디렉토리만 존재하는 단일계층으로 root 디렉토리에만 파일을 생성한다. 본 과제에서는 root 디렉터리내에 파일과 디렉터리를 생성할 수 있도록 계층구조를 구현한다.



2. 과제설명

아래의 그림은과제를 구현 한후에 root 디렉토리 내에 디렉토리와 파일을 생성할 수 있게 된 것을 표현한 것이다.



1. 현재 single directory에서 디렉토리가 계층 구조를 구현하도록 하여야 한다. 디렉토리 엔트리 구조를 -> 파일과 다른 디렉토리들을 포인팅 하도록 수정하여야 한다. '.' , '..' 기능을 구현하여야 한다.
2. 스레드의 현재 작업디렉토리 정보를 추가 해줘야한다. 절대경로, 상대 경로 기능을 구현하여야 한다. 이는 '/'구분자에 의해 디렉토리를 구분할 수 있도록 한다. (userprogram과제를 하면서 공부했었던 '/'를 기점으로 파싱을 해주는 방법을 사용하면 된다.)
3. create, open, remove 시스템콜을 수정하고 디렉토리 관련 시스템콜을 추가한다.

3. 함수 수정 및 구현

과제를 아직 수행하지 못하였습니다. 죄송합니다.

make check 결과

```
FAIL tests/filesys/extended/dir-rm-root-persistence
FAIL tests/filesys/extended/dir-rm-tree-persistence
FAIL tests/filesys/extended/dir-rmdir-persistence
FAIL tests/filesys/extended/dir-under-file-persistence
FAIL tests/filesys/extended/dir-vine-persistence
FAIL tests/filesys/extended/grow-create-persistence
FAIL tests/filesys/extended/grow-dir-lg-persistence
FAIL tests/filesys/extended/grow-file-size-persistence
FAIL tests/filesys/extended/grow-root-lg-persistence
FAIL tests/filesys/extended/grow-root-sm-persistence
FAIL tests/filesys/extended/grow-seq-lg-persistence
FAIL tests/filesys/extended/grow-seq-sm-persistence
FAIL tests/filesys/extended/grow-sparse-persistence
FAIL tests/filesys/extended/grow-tell-persistence
FAIL tests/filesys/extended/grow-two-files-persistence
FAIL tests/filesys/extended/syn-rw-persistence
32 of 121 tests failed.
make[1]: *** [check] 오류 1
make[1]: Leaving directory `/home/ckdqja0225/project/src_changbeom/filesys/build'
make: *** [check] 오류 2
ckdqja0225@ubuntu:~/project/src_changbeom/filesys$
```

bitbucket 주소

git clone

https://seomooneunji@bitbucket.org/pintos_ejcb/project.git

에서 " **src_changbeom 폴더** " 입니다.