گزارش پروژه چهارم پینتوس

استاد:

دکتر شهاب الدین نبوی

اعضای گروه:

بهار امیریان ورنوسفادرانی

مهتاب سرلک

کیانا شهراسبی

سهند صفی زاده

محمد رضا فریدونی

در این گزارش، به بررسی پروژه چهارم و راه حل ارائه شده برای آن می‌پردازیم. مشکلات موجود، راه حل‌های در نظر گرفته شده و نتایج آن در هر بخش ضمن توضیحات فایل‌های مرتبط توضیح داده شده‌اند.

فایل inode.c

در این فایل ابتدا include های مورد نظر قرار داده شده است.

```c
#include "filesys/inode.h"
#include <list.h>
#include <debug.h>
#include <round.h>
#include <string.h>
#include "filesys/cache.h"
#include "filesys/filesys.h"
#include "filesys/free-map.h"
#include "threads/malloc.h"
```

سپس برای مشخص کردن یک inode، نیاز به چند define داریم. نام هر کدام کاربرد آن را مشخص می‌کند و در ادامه از آنها استفاده خواهد شد.

یکی از مشکلات اساسی در پیاده سازی‌های قبلی، عدم پشتیبانی از فایل‌ها و پوشه بندی‌های تو در تو و فایل‌هایی با اندازه زیاد و غیر ثابت بود. برای حل این مشکلات به سراغ ساختار داده inode رفته ایم.

فرض می‌کنیم هر فایل از تعدادی block sector تشکیل شده است و به ازای هر فایل، باید شماره sector های تشکیل دهنده آن را در inode متناظر با آن نگه داریم. این روش باعث می‌شود محدویتی در افزایش حجم فایل وجود نداشته باشد و بتوانیم اجزای مختلف فایل را در بخش‌های فیزیکی مختلف و مجزایی از دیسک قرار دهیم و بابت وجود فضای خالی پشت سرهم در دیسک نگران نباشیم. البته برای عدم کاهش عمر دیسک و سرعت کلی سیستم، موضوع defragment و الگوریتم‌های آن باید موجود باشند.

با توجه به مستندات رسمی سیستم عامل پینتوس، در این پروژه باید پشتیبانی از فایل‌هایی با حجم حداکثر ۸ مگابایت پیاده سازی شود. پس از آنجایی که در حالت indirect block، تنها ۴ مگابایت قابل آدرس دهی است، باید از doubly-indirect block ها استفاده شود تا بتوانیم تا ۴ گیگابایت را آدرس دهی کنیم.

direct block ها در یک مرحله، indirect ها در دو مرحله و doubly-indirect ها در سه مرحله به داده اشاره می‌کنند.

```c
/* Identifies an inode. */
#define INODE_MAGIC 0x494e4f44

#define DIRECT_BLOCKS 4
#define INDIRECT_BLOCKS 9
#define DOUBLE_INDIRECT_BLOCKS 1

#define DIRECT_INDEX 0
#define INDIRECT_INDEX 4
#define DOUBLE_INDIRECT_INDEX 13

#define INDIRECT_BLOCK_PTRS 128
#define INODE_BLOCK_PTRS 14

/* 8 megabyte file size limit */
#define MAX_FILE_SIZE 8980480
```

در بخش بعد، struct برای disk و indirect-block ساخته شده است. on-disk inode باید سایزی دقیقا برابر با طول BLOCK-SECTOR-SIZE داشته باشد و شامل تعداد بایت اندازه فایل، مقدار magic، که نشان می‌دهد در حال کار با inode هستیم، اندیس‌هایی برای direct، indirect و double-indirect، bool برای directory بودن یا نبودن، یک block sector به عنوان parent، یک آرایه استفاده نشده و آرایه ای از اشاره گر به block ها است.

Indirect-block نیز شامل آرایه از pointer ها به block های indirect است.

```
struct inode_disk
  {
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    uint32_t direct_index;
    uint32_t indirect_index;
    uint32_t double_indirect_index;
    bool isdir;
    block_sector_t parent;
    uint32_t unused[107];                   /* Not used. */
    block_sector_t ptr[INODE_BLOCK_PTRS];     /* Pointers to blocks */
  };

struct indirect_block
  {
    block_sector_t ptr[INDIRECT_BLOCK_PTRS];
  };
```

بخش بعدی مربوط به prototype توابع است که در ادامه توضیح داده می‌شوند.

```
bool inode_alloc (struct inode_disk *disk_inode);
off_t inode_expand (struct inode *inode, off_t new_length);
size_t inode_expand_indirect_block (struct inode *inode,
                    size_t new_data_sectors);
size_t inode_expand_double_indirect_block (struct inode *inode,
                      size_t new_data_sectors);
size_t inode_expand_double_indirect_block_lvl_two (struct inode *inode,
                      size_t new_data_sectors,
                      struct indirect_block *outer_block);

void inode_dealloc (struct inode *inode);
void inode_dealloc_indirect_block (block_sector_t *ptr, size_t data_ptrs);
void inode_dealloc_double_indirect_block (block_sector_t *ptr,
                     size_t indirect_ptrs,
                     size_t data_ptrs);
```

- **تابع bytes_to_data_sectors:**

تعداد sector هایی که باید برای یک inode با اندازه SIZE، allocate شوند را با DIV_ROUND_UP برمی‌گرداند.

```
static inline size_t
bytes_to_data_sectors (off_t size)
{
  return DIV_ROUND_UP (size, BLOCK_SECTOR_SIZE);
}
```

- **تابع bytes_to_indirect _sectors:**

مشابه تابع بالا ولی برای sector های indirect است. ابتدا size را با BLOCK_SECTOR_SIZE*DIRECT_BLOCKS یعنی اندازه کل block های indirect مقایسه می‌کند. اگر size کوچکتر بود، ۰ را برمی‌گرداند و در غیر این صورت، این مقدار را از size کم کرده و مقدار را با DIV_ROUND_UP برمی‌گرداند. این مقایسه برای بررسی نیاز به استفاده از indirect-block ها است.

```
static size_t
bytes_to_indirect_sectors (off_t size)
{
  if (size <= BLOCK_SECTOR_SIZE*DIRECT_BLOCKS)
    {
      return 0;
    }
  size -= BLOCK_SECTOR_SIZE*DIRECT_BLOCKS;
  return DIV_ROUND_UP(size, BLOCK_SECTOR_SIZE*INDIRECT_BLOCK_PTRS);
}
```

- **تابع bytes_to_double_indirect _sectors:**

مشابه توابع بالا است ولی برای sector های double-indirect استفاده می‌شود. ابتدا size را با BLOCK_SECTOR_SIZE*(DIRECT_BLOCKS + INDIRECT_BLOCKS*INDIRECT_BLOCKS_PTRS) مقایسه می‌کند. اگر size کوچکتر بود، ۰ را برمی‌گرداند و در غیر این صورت، DOUBLE_INDIRECT_BLOCKS را برمی‌گرداند. این مقایسه برای بررسی نیاز به استفاده از double-indirect-block ها است.

```
static size_t bytes_to_double_indirect_sector (off_t size)
{
  if (size <= BLOCK_SECTOR_SIZE*(DIRECT_BLOCKS +
              INDIRECT_BLOCKS*INDIRECT_BLOCK_PTRS))
    {
      return 0;
    }
  return DOUBLE_INDIRECT_BLOCKS;
}
```

در بخش بعدی یک struct برای in-memory inode ساخته شده است. شامل لیستی از المان‌های داخل inode، شماره sector مربوط به مکان disk، تعداد opener ها، bool برای حذف شده یا نشده، متغیری با مقدار ۰ برای ok بودن نوشتن و غیر از آن برای deny کردن آن، تعداد بایت‌های اندازه فایل، طول خواندن، اندیس‌هایی برای direct، indirect و double-indirect، bool برای directory بودن یا نبودن، یک block sector به عنوان parent، یک آرایه استفاده نشده و آرایه ای از اشاره گر به block ها و یک قفل است.

```
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. */
    off_t length;                       /* File size in bytes. */
    off_t read_length;
    size_t direct_index;
    size_t indirect_index;
    size_t double_indirect_index;
    bool isdir;
    block_sector_t parent;
    struct lock lock;
    block_sector_t ptr[INODE_BLOCK_PTRS];  /* Pointers to blocks */
  };
```

- تابع **bytes_to_sector**:

block device sector شامل byte offset POS در INODE را برمی‌گرداند. اگر INODE شامل داده برای بایتی در offset pos نباشد، ۱- برمی‌گرداند. برای بررسی آن، POS را با LENGTH چک می‌کند. اگر POS بزرگتر یا مساوی آن بود، ۱- برمی‌گرداند. در غیر این صورت، سه حالت ایجاد می‌شود.

در حالت اول، POS از BLOCK_SECTOR_SIZE*DIRECT_BLOCKS کوچکتر است. در حالت دوم، از آن کوچتر نیست ولی از BLOCK_SECTOR_SIZE*(DIRECT_BLOCKS + INDIRECT_BLOCKS*INDIRECT_BLOCKS_PTRS) کوچکتر است. در حالت سوم نیز در دو شرط بالا صدق نمی‌کند. این شرایط به ترتیب یعنی direct، indirect و doubly-indirect هستند. برای هر کدام از حالات، مقدار مناسب محاسبه شده و برگردانده می‌شود.

```
static block_sector_t
byte_to_sector (const struct inode *inode, off_t length, off_t pos)
{
  ASSERT (inode != NULL);
  if (pos < length)
    {
      uint32_t idx;
      uint32_t indirect_block[INDIRECT_BLOCK_PTRS];
      if (pos < BLOCK_SECTOR_SIZE*DIRECT_BLOCKS)
    {
      return inode->ptr[pos / BLOCK_SECTOR_SIZE];
    }
      else if (pos < BLOCK_SECTOR_SIZE*(DIRECT_BLOCKS +
                  INDIRECT_BLOCKS*INDIRECT_BLOCK_PTRS))
    {
      pos -= BLOCK_SECTOR_SIZE*DIRECT_BLOCKS;
      idx = pos / (BLOCK_SECTOR_SIZE*INDIRECT_BLOCK_PTRS) + DIRECT_BLOCKS;
      block_read(fs_device, inode->ptr[idx], &indirect_block);
      pos %= BLOCK_SECTOR_SIZE*INDIRECT_BLOCK_PTRS;
      return indirect_block[pos / BLOCK_SECTOR_SIZE];
    }
```

```
      else
    {
      block_read(fs_device, inode->ptr[DOUBLE_INDIRECT_INDEX],
            &indirect_block);
      pos -= BLOCK_SECTOR_SIZE*(DIRECT_BLOCKS +
                    INDIRECT_BLOCKS*INDIRECT_BLOCK_PTRS);
      idx = pos / (BLOCK_SECTOR_SIZE*INDIRECT_BLOCK_PTRS);
      block_read(fs_device, indirect_block[idx], &indirect_block);
      pos %= BLOCK_SECTOR_SIZE*INDIRECT_BLOCK_PTRS;
      return indirect_block[pos / BLOCK_SECTOR_SIZE];
    }
    }
  else
    {
      return -1;
    }
}
```

در اینجا لیستی از node های open معرفی می‌شود، تا دو بار باز کردن یک inode، یک struct inode را برگرداند.

```
/* List of open inodes, so that opening a single inode twice
   returns the same `struct inode'. */
static struct list open_inodes;
```

- **تابع inode_init:**

برای مقداردهی اولیه به ماژول inode استفاده می‌شود.

```
/* Initializes the inode module. */
void
inode_init (void)
{
  list_init (&open_inodes);
}
```

- **تابع inode_creat:**

یک inode با LENGTH بایت از داده را مقداردهی اولیه می‌کند و inode جدید را در SECTOR فایل سیستم می‌نویسد. اگر عملیات به درستی انجام شود true و اگر allocation در memory یا disk با مشکل مواجه شود false برمی‌گرداند.

LENGTH باید بیشتر یا مساوی ۰ باشد، همچنین اندازه disk_inode باید با BLOCK_SECTOE_SEIZE برابر باشد. در غیر اینصورت، یعنی اگر ساختار inode دقیقا اندازه یک sector را نداشته باشد، باید آن را اصلاح کنیم.

پس از این یک calloc برای disk_inode انجام می‌شود و در صورتی که null نبود، مقداردهی‌ها انجام شده و عملیات نوشتن آن نیز انجام می‌شود. در صورتی که به درستی انجام شود، success، true می‌شود.

```
bool
inode_create (block_sector_t sector, off_t length, bool isdir)
{
  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)
    {
      disk_inode->length = length;
      if (disk_inode->length > MAX_FILE_SIZE)
    {
      disk_inode->length = MAX_FILE_SIZE;
```

```
    }
    disk_inode->magic = INODE_MAGIC;
    disk_inode->isdir = isdir;
    disk_inode->parent = ROOT_DIR_SECTOR;
    if (inode_alloc(disk_inode))
      {
        block_write (fs_device, sector, disk_inode);
        success = true;
      }
    free (disk_inode);
  }
 return success;
}
```

- **تابع inode_open:**

یک inode را از SECTOR می‌خواند و یک struct inode که شامل آن است را برمی‌گرداند. اگر memory allocation با مشکل مواجه شود، null برمی‌گرداند.

برای اینکار ابتدا چک می‌کند که inode مورد نظر از قبل باز است یا نه. اینکار با یک حلقه روی لیست inode ها باز انجام می‌شود. اگر باز بود و پیدا شد، inode_reopen را فراخوانی کرده و آن را باز می‌گرداند. در غیر این صورت، malloc و مقداردهی‌ها انجام شده، در نهایت بازگردانده می‌شود.

```
struct inode *
inode_open (block_sector_t sector)
{
  struct list_elem *e;
  struct inode *inode;

  /* Check whether this inode is already open. */
  for (e = list_begin (&open_inodes); e != list_end (&open_inodes);
       e = list_next (e))
    {
      inode = list_entry (e, struct inode, elem);
      if (inode->sector == sector)
        {
          inode_reopen (inode);
          return inode;
        }
    }

  /* Allocate memory. */
  inode = malloc (sizeof *inode);
  if (inode == NULL)
    return NULL;
```

```
  /* Initialize. */
  list_push_front (&open_inodes, &inode->elem);
  inode->sector = sector;
  inode->open_cnt = 1;
  inode->deny_write_cnt = 0;
  inode->removed = false;
  lock_init(&inode->lock);
  struct inode_disk data;
  block_read(fs_device, inode->sector, &data);
  inode->length = data.length;
  inode->read_length = data.length;
  inode->direct_index = data.direct_index;
  inode->indirect_index = data.indirect_index;
  inode->double_indirect_index = data.double_indirect_index;
  inode->isdir = data.isdir;
  inode->parent = data.parent;
  memcpy(&inode->ptr, &data.ptr, INODE_BLOCK_PTRS*sizeof(block_sector_t));
  return inode;
}
```

- **تابع inode_reopen:**

اگر inode باز باشد، فقط تعداد open ها یکی اضافه می‌شود و همان بازگردانده می‌شود.

```
struct inode *
inode_reopen (struct inode *inode)
{
  if (inode != NULL)
    inode->open_cnt++;
  return inode;
}
```

- **تابع inode_get_number:**

شماره INODE مورد نظر را که همان sector است، برمیگرداند.

```
block_sector_t
inode_get_inumber (const struct inode *inode)
{
  return inode->sector;
}
```

- **تابع inode_close:**

INODE را می‌بندد و آن را روی دیسک می‌نویسد. اگر آخرین رفرنس به INODE بود، حافظه آن را آزاد می‌کند. همچنین اگر یک inode حذف شده بود، block آن را آزاد می‌کند.

برای اینکار ابتدا اگر INODE، null بود، تابع متوقف می‌شود. سپس اگر آخرین opener بود، منابع باید آزاد شوند. ابتدا از لیست open ها حذف می‌شود و قفل آزاد می‌شود. اگر از قبل حذف شده بود، block ها را deallocate می‌کند. در غیر این صورت، کپی آن را وارد دیسک می‌کند. در نهایت INODE آزاد می‌شود.

```
void
inode_close (struct inode *inode)
{
  /* Ignore null pointer. */
  if (inode == NULL)
    return;

  /* Release resources if this was the last opener. */
  if (--inode->open_cnt == 0)
    {
      /* Remove from inode list and release lock. */
      list_remove (&inode->elem);

      /* Deallocate blocks if removed. */
      if (inode->removed)
        {
          free_map_release (inode->sector, 1);
      inode_dealloc(inode);
        }
```

```
    else
    {
      struct inode_disk disk_inode = {
        .length = inode->length,
        .magic = INODE_MAGIC,
        .direct_index = inode->direct_index,
        .indirect_index = inode->indirect_index,
        .double_indirect_index = inode->double_indirect_index,
        .isdir = inode->isdir,
        .parent = inode->parent,
      };
      memcpy(&disk_inode.ptr, &inode->ptr,
        INODE_BLOCK_PTRS*sizeof(block_sector_t));
      block_write(fs_device, inode->sector, &disk_inode);
    }
    free (inode);
  }
}
```

- **تابع inode_remove:**

نشان می‌دهد که آخرین caller از INODE آن را بسته است. برای اینکار پارامتر removed آن را true می‌کند.

```
/* Marks INODE to be deleted when it is closed by the last caller who
   has it open. */
void
inode_remove (struct inode *inode)
{
  ASSERT (inode != NULL);
  inode->removed = true;
}
```

- **تابع inode_read_at:**

به اندازه SIZE بایت از INODE خوانده و آن را داخل BUFFER با position اولیه OFFSET قرار می‌دهد و تعداد بایت‌هایی را که واقعا خوانده شده اند را برمی‌گرداند. این مقدار می‌تواند از SIZE کوچکتر باشد، چون ممکن است به انتهای file برسیم و یا error رخ دهد.

اگر OFFSET از طول INODE بیشتر باشد، ۰ را برمی‌گرداند. در غیر این صورت تا وقتی که SIZE از ۰ بزرگتر است، داده را chunk شده می‌خواند و SIZE باقی مانده را اصلاح می‌کند. اگر مقدار chunk زمانی کمتر از مقدار باقی مانده شود نیز به همان اندازه باقی مانده خوانده می‌شود.

```
off_t
inode_read_at (struct inode *inode, void *buffer_, off_t size, off_t offset)
{
  uint8_t *buffer = buffer_;
  off_t bytes_read = 0;

  off_t length = inode->read_length;

  if (offset >= length)
    {
      return bytes_read;
    }

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

      /* Bytes left in inode, bytes left in sector, lesser of the two. */
      off_t inode_left = length - offset;
      int sector_left = BLOCK_SECTOR_SIZE - sector_ofs;
      int min_left = inode_left < sector_left ? inode_left : sector_left;

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

      struct cache_entry *c = filesys_cache_block_get(sector_idx, false);
      memcpy (buffer + bytes_read, (uint8_t *) &c->block + sector_ofs,
          chunk_size);
      c->accessed = true;
      c->open_cnt--;

      /* Advance. */
      size -= chunk_size;
      offset += chunk_size;
      bytes_read += chunk_size;
    }

  return bytes_read;
}
```

- **تابع inode_write_at:**

مقدار SIZE بایت را از BUFFER با نقطه شروع OFFSET می‌خواند و در INODE می‌نویسد. تعداد بایت‌هایی که واقعا نوشته شده است را برمی‌گرداند که می‌تواند کمتر از SIZE باشد، چون ممکن است به انتهای file برسیم یا خطایی رخ دهد. البته معمولا نوشتن در انتهای فایل یاعث افزایش اندازه INODE می‌شود ولی در اینجا این افزایش پیاده سازی نشده است. انجام اینکار تقریبا مشابه تابع بالا، ولی به صورت معکوس است.

```
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;

  if (inode->deny_write_cnt)
    return 0;

  if (offset + size > inode_length(inode))
    {
      if (!inode->isdir)
      {
        inode_lock(inode);
      }
      inode->length = inode_expand(inode, offset + size);
      if (!inode->isdir)
      {
        inode_unlock(inode);
      }
    }
```

```
while (size > 0)
  {
    /* Sector to write, starting byte offset within sector. */
    block_sector_t sector_idx = byte_to_sector (inode,inode_length(inode), offset);
    int sector_ofs = offset % BLOCK_SECTOR_SIZE;
    /* Bytes left in inode, bytes left in sector, lesser of the two. */
    off_t inode_left = inode_length(inode) - offset;
    int sector_left = BLOCK_SECTOR_SIZE - sector_ofs;
    int min_left = inode_left < sector_left ? inode_left : sector_left;
    /* Number of bytes to actually write into this sector. */
    int chunk_size = size < min_left ? size : min_left;
    if (chunk_size <= 0) break;
    struct cache_entry *c = filesys_cache_block_get(sector_idx, true);
    memcpy ((uint8_t *) &c->block + sector_ofs, buffer + bytes_written,
        chunk_size);
    c->accessed = true;
    c->dirty = true;
    c->open_cnt--;
    /* Advance. */
    size -= chunk_size;
    offset += chunk_size;
    bytes_written += chunk_size;
  }
inode->read_length = inode_length(inode);
return bytes_written;
```

- تابع **inode_deny_write**:

نوشتن در INODE را disable می‌کند. برای هر inode opener حداکثر یکبار فراخوانی می‌شود. برای اینکار هم پارامتر deny_wrie_cnt را در INODE یکی اضافه می‌کند. تعداد این پارامتر باید کمتر از تعداد INODE های باز باشد.

```
/* Disables writes to INODE.
   May be called at most once per inode opener. */
void
inode_deny_write (struct inode *inode)
{
  inode->deny_write_cnt++;
  ASSERT (inode->deny_write_cnt <= inode->open_cnt);
}
```

- تابع **inode_allow_write**:

نوشتن در INODE را مجددا enable می‌کند. باید برای هر inode opener که inode_deny_write را فراخوانی کرده است، قبل از بسته شدن inode، فراخوانی شود.

```
/* Re-enables writes to INODE.
   Must be called once by each inode opener who has called
   inode_deny_write() on the inode, before closing the inode. */
void
inode_allow_write (struct inode *inode)
{
  ASSERT (inode->deny_write_cnt > 0);
  ASSERT (inode->deny_write_cnt <= inode->open_cnt);
  inode->deny_write_cnt--;
}
```

- **تابع inode_length:**

طول داده INODE را به صورت تعداد بایت برمی‌گرداند.

```
/* Returns the length, in bytes, of INODE's data. */
off_t
inode_length (struct inode *inode)
{
  return inode->length;
}
```

- **Inode_dealloc**

هنگام ایجاد فایل و inode allocation بر اساس سایز فایل موردنیاز تعدادی inode ایجاد می‌شود، هنگامی که لازم باشد تغییراتی در فایل ایجاد و ثبت شود که منجر به کاهش حجم فایل می‌شود باید inode های ضروری و غیرلازم را حذف کرد.

تابع inode_dealloc جهت آزاد کردن inode های غیرلازم تعریف شده است.

همانگونه که در ابتدا براساس سایز فایل موردنظر بلوک های موردنیاز ابتدا به صورت direct block، سپس به صورت indirect block و در نهایت به صورت doubly indirect block ذخیره می‌شوند، برای آزاد کردن بلوک ها نیز باید این حالت سلسه مراتبی را رعایت نمود.

در نتیجه در این تابع بعد از آن که تعداد data_sectors، indirect_sectors و double_indirect_sector مشخص شد ابتدا باید بلوک های direct و سپس بلوک های indirect و درنهایت در صورت وجود بلوک های double indirect آنها را حذف نمود.

همانگونه که مشاهده می‌شود عملیات deallocation این بلاک ها با استفاده از توابع کمکی inode_dealloc_indirect_block و inode_dealloc_double_indirect_block انجام می‌پذیرد.

```
453 ⇆ void inode_dealloc (struct inode *inode)
454     {
455       size_t data_sectors = bytes_to_data_sectors(inode->length);
456       size_t indirect_sectors = bytes_to_indirect_sectors(inode->length);
457       size_t double_indirect_sector = bytes_to_double_indirect_sector(
458                                       inode->length);
459       unsigned int idx = 0;
460       while (data_sectors && idx < INDIRECT_INDEX)
461         {
462           free_map_release (inode->ptr[idx], 1);
463           data_sectors--;
464           idx++;
465         }
466       while (indirect_sectors && idx < DOUBLE_INDIRECT_INDEX)
467         {
468           size_t data_ptrs = data_sectors < INDIRECT_BLOCK_PTRS ? \
469         data_sectors : INDIRECT_BLOCK_PTRS;
470           inode_dealloc_indirect_block(&inode->ptr[idx], data_ptrs);
471           data_sectors -= data_ptrs;
472           indirect_sectors--;
473           idx++;
474         }
```

```
  if (double_indirect_sector)
    {
      inode_dealloc_double_indirect_block(&inode->ptr[idx],
                    indirect_sectors,
                    data_sectors);
    }
}
```

- **inode_dealloc_double_indirect_block**

این تابع برای آزاد کردن بلوک های double_indirect از inode به کار می‌رود. این تابع برای انجام این کار در ابتدا یک بلوک از نوع Indirect_block تعریف می‌کند و سپس برای هر بلاک از تابع کمکی inode_dealloc_indirect_block استفاده کرده و عملیات deallocation را انجام می‌دهد.

```
void inode_dealloc_double_indirect_block (block_sector_t *ptr,
                    size_t indirect_ptrs,
                    size_t data_ptrs)
{
  unsigned int i;
  struct indirect_block block;
  block_read(fs_device, *ptr, &block);
  for (i = 0; i < indirect_ptrs; i++)
    {
      size_t data_per_block = data_ptrs < INDIRECT_BLOCK_PTRS ? data_ptrs : \
    INDIRECT_BLOCK_PTRS;
      inode_dealloc_indirect_block(&block.ptr[i], data_per_block);
      data_ptrs -= data_per_block;
    }
  free_map_release(*ptr, 1);
}
```

- **inode_dealloc_indirect_block**

این تابع برای آزاد کردن و حذف بلوک های indirect به کار می‌رود. در این تابع نیز ابتدا یک بلوک از نوع indirect_block ساخته می‌شود و سپس به کمک تابع free_map_release، بلوک های غیرضروی inode حذف می‌شوند.

```
void inode_dealloc_indirect_block (block_sector_t *ptr,
                    size_t data_ptrs)
{
  unsigned int i;
  struct indirect_block block;
  block_read(fs_device, *ptr, &block);
  for (i = 0; i < data_ptrs; i++)
    {
      free_map_release(block.ptr[i], 1);
    }
  free_map_release(*ptr, 1);
}
```

- **inode_expand**

این تابع برای توسعه‌ی بلاک سکتور ها به کار می‌رود. به این صورت که ورودی این تابع شامل یک inode و اندازه‌ی آن با توجه به فایل موردنظر است.

از آنجایی که برای ایجاد بلوک های inode به صورت سلسله مراتبی عمل می‌شود در اینجا نیز باید ابتدا بلوک های direct و سپس بلوک های indirect و بعد از آن بلوک های doubly indirect گسترش یابند. در این تابع نیز به همین جهت ابتدا مشخصه ی direct_index مربوط به inode با INDIRECT_INDEX مقایسه می‌شود و در صورت برقرار بودن شرایط بلوک های direct به inode تخصیص داده می‌شوند و سپس دوباره در حلقه‌ی بعدی مشخصه‌ی direct_index با DOUBLE_INDIRECT_INDEX مقایسه می‌شود و در صورت برقرار بودن شرایط، بلوک های indirect به کمک تابع inode_expand_indirect_block به inode تخصیص داده می‌شود و درنهایت در صورت برابر بودن مقدارهای direct_indexو DOUBLE_INDIRECT_INDEX با استفاده از تابع کمکی inode_expand_double_indirect_block بلوک های double indirect به inode تخصیص داده می‌شود.

```
off_t inode_expand (struct inode *inode, off_t new_length)
{
  static char zeros[BLOCK_SECTOR_SIZE];
  size_t new_data_sectors = bytes_to_data_sectors(new_length) - \
    bytes_to_data_sectors(inode->length);

  if (new_data_sectors == 0)
    {
      return new_length;
    }
  while (inode->direct_index < INDIRECT_INDEX)
    {
      free_map_allocate (1, &inode->ptr[inode->direct_index]);
      block_write(fs_device, inode->ptr[inode->direct_index], zeros);
      inode->direct_index++;
      new_data_sectors--;
      if (new_data_sectors == 0)
    {
      return new_length;
    }
    }
```

```
  while (inode->direct_index < DOUBLE_INDIRECT_INDEX)
    {
      new_data_sectors = inode_expand_indirect_block(inode, new_data_sectors);
      if (new_data_sectors == 0)
    {
      return new_length;
    }
    }
  if (inode->direct_index == DOUBLE_INDIRECT_INDEX)
    {
      new_data_sectors = inode_expand_double_indirect_block(inode,
                                new_data_sectors);
    }
  return new_length - new_data_sectors*BLOCK_SECTOR_SIZE;
}
```

- **inode_expand_double_indirect_block**

این تابع برای گسترش بلوک های double indirect مربوط به inode تعریف شده است. در این تابع پس از آن که یک بلوک از نوع indirect block تعریف شد در ابتدا بررسی می‌شود که اگر مشخصه های double_indirect_index و indirect_index برابر صفر بودند، تنها برای توسعه‌ی بلوک ها از تابع free_map_allocate استفاده شود. در ادامه در حلقه ی while تا زمانی که مشخصه‌ی indirect_index مربوط به inode کمتر از INDIRECT_BLOCK_PTRS باشد، سکتور های جدیدی ایجاد می‌شود و نهایت این سکتورهای جدید به عنوان خروجی این تابع برگردانده می‌شوند.

```
size_t inode_expand_double_indirect_block (struct inode *inode,
                      size_t new_data_sectors)
{
  struct indirect_block block;
  if (inode->double_indirect_index == 0 && inode->indirect_index == 0)
    {
      free_map_allocate(1, &inode->ptr[inode->direct_index]);
    }
  else
    {
      block_read(fs_device, inode->ptr[inode->direct_index], &block);
    }
  while (inode->indirect_index < INDIRECT_BLOCK_PTRS)
    {
      new_data_sectors = inode_expand_double_indirect_block_lvl_two(inode,
                        new_data_sectors, &block);
      if (new_data_sectors == 0)
    {
      break;
    }
    }
  block_write(fs_device, inode->ptr[inode->direct_index], &block);
  return new_data_sectors;
```

- **inode_expand_double_indirect_block_lvl_two**

این تابع مشابه تابع قبلی است و تنها تفاوتش این است که این عملیات توسعه‌ی بلوک ها گویی در دو مرحله انجام می‌شود. (یک مرحله بررسی برابری double_indirect_index با INDIRECT_BLOCK_PTRS اضافه شده است.)در این تابع یک inner block تعریف می‌شود و درنهایت باز هم مانند تابع قبلی سکتور های جدید ایجاد شده بازگردانده می‌شوند.

```
size_t inode_expand_double_indirect_block_lvl_two (struct inode *inode,
                      size_t new_data_sectors,
                      struct indirect_block* outer_block)
{
  static char zeros[BLOCK_SECTOR_SIZE];
  struct indirect_block inner_block;
  if (inode->double_indirect_index == 0)
    {
      free_map_allocate(1, &outer_block->ptr[inode->indirect_index]);
    }
  else
    {
      block_read(fs_device, outer_block->ptr[inode->indirect_index],
         &inner_block);
    }
  while (inode->double_indirect_index < INDIRECT_BLOCK_PTRS)
    {
      free_map_allocate(1, &inner_block.ptr[inode->double_indirect_index]);
      block_write(fs_device, inner_block.ptr[inode->double_indirect_index],
          zeros);
      inode->double_indirect_index++;
      new_data_sectors--;
```

```
      if (new_data_sectors == 0)
    {
      break;
    }
    }
  block_write(fs_device, outer_block->ptr[inode->indirect_index], &inner_block);
  if (inode->double_indirect_index == INDIRECT_BLOCK_PTRS)
    {
      inode->double_indirect_index = 0;
      inode->indirect_index++;
    }
  return new_data_sectors;
}
```

- **inode_expand_indirect_block**

این تابع کمکی نیز برای توسعه‌ی بلوک های indirect مربوط به Inode تعریف شده است. انجام این عملیات مشابه با موارد ذکر شده در بالاست.

```
size_t inode_expand_indirect_block (struct inode *inode,
                  size_t new_data_sectors)
{
  static char zeros[BLOCK_SECTOR_SIZE];
  struct indirect_block block;
  if (inode->indirect_index == 0)
    {
      free_map_allocate(1, &inode->ptr[inode->direct_index]);
    }
  else
    {
      block_read(fs_device, inode->ptr[inode->direct_index], &block);
    }
  while (inode->indirect_index < INDIRECT_BLOCK_PTRS)
    {
      free_map_allocate(1, &block.ptr[inode->indirect_index]);
      block_write(fs_device, block.ptr[inode->indirect_index], zeros);
      inode->indirect_index++;
      new_data_sectors--;
      if (new_data_sectors == 0)
    {
      break;
    }
```

```
    }
  block_write(fs_device, inode->ptr[inode->direct_index], &block);
  if (inode->indirect_index == INDIRECT_BLOCK_PTRS)
    {
      inode->indirect_index = 0;
      inode->direct_index++;
    }
  return new_data_sectors;
}
```

پس به صورت کلی هر یک از توابع expand بالا، inode مربوطه را توسعه می‌دهند و به این صورت عمل می‌کنند که در ابتدا بررسی می‌کنند که آینود پر است یا خیر و در صورتی که inode پر باشد به تعداد لازم و موردنیاز sector از نو تخصیص داده می‌شود و new_sectors داده شده به آنها را در این مکان expand شده می‌ریزند.

- **inode_alloc**

در این تابع ابتدا یک struct از نوع inode با مشخصه های مربوط به آن ساخته می‌شود، سپس یک inode-disk جدید ساخته و مقداردهی می‌شود.

```
bool inode_alloc (struct inode_disk *disk_inode)
{
  struct inode inode = {
    .length = 0,
    .direct_index = 0,
    .indirect_index = 0,
    .double_indirect_index = 0,
  };
  inode_expand(&inode, disk_inode->length);
  disk_inode->direct_index = inode.direct_index;
  disk_inode->indirect_index = inode.indirect_index;
  disk_inode->double_indirect_index = inode.double_indirect_index;
  memcpy(&disk_inode->ptr, &inode.ptr,
     INODE_BLOCK_PTRS*sizeof(block_sector_t));
  return true;
}
```

- **inode_is_dir**

این تابع بررسی می‌کند که inode یک دایرکتوری است یا خیر و در صورتی که دایرکتوری باشد true برمی‌گرداند.

```
bool inode_is_dir (const struct inode *inode)
{
  return inode->isdir;
}
```

- **Inode_get_open_cnt**

این تابع تعداد opener های مربوط به inode موردنظر را برمی‌گرداند.

```
int inode_get_open_cnt (const struct inode *inode)
{
  return inode->open_cnt;
}
```

- **inode_get_parent**

این تابع، والد (parent) inode موردنظر را بر می‌گرداند.

```
block_sector_t inode_get_parent (const struct inode *inode)
{
  return inode->parent;
}
```

- **inode_add_parent**

این تابع به inode مورنظر یک والد (parent) اضافه می‌کند. درواقع این تابع در ابتدا بررسی می‌کند که inode موردنظر وجود دارد یا خیر و سپس در صورتی که inode موجود باشد برای آن یک والد (parent_sector) تعیین می‌کند.

```
bool inode_add_parent (block_sector_t parent_sector,
                       block_sector_t child_sector)
{
  struct inode* inode = inode_open(child_sector);
  if (!inode)
    {
      return false;
    }
  inode->parent = parent_sector;
  inode_close(inode);
  return true;
}
```

- **inode_lock**

این تابع برای گرفتن قفل توسط inode موردنظر تعریف شده است.

```
void inode_lock (const struct inode *inode)
{
  lock_acquire(&((struct inode *)inode)->lock);
}
```

- **inode_unlock**

این تابع برای آزاد کردن قفل های گرفته شده توسط inode موردنظر طراحی شده است.

```
void inode_unlock (const struct inode *inode)
{
  lock_release(&((struct inode *) inode)->lock);
}
```

فایل filesys/cache.h

در این فایل، تعریف چهار تابع اصلی مربوط به ساختار cache شامل init، close، read و write به همراه معرفی کتابخانه‌ی block.h جهت توصیف ورودی توابع cache و sectorها انجام شده است.

```
#ifndef FILESYS_CACHE_H
#define FILESYS_CACHE_H

#include "devices/block.h"

void buffer_cache_init (void);
void buffer_cache_close (void);
void buffer_cache_read (block_sector_t sector, void *target);
void buffer_cache_write (block_sector_t sector, const void *source);

#endif
```

فایل filesys/cache.c

در این فایل، ابتدا کتابخانه‌های مورد نیاز معرفی شده‌اند و یکی از موارد مهم آن‌ها، sync.h جهت استفاده از ساختار **سمافور و قفل** در هماهنگ‌سازی عملیات روی آرایه‌ی مشترک cache توسط lock می‌باشد. در قدم بعد، ساختار هر یک از 64 عنصر آرایه‌ی معرف cache، ارائه شده است. هر عنصر متناظر با یک sector مشخص از دیسک بوده و محتویات آن بخش را در آرایه‌ای به اندازه‌ی یک block_size از دیسک در حافظه‌ای سریع‌تر ارائه می‌کند. سازگاری یا عدم سازگاری محتویات ذخیره شده در هر عنصر cache با sector متناظر در دیسک به دنبال عمل **نوشتن**، توسط مقدار منطقی dirty مشخص می‌شود. همچنین کنترل مورد استفاده بودن عنصر cache جهت درج عنصری جدید با مقدار منطقی occupied انجام می‌شود. در نهایت، برای هر عنصر مقدار منطقی access برای انتخاب انقضای آن با توجه به الگوریتم *eviction* ذخیره‌سازی می‌شود.

```c
#include <debug.h>
#include <string.h>
#include "filesys/cache.h"
#include "filesys/filesys.h"
#include "threads/synch.h"

#define BUFFER_CACHE_SIZE 64

struct buffer_cache_entry_t {
  bool occupied;  // true only if this entry is valid cache entry

  block_sector_t disk_sector;
  uint8_t buffer[BLOCK_SECTOR_SIZE];

  bool dirty;     // dirty bit
  bool access;    // reference bit, for clock algorithm
};

/* Buffer cache entries. */
static struct buffer_cache_entry_t cache[BUFFER_CACHE_SIZE];

/* A global lock for synchronizing buffer cache operations. */
static struct lock buffer_cache_lock;
```

حال با دانستن ساختار cache و استفاده از آن جهت **کاهش دسترسی مستقیم به دیسک** برای بلوک‌های پرکاربرد در وضعیت فعلی، به معرفی توابع ارائه شده در این فایل می‌پردازیم:

- **buffer_cache_init**:

پس از مقداردهی متغیر جهانی **lock**، تمامی عناصر آرایه‌ی مشترک cache را در وضعیت خالی و قابل استفاده قرار می‌دهد.

```
void
buffer_cache_init (void)
{
  lock_init (&buffer_cache_lock);

  // initialize entries
  size_t i;
  for (i = 0; i < BUFFER_CACHE_SIZE; ++ i)
  {
    cache[i].occupied = false;
  }
}
```

- **buffer_cache_flush**:

در صورت وجود وضعیت ناسازگار میان بایت‌های ذخیره شده در یک عنصر از cache و بلوک متناظر با آن در دیسک، آن را با نوشتن محتویات عنصر مورد نظر در دیسک توسط تابع block_write از کتابخانه‌ی block.h و صفر کردن مقدار منطقی dirty برای به حالت سازگار برمی‌گرداند. اجرای این تابع مشروط به معتبر بودن عنصر مورد نظر از cache و قرار گیری در ناحیه‌ای بحرانی متناظر با lock در نخ جاری است.

```
static void
buffer_cache_flush (struct buffer_cache_entry_t *entry)
{
  ASSERT (lock_held_by_current_thread(&buffer_cache_lock));
  ASSERT (entry != NULL && entry->occupied == true);

  if (entry->dirty) {
    block_write (fs_device, entry->disk_sector, entry->buffer);
    entry->dirty = false;
  }
}
```

- **buffer_cache_close**:

در ناحیه‌ای بحرانی، تابع پیشین را روی تک تک عناصر cache در صورتی که معتبر و مورد استفاده باشند فراخوانی کرده و آن را کاملا روی دیسک قرار می‌دهد.

```
void
buffer_cache_close (void)
{
  // flush buffer cache entries
  lock_acquire (&buffer_cache_lock);

  size_t i;
  for (i = 0; i < BUFFER_CACHE_SIZE; ++ i)
  {
    if (cache[i].occupied == false) continue;
    buffer_cache_flush( &(cache[i]) );
  }

  lock_release (&buffer_cache_lock);
}
```

- **buffer_cache_lookup**:

با دریافت آدرس یک sector مشخص از دیسک، تمامی عناصر آرایه‌ی مشترک cache را پیمایش کرده و عنصر معتبر متناظر با ورودی تابع را در صورت وجود برمی‌گرداند.

```
static struct buffer_cache_entry_t*
buffer_cache_lookup (block_sector_t sector)
{
  size_t i;
  for (i = 0; i < BUFFER_CACHE_SIZE; ++ i)
  {
    if (cache[i].occupied == false) continue;
    if (cache[i].disk_sector == sector) {
      // cache hit.
      return &(cache[i]);
    }
  }
  return NULL; // cache miss
}
```

- **buffer_cache_evict**:

در صورت وجود نخ اجرایی فعلی در ناحیه‌ی بحرانی متناظر با lock، یک فضای خالی را در آرایه‌ی cache جهت درج یک sector جدید پیدا کرده و برمی‌گرداند. یافتن فضای خالی مناسب توسط الگوریتم ***clock*** انجام می‌شود که در آن، آرایه‌ی cache به صورت نامتناهی از اول به آخر پیمایش می‌شود. اگر مقدار منطقی occupied در عنصر جاری صفر باشد، الگوریتم پایان می‌یابد چرا که یک فضای بلااستفاده وجود دارد. در غیر این صورت، cache پر شده است و برای ایجاد فضای جدید، حذف یکی از عناصر معتبر لازم است. این تصمیم با توجه به مقدار منطقی access گرفته می‌شود. به عبارتی، اگر یک عنصر اخیرا مورد دسترسی قرار گرفته شده باشد، مقدار access برای آن صفر می‌شود اما تا پیمایش بعد حذف نشده و اصطلاحا **شانسی دوباره** به آن داده می‌شود. در غیر این صورت، جایگاه فعلی گزینه‌ای مناسب برای حذف است. با داشتن جایگاه مناسب، ابتدا در صورت لزوم تابع buffer_cache_flush جهت ایجاد سازگاری روی آن فراخوانی می‌شود. سپس با تغییر وضعیت آن به خالی، در خروجی تابع برگردانده می‌شود.

```
static struct buffer_cache_entry_t*
buffer_cache_evict (void)
{
  ASSERT (lock_held_by_current_thread(&buffer_cache_lock));

  // clock algorithm
  static size_t clock = 0;
  while (true) {
    if (cache[clock].occupied == false) {
      // found an empty slot -- use it
      return &(cache[clock]);
    }

    if (cache[clock].access) {
      // give a second chance
      cache[clock].access = false;
    }
    else break;

    clock ++;
    clock %= BUFFER_CACHE_SIZE;
  }

  // evict cache[clock]
  struct buffer_cache_entry_t *slot = &cache[clock];
  if (slot->dirty) {
    // write back into disk
    buffer_cache_flush (slot);
  }

  slot->occupied = false;
  return slot;
}
```

- **buffer_cache_read**:

با دریافت نقطه‌ی شروع یک sector در دیسک، به اندازه‌ی یک بلوک از آن نقطه خوانده و محتویات حافظه را در محل متناظر با اشاره‌گر ورودی یعنی target کپی می‌کند. در بدنه‌ی تابع، ابتدا sector مورد نظر با فراخوانی تابع buffer_cache_lookup در cache جست و جو می‌شود. اگر جست و جو ناموفق باشد، لازم است با فراخوانی تابع پیشین یک جایگاه برای درج عنصر جدیدی در cache متناظر با sector ورودی پیدا شود. جایگاه معتبر جدید به sector ورودی اشاره داده شده و با فراخوانی تابع block_read، محتویات سازگار دیسک در آن قرار می‌گیرد. در نهایت مقدار access برای جایگاه جست و جو شده یا جدید، یک شده و مقدار دخیره شده در cache در حافظه‌ی متناظر با target کپی می‌شود.

```
void
buffer_cache_read (block_sector_t sector, void *target)
{
  lock_acquire (&buffer_cache_lock);

  struct buffer_cache_entry_t *slot = buffer_cache_lookup (sector);
  if (slot == NULL) {
    // cache miss: need eviction.
    slot = buffer_cache_evict ();
    ASSERT (slot != NULL && slot->occupied == false);

    // fill in the cache entry.
    slot->occupied = true;
    slot->disk_sector = sector;
    slot->dirty = false;
    block_read (fs_device, sector, slot->buffer);
  }

  // copy the buffer data into memory.
  slot->access = true;
  memcpy (target, slot->buffer, BLOCK_SECTOR_SIZE);

  lock_release (&buffer_cache_lock);
}
```

- **buffer_cache_write**:

عملکرد این تابع مشابه تابع پیشین است. با این تفاوت که پس از یافتن جایگاه مناسب برای عنصر cache، مقدار منطقی dirty را برای آن یک کرده و محتوای دریافت شده از حافظه‌ی اصلی در source را در آرایه‌ی بایت‌های موجود در cache به صورت ناسازگار با مقدار اصلی در دیسک می‌نویسد.

```
void
buffer_cache_write (block_sector_t sector, const void *source)
{
  lock_acquire (&buffer_cache_lock);

  struct buffer_cache_entry_t *slot = buffer_cache_lookup (sector);
  if (slot == NULL) {
    // cache miss: need eviction.
    slot = buffer_cache_evict ();
    ASSERT (slot != NULL && slot->occupied == false);

    // fill in the cache entry.
    slot->occupied = true;
    slot->disk_sector = sector;
    slot->dirty = false;
    block_read (fs_device, sector, slot->buffer);
  }

  // copy the data form memory into the buffer cache.
  slot->access = true;
  slot->dirty = true;
  memcpy (slot->buffer, source, BLOCK_SECTOR_SIZE);

  lock_release (&buffer_cache_lock);
}
```

فایل userprog/process.c

در راه حل مورد استفاده، تغییرات برنامه‌ی کاربری متناظر با پروژه‌ی چهارم تماما در فایل بعد قرار گرفته‌اند.

فایل userprog/syscall.c

تغییرات این فایل نسبت به پروژه‌ی دوم، شامل تعریف **پنج** ***system call*** جدید مطابق توابع زیر می‌شود:

```
bool sys_chdir(const char *filename);
bool sys_mkdir(const char *filename);
bool sys_readdir(int fd, char *filename);
bool sys_isdir(int fd);
int sys_inumber(int fd);
```

کدهای موجود در هر case از دستور switch شامل خواندن پارامترهای ورودی هر *system call* از stack فرآیند آن با استفاده از تابع memread_user، فراخوانی تابع تعریف شده در سطح سیستم با آن پارامترها و دریافت مقادیر بازگشتی در حافظه‌ی سطح پردازنده‌ی eax می‌شود:

```
case SYS_CHDIR: // 15
  {
    const char* filename;
    int return_code;

    memread_user(f->esp + 4, &filename, sizeof(filename));

    return_code = sys_chdir(filename);
    f->eax = return_code;
    break;
  }

case SYS_MKDIR: // 16
  {
    const char* filename;
    int return_code;

    memread_user(f->esp + 4, &filename, sizeof(filename));

    return_code = sys_mkdir(filename);
    f->eax = return_code;
    break;
  }

case SYS_READDIR: // 17
  {
    int fd;
    char *name;
    int return_code;

    memread_user(f->esp + 4, &fd, sizeof(fd));
    memread_user(f->esp + 8, &name, sizeof(name));

    return_code = sys_readdir(fd, name);
    f->eax = return_code;
    break;
  }
```

```
case SYS_ISDIR: // 18
  {
    int fd;
    int return_code;

    memread_user(f->esp + 4, &fd, sizeof(fd));
    return_code = sys_isdir(fd);
    f->eax = return_code;
    break;
  }

case SYS_INUMBER: // 19
  {
    int fd;
    int return_code;

    memread_user(f->esp + 4, &fd, sizeof(fd));
    return_code = sys_inumber(fd);
    f->eax = return_code;
    break;
  }
```

حال به ارائه توضیحات لازم برای توابع متناظر با هر یک از *system call*های جدید می‌پردازیم:

- **sys_chdir:**

بدنه‌ی تابع شامل فراخوانی تابع filesys_chdir در ناحیه‌ی بحرانی متناظر با قفل تعریف شده در فایل filesys.c مشروط به وجود دسترسی برای کاربر است.

```
bool sys_chdir(const char *filename)
{
  bool return_code;
  check_user((const uint8_t*) filename);

  lock_acquire (&filesys_lock);
  return_code = filesys_chdir(filename);
  lock_release (&filesys_lock);

  return return_code;
}
```

- **sys_mkdir:**

بدنه‌ی تابع شامل فراخوانی تابع filesys_create در ناحیه‌ی بحرانی متناظر با قفل تعریف شده در فایل filesys.c مشروط به وجود دسترسی برای کاربر است.

```
bool sys_mkdir(const char *filename)
{
  bool return_code;
  check_user((const uint8_t*) filename);

  lock_acquire (&filesys_lock);
  return_code = filesys_create(filename, 0, true);
  lock_release (&filesys_lock);

  return return_code;
}
```

- **sys_isdir:**

بدنه‌ی تابع شامل فراخوانی توابع find_file_desc و file_get_inode از file.c و فراخوانی تابع inode_is_directory از inode.c به ترتیب جهت دریافت اطلاعات فایل، دریافت *inode* متناظر با آن و تشخیص پوشه بودن آن در ناحیه‌ی بحرانی متناظر با قفل تعریف شده در فایل filesys.c است.

```
bool sys_isdir(int fd)
{
  lock_acquire (&filesys_lock);

  struct file_desc* file_d = find_file_desc(thread_current(), fd, FD_FILE | FD_DIRECTORY);
  bool ret = inode_is_directory (file_get_inode(file_d->file));

  lock_release (&filesys_lock);
  return ret;
}
```

- **sys_inumber:**

بدنه‌ی تابع شامل فراخوانی توابع find_file_desc و file_get_inode از file.c مطابق تابع پیشین و فراخوانی تابع inode_get_number از inode.c جهت دریافت شماره‌ی *inode* متناظر با فایل در ناحیه‌ی بحرانی متناظر با قفل تعریف شده در فایل filesys.c است.

```
int sys_inumber(int fd)
{
  lock_acquire (&filesys_lock);

  struct file_desc* file_d = find_file_desc(thread_current(), fd, FD_FILE | FD_DIRECTORY);
  int ret = (int) inode_get_inumber (file_get_inode(file_d->file));

  lock_release (&filesys_lock);
  return ret;
}
```

- **sys_readdir:**

بدنه‌ی تابع شامل فراخوانی توابع ذکر شده در بخش پیشین جهت بررسی معتبر بودن اطلاعات و شناسه‌های یک پوشه در ناحیه‌ی بحرانی متناظر با قفل تعریف شده در فایل filesys.c است. تشخیص نهایی اعتبار محتوای پوشه با فراخوانی تابع dir_readdir از فایل dir.c انجام می‌شود.

```
bool sys_readdir(int fd, char *name)
{
  struct file_desc* file_d;
  bool ret = false;

  lock_acquire (&filesys_lock);
  file_d = find_file_desc(thread_current(), fd, FD_DIRECTORY);
  if (file_d == NULL) goto done;

  struct inode *inode;
  inode = file_get_inode(file_d->file); // file descriptor -> inode
  if(inode == NULL) goto done;

  // check whether it is a valid directory
  if(! inode_is_directory(inode)) goto done;

  ASSERT (file_d->dir != NULL); // see sys_open()
  ret = dir_readdir (file_d->dir, name);

done:
  lock_release (&filesys_lock);
  return ret;
}
```

فایل Directory

در این قسمت هدف پیاده سازی توابعی برای اضافه کردن قابلیت فایل سیستم سلسله مراتبی است. به طور کلی می خواهیم هر دایرکتوری بتواند entry های دیگر شامل فایل ها و دایرکتوری های دیگری درون خودش داشته باشد و file system از یک حالت flat که همه چیز در root قرار دارد خارج شود.

توضیح تابع هایی که در فایل directory.c وجود دارند به طور کلی در مورد ایجاد و حذف یک دایرکتوری و اضافه کردن child entry یا حذف آن ها، مسائلی مانند بدست آوردن دایرکتوری parent برای entry فعلی و گرفتن inode متناظر برای یک دایرکتوری به منظور کار با آن، که مفهوم باز کردن یک دایرکتوری است و همچنین باز کردن دایرکتوری parent و کار هایی از این قبیل است.

سایر عملیات روی دایرکتوری ها مانند دستور chdir یا همان change directory و pwd یا گرفتن آدرس دایرکتوری فعلی و ... در توضیحات قسمت filesys.c آمده است.

فایل directory.h

در این فایل prototype تابع ها و متغیر NAME_MAX که برای مشخص کردن حداکثر کاراکتر نام یک دایرکتوری است، آمده اند.

تعریف prototype تابع ها در دو بخش کلی شامل تابع های برای باز و بسته کردن دایرکتوری و تابع های خواندن و نوشتن در دایرکتوری ها آمده است.

در شکل زیر تصویر این فایل آمده است که به تفصیل به توضیح هر یک در فایل directory.c می پردازیم.

filesys > C directory.h

```c
#ifndef FILESYS_DIRECTORY_H
#define FILESYS_DIRECTORY_H

#include <stdbool.h>
#include <stddef.h>
#include "devices/block.h"

/* Maximum length of a file name component.
   This is the traditional UNIX maximum length.
   After directories are implemented, this maximum length may be
   retained, but much longer full path names must be allowed. */
#define NAME_MAX 14

struct inode;

/* Opening and closing directories. */
bool dir_create (block_sector_t sector, size_t entry_cnt);
struct dir *dir_open (struct inode *);
struct dir *dir_open_root (void);
struct dir *dir_reopen (struct dir *);
void dir_close (struct dir *);
struct inode *dir_get_inode (struct dir *);

/* Reading and writing. */
bool dir_lookup (const struct dir *, const char *name, struct inode **);
bool dir_add (struct dir *, const char *name, block_sector_t);
bool dir_remove (struct dir *, const char *name);
bool dir_readdir (struct dir *, char name[NAME_MAX + 1]);

bool dir_is_root (struct dir* dir);
bool dir_get_parent (struct dir* dir, struct inode **inode);

#endif /* filesys/directory.h */

```

فایل directory.c

در این فایل ابتدا دو struct به منظور تعریف تایپ های dir و dir_entry آمده که همانطور که از اسم آن ها مشخص است، به ترتیب نمایانگر یک دایرکتوری و یک entry در دایرکتوری هستند.

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

```
struct dir_entry
  {
    block_sector_t inode_sector;        /* Sector number of header. */
    char name[NAME_MAX + 1];            /* Null terminated file name. */
    bool in_use;                        /* In use or free? */
  };
```

در ادامه پیاده سازی توابع آمده که به توضیح هر یک می پردازیم. در ابتدا تعریف هر تابع آمده و زیر آن توضیح مرتبط با تابع نوشته شده است.

```
bool
dir_create (block_sector_t sector, size_t entry_cnt)
{
  return inode_create (sector, entry_cnt * sizeof (struct dir_entry), true);
}
```

این تابع با گرفتن یک sector و تعداد entry های موجود روی آن sector، یک indoe ساخته و خروجی این تابع در صورت موفقیت آمیز بودن ساخت inode مقدار true خواهد بود. تابع inode_create در قسمت فایل inode توضیح داده شده است.

```
struct dir *
dir_open (struct inode *inode)
{
  struct dir *dir = calloc (1, sizeof *dir);
  if (inode != NULL && dir != NULL)
    {
      dir->inode = inode;
      dir->pos = 0;
      return dir;
    }
  else
    {
      inode_close (inode);
      free (dir);
      return NULL;
    }
}
```

این تابع یک inode ورودی می گیرد و آن را به عنوان inode آن دایرکتوری ست می کند. اگر نتواند فضای کافی برای directory اختصاص دهد و یا indoe داده شده قبلا به آن حافظه ای تخصیص داده نشده و مقدار آن NULL باشد، تابع NULL برمیگرداند و نمی تواند یک دایرکتوری باز کند.

در حقیقت برای باز کردن یک دایرکتوری، نیاز به تخصیص inode برای نگه داشتن مشخصات آن داریم که بتوان با محتوای آن کار کرد و sector های متناظر روی حافظه را شناخت که این تابع این وظیفه را به عهده دارد و یک dir که از جنس struct ای است که در ابتدا تعریف شده بود برمیگرداند.

```
struct dir *
dir_open_root (void)
{
  return dir_open (inode_open (ROOT_DIR_SECTOR));
}
```

این تابع مشابه کاری مشابه تابع قبلی را با صدا زدن آن انجام میدهد. تنها تفاوت آن این است که یک dir برای root برمیگرداند. منظور از root همان آدرس / یا دایرکتوری ای است که همه ی دایرکتوری های دیگر را شامل می شود. در ابتدا فایل سیستم پینتوس تنها از root پشتیبانی می کرد و همه ی فایل ها در آن قرار می گرفت اما در این پروژه امکان دایرکتوری های سلسله مراتبی اضافه شده است.

```
struct dir *
dir_reopen (struct dir *dir)
{
  return dir_open (inode_reopen (dir->inode));
}
```

این تابع یک دایرکتوری دیگر برای inode موجود در دایرکتوری که پاس داده شده باز می کند و برمیگرداند. در صورت ناموفق بودن عملیات مقدار NULL برگشت داده می شود.

```
void
dir_close (struct dir *dir)
{
  if (dir != NULL)
    {
      inode_close (dir->inode);
      free (dir);
    }
}
```

دایرکتوری گرفته شده را می بندد و تمامی منابع گرفته شده توسط آن مانند inode و حافظه ی خود dir را آزاد می کند.

```
struct inode *
dir_get_inode (struct dir *dir)
{
  return dir->inode;
}
```

Inode درون یک دایرکتوری را برمیگرداند. این کار با دسترسی مستقیم به واسطه ی پوینتر به مقدار inode هم امکان پذیر است.

```
static bool
lookup (const struct dir *dir, const char *name,
        struct dir_entry *ep, off_t *ofsp)
{
  struct dir_entry e;
  size_t ofs;

  ASSERT (dir != NULL);
  ASSERT (name != NULL);

  for (ofs = 0; inode_read_at (dir->inode, &e, sizeof e, ofs) == sizeof e;
       ofs += sizeof e)
    if (e.in_use && !strcmp (name, e.name))
      {
        if (ep != NULL)
          *ep = e;
        if (ofsp != NULL)
          *ofsp = ofs;
        return true;
      }
  return false;
}
```

کل یک دایرکتوری را برای پیدا کردن یک فایل با نام ورودی گرفته شده جستجو می کند. اگر فایل را پیدا کرد مقدار true بر میگرداند و مقدار ep را به آن directory entry ست می کند. همچنین مقدار offset pointer را که با متغیر به نام ofsp ورودی گرفته است برمیگرداند.

```
bool
dir_lookup (const struct dir *dir, const char *name,
            struct inode **inode)
{
  struct dir_entry e;

  ASSERT (dir != NULL);
  ASSERT (name != NULL);

  inode_lock(dir_get_inode((struct dir *) dir));
  if (lookup (dir, name, &e, NULL))
    *inode = inode_open (e.inode_sector);
  else
    *inode = NULL;
  inode_unlock(dir_get_inode((struct dir *) dir));

  return *inode != NULL;
}
```

این تابع با گرفتن نام یک directory به دنبال آن در دایرکتوری مشخص شده در ورودی می گردد. اگر دایرکتوری با چنین اسمی پیدا شد، inode ورودی گرفته شده را به آن ست می کند و true برمی گرداند.

نکته ی مهم این است که در صورت برگشتن مقدار false از این تابع، وظیفه ی caller است که inode پاس داده شده را آزاد کند و فضای حافظه ی آن را برگرداند.

```
bool
dir_add (struct dir *dir, const char *name, block_sector_t inode_sector)
{
  struct dir_entry e;
  off_t ofs;
  bool success = false;

  ASSERT (dir != NULL);
  ASSERT (name != NULL);

  inode_lock(dir_get_inode(dir));
  /* Check NAME for validity. */
  if (*name == '\0' || strlen (name) > NAME_MAX)
    goto done;

  /* Check that NAME is not in use. */
  if (lookup (dir, name, NULL, NULL))
    goto done;
```

```
  if (!inode_add_parent(inode_get_inumber(dir_get_inode(dir)),
     inode_sector))
   {
    goto done;
   }

  for (ofs = 0; inode_read_at (dir->inode, &e, sizeof e, ofs) == sizeof e;
       ofs += sizeof e)
    if (!e.in_use)
     break;

  /* Write slot. */
  e.in_use = true;
  strlcpy (e.name, name, sizeof e.name);
  e.inode_sector = inode_sector;
  success = inode_write_at (dir->inode, &e, sizeof e, ofs) == sizeof e;

 done:
  inode_unlock(dir_get_inode(dir));
  return success;
}
```

این تابع یک فایل با اسم ورودی گرفته شده به دایرکتوری که ورودی گرفته اضافه می کند. Inode فایلی که می خواهد به دایرکتوری اضافه شود در inode_sector قرار دارد که تابع آن را هم به عنوان ورودی دیگر دریافت می کند.

اگر نتواند عملیات خواسته شده را انجام دهد، مقدار false برگشت داده می شود. این اتفاق زمانی است که نام فایل داده شده معتبر نباشد یا بسیار طولانی باشد یا از جنبه ی دیگر یک disk or memory error اتفاق بیفتد.

نحوه ی کار به این صورت است که یک متغیر success تعریف شده که مقدار آن false است. در if های ابتدایی چک می شود تا اگر شرایط مطلوب مثل خاتمه یافتن string با مقدار ۰/ اتفاق نیفتد یا طول آن زیاد باشد یا همچین اسمی وجود داشته باشد، به برچسب done پرش می کند بدون اینکه مقدار success برابر true شود و بنابراین false برگشت داده می شود.

در ادامه در یک حلقه آنقدر offset را جلو می برد تا به یک offset خالی و بدون استفاده برسد و با مشخص شدن محل offset و entry متناظر، آن ها را در inode دایرکتوری که به تابع پاس شده اضافه می کند و مقدار success در صورت موفق بودن برابر true می شود.

```
bool
dir_remove (struct dir *dir, const char *name)
{
  struct dir_entry e;
  struct inode *inode = NULL;
  bool success = false;
  off_t ofs;
```

```
  ASSERT (dir != NULL);
  ASSERT (name != NULL);

  inode_lock(dir_get_inode(dir));
  /* Find directory entry. */
  if (!lookup (dir, name, &e, &ofs))
    goto done;

  /* Open inode. */
  inode = inode_open (e.inode_sector);
  if (inode == NULL)
    goto done;

  /* Directory to be deleted is used by other processes */
  if (inode_is_dir(inode) && inode_get_open_cnt(inode) > 1)
    goto done;

  /* Directory to be deleted is nonempty */
  if (inode_is_dir(inode) && !dir_is_empty(inode))
    goto done;

  /* Erase directory entry. */
  e.in_use = false;
  if (inode_write_at (dir->inode, &e, sizeof e, ofs) != sizeof e)
    goto done;

  /* Remove inode. */
  inode_remove (inode);
  success = true;

done:
  inode_close (inode);
  inode_unlock(dir_get_inode(dir));
  return success;
}
```

این تابع با گرفتن یک نام و یک دایرکتوری، entry متناظر با آن نام را از دایرکتوری حذف می کند. اگر ناموفق بود false برمیگرداند که برای زمانی است که هیچ entry با نام داده شده پیدا نشود.

طرز کار مشابه تابع قبلی است و با تعریف متغیر success = false در ابتدای تابع و چک کردن شرط هایی که موجب برقرار ماندن این مقدار می شود در ابتدای کار، جلو می رود و اگر هر یک از این شرایط به وقوع پیوست به برچسب done پرش می کند.

ابتدا با تابع lookup به دنبال entry با آن اسم می گردد که اگر پیدا نکرد به done می رود.

سپس یک inode برای اشاره به sector مورد نظر باز می کند. اگر دایرکتوری متناظر با inode در حال استفاده بود امکان حذف آن نیست و به انتهای تابع می رود. اگر دایرکتوری شامل entry های دیگر بود و ابتدا خالی نشده بود هم آن را حذف نمی کند و به انتهای تابع پرش می کند.

در ادامه اگر هیچ یک از شرایط گفته شده برقرار نبود، دایرکتوری را حذف و inode را آزاد می کند و مقدار true برمیگرداند.

نکته ی مهم این است که هنگام گرفتن inode یک دایرکتوری و کار کردن با آن، inode را lock می کند تا قفل آن را در اختیار بگیرد و از تغییر همزمان inode در قسمت های دیگر برنامه جلوگیری شود.

```
bool
dir_readdir (struct dir *dir, char name[NAME_MAX + 1])
{
  struct dir_entry e;
  inode_lock(dir_get_inode(dir));
  while (inode_read_at (dir->inode, &e, sizeof e, dir->pos) == sizeof e)
    {
      dir->pos += sizeof e;
      if (e.in_use)
        {
          strlcpy (name, e.name, NAME_MAX + 1);
    inode_unlock(dir_get_inode(dir));
          return true;
        }
    }
  inode_unlock(dir_get_inode(dir));
  return false;
}
```

این تابع با هر بار فراخوانی entry بعدی در یک دایرکتوری را میخواند و اسم آن را در متغیر name که ورودی گرفته قرار می دهد.

برای این منظور entry ها را تا وقتی به انتها نرسیده پیمایش می کند و pos که عضو struct دایرکتوری پاس داده شده است را به اندازه ی entry فعلی جلو می برد تا به position فایل یا فولدر یا به طور کلی تر entry بعدی برسد.

اگر entry بعدی به inode ای اختصاص داده شده بود و در حال استفاده بود، متغیر name را با اسم آن entry پر می کند و مقدار true برمیگرداند. در غیر اینصورت قفل inode متناظر با دایرکتوری گرفته شده را آزاد میکند و مقدار false برمیگرداند.

```
bool dir_is_empty (struct inode *inode)
{
  struct dir_entry e;
  off_t pos = 0;
  while (inode_read_at (inode, &e, sizeof e, pos) == sizeof e)
    {
      pos += sizeof e;
      if (e.in_use)
        {
          return false;
        }
    }
  return true;
}
```

این تابع با گرفتن یک inode، تک تک entry های آن را چک می کند و اگر همه ی entry ها آزاد بودند، مقدار true برمیگرداند. در حقیقت چک می کند دایرکتوری متناظر با inode پاس داده شده خالی است یا خیر.

```
bool dir_is_root (struct dir* dir)
{
  if (!dir)
    {
      return false;
    }
  if (inode_get_inumber(dir_get_inode(dir)) == ROOT_DIR_SECTOR)
    {
      return true;
    }
  return false;
}
```

این تابع چک می کند یک دایرکتوری root هست یا خیر. برای این کار یک اشاره گر به دایرکتوری ورودی می گیرد و در ابتدا چک می کند که متغیر پاس داده شده دایرکتوری باشد. سپس اگر inode متناظر با این دایرکتوری برابر inode دایرکتوری root باشد مقدار true برمیگرداند.

تعریف دایرکتوری root در بالاتر آمده است با این حال منظور از root دایرکتوری است که تمامی دایرکتوری و فایل های دیگر در file system را در بر می گیرد و مبدا آدرس دهی با آدرس / است.

```
inode dir_get_parent (struct dir* dir, struct inode **inode)
{

  block_sector_t sector = inode_get_parent(dir_get_inode(dir));
  *inode = inode_open (sector);
  return *inode;
}
```

این تابع با گرفتن یک استراکت از جنس dir و یک اشاره گر به آدرس یک inode مقدار parent یا شاخه ی بالاتر در سلسله مراتب file system را برای دایرکتوری پاس داده شده بدست می آورد و مقدار inode آن را برمیگرداند.

فایل filesys.c

در این پروژه، لازم است تا یک system file برای پینتوس طراحی شود که از قابلیت های آن، امکان افزایش اندازهی فایل ها و سکتورها، امکان caching و امکان directory های تو در تو در این سیستم عامل می باشد. در زیر فایل filesys.c را شرح می دهیم.

تابع filesys_init ماژول سیستم فایل را آغاز می کند. اگر FORMAT مقدار true داشته باشد ، تابع do_format را فراخوانی می کند. همچنین تابع filesys_cache_init را نیز فراخوانی می کند.

```
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 ();
  filesys_cache_init();
  free_map_init ();

  if (format)
    do_format ();

  free_map_open ();
}
```

تابع filesys_done ماژول سیستم فایل را خاموش می کند و هرگونه داده نانوشته را روی دیسک می نویسد و filesys_cache_write_to_disk فراخوانی می شود .

```
void
filesys_done (void)
{
  filesys_cache_write_to_disk(true);
  free_map_close ();
}
```

تابع filesys_create با INITIAL_SIZE داده شده فایلی به نام NAME ایجاد می کند. در صورت موفقیت ، true برمی گردد ، در غیر این صورت false است. اگر فایلی به نام NAME از قبل وجود داشته باشد ، یا اگر تخصیص حافظه داخلی fail شده باشد ، fail می شود.همچنین شرطی اضافه شده تا نام فایل شامل . و .. نباشد.

```
bool
filesys_create (const char *name, off_t initial_size, bool isdir)
{
  block_sector_t inode_sector = 0;
  struct dir *dir = get_containing_dir(name);
  char* file_name = get_filename(name);
  bool success = false;
  if (strcmp(file_name, ".") != 0 && strcmp(file_name, "..") != 0)
    {
      success = (dir != NULL
     && free_map_allocate (1, &inode_sector)
     && inode_create (inode_sector, initial_size, isdir)
     && dir_add (dir, file_name, inode_sector));
    }
  if (!success && inode_sector != 0)
    free_map_release (inode_sector, 1);
  dir_close (dir);
  free(file_name);

  return success;
}
```

تابع open_filesys فایلی را با NAME داده شده باز می کند. در صورت موفقیت فایل جدید را بر می گرداند یا در غیر اینصورت یک اشاره گر null برمی گرداند. اگر هیچ فایلی به نام NAME وجود نداشته باشد ، یا اگر تخصیص حافظه داخلی fail شود ، fail می شود.

همچنین ابتدا یک شرط اضافه شده که اندازه ی name را بررسی میکند تا صفر نباشد. قبل از فراخوانی lookup_dir نیز شرط هایی بررسی می شود. شرط هایی مانند NULL نبودن directory ،نبودن .. .

```
   or if an internal memory allocation fails. */
struct file *
filesys_open (const char *name)
{
  if (strlen(name) == 0)
    {
      return NULL;
    }
  struct dir* dir = get_containing_dir(name);
  char* file_name = get_filename(name);
  struct inode *inode = NULL;

  if (dir != NULL)
    {
      if (strcmp(file_name, "..") == 0)
  {
    if (!dir_get_parent(dir, &inode))
      {
        free(file_name);
        return NULL;
      }
  }
      else if ((dir_is_root(dir) && strlen(file_name) == 0) ||
          strcmp(file_name, ".") == 0)
  {
    free(file_name);
    return (struct file *) dir;
  }
      else
  {
    dir_lookup (dir, file_name, &inode);
  }
    }

  dir_close (dir);
  free(file_name);
```

```
  dir_close (dir);
  free(file_name);

  if (!inode)
    {
      return NULL;
    }

  if (inode_is_dir(inode))
    {
      return (struct file *) dir_open(inode);
    }
  return file_open (inode);
}
```

تابع filesys_remove فایلی با نام NAME را حذف می کند. در صورت موفقیت true ، در غیر این صورت false برمی گردد. اگر هیچ فایلی به نام NAME وجود نداشته باشد ، یا اگر تخصیص حافظه داخلی fail شود ، fail می شود.

```
bool
filesys_remove (const char *name)
{
  struct dir* dir = get_containing_dir(name);
  char* file_name = get_filename(name);
  bool success = dir != NULL && dir_remove (dir, file_name);
  dir_close (dir);
  free(file_name);

  return success;
}
```

تابع do_format سیستم فایل را قالب بندی می کند.

```
static void
do_format (void)
{
  printf ("Formatting file system...");
  free_map_create ();
  if (!dir_create (ROOT_DIR_SECTOR, 16))
    PANIC ("root directory creation failed");
  free_map_close ();
  printf ("done.\n");
}
```

این تابع، sys file داده شده به آن را حذف می کند. در صورتی که این filesys دارای فرزند و یا والد باشد، آن ها را نیز می بندد

```
bool filesys_chdir (const char* name)
{
  struct dir* dir = get_containing_dir(name);
  char* file_name = get_filename(name);
  struct inode *inode = NULL;

  if (dir != NULL)
    {
      if (strcmp(file_name, "..") == 0)
  {
    if (!dir_get_parent(dir, &inode))
      {
        free(file_name);
        return false;
      }
  }
      else if ((dir_is_root(dir) && strlen(file_name) == 0) ||
    strcmp(file_name, ".") == 0)
  {
    thread_current()->cwd = dir;
    free(file_name);
    return true;
  }
      else
  {
    dir_lookup (dir, file_name, &inode);
  }
    }

  dir_close (dir);
  free(file_name);

  dir = dir_open (inode);
  if (dir)
    {
```

```
    dir_close (dir);
    free(file_name);

    dir = dir_open (inode);
    if (dir)
      {
        dir_close(thread_current()->cwd);
        thread_current()->cwd = dir;
        return true;
      }
    return false;
}
```

این تابع، directory ای که شامل sys file داده شده است را بر می گرداند. این کار را از طریق چک کردن directory ها از بالا به درون انجام می دهد و زمانی که به مسیر داده شده رسید، directory مربوطه را بر می گرداند.

```
struct dir* get_containing_dir (const char* path)
{
  char s[strlen(path) + 1];
  memcpy(s, path, strlen(path) + 1);

  char *save_ptr, *next_token = NULL, *token = strtok_r(s, "/", &save_ptr);
  struct dir* dir;
  if (s[0] == ASCII_SLASH || !thread_current()->cwd)
    {
      dir = dir_open_root();
    }
  else
    {
      dir = dir_reopen(thread_current()->cwd);
    }

  if (token)
    {
      next_token = strtok_r(NULL, "/", &save_ptr);
    }
  while (next_token != NULL)
    {
      if (strcmp(token, ".") != 0)
  {
    struct inode *inode;
    if (strcmp(token, "..") == 0)
      {
        if (!dir_get_parent(dir, &inode))
  {
    return NULL;
  }
      }
    else
      {
        if (!dir_lookup(dir, token, &inode))
  {
    return NULL;
  }
```

```
      }
    else
      {
        if (!dir_lookup(dir, token, &inode))
  {
    return NULL;
  }
      }
    if (inode_is_dir(inode))
      {
        dir_close(dir);
        dir = dir_open(inode);
      }
    else
      {
        inode_close(inode);
      }
  }
      token = next_token;
      next_token = strtok_r(NULL, "/", &save_ptr);
    }
  return dir;
}
```

تابع get_filename، یک مسیر را به عنوان ورودی میگیرد و نام فایل مربوط به آن مسیر را بر می گرداند.

```
char* get_filename (const char* path)
{
  char s[strlen(path) + 1];
  memcpy(s, path, strlen(path) + 1);

  char *token, *save_ptr, *prev_token = "";
  for (token = strtok_r(s, "/", &save_ptr); token != NULL;
       token = strtok_r (NULL, "/", &save_ptr))
    {

      prev_token = token;
    }
  char *file_name = malloc(strlen(prev_token) + 1);
  memcpy(file_name, prev_token, strlen(prev_token) + 1);
  return file_name;
}
```