Operating System Labs

Yuanbin Wu
CS@ECNU

Operating System Labs

- Project 4 (multi-thread & lock):
  - Due: 10 Dec
    - Code & experiment report
- 18 Dec.
  - Oral test (proj 4): 9:30am
  - Lectures: Q&A
- Project 5:
  - Due: 31 Dec
  - Oral test: 8 Jan, 9:30am

Operating System Labs

- Overview of file system
  - File system API
  - File system implementation
- Project 5

File System API

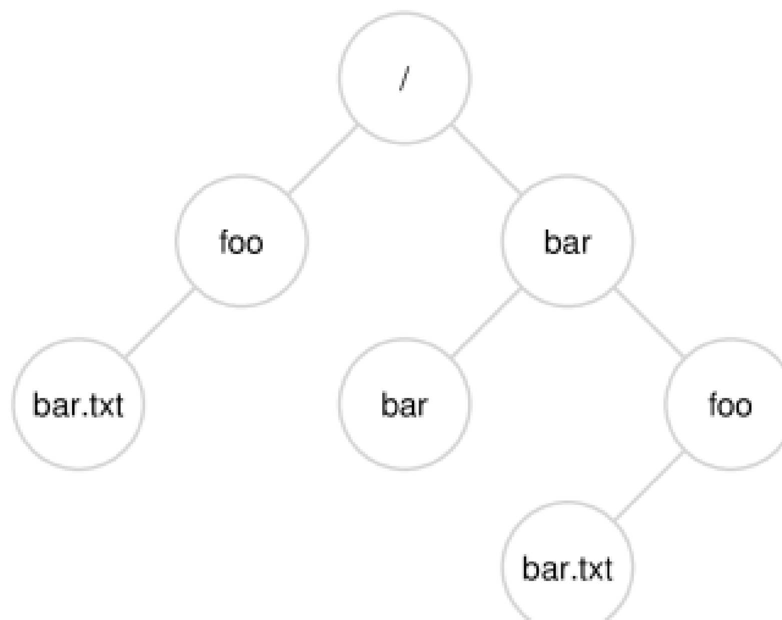
- Previous
  - CPU: process, thread
  - Memory: address space, virtual memory management
- Now
  - File system: persistent storage

File System API

- Regular File
  - File name: user readable
  - **inode number**: low-level file name
  - Contents: figure, text, video
- Directory
  - Directory name: user readable
  - **inode number**: low-level directory name
  - Contents: file and sub-directories

File System API

- Directories
  - Content :
    - tuples: (user-readable name, inode number)
  - Directory tree



File System API

- File System APIs
  - Basic I/O interface (lecture2)
    - File descriptor
    - open, read, write, close, lseek
    - buffer
  - strace

```
% strace cat foo
```

  - Other APIs

File System API

- Renaming files

```
% strace mv foo bar
```

```
#include <stdio.h>

int rename(char *old, char *new)
```

File System API

- Renaming files
  - Atomic
    - the system can crash during renaming
    - either old name or new name

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

File System API

- Get information about files
  - **meta data**

```
#include <sys/stat.h>

int fstat(int fildes, struct stat *buf);
```

```
% stat bar
```

```
struct stat {
  dev_t       st_dev;         // ID of device containing file
  ino_t       st_ino;         // inode number
  mode_t      st_mode;        // protection
  nlink_t     st_nlink;       // number of hard links
  uid_t       st_uid;         // user ID of owner
  gid_t       st_gid;         // group ID of owner
  dev_t       st_rdev;        // device ID (if special file)
  offset_t    st_size;        // total size, in bytes
  blksize_t   st_blksize;     // blocksize for filesystem I/O
  blkcnt_t    st_blocks;      // number of blocks allocated
  time_t      st_atime;       // time of last access
  time_t      st_mtime;       // time of last modification
  time_t      st_ctime;       // time of last status change
};
```

File System API

- Removing file

```
% strace rm bar
```

```
#include <unistd.h>

int unlink(const char *pathname);
```

File System API

- Making Directories

```
% strace mkdir foo
```

```
#include <unistd.h>

int mkdir(const char *pathname);
```

File System API

- Reading Directories

```
% strace ls
```

```
int getdents(unsigned int fd,
                    struct linux_dirent *dirp,
                    unsigned int count);
// no glibc wrappers (with the same name) for it
```

File System API

- Reading Directories
  - Glibc: DIR stream (recall the FILE stream)

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
int closedir(DIR *dirp);
```

```
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

File System API

- Reading Directories

```
struct dirent {
  char              d_name[256];    //filename
  ino_t             d_ino;          //inode number
  off_t             d_off;          //opaque value
  unsigned short  d_reclen;       //length of this record
  unsigned char  d_type;          //type of file
};
```

File System API

- Reading Directories
  - A simple ls

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%d %s\n", (int) d->d_ino, d->d_name);
    }
    closedir(dp);
    return 0;
}
```

File System API

- Remove Directories

```
% strace rmdir
```

```
int rmdir(const char* name);
// remove empty directory
```

File System API

- Hard links
  - Link(): create a new way to refer the same file

```
#include <unistd.h>

int link(const char* old, const char* new);
```

```
% cat file
% ln file file1
% cat file1
% ls -i file file1
```

File System API

- Hard links
  - unlink(): the reverse of link()

```
#include <unistd.h>

int unlink(const char* filename);
```

```
% rm file
% cat file1
% ls -i file1
```

File System API

- Hard links
  - A field in inode structure: reference count

```
% ln file file1
% stat file
% ln file1 file2
% stat file
% ln file2 file3
% stat file
% rm file1
% stat file
```

File System API

- Symbolic links
  - Limitations of hard links
    - Can not link directories → cycles are not allowed
    - Can not hard link across partitions
  - Symbolic links
    - A new file type (regular file, directory, symbolic link)
    - Different from the original file
    - The content of a symbolic links
      - Pathname of the linked-to file

File System API

- Symbolic links

```
% ln -s file file1
% stat file
% stat file1
% ls -al
% rm file
% cat file1

% echo hello > verylongfile
% ln -s verylongfile file
% ls -al
```

File System API

- Making and Mounting file systems
  - mkfs:
    - Input: a partition and a fs type
    - Output: a file system
  - Mount:
    - Put the new file system in the current directory tree

File System API

- Summary
  - File, directory, symbolic link
  - open(), read(), write(), lseek(), close()
  - link(), unlink()
  - readdir(), mkdir()

File System Implementation

- A very simple file system (vsfs)
  - pure software (different from process/vm)
- The way to think about a file system
  - Data structures
  - Access methods

File System Implementation

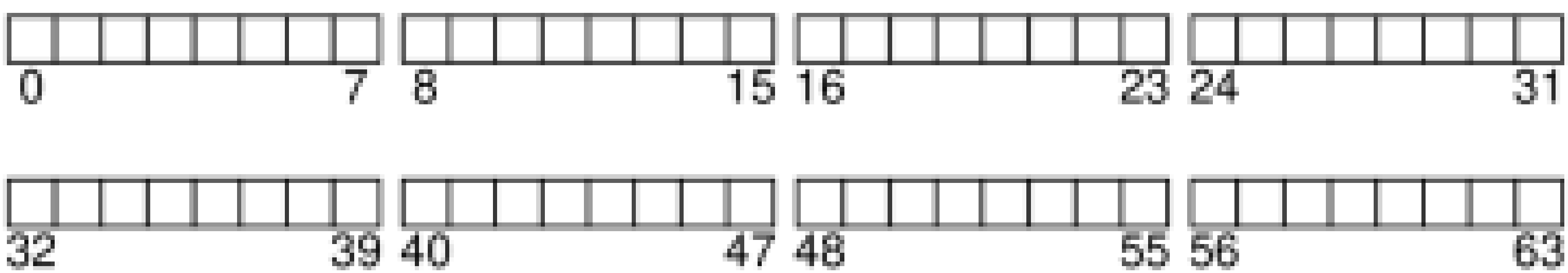
- The way to think about a file system
  - Data structures
    - How to organize files?
  - Things to manage
    - Files
    - Meta data of files (inode)
    - Free space

File System Implementation

- The way to think about a file system
  - Access methods
    - open(), read(), write()
    - opendir(), readdir()
    - link(), unlink()

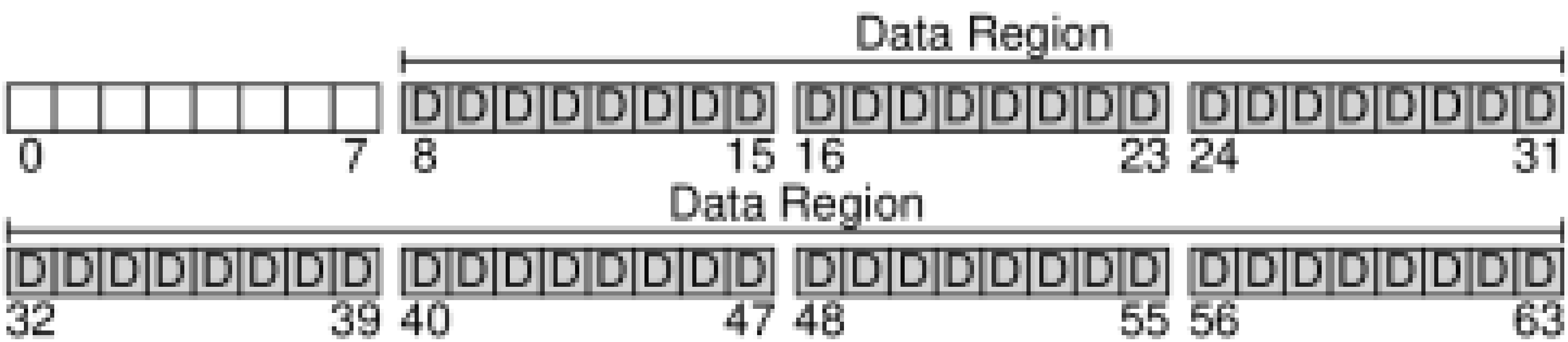
File System Implementation

- Data structure: overall organization
  - Block
    - A file system manipulate blocks (not byte)
    - Commonly used: 4KB
  - We have a disk with 64 blocks (256KB)



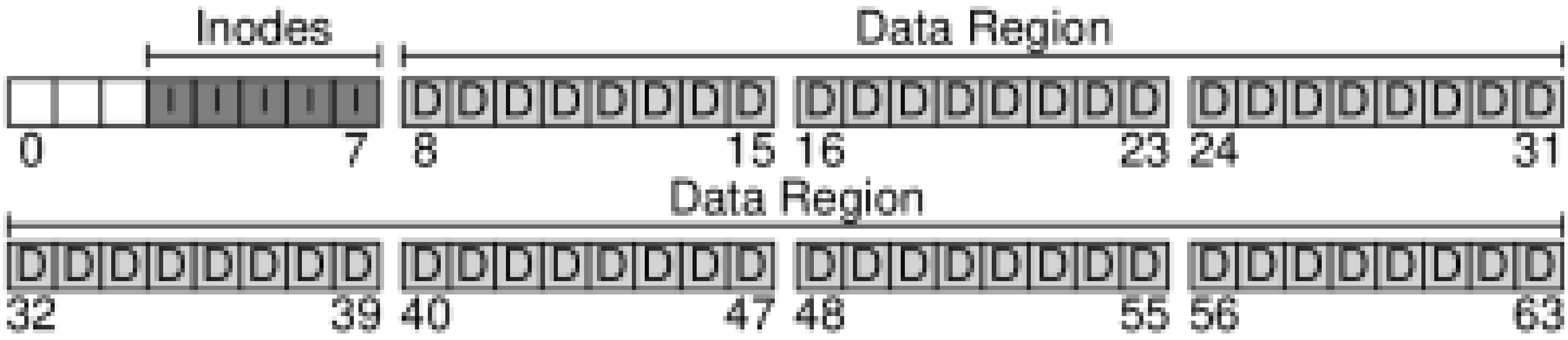
File System Implementation

- Data structure: overall organization
  - Data region
  - We have 56 blocks for storing data (D)



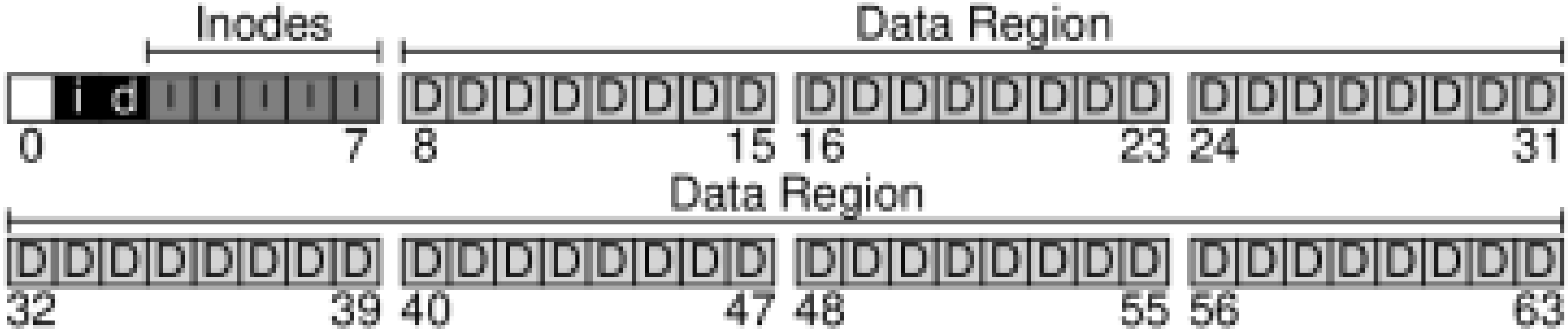
File System Implementation

- Data structure: overall organization
  - Meta data: information about files
    - size, reference count, protection, access time
    - Inode
  - We have 5 blocks for inodes (I)
    - Assume each inode 256B (16 inodes per block)
    - We can handle 80 files



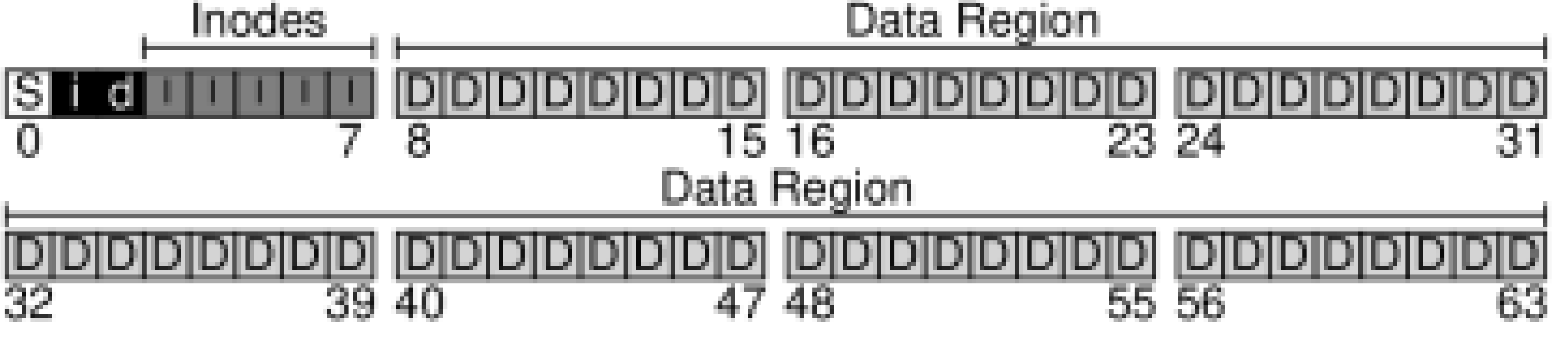
File System Implementation

- Data structure: overall organization
  - Allocation structures (free list)
    - Which blocks are allocated?
  - We will use the “bitmap”
    - Each bit indicates whether a block is used
    - one for data region (d), one for inode table (i)
    - What are sizes of the bitmaps?



File System Implementation

- Data structure: overall organization
  - Superblock
    - Metadata of the whole file system
    - How many inodes and data blocks?
    - The start of inode table/data region
  - We use the left 1 block as superblock (S)

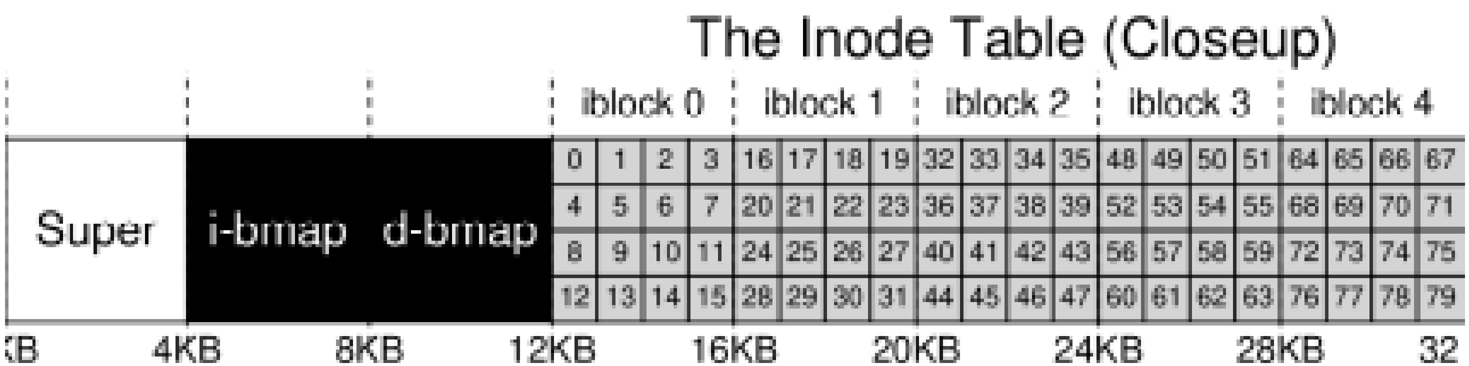


File System Implementation

- Summary
  - Data structure: overall organization
    - Data region
    - Inode table
    - Bitmaps
    - Superblock

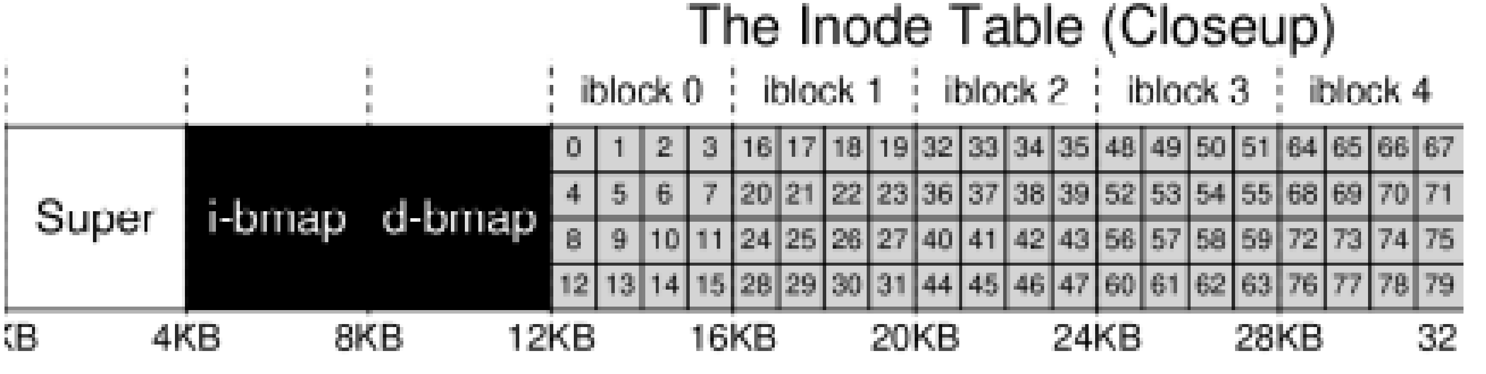
File System Implementation

- Data structure: the inode
  - Inode number:
    - its index in the inode table
    - Low-level name of the file



File System Implementation

- Data structure: the inode
  - Locating an inode through inode number
  - Example: file with inode number 32
    - Offset: 32*256 + 12K = 8K + 12K = 20K
    - For a read from disk (only read *sectors*)
      - Sector size: 512
    - Finally the disk will read sector: 40 (20K/512)



File System Implementation

- Data structure: the inode
  - An inode contains
    - The data blocks
    - Type (file/directory/symbolic link)
    - Reference count (link/unlink)
    - Size (#blocks)
    - Protection
    - Time information

- The Ext2 inode

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists
4	faddr	an unsupported field
12	i_osd2	another OS-dependent field

File System Implementation

- Data structure: the inode
  - An inode contains
    - The data blocks
    - Type (file/directory/symbolic link)
    - Reference count (link/unlink)
    - Size (#blocks)
    - Protection
    - Time information

How to organize data blocks in inodes?

File System Implementation

- Data structure: the inode
  - How to locate data blocks
    - direct pointers in inode structure
    - Can not hold large files
  - The multi-level Index
    - Indirect pointers
    - Point to data blocks which contain direct pointers

File System Implementation

- Data structure: the inode
  - Example of multi-level Index
    - An inode contains 12 direct pointers
    - 1 indirect pointers
      - Block size: 4K
      - Block number: an int (4 Bytes)
      - #direct pointers per block: 1K
    - #direct pointers: 12 + 1K
    - File size: (12 + 1K)*4K = 4144KB

File System Implementation

- Data structure: the inode
  - Double indirect pointer
    - # direct pointers: 1024 * 1024 = 1M
    - File size: $(12 + 1024 + 1024^2) * 4K \approx 4G$
  - Triple indirect pointer
  - An imbalanced tree
    - Most files are small

File System Implementation

- Summary
  - Data structure: the inode
    - Inode number
    - Locating an inode
    - Contents of an inode
    - How to index data blocks

File System Implementation

- Data structure: directory
  - Again: a directory is a file!
    - An inode
    - Data blocks
  - The contents of its data blocks
    - List of (entry name, inode number)
    - Other data structures: B-trees, hash tables

File System Implementation

- Data structure: directory
  - Example
    - Directory: dir(5)
    - Files: dir/foo(12), dir/bar(13), dir/foobar(24)

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

  - Delete a file in the directory
    - Can we reuse the entries?

File System Implementation

- Data structure: free space management
  - Bitmaps
    - Inode table
    - Data region
  - Other data structures: B-tree
  - Pre-allocation

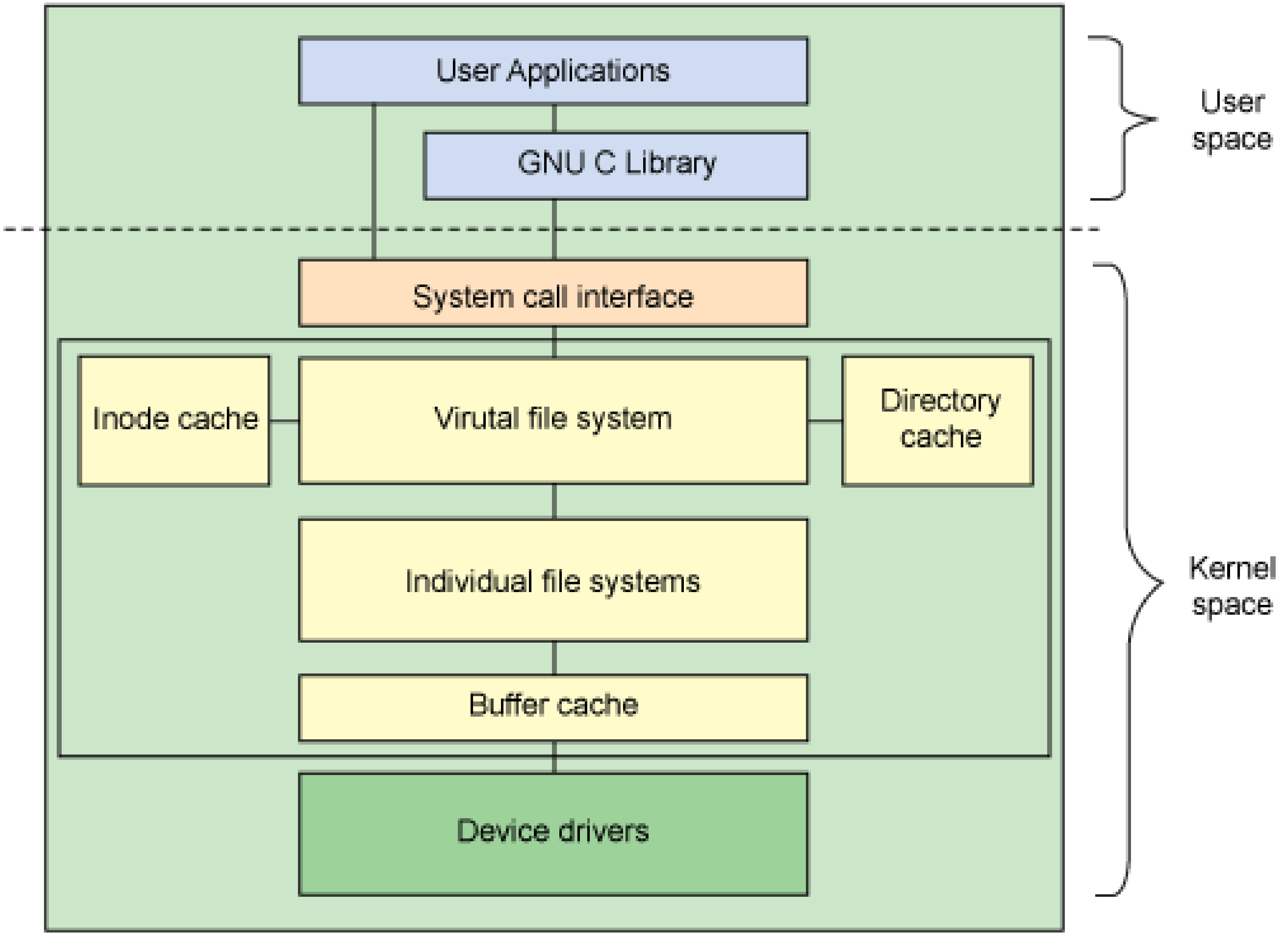
File System Implementation

- Summary
  - Data structures for implementing an fs
    - Overall organization
    - Inode
    - Directory
    - Free list management

File System Implementation

- Different types of fs
  - ext2, ext3, ext4, proc, cgroup
  - The concept: *virtual file system*
    - Provide unified view of different file systems
  - For each file system
    - Data structures: Inode, dentry, superblock
    - Operations:
      - Superblock operations: alloc_inode(), distroy_inode(), read_inode(), write_inode();
      - inode operations: create_inode(), lookup(), mkdir(), rename()
      - File operations: read(), write(), open(); close(); lseek()

File System Implementation



File System Implementation

- Access methods
  - read(), write()
  - readdir()
  - link(), unlink()

File System Implementation

- Access methods: read a file
  - open("/foo/bar", O RDONLY); and read it

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			**read**							
						read				
				read						
							read			
					read					
read()					**read**					
								read		
					write					
read()					**read**					
									read	
					write					
read()					**read**					
										read
					write					

File System Implementation

- Access methods: create and write a file
  - open("/foo/bar", O RDONLY); and write it

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)			read							
						read				
				read						
							read			
		read								
		write								
							write			
					read					
					write					
				write						
write()					read					
	read									
	write									
								write		
					write					
write()					read					
	read									
	write									
									write	
					write					
write()					read					
	read									
	write									
										write
					write					

There are more operations!

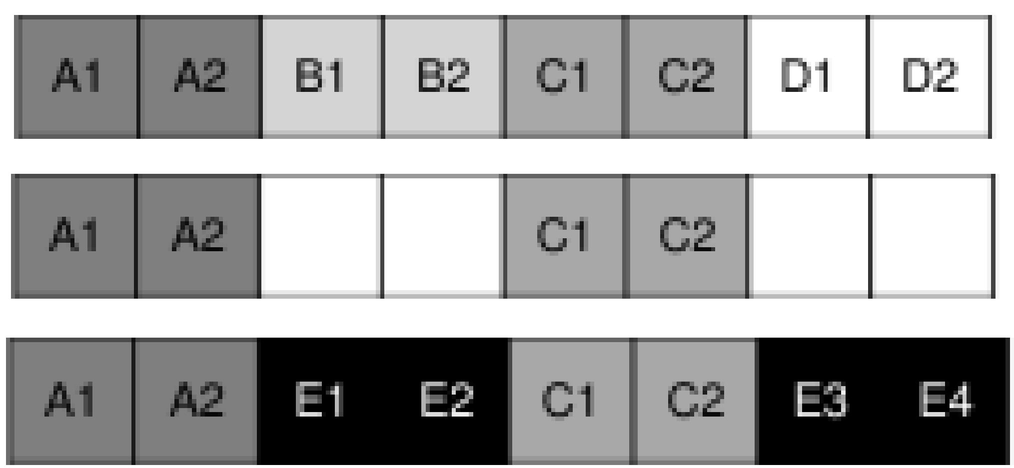
Can we change the order?

File System Implementation

- Access methods: how to speed up?
  - Cache
  - Buffering

File System Implementation

- Summary
  - The way to think about a file system
    - Data structures
    - Access methods
- Problems
  - Locality is not preserved



File System Implementation

- Project 5
  - defragmentation