Project4: File Systems

PintOS

陈震雄

2025 年 3 月 31 日

武汉大学

Table of contents

Introduction

Introduction

In the previous two assignments, you made extensive use of a file system without actually worrying about how it was implemented underneath. For this last assignment, you will **improve the implementation of the file system.**

Task 3: Buffer Cache

Exercise 3.1

Modify the file system to keep a cache of file blocks.

When a request is made to read or write a block, **check to see if it is in the cache**, and if so, use the cached data without going to disk.

Otherwise, **fetch the block from disk into the cache**, evicting an older entry if necessary.

You are limited to a cache no greater than 64 sectors in size.

Implement buffer cache and cache replacement algorithm.

You must implement a cache replacement algorithm that is at least as good as the ”clock” algorithm.

write-behind,read-ahead

Data structure

```
1   struct disk_cache
2   {
3       uint8_t block[BLOCK_SECTOR_SIZE];   /**< 512 Bytes */
4       block_sector_t disk_sector;         /**< disk sector */
5
6       bool is_free;                       /**< is free */
7       int open_cnt;                       /**< open count */
8       bool accessed;                      /**< accessed */
9       bool dirty;                         /**< dirty */
10  };
11
12  struct lock cache_lock;                 /**< cache lock */
13  struct disk_cache cache_array[64];      /**< cache array */
```

Read

```
int access_cache_entry(block_sector_t
disk_sector, bool dirty) {
  lock_acquire(&cache_lock);
  int idx = get_cache_entry(disk_sector);
  if(idx == -1)
    idx = replace_cache_entry(disk_sector, dirty);
  else {
    cache_array[idx].open_cnt++;
    cache_array[idx].accessed = true;
    cache_array[idx].dirty |= dirty;
  }
  lock_release(&cache_lock);
  return idx;
}

int replace_cache_entry(block_sector_t
disk_sector, bool dirty)
{
  int idx = get_free_entry();
  int i = 0;
  if(idx == -1) /**< cache is full. */
  {
    for(i = 0; ; i = (i + 1) % CACHE_MAX_SIZE)
    {
      if(cache_array[i].open_cnt > 0)
        continue;
      if(cache_array[i].accessed == true)
        cache_array[i].accessed = false;
```

```
    /* evict it */
        else
        {
          /* write back */
          if(cache_array[i].dirty == true)
          {
            block_write(fs_device,
            cache_array[i].disk_sector,
              &cache_array[i].block);
          }

          init_entry(i);
          idx = i;
          break;
        }
      }
    }
    cache_array[idx].disk_sector = disk_sector;
    cache_array[idx].is_free = false;
    cache_array[idx].open_cnt++;
    cache_array[idx].accessed = true;
    cache_array[idx].dirty = dirty;
    block_read(fs_device,
    cache_array[idx].disk_sector,
    &cache_array[idx].block);
    return idx;
}
```

Write

```
1   void
2   write_back(bool clear)
3   {
4       int i;
5       lock_acquire(&cache_lock);
6
7       for(i = 0; i < CACHE_MAX_SIZE; i++)
8       {
9           if(cache_array[i].dirty == true)
10          {
11              block_write(fs_device, cache_array[i].disk_sector, &cache_array[i].block);
12              cache_array[i].dirty = false;
13          }
14
15          /* clear cache line (filesys done) */
16          if(clear)
17            init_entry(i);
18      }
19
20      lock_release(&cache_lock);
21  }
```

Read-ahead and Write -behind

```
1   // Write-behind
2   void init_cache(void) {
3     int i;
4     lock_init(&cache_lock);
5     for(i = 0; i < CACHE_MAX_SIZE; i++)
6       init_entry(i);
7     thread_create("cache_writeback", PRI_MIN, func_periodic_writer, NULL);
8   }
9   // Read-ahead
10  void func_read_ahead(void *aux) {
11      block_sector_t disk_sector = *(block_sector_t *)aux;
12      lock_acquire(&cache_lock);
13
14      int idx = get_cache_entry(disk_sector);
15
16      /* need eviction */
17      if (idx == -1)
18          replace_cache_entry(disk_sector, false);
19
20      lock_release(&cache_lock);
21      free(aux);
22  }
23
24  void ahead_reader(block_sector_t disk_sector) {
25      block_sector_t *arg = malloc(sizeof(block_sector_t));
26      *arg = disk_sector + 1;  /**< next block */
27      thread_create("cache_read_ahead", PRI_MIN, func_read_ahead, arg);
28  }
```

Task 1: Indexed and Extensible Files

Exercise 1.1

The basic file system allocates files as a single extent, making it vulnerable to **external fragmentation**, that is, it is possible that an n-block file cannot be allocated even though n blocks are free.

Eliminate this problem by modifying the on-disk inode structure. In practice, this probably means **using an index structure with direct, indirect, and doubly indirect blocks.**

Modify inode structure to accommodate external fragmentation.

Data Structure

```
1   struct inode_disk
2     {
3       off_t length;                       /**< File size in bytes. */
4       unsigned magic;                     /**< Magic number. */
5       uint32_t unused[107];               /**< Not used. */
6
7       uint32_t direct_index;              /**< Number of direct blocks. */
8       uint32_t indirect_index;            /**< Number of indirect blocks. */
9       uint32_t double_indirect_index;     /** Number of double indirect blocks. */
10      block_sector_t blocks[14];          /**< Block pointers. */
11      bool is_dir;                        /** True if directory. */
12      block_sector_t parent;              /** Parent block sector. */
13    };
14  struct inode
15    {
16      struct list_elem elem;              /**< Element in inode list. */
17      block_sector_t sector;              /**< Sector number of disk location. */
18      int open_cnt;                       /**< Number of openers. */
19      bool removed;                       /**< True if deleted, false otherwise. */
20      int deny_write_cnt;                 /**< 0: writes ok, >0: deny writes. */
21
22      off_t length;                       /**< File size in bytes. */
23      off_t read_length;                  /** File size in bytes. */
24      uint32_t direct_index;              /**< Number of direct blocks. */
25      uint32_t indirect_index;            /**< Number of indirect blocks. */
26      uint32_t double_indirect_index;     /** Number of double indirect blocks. */
27      block_sector_t blocks[14];          /** Block pointers. */
28      bool is_dir;                        /** True if directory. */
29      block_sector_t parent;              /** Parent block sector. */
30      struct lock lock;                   /** Lock for inode. */
31    };
```

Inode create

```
bool
inode_create (block_sector_t sector, off_t length, bool is_dir)
{
  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)
    {
      /* ADDED */
      disk_inode->length = length;
      disk_inode->magic = INODE_MAGIC;
      disk_inode->is_dir = is_dir;
      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;
}
```

```
bool
inode_alloc (struct inode_disk *inode_disk)
{
  struct inode inode;
  inode.length = 0;
  inode.direct_index = 0;
  inode.indirect_index = 0;
  inode.double_indirect_index = 0;

  inode_grow(&inode, inode_disk->length);
  inode_disk->direct_index = inode.direct_index;
  inode_disk->indirect_index = inode.indirect_index;
  inode_disk->double_indirect_index =
  inode.double_indirect_index;
  memcpy(&inode_disk->blocks, &inode.blocks,
  INODE_PTRS * sizeof(block_sector_t));
  return true;
}
```

Inode grow

```
off_t
inode_grow (struct inode *inode, off_t length)
{
  static char zeros[BLOCK_SECTOR_SIZE];
  size_t grow_sectors = bytes_to_sectors(length) -
  bytes_to_sectors(inode->length);
  if (grow_sectors == 0)
    return length;

  /* direct blocks (index < 4) */
  while (inode->direct_index < DIRECT_BLOCKS
  && grow_sectors != 0)
  {
    if (!free_map_allocate(1, &inode->
    blocks[inode->direct_index])) {
      return -1; /**< Allocation failed */
    }
    block_write(fs_device, inode->
    blocks[inode->direct_index], zeros);
    inode->direct_index++;
    grow_sectors--;
  }
```

```
/* indirect blocks (index < 13) */
  while (inode->direct_index < (int) DIRECT_BLOCKS
  + INDIRECT_BLOCKS && grow_sectors != 0)
  {
    block_sector_t blocks[128];
    if (inode->indirect_index == 0) {
      if (!free_map_allocate(1, &inode->
      blocks[inode->direct_index])) {
        return -1;
      }
    } else {
      block_read(fs_device, inode->
      blocks[inode->direct_index], &blocks);
    }
    while (inode->indirect_index <
    INDIRECT_PTRS && grow_sectors != 0)
    {
      if (!free_map_allocate(1,
      &blocks[inode->indirect_index])) {
        return -1;
      }
      block_write(fs_device,
      blocks[inode->indirect_index], zeros);
      inode->indirect_index++;
      grow_sectors--;
    }
    block_write(fs_device, inode->
    blocks[inode->direct_index], &blocks);
    if (inode->indirect_index
    == INDIRECT_PTRS) {
      inode->indirect_index = 0;
      inode->direct_index++;
    }
  }
```

Inode grow

```
1  if (inode->direct_index == INODE_PTRS - 1 && grow_sectors != 0) {
2      block_sector_t level_one[128];
3      block_sector_t level_two[128];
4      if (inode->double_indirect_index == 0 && inode->indirect_index == 0) {
5        if (!free_map_allocate(1, &inode->blocks[inode->direct_index]))
6          return -1;
7      } else
8        block_read(fs_device, inode->blocks[inode->direct_index], &level_one);
9      while (inode->indirect_index < INDIRECT_PTRS && grow_sectors != 0) {
10       if (inode->double_indirect_index == 0) {
11         if (!free_map_allocate(1, &level_one[inode->indirect_index])) {
12           return -1;
13         }
14       } else
15         block_read(fs_device, level_one[inode->indirect_index], &level_two);
16       while (inode->double_indirect_index < INDIRECT_PTRS && grow_sectors != 0) {
17         if (!free_map_allocate(1, &level_two[inode->double_indirect_index])) {
18           return -1;
19         }
20         block_write(fs_device, level_two[inode->double_indirect_index], zeros);
21         inode->double_indirect_index++;
22         grow_sectors--;
23       }
24       block_write(fs_device, level_one[inode->indirect_index], &level_two);
25       if (inode->double_indirect_index == INDIRECT_PTRS) {
26         inode->double_indirect_index = 0;
27         inode->indirect_index++;
28       }
29     }
30     block_write(fs_device, inode->blocks[inode->direct_index], &level_one);
31   }
32   return length;
33 }
```

read

```
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 0;

  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;

      int cache_idx =
      access_cache_entry(sector_idx, false);
      memcpy(buffer + bytes_read,
       cache_array[cache_idx].block + sector_ofs,
       chunk_size);
      cache_array[cache_idx].accessed = true;
      cache_array[cache_idx].open_cnt--;
      /* Advance. */
      size -= chunk_size;
      offset += chunk_size;
      bytes_read += chunk_size;
    }
  return bytes_read;
}
```

Inode write

```
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;

  /* beyond EOF, need extend */
  if(offset + size > inode_length(inode))
  {
    /* no sync required for dirs */
    if(!inode->is_dir)
      lock_acquire(&inode->lock);

    inode->length = inode_grow(inode, offset + size);

    if(!inode->is_dir)
      lock_release(&inode->lock);
  }

  while (size > 0)
    {
      block_sector_t sector_idx = byte_to_sector
      (inode, inode_length(inode),
       offset);
      int sector_ofs = offset % BLOCK_SECTOR_SIZE;
      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;

      int chunk_size = size < min_left ? size :
      min_left;
      if (chunk_size <= 0)
        break;
      int cache_idx = access_cache_entry(sector_idx,
      true);
      memcpy(cache_array[cache_idx].block +
      sector_ofs, buffer + bytes_written, chunk_size);
      cache_array[cache_idx].accessed = true;
      cache_array[cache_idx].dirty = true;
      cache_array[cache_idx].open_cnt--;
      /* Advance. */
      size -= chunk_size;
      offset += chunk_size;
      bytes_written += chunk_size;
    }

  inode->read_length = inode_length(inode);
  return bytes_written;
}
```

Other funcs

```
1   struct bitmap;
2
3   void inode_init (void);
4   bool inode_create (block_sector_t, off_t, bool);
5   struct inode *inode_open (block_sector_t);
6   struct inode *inode_reopen (struct inode *);
7   block_sector_t inode_get_inumber (const struct inode *);
8   void inode_close (struct inode *);
9   void inode_remove (struct inode *);
10  off_t inode_read_at (struct inode *, void *, off_t size, off_t offset);
11  off_t inode_write_at (struct inode *, const void *, off_t size, off_t offset);
12  void inode_deny_write (struct inode *);
13  void inode_allow_write (struct inode *);
14  off_t inode_length (const struct inode *);
15  bool inode_is_dir (const struct inode *);
16  block_sector_t inode_get_parent (const struct inode *);
17
18  /* ADDED */
19  bool inode_is_dir (const struct inode *);
20  int inode_get_open_cnt (const struct inode *);
21  block_sector_t inode_get_parent (const struct inode *);
22  bool inode_set_parent (block_sector_t parent, block_sector_t child);
23  void inode_lock (const struct inode *inode);
24  void inode_unlock (const struct inode *inode);
25
26  /* ADDED */
27  bool inode_alloc (struct inode_disk *inode_disk);
28  off_t inode_grow (struct inode* inode, off_t length);
29  void inode_free (struct inode *inode)
```

Task 2: Subdirectories

Exercise 2.1

Implement a hierarchical name space. In the basic file system, all files live in a **single** directory.

Modify this to allow directory entries to point to files or to other directories.

Implement a hierarchical name space.

In the basic file system, all files live in a single directory.

Modify this to allow directory entries to point to files or to other directories.

dir_add and other funcs

```
bool
dir_add (struct dir *dir, const char *name, block_sector_t inode_sector)
{
    ...
  /* set parent of added file to this dir */
  if (!inode_set_parent(inode_get_inumber(dir_get_inode(dir)), inode_sector))
    goto done;
    ...
}
/** Returns true if DIR is the root directory. */
bool
dir_is_root(struct dir* dir)
{
  if (dir != NULL && inode_get_inumber(dir_get_inode(dir)) == ROOT_DIR_SECTOR)
    return true;
  else
    return false;
}

/** Returns the parent inode of DIR. */
struct inode*
dir_parent_inode(struct dir* dir)
{
  if(dir == NULL) return NULL;

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

Exercise 2.2

Maintain a separate current directory for each process.

At startup, set the **root** as **the initial process's current directory.**

When one process starts another with the **exec** system call, **the child process inherits its parent's current directory.** After that, the two processes' current directories are **independent**, so that either changing its own current directory has no effect on the other. (This is why, under Unix, the cd command is a shell built-in, not an external program.)

Update the system calls to support hierarchical file names. an absolute or relative path name may be used. Update the open system call so that it can also open directories.

Current directory

```
1  struct thread {
2      ...
3  #ifdef FILESYS
4     struct dir *dir;          /** Current directory. */
5  #endif
6      ...
7  }
8  static void init_thread (struct thread *t, const char *name, int priority) {
9      ...
10 #ifdef FILESYS
11     t->dir = NULL;
12 #endif
13     ...
14 }
15 tid_t thread_create (const char *name, int priority,
16                 thread_func *function, void *aux) {
17     ...
18 #ifdef FILESYS
19   if(thread_current()->dir)
20     t->dir = dir_reopen(thread_current()->dir);
21   else
22     t->dir = NULL;
23 #endif
24     ...
25 }
26 static void start_process(void *pcb_) {
27     ...
28 #ifdef FILESYS
29   /* Set the working directory to the root directory. */
30   if (!thread_current()->dir)
31     thread_current()->dir = dir_open_root();
32 #endif
33     ...
34 }
```

Support hierarchical file names

```
/** Returns the name of the file in PATH. */
char*
path_to_name(const char* path_name)
{
  int length = strlen(path_name);
  char path[length + 1];
  memcpy(path, path_name, length + 1);

  char *cur, *ptr, *prev = "";
  for(cur = strtok_r(path, "/", &ptr); cur != NULL;
  cur = strtok_r(NULL, "/", &ptr))
    prev = cur;

  char* name = malloc(strlen(prev) + 1);
  memcpy(name, prev, strlen(prev) + 1);
  return name;
}
/** Returns the directory of the file in PATH. */
struct dir*
path_to_dir(const char* path_name)
{
  int length = strlen(path_name);
  char path[length + 1];
  memcpy(path, path_name, length + 1);
```

```

  struct dir* dir;
  if(path[0] == '/' || !thread_current()->dir)
    dir = dir_open_root();
  else
    dir = dir_reopen(thread_current()->dir);

  char *cur, *ptr, *prev;
  prev = strtok_r(path, "/", &ptr);
  for(cur = strtok_r(NULL, "/", &ptr); cur != NULL;
    prev = cur, cur = strtok_r(NULL, "/", &ptr))
  {
    struct inode* inode;
    if(strcmp(prev, ".") == 0) continue;
    else if(strcmp(prev, "..") == 0)
    {
      inode = dir_parent_inode(dir);
      if(inode == NULL) return NULL;
    }
    else if(dir_lookup(dir, prev, &inode) == false)
      return NULL;

    if(inode_is_dir(inode))
    {
      dir_close(dir);
      dir = dir_open(inode);
    }
    else
      inode_close(inode);
  }

  return dir;
}
```

Support hierarchical file names

```
bool
filesys_create (const char *name, off_t initial_size, bool is_dir)
{
  block_sector_t inode_sector = 0;
  struct dir *dir = path_to_dir(name);
  char* file_name = path_to_name(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, is_dir)
                && 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;
}
bool
filesys_remove (const char *name)
{
  struct dir* dir = path_to_dir(name);
  char* file_name = path_to_name(name);
  bool success = dir != NULL && dir_remove (dir, file_name);
  dir_close (dir);
  free(file_name);

  return success;
}
```

Support hierarchical file names

```
1  struct file *
2  filesys_open (const char *name)
3  {
4    if(strlen(name) == 0)
5      return NULL;
6
7    struct dir* dir = path_to_dir(name);
8    char* file_name = path_to_name(name);
9    struct inode *inode = NULL;
10
11   if (dir != NULL)
12   {
13     /* root, current dir */
14     if (dir_is_root(dir) && strlen(file_name) == 0)
15     {
16       free(file_name);
17       return (struct file *) dir;
18     }
19
20     else if (dir_lookup(dir, file_name, &inode) == false)
21     {
22       free(file_name);
23       dir_close(dir);
24       return NULL;
25     }
26   }
27
28   free(file_name);
29   dir_close(dir);
30
31   if (inode_is_dir(inode))
32     return (struct file *) dir_open(inode);
33   return file_open(inode);
34 }
```

Exercise 2.3

Implement the following new system calls:

System Call: bool chdir (const char *dir)

System Call: bool mkdir (const char *dir)

System Call: bool readdir (int fd, char *name)

System Call: bool isdir (int fd)

System Call: int inumber (int fd)

chdir

```
1   bool
2   filesys_chdir(const char* path)
3   {
4     struct dir* dir = path_to_dir(path);
5     char* name = path_to_name(path);
6     struct inode *inode = NULL;
7
8     if(dir == NULL)
9     {
10      free(name);
11      return false;
12    }
13    /* special case: go to parent dir */
14    else if(strcmp(name, "..") == 0)
15    {
16      inode = dir_parent_inode(dir);
17      if(inode == NULL)
18      {
19        free(name);
20        return false;
21      }
22    }
23    /* special case: current dir */
24    else if(strcmp(name, ".") == 0 ||
25    (strlen(name) == 0 && dir_is_root(dir)))
26    {
27      thread_current()->dir = dir;
28      free(name);
29      return true;
30    }
31    else dir_lookup(dir, name, &inode);
```

```
1     dir_close(dir);
2
3     /* now open up target dir */
4     dir = dir_open(inode);
5
6     if(dir == NULL)
7     {
8       free(name);
9       return false;
10    }
11    else
12    {
13      dir_close(thread_current()->dir);
14      thread_current()->dir = dir;
15      free(name);
16      return true;
17    }
18  }
```

Syscalls

```
bool
sys_chdir(char* path)
{
    check_user((const uint8_t*) path);
    bool success = filesys_chdir(path);
    return success;
}

bool
sys_mkdir(char* path)
{
    bool success = filesys_create(path, 0, true);
    return success;
}

bool
sys_readdir(int fd, char* path)
{
    ASSERT (fd >= 0);

    struct file* file = find_file_desc(
    thread_current(), fd)->file;
    if (file == NULL) return false;

    struct inode* inode = file_get_inode(file);
    if(inode == NULL) return false;
    if(!inode_is_dir(inode)) return false;

    struct dir* dir = (struct dir*) file;
    if(!dir_readdir(dir, path)) return false;

    return true;
}
```

```
bool
sys_isdir(int fd)
{
    ASSERT (fd >= 0);

    struct file* file = find_file_desc
    (thread_current(), fd)->file;
    if (file == NULL) return false;

    struct inode* inode = file_get_inode(file);
    if(inode == NULL) return false;
    if(!inode_is_dir(inode)) return false;

    return true;
}

int
sys_inumber(int fd)
{
    ASSERT (fd >= 0);

    struct file* file = find_file_desc
    (thread_current(), fd)->file;
    if (file == NULL) return -1;

    struct inode* inode = file_get_inode(file);
    if(inode == NULL) return -1;

    block_sector_t inumber =
    inode_get_inumber(inode);
    return inumber;
}
```

Task 4: Synchronization

Exercise 4.1

The provided file system requires external synchronization, that is, callers must ensure that only one thread can be running in the file system code at once.

Your submission must adopt a **finer-grained** synchronization strategy that **does not require external synchronization**.

To the extent possible, operations on independent entities should be independent, so that they do not need to wait on each other.

Support finer-grained synchronization of file system.

Operations on different cache blocks must be independent. Multiple processes must be able to access a single file at once. On the other hand, extending a file and writing data into the new section must be atomic.

Operations on different directories should take place concurrently. Operations on the same directory may wait for one another.

Inode

```
1  struct inode {
2      ...
3       off_t length;                    /**< File size in bytes. */
4      off_t read_length;               /** File size in bytes. */
5      ...
6      struct lock lock;                /** Lock for inode. */
7  }
```

Inode lock

```
1   off_t
2   inode_write_at (struct inode *inode, const void *buffer_, off_t size,
3                   off_t offset)
4                    /* beyond EOF, need extend */
5   {
6       ...
7     if(offset + size > inode_length(inode))
8     {
9       // no sync required for dirs
10      if(!inode->is_dir)
11        lock_acquire(&inode->lock);
12
13      inode->length = inode_grow(inode, offset + size);
14
15      if(!inode->is_dir)
16        lock_release(&inode->lock);
17    }
18      ...
19      // block_write
20      inode->read_length = inode_length(inode);
21  }
```

dir lock

```
1   bool dir_lookup (const struct dir *dir, const char *name, struct inode **inode)
2   bool dir_add (struct dir *dir, const char *name, block_sector_t inode_sector)
3   bool dir_remove (struct dir *dir, const char *name)
4   bool dir_readdir (struct dir *dir, char name[NAME_MAX + 1])
```

cache lock

```
1  struct lock cache_lock;                  /**< cache lock */
2  int access_cache_entry(block_sector_t disk_sector, bool dirty) {
3    lock_acquire(&cache_lock);
4    int idx = get_cache_entry(disk_sector);
5    if(idx == -1)
6      idx = replace_cache_entry(disk_sector, dirty);
7    else {
8      cache_array[idx].open_cnt++;
9      cache_array[idx].accessed = true;
10     cache_array[idx].dirty |= dirty;
11   }
12   lock_release(&cache_lock);
13   return idx;
14 }
15
16 // func_read_ahead(void *aux)
17
18 void write_back(bool clear) {
19     int i;
20     lock_acquire(&cache_lock);
21     for(i = 0; i < CACHE_MAX_SIZE; i++)
22     {
23         if(cache_array[i].dirty == true)
24         {
25             block_write(fs_device, cache_array[i].disk_sector, &cache_array[i].block);
26             cache_array[i].dirty = false;
27         }
28
29         /* clear cache line (filesys done) */
30         if(clear)
31           init_entry(i);
32     }
33
34     lock_release(&cache_lock);
35 }
```

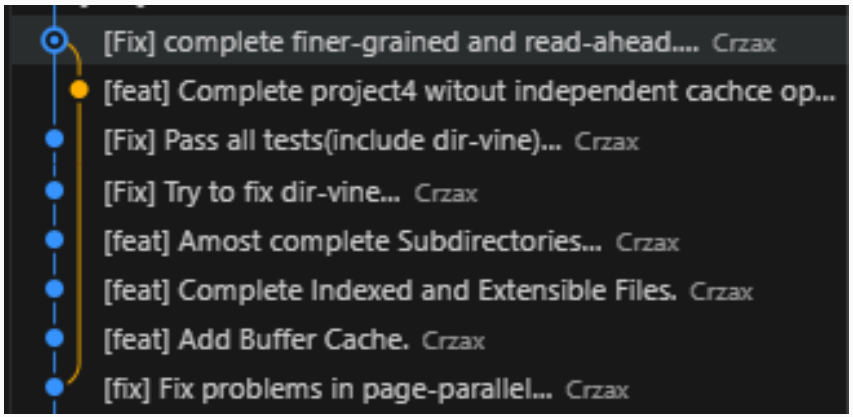
Results

A terrible test-vine

This code tests a file system by creating a deep chain of directories and files (e.g., /dir0/dir1/dir2/...) until the disk is full, then deletes most of them, keeping only the top 10 levels. It’s meant to stress-test file system limits and behavior.

Coarser-grained -> Finer-grained

- sector number undefined.
- dead lock.
- pt-write-code2(open unexisted file) can’t exit.



You should think about synchronization throughout.

a strange thing

To solve sector number undefined:

-

```
bool
get_free_map_empty_size(void)
{
  return bitmap_count(free_map, 0, bitmap_size(free_map), false);
}
```

  - should defined in free-map.h but I define it in bitmap.h -> pass
  - Fix warnings -> define in free-map.h PANIC when pintos inits.
  - delete it -> pass

Two enhancement

- **page-parallel** runs four ”child-linear” processes simultaneously and waits for each to finish, checking that they return the value 0x42. It tests the system’s ability to handle multiple child processes and process synchronization.
  Frame exists but supt can’t be found by frame ? -> only evict frames belonging to itself.
- Cache lock -> each cache lock (Finer-grained lock)

Results

```
pass tests/filesys/extended/grow-seq-lg-persistence
pass tests/filesys/extended/grow-seq-sm-persistence
pass tests/filesys/extended/grow-sparse-persistence
pass tests/filesys/extended/grow-tell-persistence
pass tests/filesys/extended/grow-two-files-persistence
pass tests/filesys/extended/syn-rw-persistence
All 159 tests passed.
```

```
src/userprog/syscall.c  | 281 ++++++++++++++++++++++++++++++++++++++++++++++++
src/userprog/syscall.h  |  12 ++++++-
src/vm/page.c           |   3 +-
src/vm/page.h           |   2 +-
28 files changed, 1296 insertions(+), 206 deletions(-)
```

Reference:30 files changed, 2721 insertions(+), 286 deletions(-)

Thank you!
Questions?