ECS 150 - Project 4

Prof. Joël Porquet

UC Davis - Spring Quarter 2019

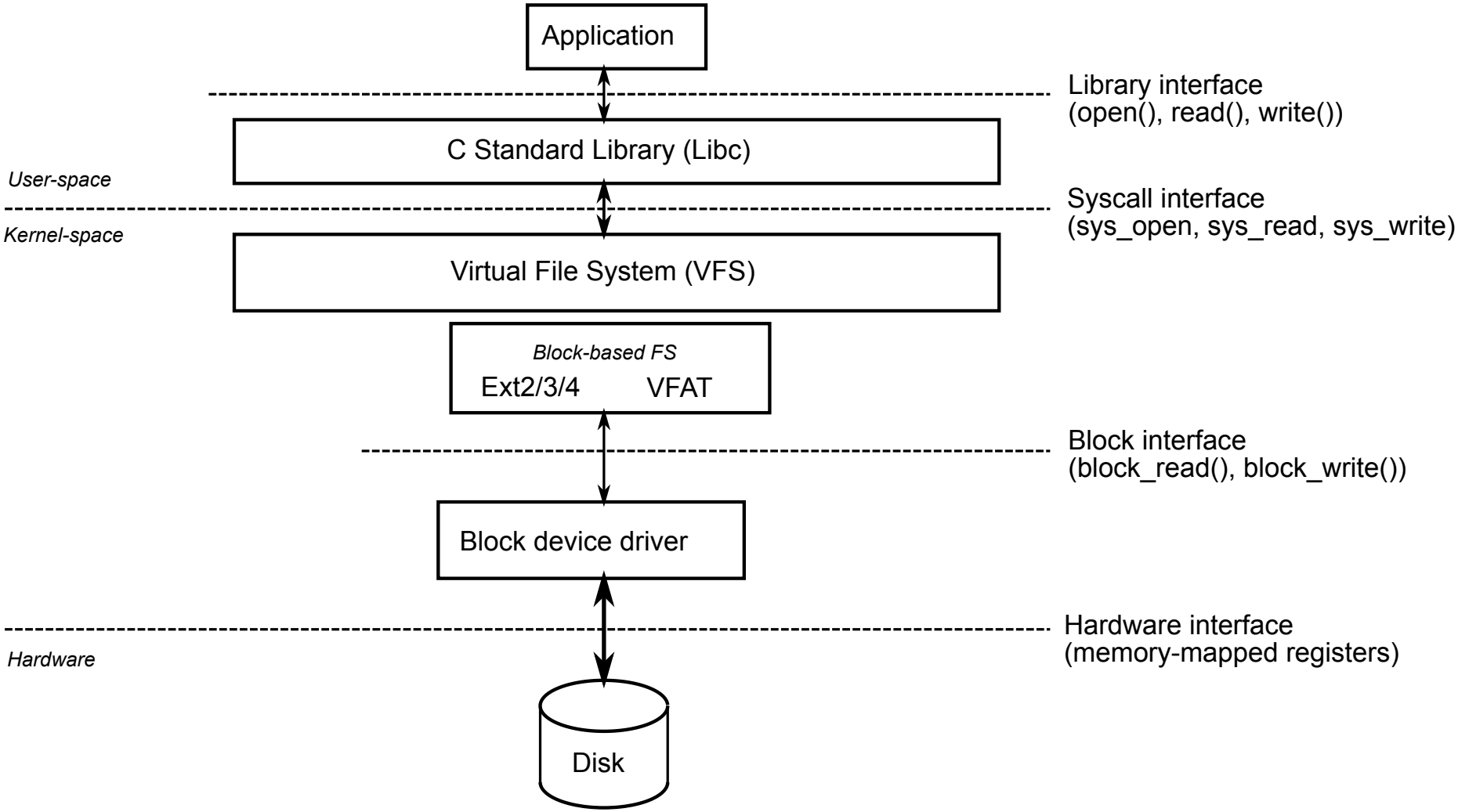
Goal

The goal of this project is to implement the support for a very simple *file system*: **ECS150-FS**.

Applications will have the possibility to read/write files from/to this file system.

```
int fs_mount(const char *diskname);            /* Mounting the file system */
int fs_umount(void);
int fs_info(void);

int fs_create(const char *filename);           /* Creating files */
int fs_delete(const char *filename);
int fs_ls(void);

int fs_open(const char *filename);             /* Opening files */
int fs_close(int fd);
int fs_stat(int fd);
int fs_lseek(int fd, size_t offset);

int fs_write(int fd, void *buf, size_t count); /* Modifying files */
int fs_read(int fd, void *buf, size_t count);
```

Big picture: reality



Problems:

- The vast majority the file system management is in kernel mode!
- And we need a physical disk...

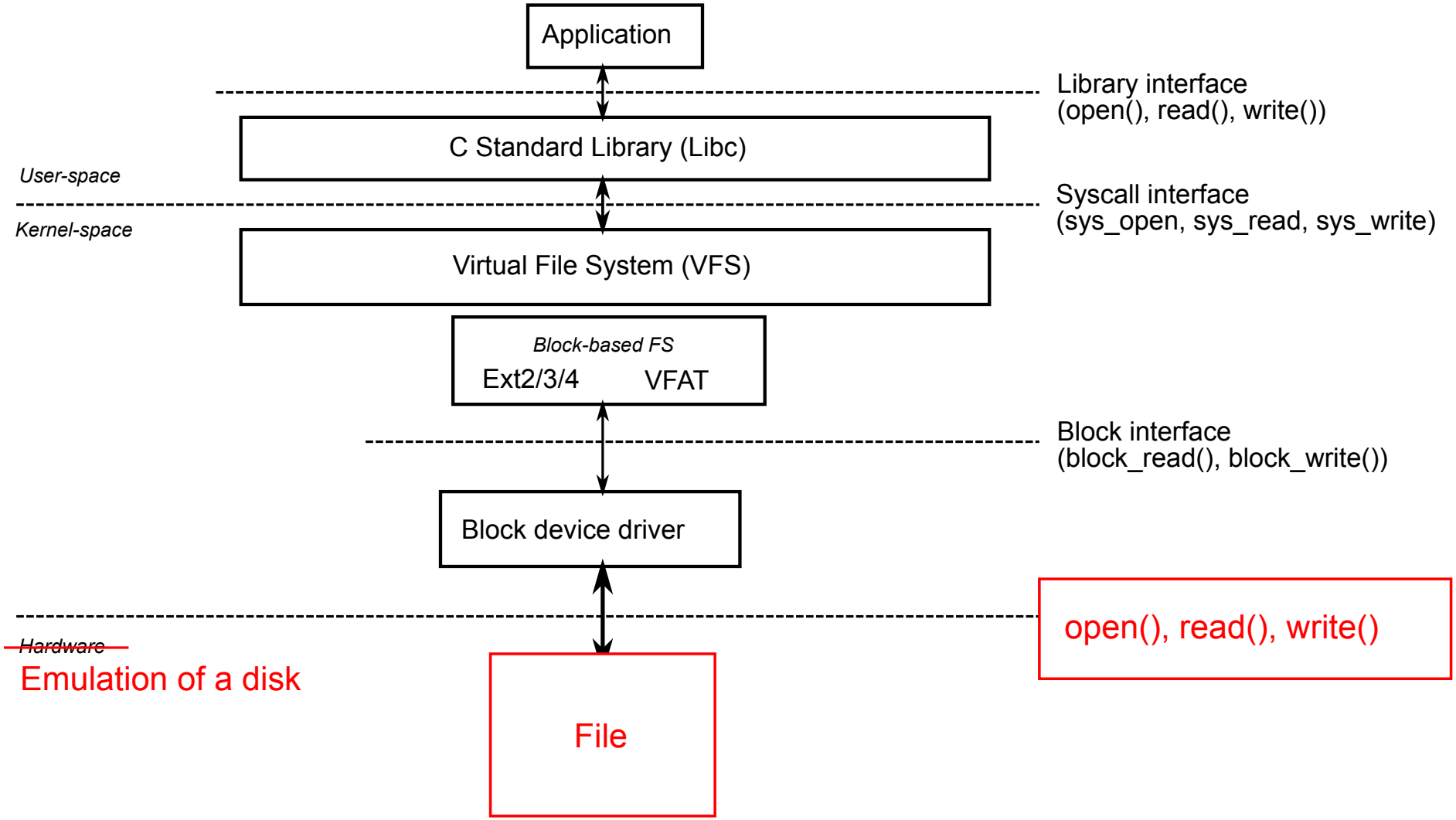
Emulating a disk with a file

A disk, or a partition on a disk, merely represents contiguous binary data storage.

How can we easily emulate any size of contiguous data?... With a file!

```
$ dd if=/dev/zero of=emulated_disk_space bs=4K count=8192

$ ls -l emulated_disk_space
-rw-r--r-- 1 joel joel 32M 2017-03-01 13:52 emulated_disk_space

$ xxd emulated_disk_space
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
...
01ffff0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

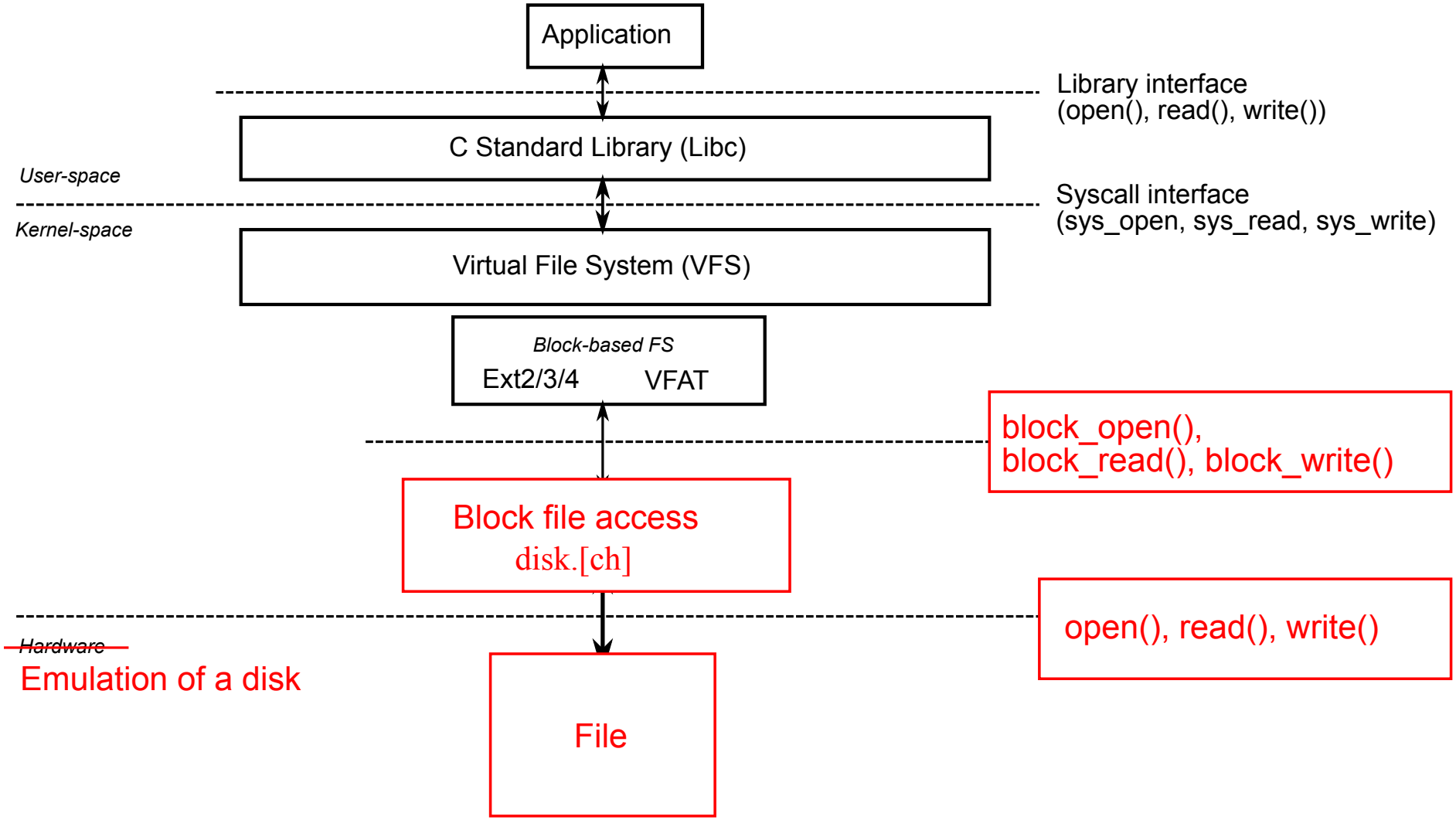
Replacing the disk



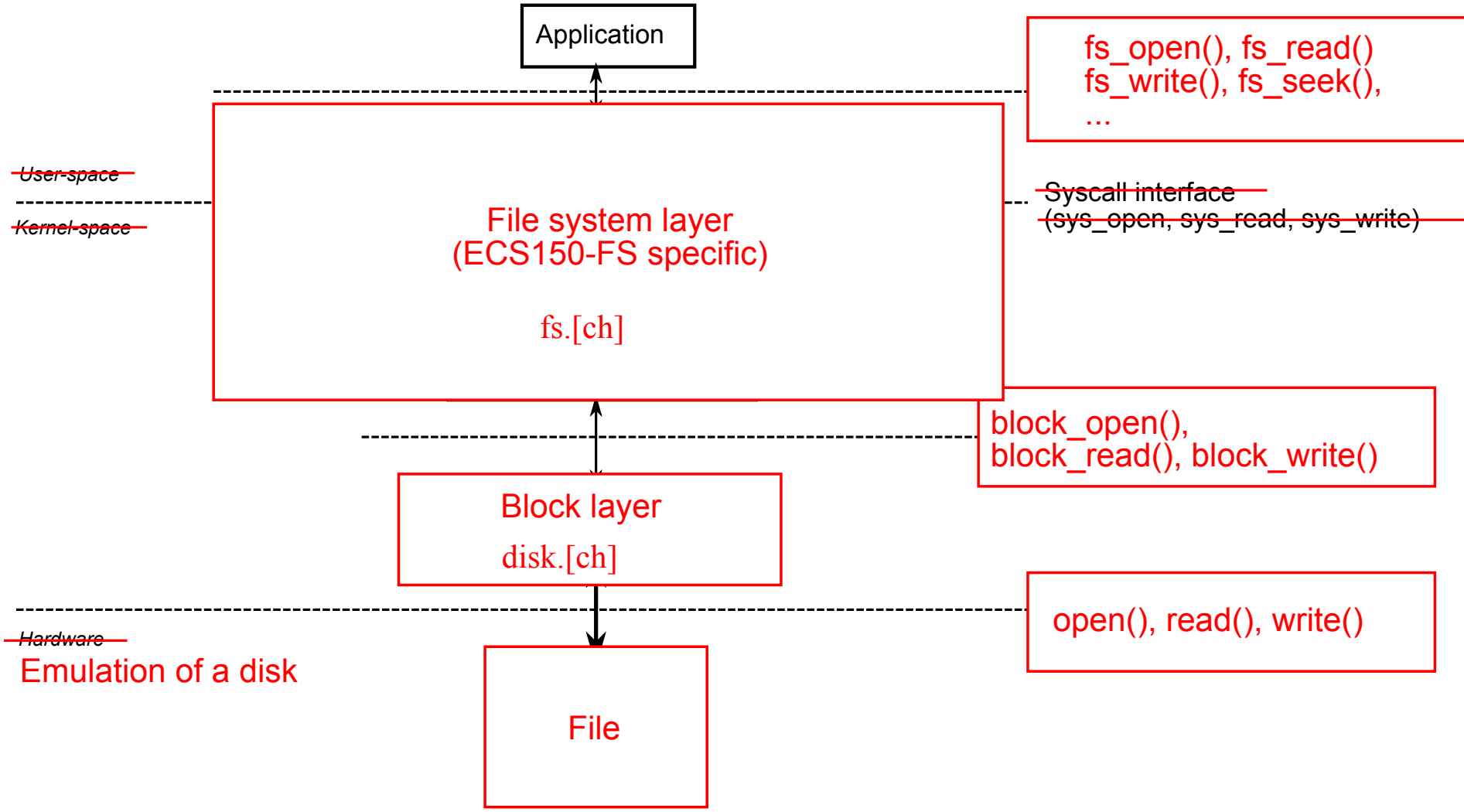
Accessing a file by blocks

```c
#define BLOCK_SIZE 4096

int fd;

int block_open(char *disk_filename)
{
    fd = open(disk_filename, O_RDWR);
}

int block_read(size_t block_nr, void *buf)
{
    lseek(fd, block_nr * BLOCK_SIZE);
    read(fd, buf, BLOCK_SIZE);
}

int block_write(size_t block_nr, void *buf)
{
    lseek(fd, block_nr * BLOCK_SIZE);
    write(fd, buf, BLOCK_SIZE);
}
```

Replacing the block device driver



Replacing the libc/vfs/fs drivers



- *That's the project!*

ECS150-FS

Layout

Each block is 4096 bytes.

Example with file system embedding 8192 data blocks:

```
$ ./fs_make.x disk.fs 8192
Creating virtual disk 'disk.fs' with '8192' data blocks
```

- Amount of data blocks: 8192
- Number of blocks for FAT: (8192 * 2) / 4096 = 4
- Total amount of blocks: 1 + 4 + 1 + 8192 = 8198
- Root directory block index: 5
- Data block start index: 6

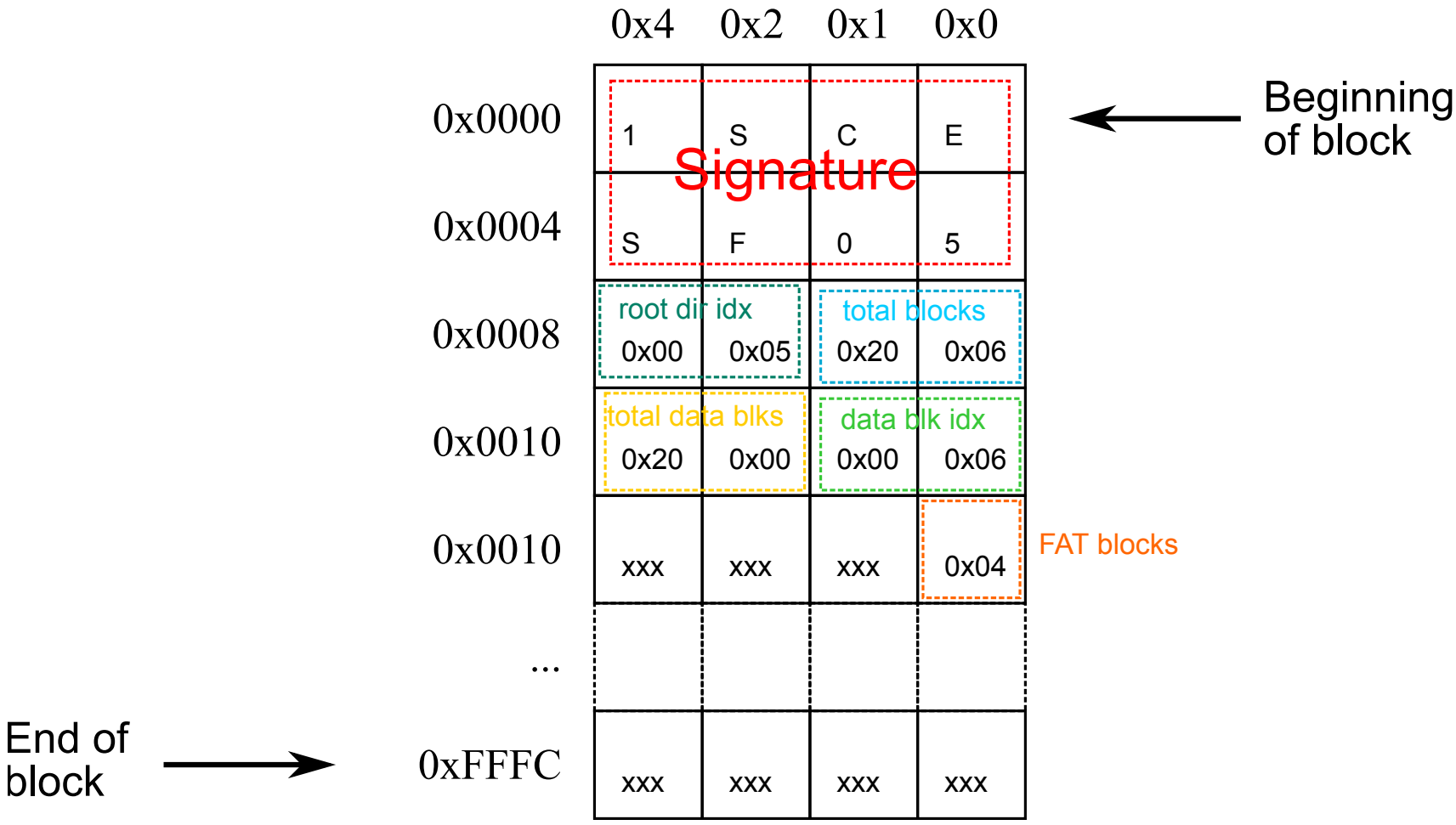
ECS150-FS

Superblock: high-level layout

Offset	Length (bytes)	Description
0x00	8	Signature (must be equal to "ECS150FS")
0x08	2	Total amount of blocks of virtual disk
0x0A	2	Root directory block index
0x0C	2	Data block start index
0x0E	2	Amount of data blocks
0x10	1	Number of blocks for FAT
0x11	4079	Unused/Padding

ECS150-FS

Superblock: at byte level



ECS150-FS

Superblock: C data structure

```
struct superblock{
    ???
};
```

Key points:

- The integer types must match exactly those of the specification
- It must represent the entirety of the block, padding included!

Digression

Integer types

- Is `char` always 8 bits?
- Is `short int` always 16 bits?
- Is `int` always 32 bits?
- Etc.

Type	Specification
char	"Smallest addressable unit of the machine that can contain basic character set"
short	"Capable of containing *at least* the [−32767, +32767] range; thus, it is *at least* 16 bits in size."
int	"Capable of containing *at least* the [−32767, +32767] range; thus, it is *at least* 16 bits in size."
long	"Capable of containing *at least* the [−2147483647, +2147483647] range; thus, it is *at least* 32 bits in size."

How to guarantee a certain size then?

Digression

Integer types

Use integer types that have exact widths (since C99):

```
#include <stdint.h>

int8_t
int16_t
int32_t

uint8_t
uint16_t
uint32_t
```

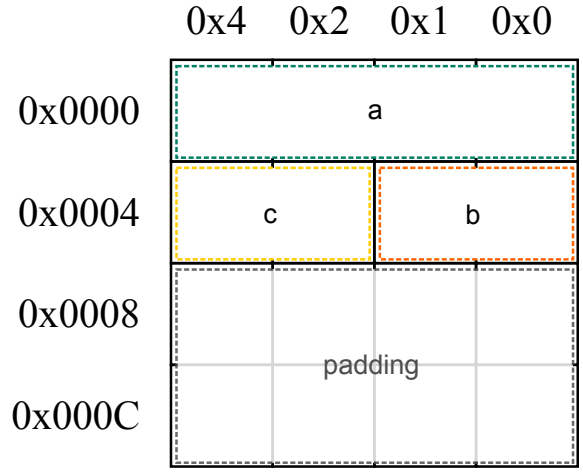
Digression

Reading data structures from a file (or buffer)

Reading data from a file (or whatever blob of data), for which I know the layout. It can easily be type-casted into a structure instance.

```
struct mystruct
{
  int32_t   a;
  int16_t   b;
  int16_t   c;
  int32_t   padding[2];
};
```

```
fd = open("file", O_RDWR);
...
char* buf[16];
read(bd, buf, 16);

struct mystruct *s = buf;

printf("s->a = %d\n", s->a);
s->a = 0;

write(bd, buf, 16);
```




```
/* or simply */
struct mystruct obj;
read(bd, &obj, sizeof(obj));

printf("obj.a = %d\n", obj.a);
obj.a = 0;

write(bd, &obj, sizeof(obj));
```

ECS150-FS

Layout

FAT

- Big array of 16-bit entries: linked-list of data blocks composing a file
- Three possible values for each entry:
  - `0`: corresponding data block is available
  - `FAT_EOC`: last data block of a file
  - `!=0 && !=FAT_EOC`: index of next data block

ECS150-FS

Layout

Root directory

1 block, 32-byte entry per file: 128 entries total

Offset	Length (bytes)	Description
0x00	16	Filename (including NULL character)
0x10	4	Size of the file (in bytes)
0x14	2	Index of the first data block
0x16	10	Unused/Padding

ECS150-FS

Example: big file, small file, empty file

Root directory

<"test1", 22000, 2>,
<"test2", 5000, 1>,
<"test3", 0, FAT_EOC>
...

FAT

0	
1	4
2	3
3	5
4	FAT_EOC
5	6
6	7
7	8
8	FAT_EOC
9	0
10	0

Data blocks

0	
1	(test2, block #0)
2	(test1, block #0)
3	(test1, block #1)
4	(test2, block #1)
5	(test1, block #2)
6	(test1, block #3)
7	(test1, block #4)
8	(test1, block #5)
9	
10	

Implementation

Phase 1: Volume mounting

- `fs_mount()`:
  - Open the virtual disk
  - Read the metadata (superblock, fat, root directory)
- `fs_unmount()`:
  - Close virtual disk (make sure that virtual disk is up-to-date)
- `fs_info()`:
  - Show information about volume

Implementation

Phase 2: File creation/deletion

- `fs_create()`:
  - Create a new file
  - Initially, size is 0 and pointer to first data block is `FAT_EOC`
- `fs_delete()`:
  - Delete an existing file
  - Don't forget to free allocated data blocks
- `fs_ls()`:
  - List all the existing files

Implementation

Phase 3: File descriptor operations

- `fs_open()`:
  - Initialize and return file descriptor
  - 32 file descriptors max
  - Can open same file multiple times
  - Contains file's offset (initially 0)
- `fs_close()`:
  - Close file descriptor
- `fs_seek()`:
  - Move file's offset
- `fs_stat()`:
  - Return file's size

None of these functions should change the file system...

Implementation

Phase 4: File reading/writing

Most complicated phase: might take as much time as all the previous phases combined

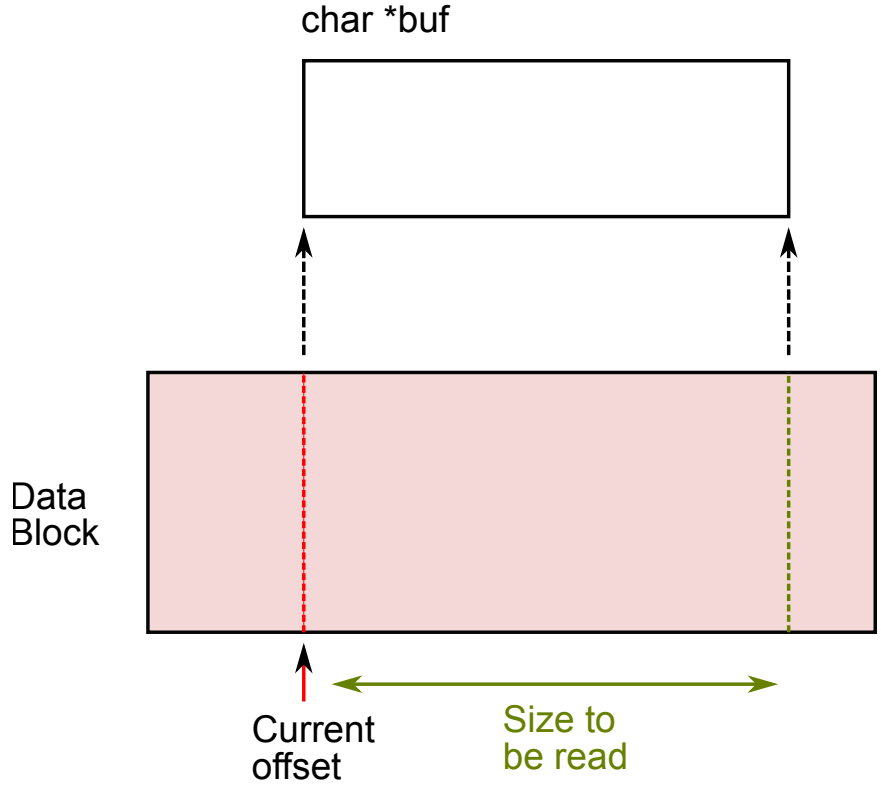
Three difficulties:

1. Small operations
2. First/last block on big operations
3. Extending writes

Implementation

Small operation: example

- Current offset is in the middle of the file, not aligned on the beginning of a block
- The size of data to read is smaller than what's remaining in this block

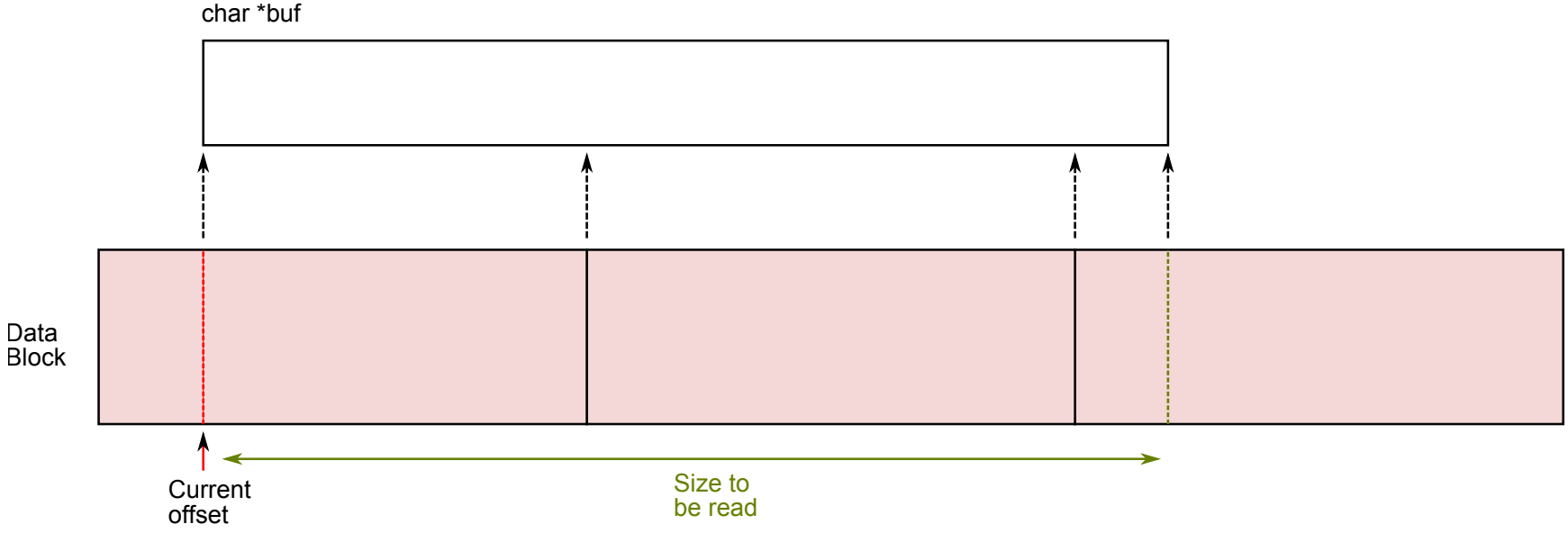


Might want to use a *bounce buffer*

Implementation

Big operation: example

- Current offset is in the middle of the file, not aligned on the beginning of a block
- The size to read spans multiple (non-consecutive) blocks
- The size of data to read is smaller than what's remaining in the last block

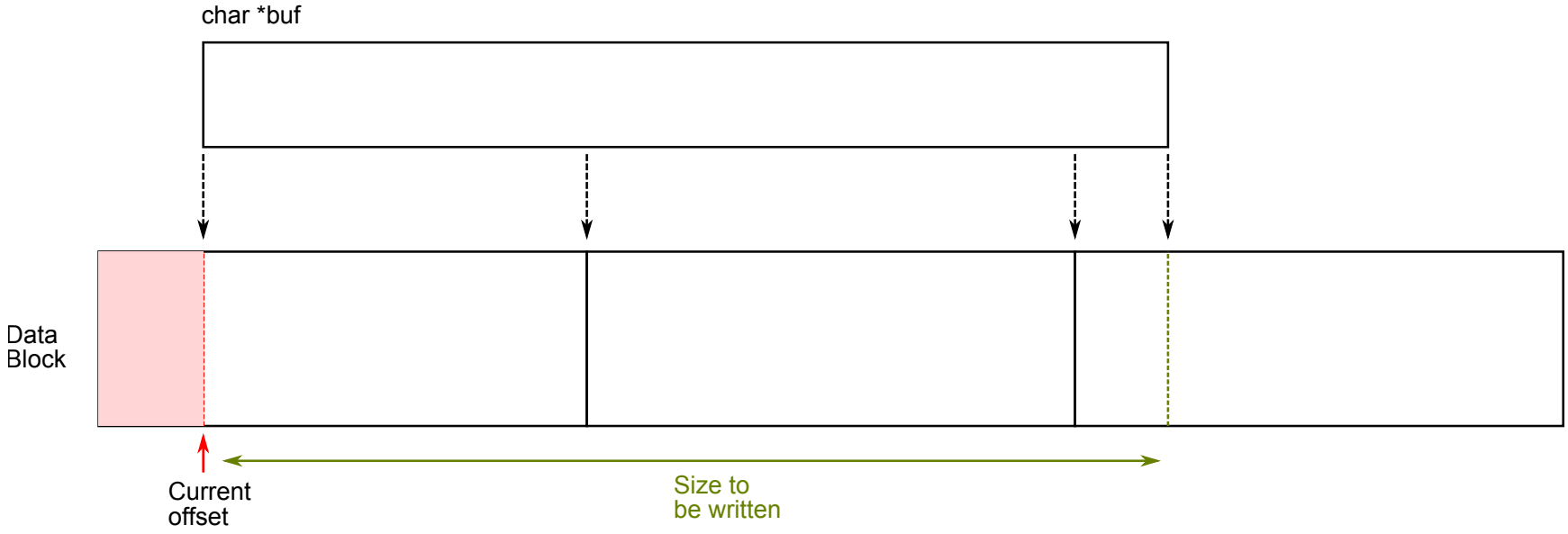


Mix of *bounce buffer* and direct copy

Implementation

Extending write: example

- Write more than what's currently allocated



- Allocation of new blocks must follow *first-fit* strategy (allocate first free data block from beginning of the FAT).

Implementation

In short

- Think of all the cases: combination of file's offset, file's size, size to be read or written, etc.
- Come up with a way to handle all these combinations in the *most* generic way (i.e., not one function or one separate if statement per case!)