# ECS 150 - Project 4

*Joël Porquet*

UC Davis - Spring Quarter 2017

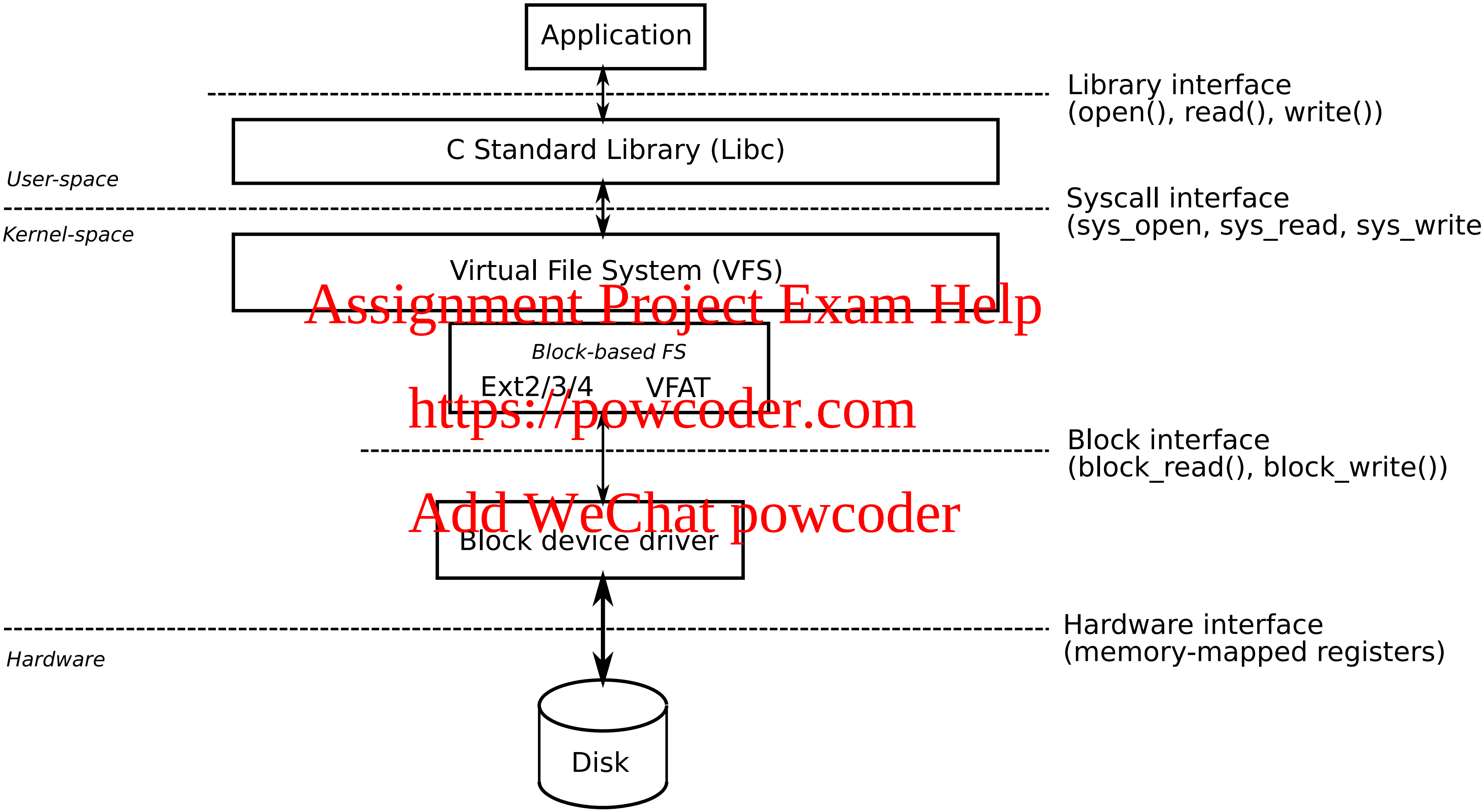
# Goal

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


Application
Library interface
(open(), read(), write())
C Standard Library (Libc)
User-space
Syscall interface
(sys_open, sys_read, sys_write
Kernel-space
Virtual File System (VFS)
Block-based FS
Ext2/3/4
VFAT
Block interface
(block_read(), block_write())
Block device driver
Hardware interface
(memory-mapped registers)
Hardware
Disk

Problem: the vast majority the file system management is in kernel mode!

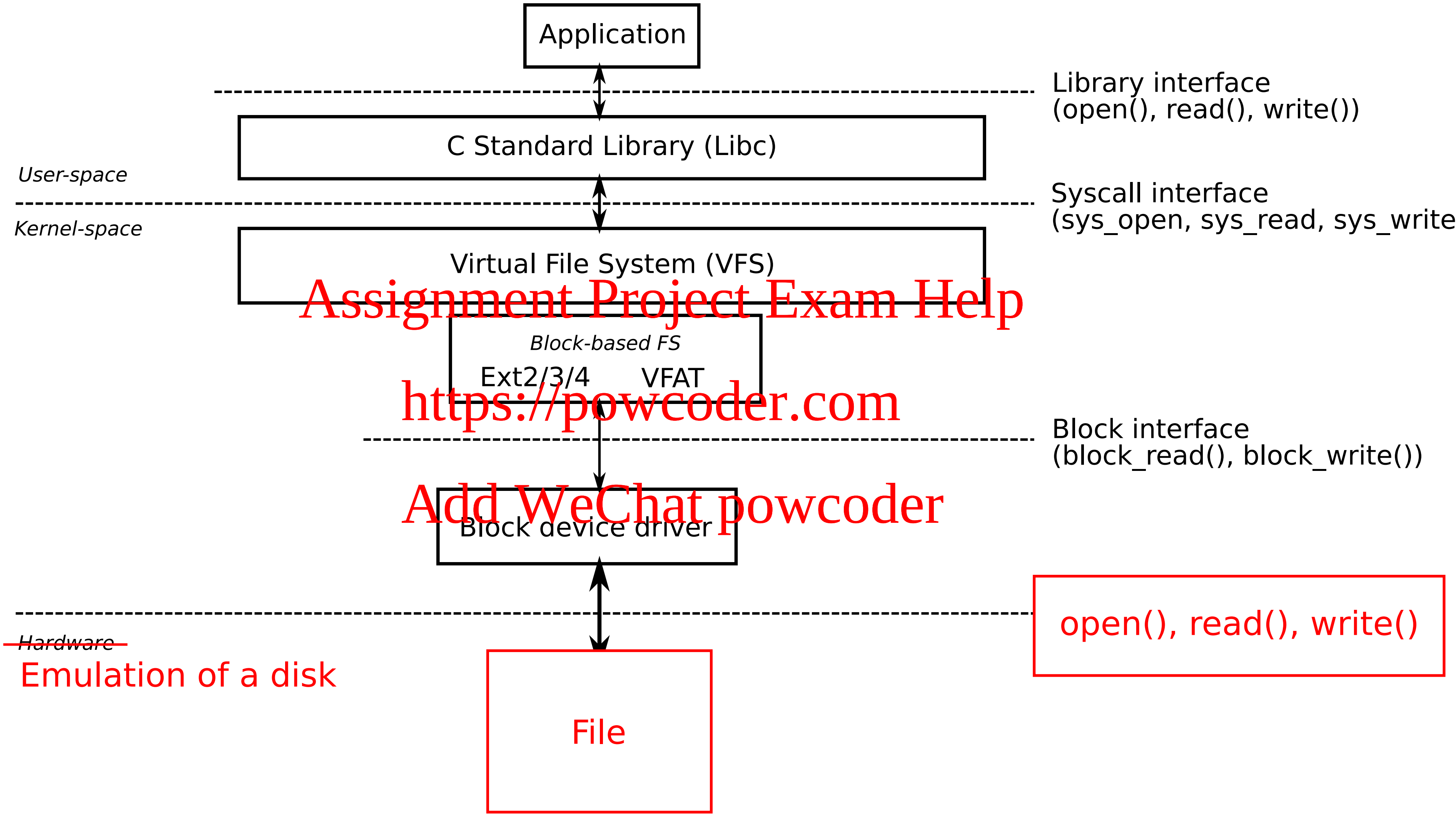
# Emulating a disk with a file

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  ................
...
01fffff0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

# Big picture: replacing the disk



Application

Library interface
(open(), read(), write())

C Standard Library (Libc)

*User-space*

Syscall interface
(sys_open, sys_read, sys_write

*Kernel-space*

Virtual File System (VFS)

*Block-based FS*
Ext2/3/4 VFAT

Block interface
(block_read(), block_write())

Block device driver

*~~Hardware~~*
Emulation of a disk

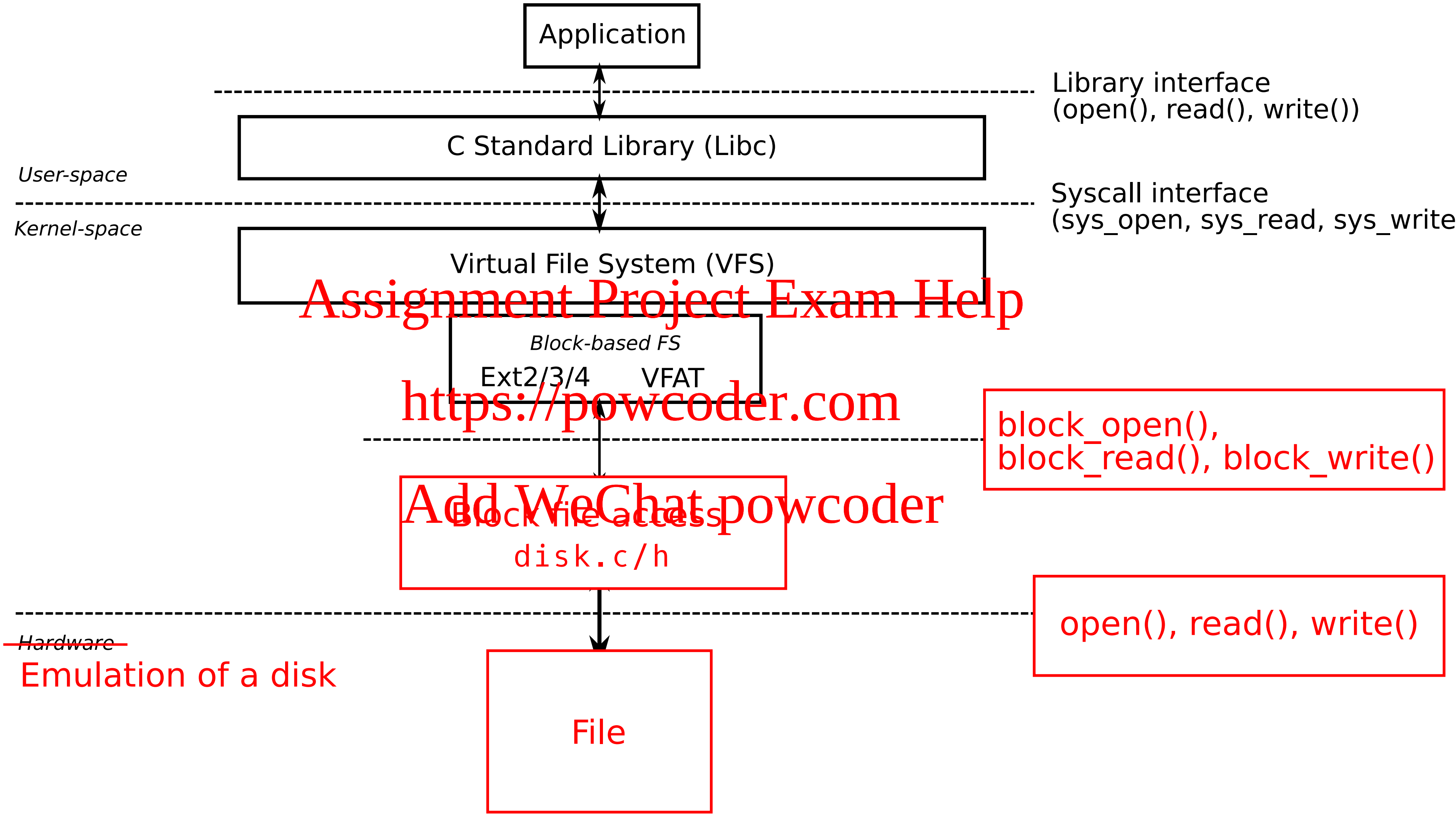
File

open(), read(), write()

# 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);
}
```

# Big picture: replacing the block device driver



Application

Library interface
(open(), read(), write())

C Standard Library (Libc)

*User-space*

Syscall interface
(sys_open, sys_read, sys_write

*Kernel-space*

Virtual File System (VFS)

*Block-based FS*

Ext2/3/4 VFAT

block_open(),
block_read(), block_write()

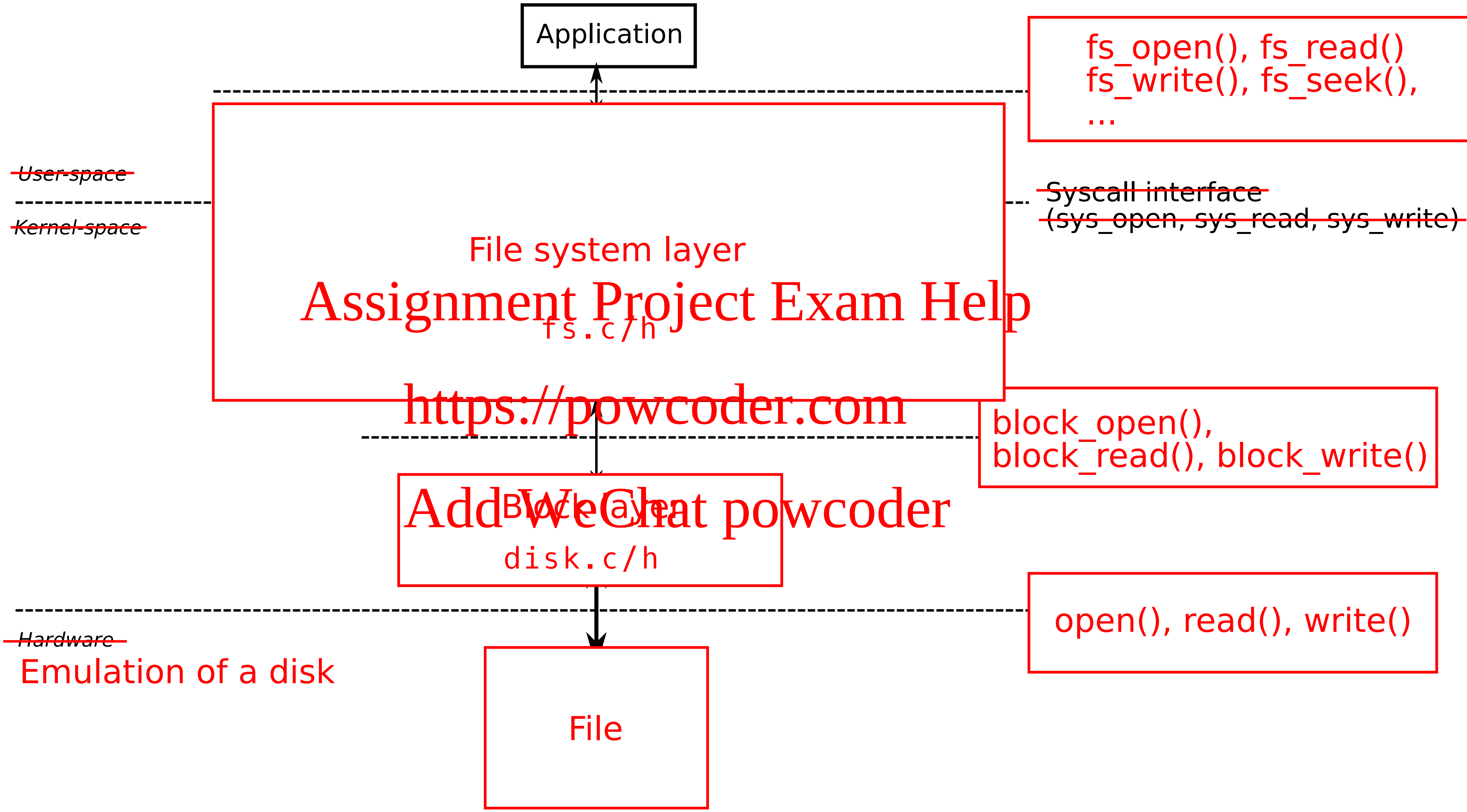
Block file access
`disk.c/h`

open(), read(), write()

~~*Hardware*~~
Emulation of a disk

File

# Big picture: replacing the libc/vfs/fs drivers



Application

fs_open(), fs_read()
fs_write(), fs_seek(),
...

~~User-space~~

~~Kernel-space~~

~~Syscall interface~~
~~(sys_open, sys_read, sys_write)~~

File system layer

`fs.c/h`

block_open(),
block_read(), block_write()

Block layer

`disk.c/h`

open(), read(), write()

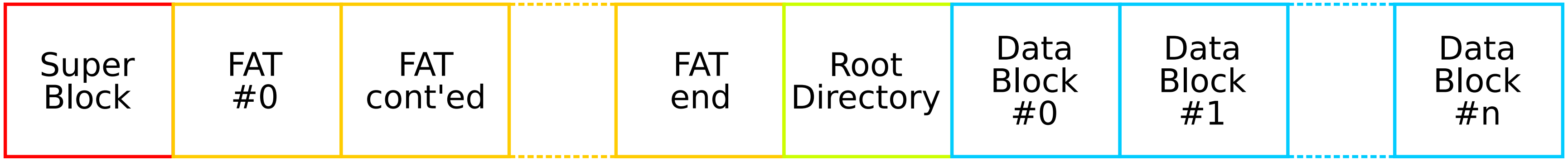
~~Hardware~~

Emulation of a disk

File

# ECS150-FS

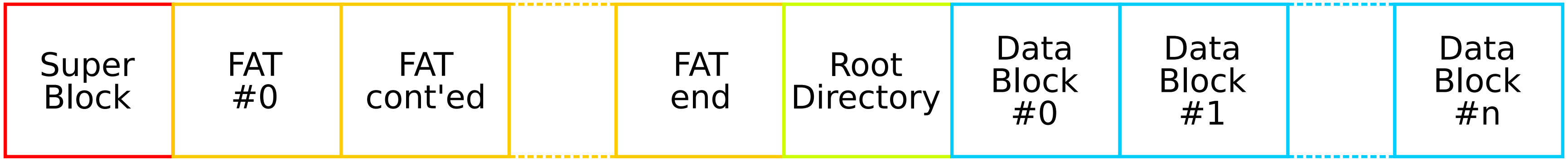
## Layout


Super Block
FAT #0
FAT cont'ed
FAT end
Root Directory
Data Block #0
Data Block #1
Data Block #n

Each block is 4096 bytes.

# ECS150-FS

## Layout



Each block is 4096 bytes.

Example with file system embedding 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

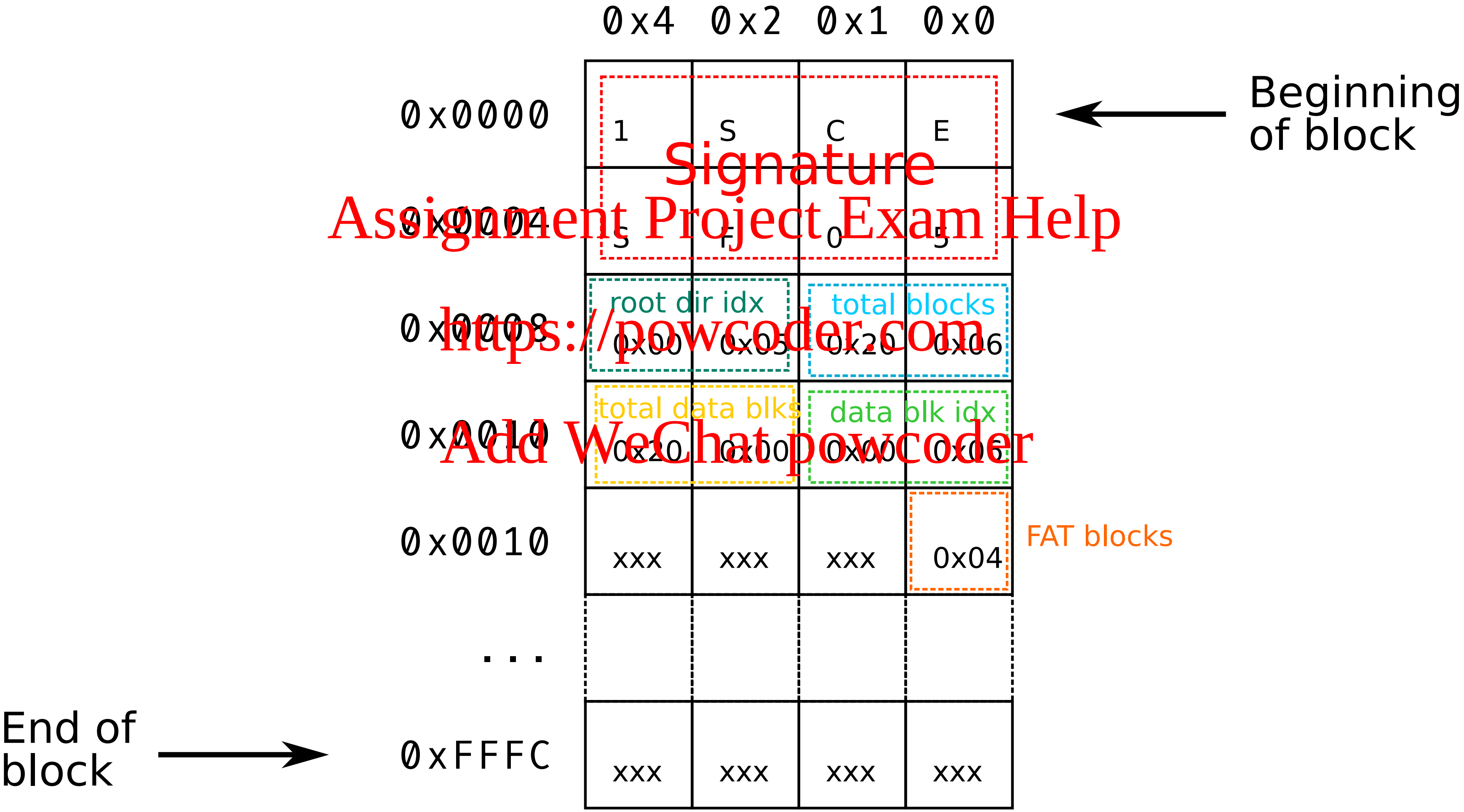
# ECS150-FS

## 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



|  | 0x4 | 0x2 | 0x1 | 0x0 |
|---|---|---|---|---|
| 0x0000 | 1 | S | C | E |
| 0x0004 | S | F | 0 | 5 |
| 0x0008 | 0x00 | 0x05 | 0x20 | 0x06 |
| 0x0010 | 0x20 | 0x00 | 0x00 | 0x06 |
| 0x0010 | xxx | xxx | xxx | 0x04 |
| ... |  |  |  |  |
| 0xFFFC | xxx | xxx | xxx | xxx |

# ECS150-FS

## Superblock: C data structure

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

Key points:

- The integer types must match exactly those of the specification
- Careful about alignment

# Digression

## Integer types

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

# 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:

```
#include <stdint.h>

int8_t
int16_t
int32_t

uint8_t
uint16_t
uint32_t
```

# Digression

## Structure alignment

Structure naturally packed:

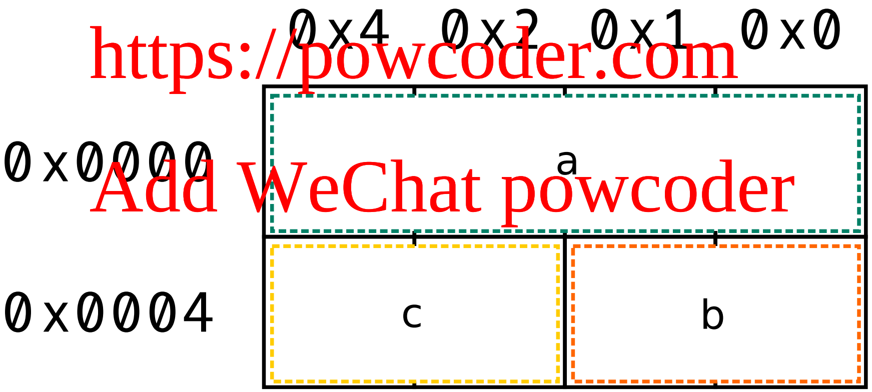
```
struct packed_s
{
  int32_t   a;
  int16_t   b;
  int16_t   c;
};
```

# Digression

## Structure alignment

Structure naturally packed:

```
struct packed_s
{
  int32_t   a;
  int16_t   b;
  int16_t   c;
};
```



# Digression

## Structure alignement

Structure fields have to aligned...

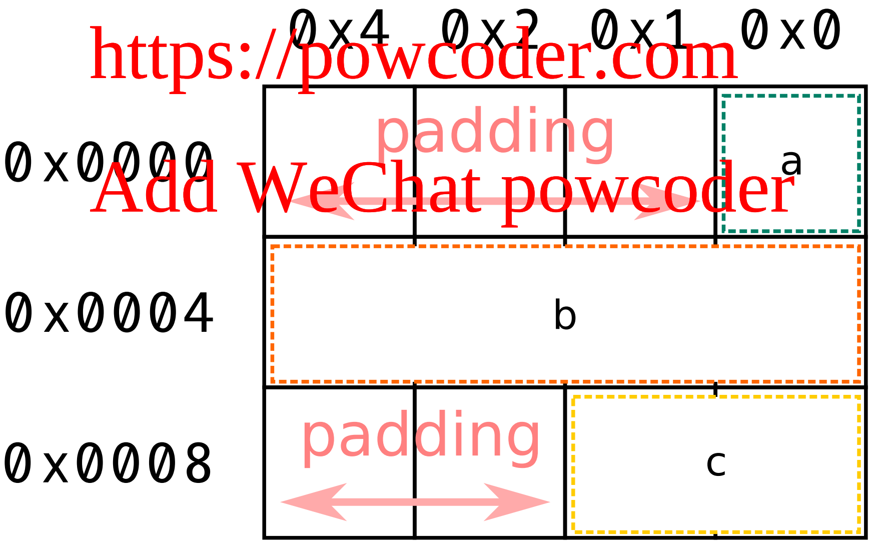
```
struct padded_s
{
  int8_t    a;
  int32_t   b;
  int16_t   c;
};
```

# Digression

## Structure alignement

Structure fields have to aligned...

```
struct padded_s
{
  int8_t    a;
  int32_t   b;
  int16_t   c;
};
```



# Digression

## Structure alignement

Force compiler to ignore alignment

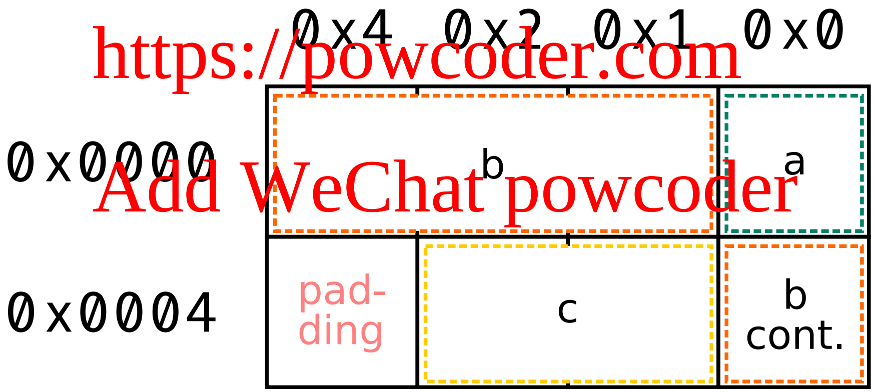
```
struct packed_s
{
  int8_t    a;
  int32_t   b;
  int16_t   c;
} __attribute__((packed));
```

# Digression

## Structure alignement

Force compiler to ignore alignment

```
struct packed_s
{
  int8_t    a;
  int32_t   b;
  int16_t   c;
} __attribute__((packed));
```



# Digression

## Structure alignement

Conclusion: when transposing a specification into data structures, always use packing!

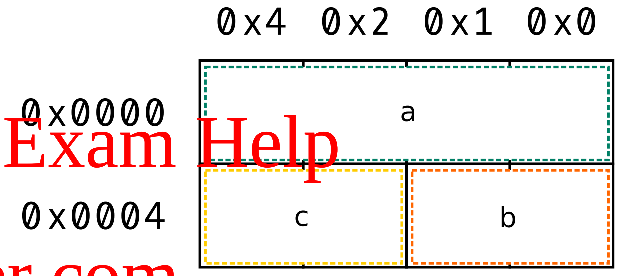
- File format
- Network protocol
- Etc.

# 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 packed_s
{
  int32_t   a;
  int16_t   b;
  int16_t   c;
};
```



```
char* buf[8];

fd = open("file", O_RDWR);
read(bd, buf, 8);

struct packed_s *s = buf;
s->a = 0;

/* or simply */
struct packed_s obj;
read(bd, &obj, sizeof(obj));
obj.a = 0;
```

# ECS150-FS

## 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

## 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

## Root directory

1 block, 16-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

<test1, 25000, 2>,
<test2, 5000, 1>,
<test3, 0, FAT_EOC>
...

FAT

| | |
|---|---|
| 0 | |
| 1 | 8 |
| 2 | 3 |
| 3 | 4 |
| 4 | 5 |
| 5 | 6 |
| 6 | 7 |
| 7 | 0xFFFF |
| 8 | 0xFFFF |
| 9 | 0 |
| 10 | 0 |

Data blocks

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

# Implementation

## Phase 1: Volume mounting

- `fs_mount()`: open the virtual disk, and 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 function should change the file system...

# Implementation

## Phase 4: File reading/writing

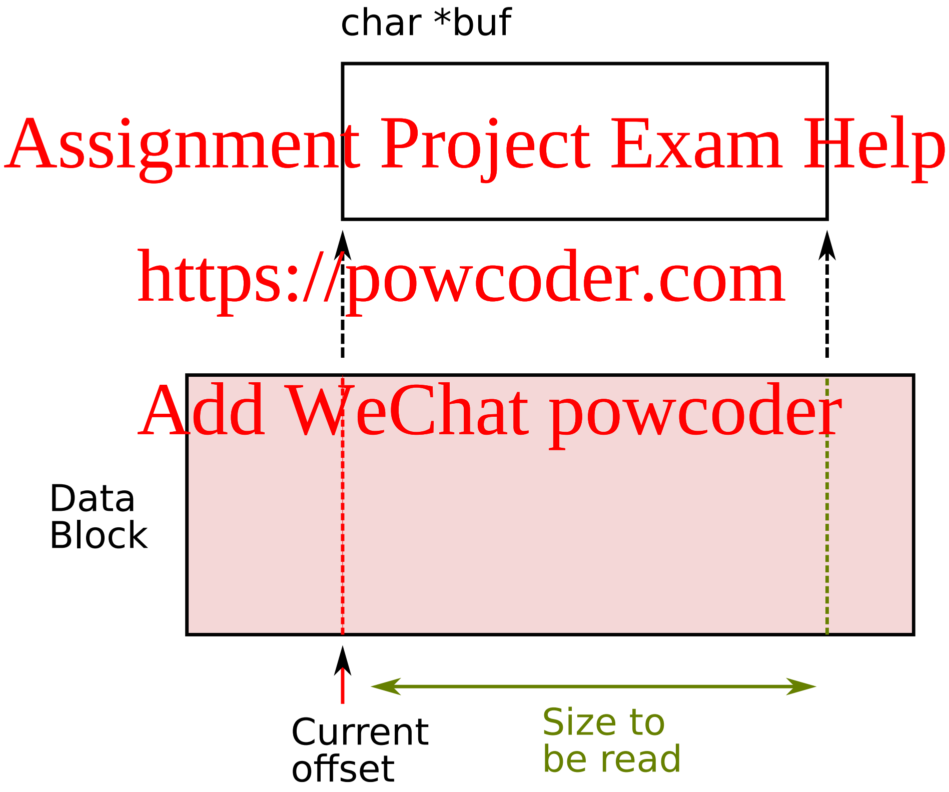
Most complicated phase: might take as much time as all the previous phases combined

- Allocation of new blocks must follow *first-fit* strategy (allocate first free data block from beginning of the FAT).
- Three difficulties:
    - Small operations
    - First/last block on big operations
    - 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

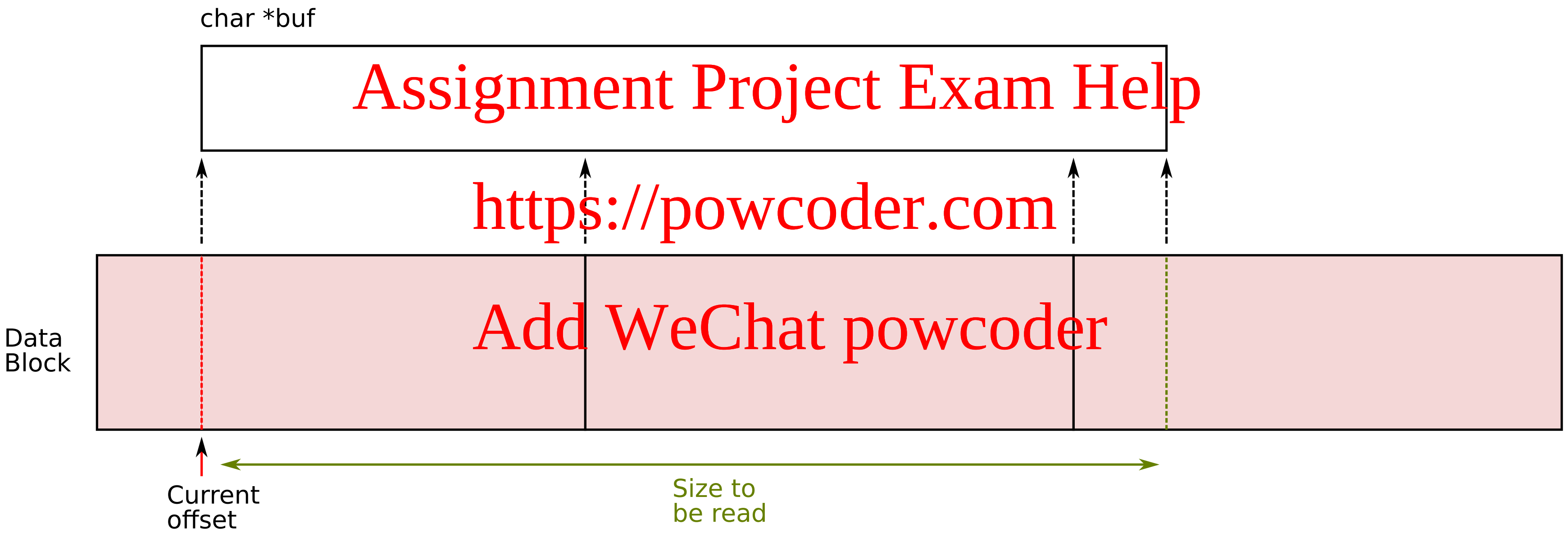


Might want to use a *bounce buffer*

# 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

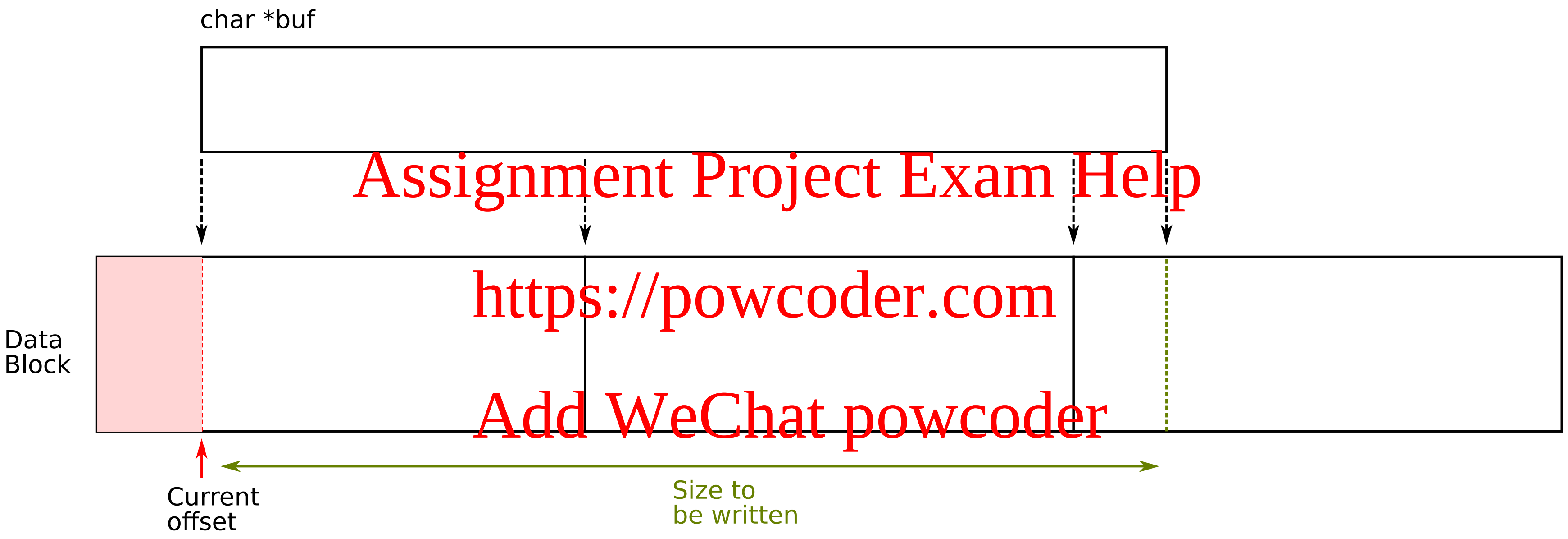


Mix of *bounce buffer* and direct copy

# Implementation

## Extending write: example

- Write more than what's currently allocated



# 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 (ie not one function per case!)