Operating System Lab
Part 4: Filesystem

KAIST EE

Youjip Won

Overview of Filesystem

- Background of Filesystem in Pintos
- To Do's in project 4
  - Buffer Cache
  - Indexed and Extensible Files
  - Subdirectories

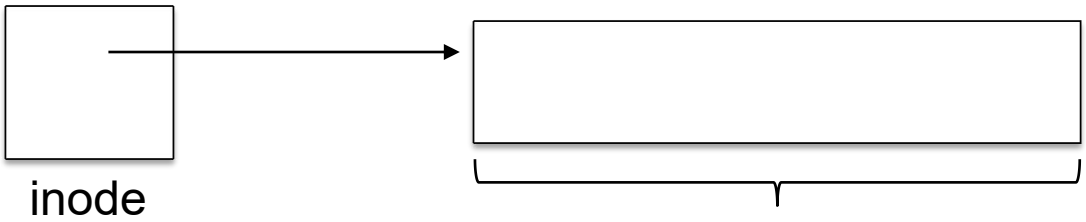
Background

Basic concepts

- inode
  - Represents a file on the disk
  - File size
  - Pointers to the disk block(s)
  - Attributes: permission, access time, modification time and etc.
  - On disk inode
  - In-memory inode = on-disk inode + on-disk location of the inode
- File object
  - Represent an "open" file.
  - Current offset to perform read/write
  - Filesystem type it belongs: EXT4

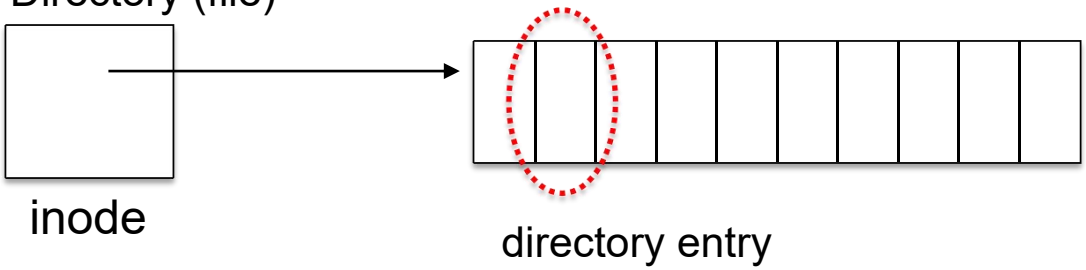
concepts

Regular file



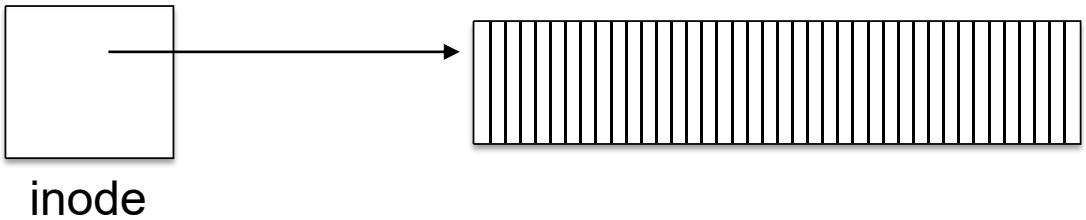
{start address, offset}

Directory (file)



Array of directory entries

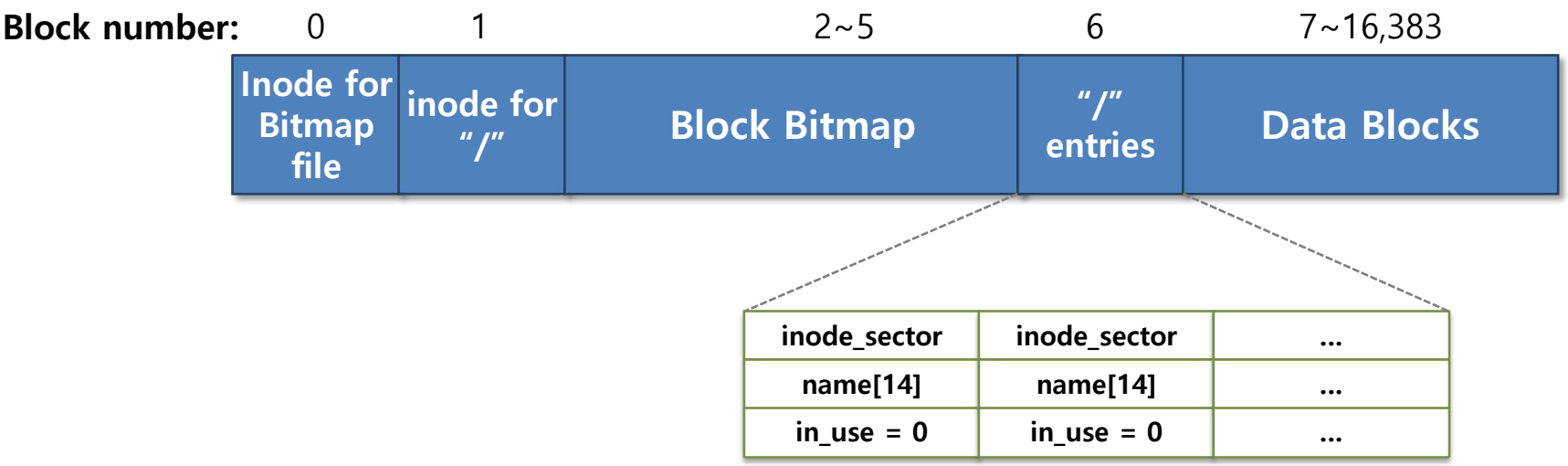
Bitmap (file)



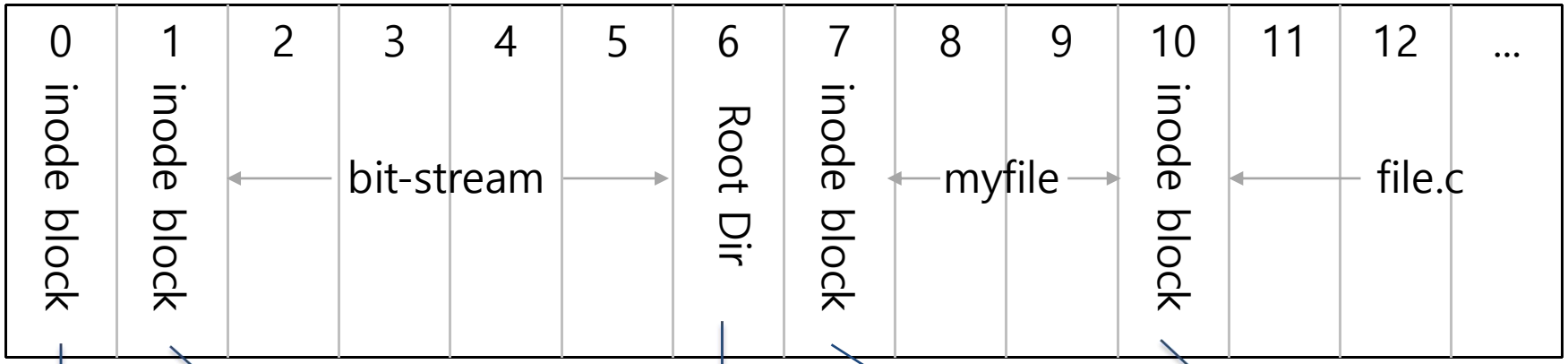
bitmap

Filesystem layout in pintos

- Example of 8MByte Filesystem in Pintos
  - Block Size (=Sector size): 512 Byte
  - 8MByte/512Byte = 16,384 = Total number of blocks
  - Block bitmap: 4*512*8 bits = 16,384 bit
  - In PINTOS, block bitmap is represented as a "file".



Sample Filesystem Layout in Pintos



Inode for bitmap

```
start : 2
length : 204
8
magic
unused[125]
```

Inode for "/"

```
start : 6
length : 320
magic
unused[125]
```

entries in "/"

fie name	inode No.
myfile	7
file.c	10
...	

myfile inode

```
start : 8
length : 1024
magic
unused[125]
```

file.c inode

```
start : 11
length : 204
8
magic
unused[125]
```

In-memory inode

- In-memory inode: `struct inode`
  - `sector` : block number where inodes are stored
  - `data` : `disk_inode` data
  - `removed` : Whether to delete the file

pintos/src/filesys/inode.c

```
struct inode {
    struct list_elem elem;  /* Element in inode list. */
    block_sector_t sector;
    int open_cnt;           /* Number of openers. */
    bool removed;
    int deny_write_cnt;     /* 0: writes ok, >0: deny writes.*/
    struct inode_disk data; /* Inode content. */
};
```

On-disk inode

- On-disk inode (`struct inode_disk`)
  - `start` : start address of file data in block address
  - `length` : size of file(byte)
  - `unused[125]` : unused area
  - One inode occupies a single sector.

pintos/src/filesys/inode.c

```
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. */
};
```

Directory object

- Represent an open directory.
  - `inode` : the pointer to the associated in-memory inode
  - `pos` : position of the next directory entry to read/write

pintos/src/filesys/directory.c

```
struct dir
{
    struct inode *inode;    /* Backing store. */
    off_t pos;              /* Current position. */
};
```

Directory entry

- Indicate information in directory entry (file or directory)
  - `inode_sector` : sector number of the inode (inode size is 512 byte)
  - file name: up to 14 characters
  - `in_use` : Whether to use `dir_entry`

pintos/src/filesys/directory.c

```
struct dir_entry
{
    block_sector_t inode_sector;
    char name[NAME_MAX + 1]; /* NAME_MAX = 14*/
    bool in_use;
};
```

Block bitmap

- `free_map`
  - bitmap to represent status of the blocks in the filesystem partition
  - `free_map_file` : bitmap is stored as a file.
  - `bit_cnt` : Number of disk blocks in entire file system

pintos/src/kernel/bitmap.c

```
static struct bitmap *free_map;
static struct file *free_map_file;
struct bitmap
{
    size_t bit_cnt;
    elem_type *bits;
};
```

File structure

- `struct file`
  - Created when a file is open.
  - `inode` : pointer to the file's in-memory inode
  - `pos` : current file offset
  - `deny_write`: indicate whether a file is writable.

pintos/src/filesys/file.c

```
struct file {
    struct inode *inode;        /* File's inode. */
    off_t pos;                  /* Current position. */
    bool deny_write;
};
```

To Do's

- Buffer cache
  - Allocate buffer cache (64 blocks).
  - Cache data blocks.
    - When read or write data blocks, read and save it in buffer cache
  - Write dirty data blocks.
    - When dirty data blocks is evicted to reclaim buffer cache entry
    - When filesystem is shut down
- Indexed and Extensible File
  - Implement block pointers in inode
    - direct, single-indirect, double-indirect
- Subdirectories
  - Implement hierarchical name space for file

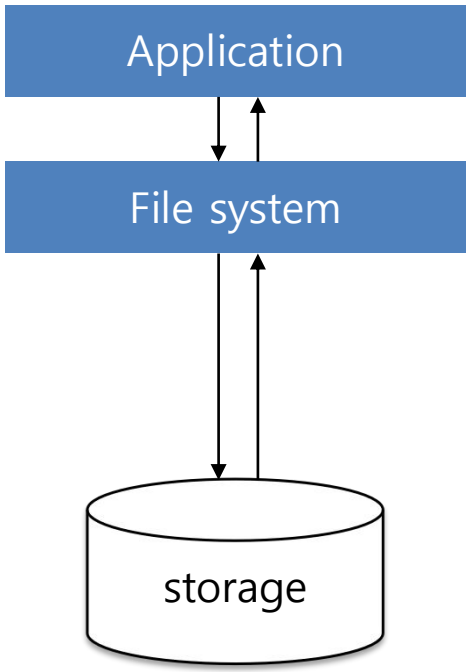
Buffer Cache

Buffer cache

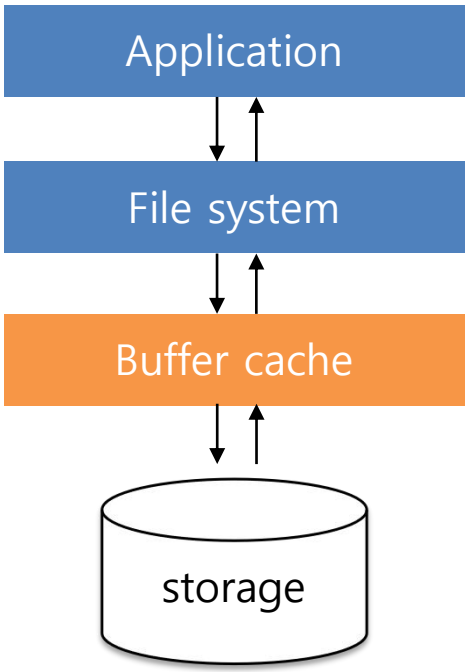
- Buffer Cache: using the part of the memory as disk.
  - Consecutive physical pages
  - Initialized when the system starts or when a filesystem is mounted.
  - Virtual Memory: using the part of the disk as memory.
- In current pintos, there is no cache for disk I/O.
- In reality, most OS's have a cache for disk I/O.
- Modify the filesystem to cache the file blocks.
  - Cache the data blocks.
  - Capacity of cache: 64 blocks
- File to modify
  - pintos/src/filesys/inode.c

Background - Pintos's read/write

- Current Pintos accesses storage on user `read/write` requests.
- Change user I/O to be performed through buffer cache (memory)



Before



After

To do's

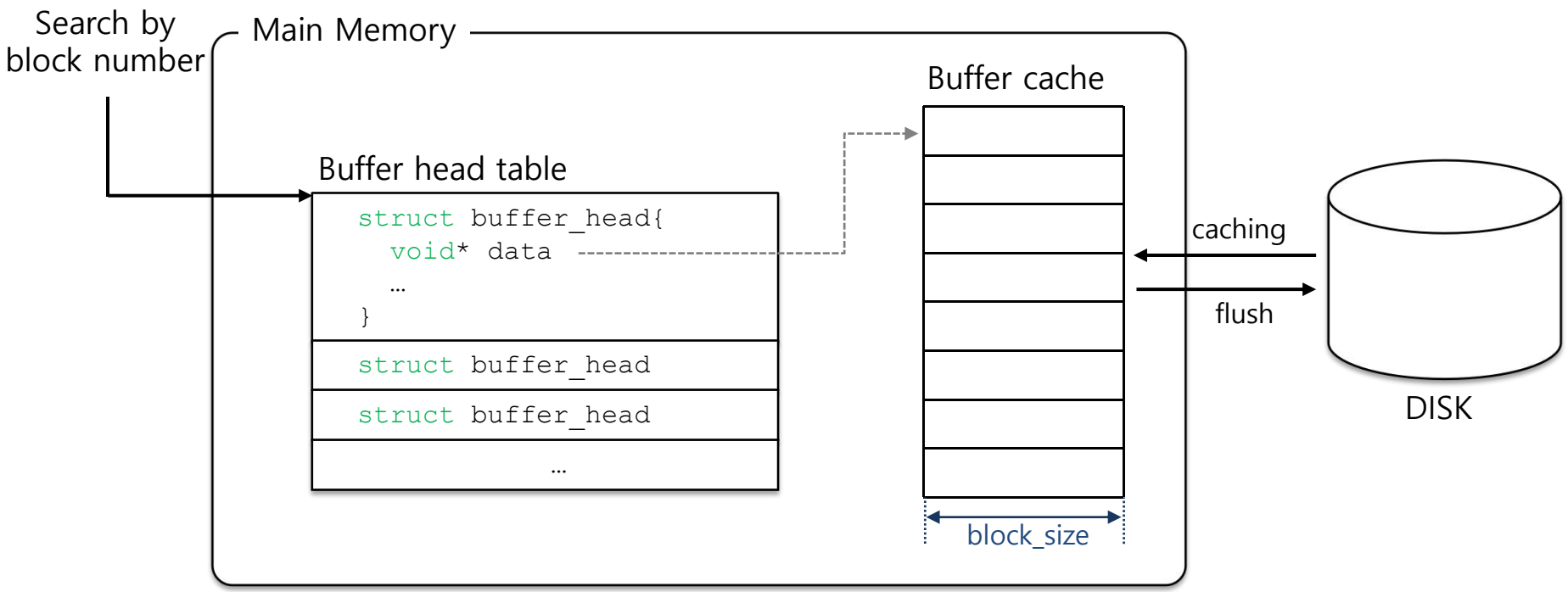
1. Define structures for buffer cache
2. Allocate and initialize buffer cache
3. When read data from file, read data from buffer cache
4. When write data to file, write data to buffer cache
5. If cache miss, read data from disk and save them in buffer cache
6. If buffer cache is full, evict buffer cache entry and reclaim free one
7. Write dirty buffer cache (sync)

To Do 1: Define structures for buffer cache

- Metadata for a buffer cache entry (ex: `struct buffer_head`)
  - The "dirty" flag
  - The flag indicating whether the entry being used or not
  - The "access" flag indicating whether the entry is accessed recently or not
  - The on-disk location
  - The virtual address of the associated buffer cache entry

- Maintain all 64 `struct buffer_head` by data structure
  - Array, List, or Hash table.

Buffer cache diagram



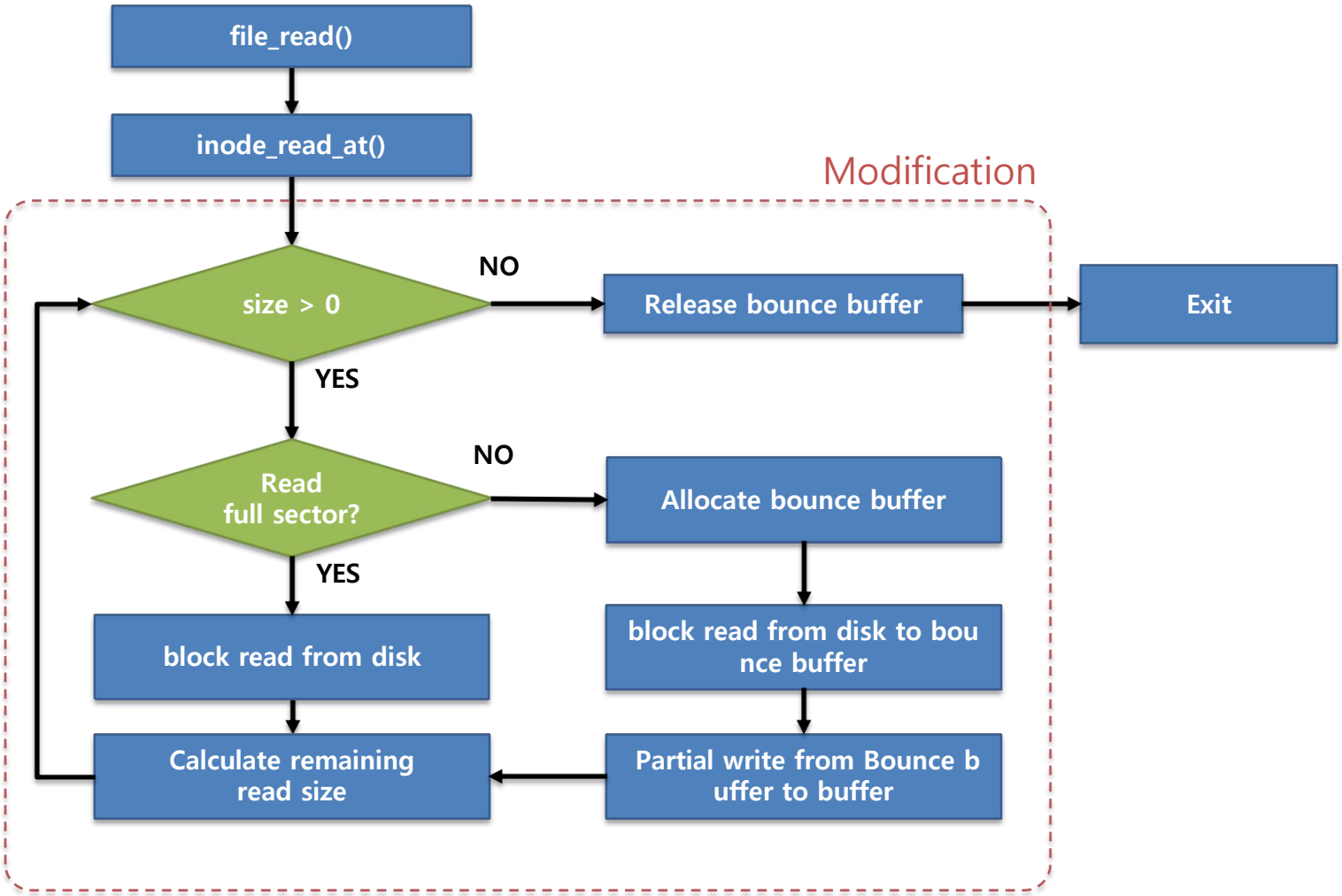
To Do 2: Allocate and initialize buffer cache

- Allocate memory for buffer cache for 64 block
  - Block size (=Sector size): 512 Byte
  - 64 * 512 Byte = 32 Kbyte
- Allocate memory for `struct buffer_head`'s
  - `sizeof (struct buffer_head)` * 64

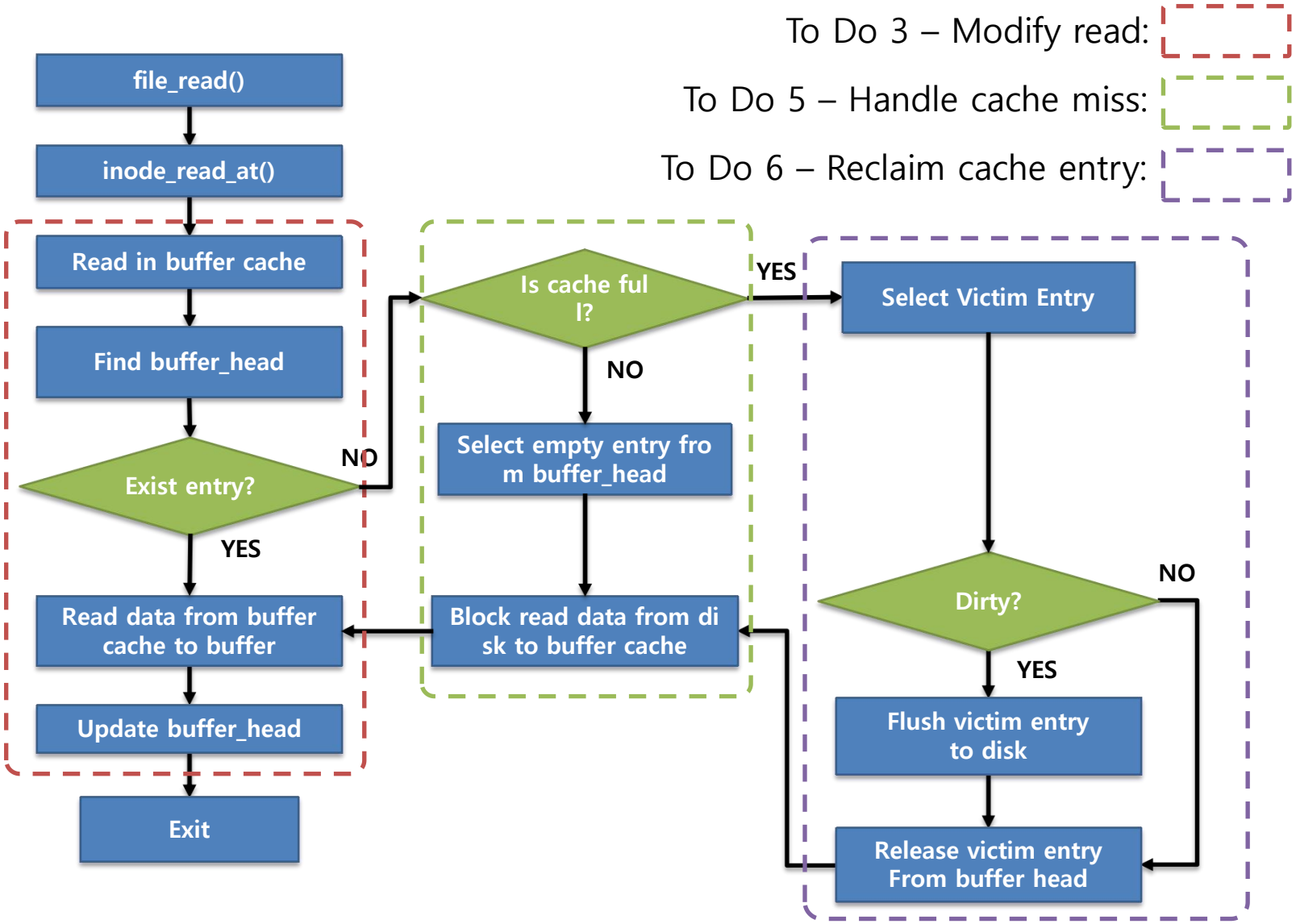
pintos/src/filesys/filesys.c

```
void filesys_init (bool format) {
  fs_device = block_get_role (BLOCK_FILESYS);
  ...
  if (format)
    do_format ();

  free_map_open ();

  /* Add code here */
}
```

READ in current pintos



Read with Buffer Cache



Modify disk read to buffer cache read

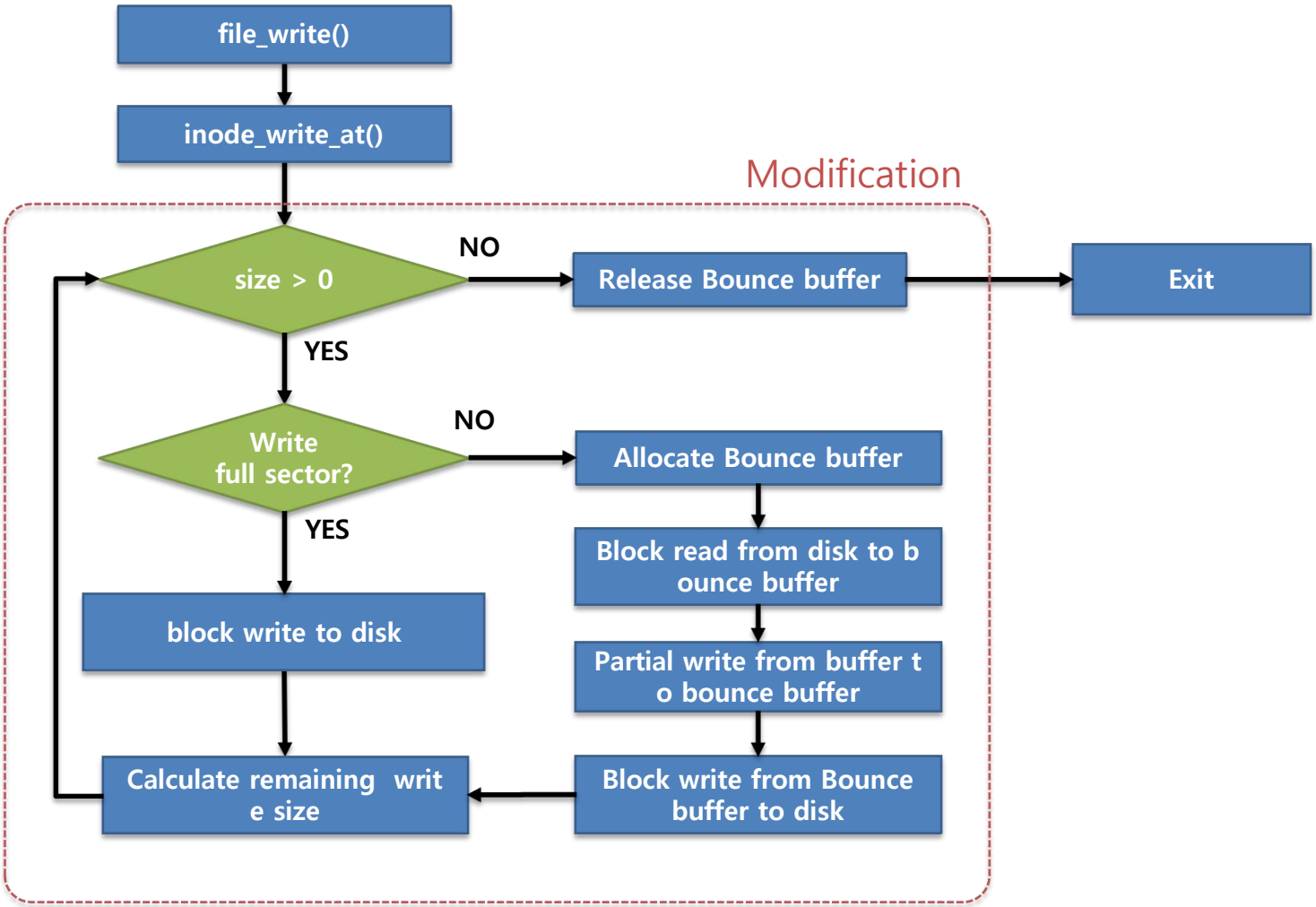
- When reading file, modify the read to read the data from the buffer c ache.

pintos/src/filesys/inode.c

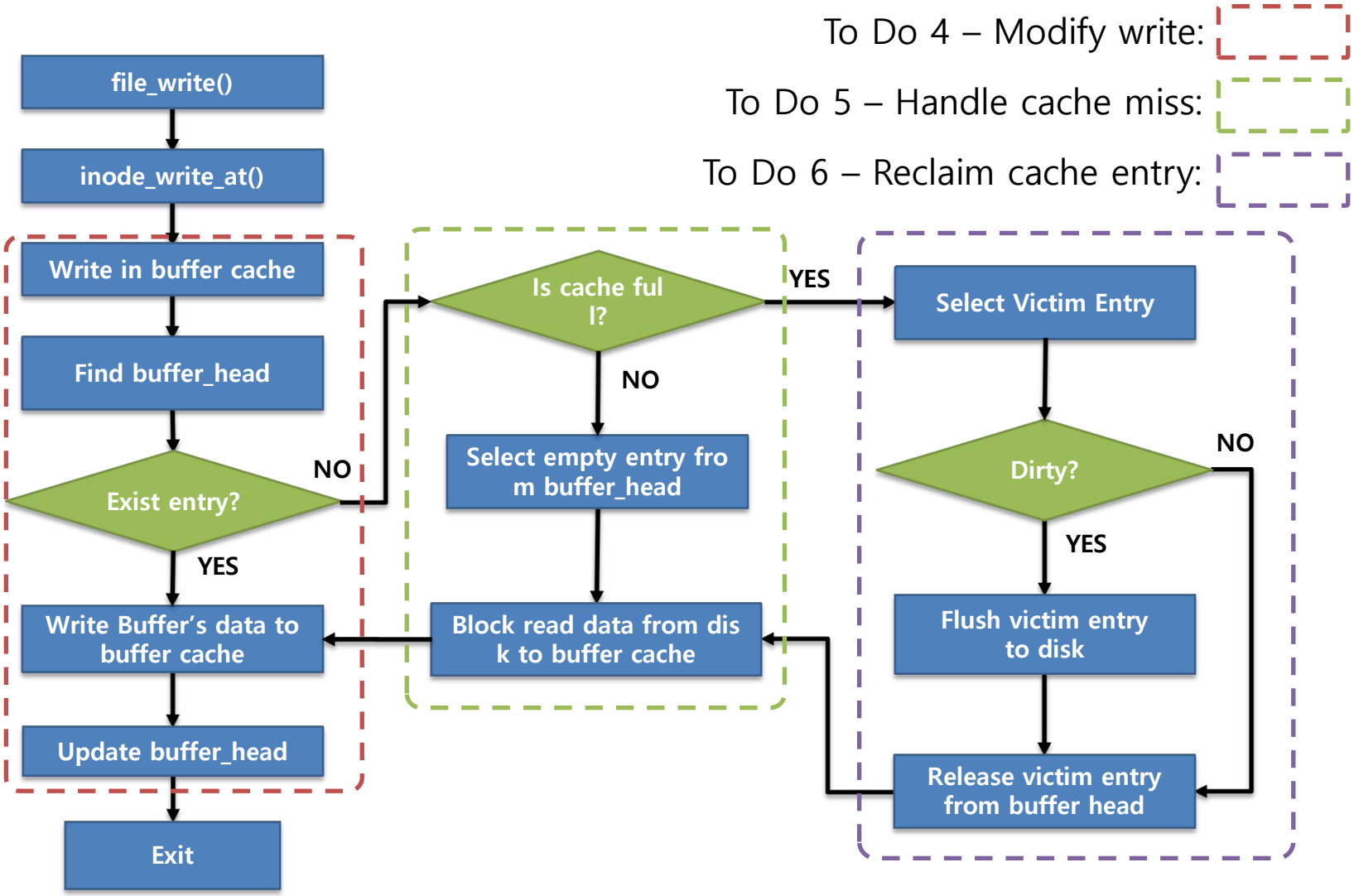
```
off_t inode_read_at (struct inode *inode, void *buffer_,
                     off_t size, off_t offset){
    …
    while( size > 0 ){
        block_sector_t sector_idx = byte_to_sector (inode, offse
t);
        int sector_ofs = offset % BLOCK_SECTOR_SIZE;
        …
        block_read (fs_device, sector_idx, buffer + bytes_read);
        …
    }
    …
}
```

modify

Write in current pintos



Write with Buffer cache



Modify the disk write to write to the buffer cache.

- When writing file, modify it to write data to buffer cache rather than to disk

pintos/src/filesys/inode.c

```
off_t inode_write_at (struct inode *inode, void *buffer_,
                      off_t size, off_t offset)
{
    …
    while (size > 0) {
        …
        if (sector_ofs == 0 && chunk_size == BLOCK_SECTOR_SIZE) {
            /* Write full sector directly to disk. */
            block_write (fs_device, sector_idx, buffer +
                         bytes_written);
            …
        }
        …
    }
}
```

modification

To Do 7: Write dirty buffer cache (sync)

- Write dirty buffer cache entries,
  - when the buffer cache entry is evicted.
  - when filesystem is shut down.

pintos/src/filesys/filesys.c

```
void
filesys_done (void)
{
  /* Add code here */
  free_map_close ();
}
```

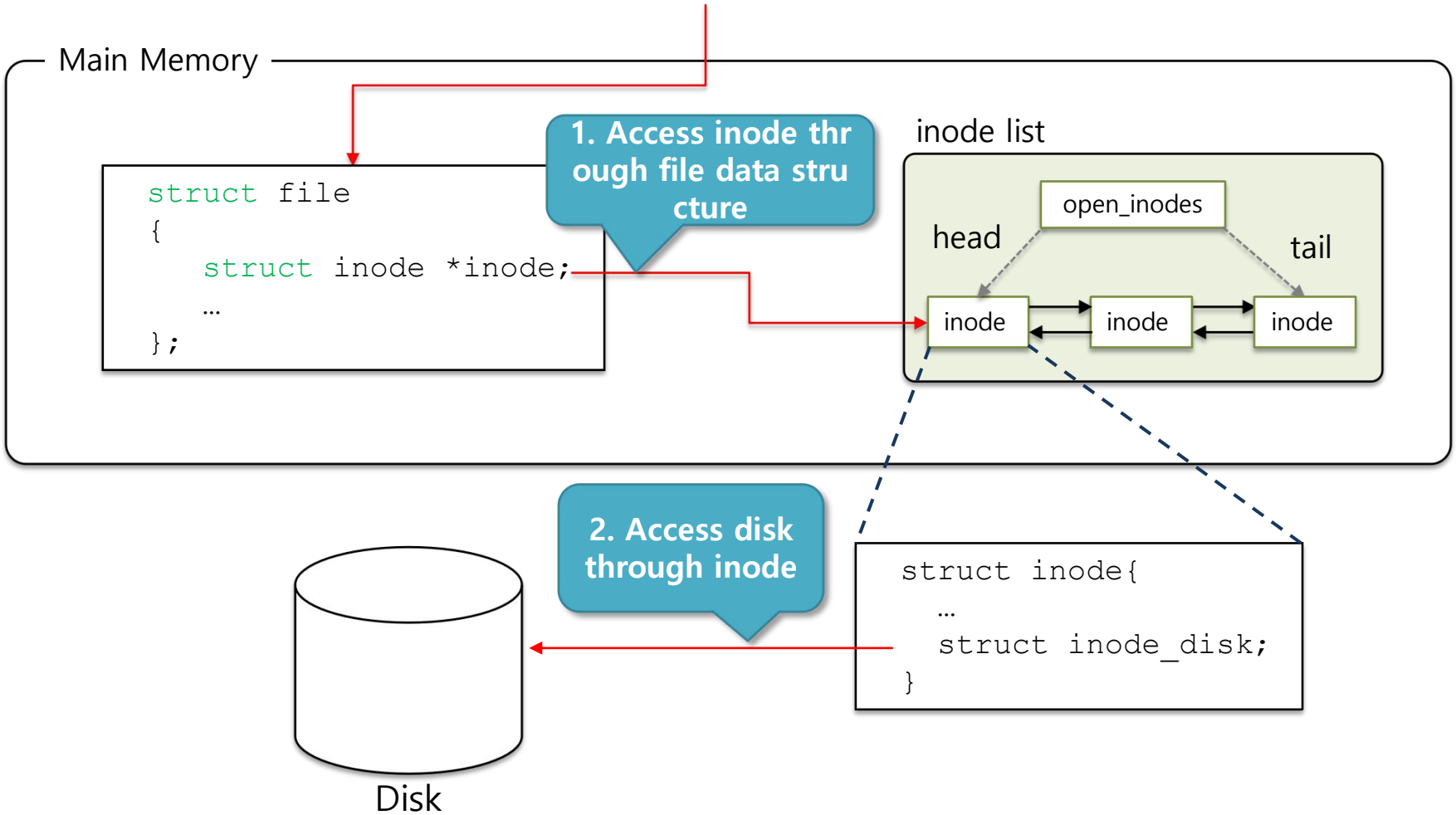
  - Periodically
    - Use timer interrupt.

Read/write in Pintos

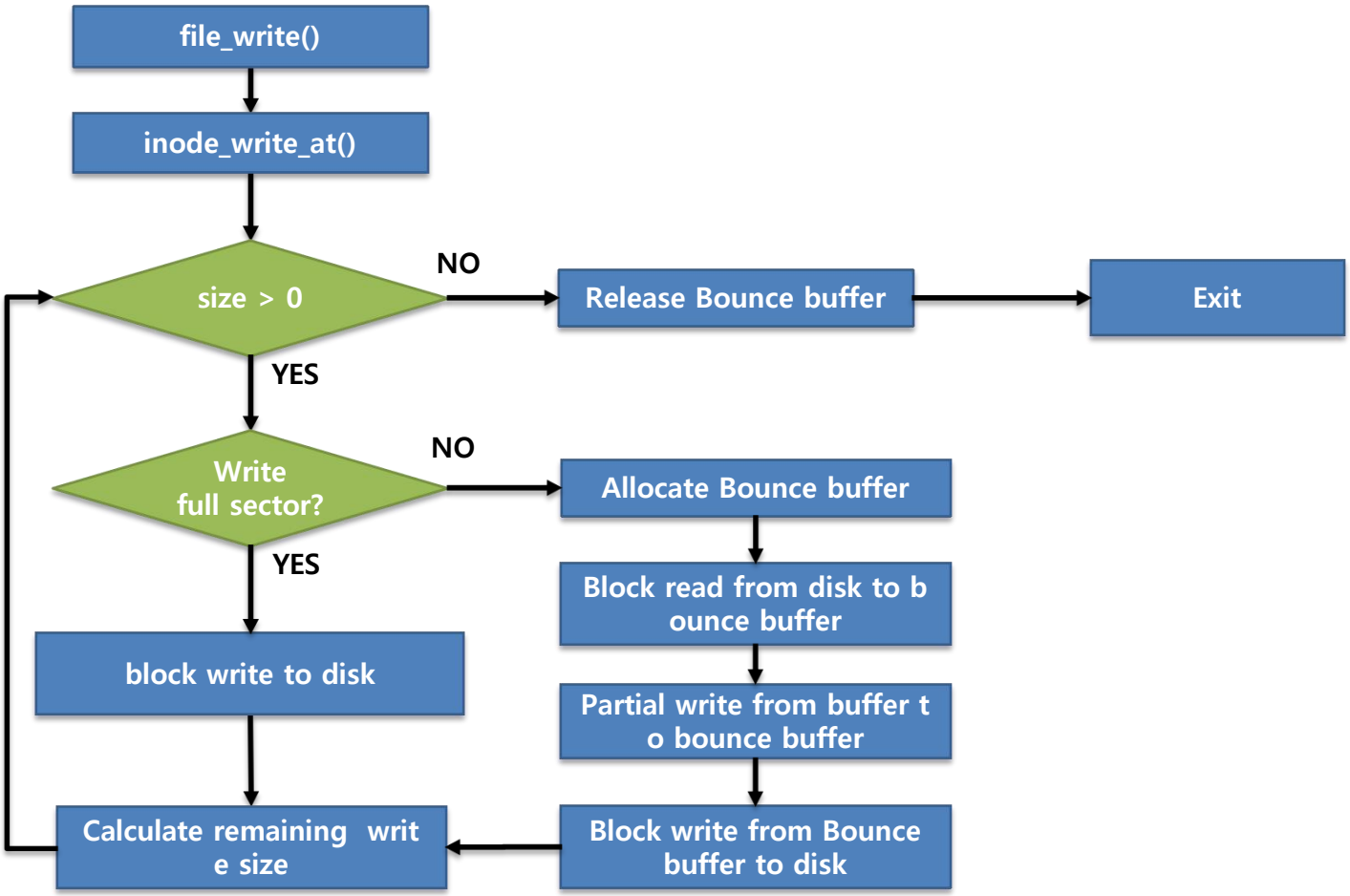
Background - Disk block access sequence in pintos

ex) reading a file

- `off_t file_read (struct file *file, void *buffer, off_t size)`



Background - Write in current pintos



Background - Write in current pintos (Cont.)

- `file_write`
  - Call `inode_write_at()` to write data in disk block
    - Change file offset by size recorded

pintos/src/filesys/file.c

```
off_t file_write (struct file *file, const void *buffer,
                  off_t size)
{
    off_t bytes_written = inode_write_at (file->inode, buffer,
size, file->pos);
    file->pos += bytes_written;
    return bytes_written;
}
```

```
struct file{
    struct inode *inode;
    off_t pos;
    …
}
```

```
struct inode{
    struct inode_disk data;
    …
}
```

Background - Write in current pintos (Cont.)

- inode_write_at
  - Record data that buffer points to disk

pintos/src/filesys/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;
```

Background - Write in current pintos (Cont.)

- Loop per disk block and write to the disk block: `block_write()`.
  - `byte_to_sector()`: Obtain the disk block number writing data.
  - `sector_ofs` : Offset within the disk block for writing the data.

pintos/src/filesys/inode.c – `inode_write_at()` (Cont.)

```
while (size > 0) {
    block_sector_t sector_idx = byte_to_sector (inode, offset);
    int sector_ofs = offset % BLOCK_SECTOR_SIZE;
    …
    if (sector_ofs == 0 && chunk_size == BLOCK_SECTOR_SIZE) {
        /* Write full sector directly to disk. */
        block_write (fs_device, sector_idx, buffer +
                     bytes_written);
    }
    …
```

Background - Write in current pintos (Cont.)

- In case of Partial Write, read the target block and save it to bounce buffer.
  - Through `memcpy()`, perform partial write on bounce buffer.
  - Record bounce buffer's data to disk : `block_write()`

pintos/src/filesys/inode.c – inode_write_at()

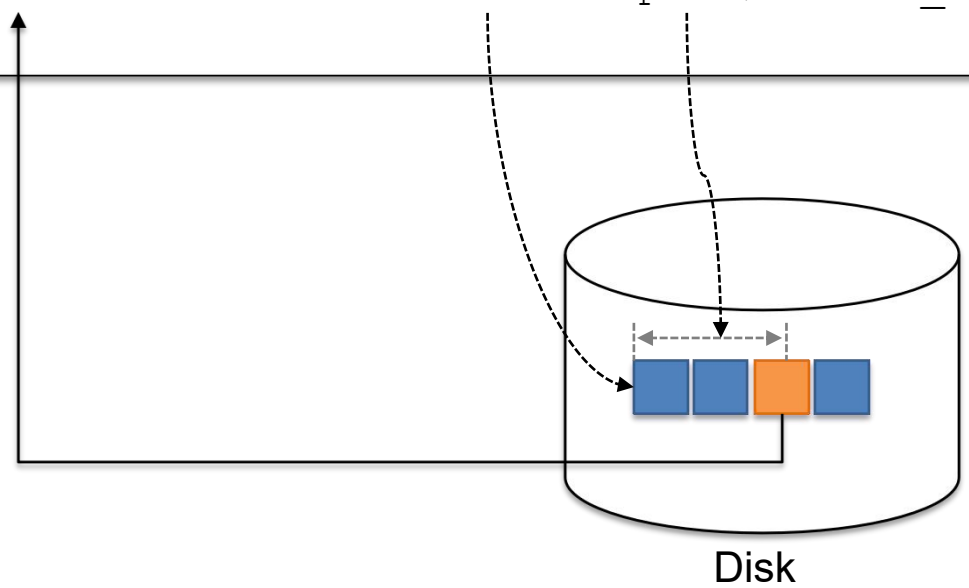
```
…
else {          /* We need a bounce buffer. */
    if (bounce == NULL)
        bounce = malloc (BLOCK_SECTOR_SIZE);
    if (sector_ofs > 0 || chunk_size < sector_left)
        block_read (fs_device, sector_idx, bounce);
    else
        memset (bounce, 0, BLOCK_SECTOR_SIZE);
    memcpy (bounce + sector_ofs, buffer + bytes_written,
            chunk_size);
    block_write (fs_device, sector_idx, bounce);
}
…
```

Background - Get disk block address from file offset

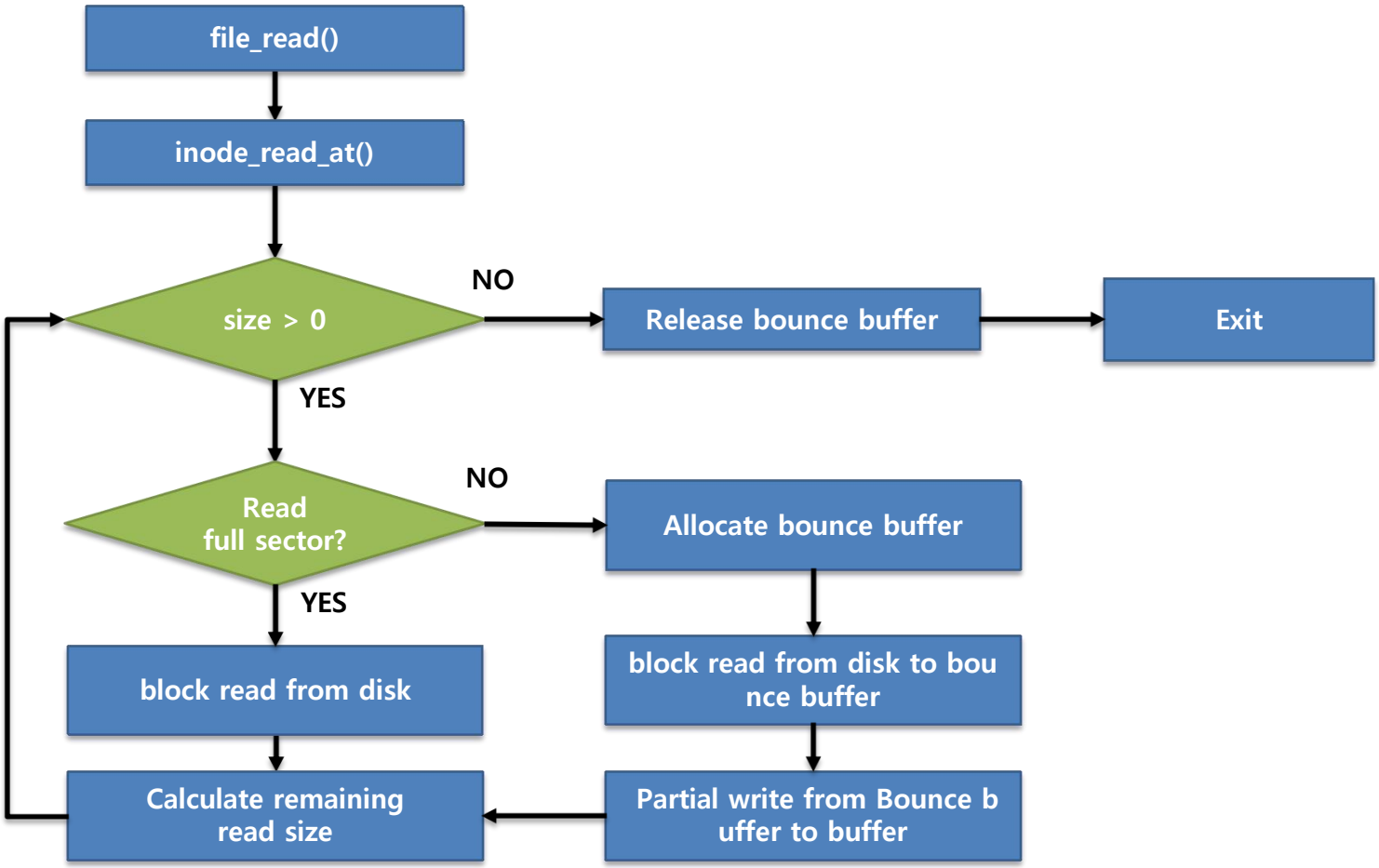
- Return the disk block number by adding offset value from file's starting block.

pintos/src/filesys/inode.c

```
static block_sector_t byte_to_sector (const struct inode *inode,
                                      off_t pos){
    …
    return inode->data.start + pos / BLOCK_SECTOR_SIZE;
}
```



Background - `read` in current pintos



Background - read in current pintos (Cont.)

- file_read
  - Call inode_read_at() to read data from disk to buffer
  - Change file offset by read size

pintos/src/filesys/file.c

```
off_t file_read (struct file *file, void *buffer, off_t size){
    off_t butes_read = inode_read_at (file->inode, buffer, size,
file->pos);
    file->pos += butes_read;
    return bytes_read;
}
```

```
struct file{
    struct inode *inode;
    off_t pos;
    …
}
```

```
struct inode{
    struct inode_disk data;
    …
}
```

Background - read in current pintos (Cont.)

- inode_read_at
  - Loop per disk block and Read data from disk: block_read()
  - byte_to_sector(): Obtain disk block number to read data
  - sector_ofs : Offset within disk block to read data

pintos/src/filesys/inode.c

```
off_t inode_read_at (struct inode *inode, void *buffer_,
                     off_t size, off_t offset){
    …
    while( size > 0 ){
        block_sector_t sector_idx = byte_to_sector (inode, offse
t);
        int sector_ofs = offset % BLOCK_SECTOR_SIZE;
        /* require replacement with cache read */
        block_read (fs_device, sector_idx, buffer + bytes_read);
    }
    …
}
```