香港中文大學
The Chinese University of Hong Kong

CSCI5550 Advanced File and Storage Systems

Programming Assignment 02: In-Storage File System (ISFS) using FUSE

Outline

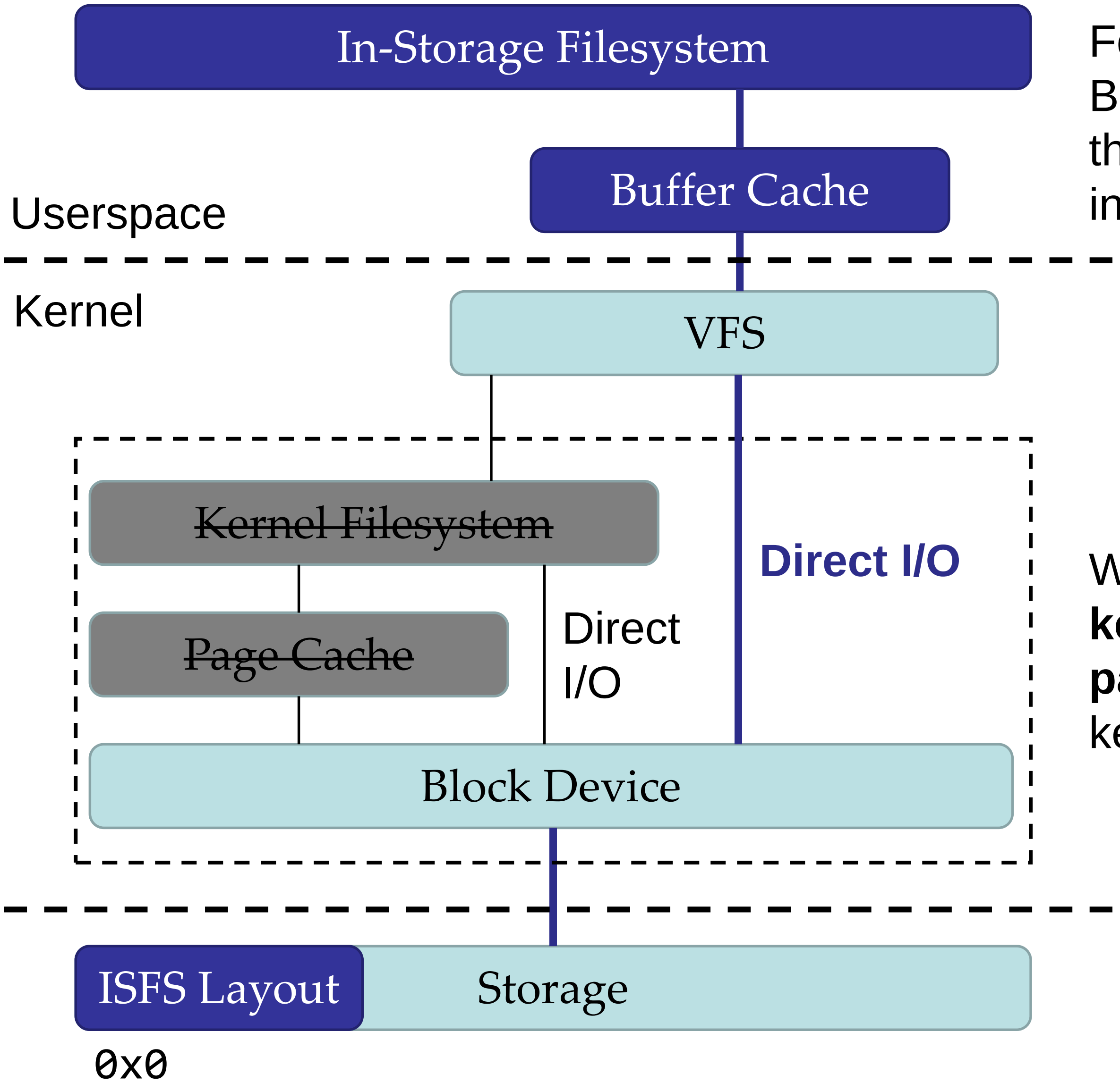
- **ISFS Introduction**
  - From IMFS to ISFS
  - Overall Structure of ISFS
- **Direct I/O to USB drive**
  - Mount a USB drive in Virtual Machine
  - Direct I/O to USB drive
- **Buffer Cache**
  - A simple buffer cache design & architecture
  - Eviction policy: LRU
  - Buffer Cache Optimization
  - Dirty Blocks Writeback
- **Grading for Programming Assignment 2**
  - Basic Part (50%) + Advanced Part (50%)
  - Bonus (10%)

From IMFS to ISFS

- For IMFS, all structures (Superblock, Bitmaps, Inode Table, and Data Region) are stored in the **memory**.

- For ISFS, you are required to persist those structures into the **storage** (e.g., a USB flash drive).
  - **Direct I/O** will be used for data read/write.
  - You will implement a **Buffer Cache** to reduce #reads/writes to the storage.

Overall structure of ISFS



For ease of implementation, the Buffer Cache is implemented in the units of **block size (512 B)** instead of page size (4 KB).

We use **direct I/O** to bypass the **kernel filesystem** and the **page cache** maintained by kernel.

Outline

- **ISFS Introduction**
  - From IMFS to ISFS
  - Overall Structure of ISFS
- **Direct I/O to USB drive**
  - Mount a USB drive in Virtual Machine
  - Direct I/O to USB drive
- **Buffer Cache**
  - A simple buffer cache design & architecture
  - Eviction policy: LRU
  - Buffer Cache Optimization
  - Dirty Blocks Writeback
- **Grading for Programming Assignment 2**
  - Basic Part (50%) + Advanced Part (50%)
  - Bonus (10%)

Mount a USB drive in Virtual Machine

- Take VirtualBox as an example

- Use *lsblk* to list block devices

```
sting@sting-VirtualBox:~$ lsblk
NAME   MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sdb      8:16   1 14.9G  0 disk
└─sdb1   8:17   1 14.9G  0 part
sr0     11:0    1 82.3M  0 rom  /media/sting/VBox_GAs_6.0.6
sda      8:0    0   50G  0 disk
├─sda2   8:2    0    1K  0 part
├─sda5   8:5    0  975M  0 part [SWAP]
└─sda1   8:1    0   49G  0 part /
sting@sting-VirtualBox:~$
```

Direct I/O

- **Direct I/O** can support that file reads/writes directly from the applications to the storage, bypassing the filesystem cache.
  - Redundant optimization if we have userspace buffer cache & OS page cache!

- An application invokes direct I/O by opening a file with the **O_DIRECT** flag. In our case, the file is the USB drive.

- When a file is opened with **O_DIRECT**:
  - Assume the smallest unit of access called sector size.
  - All I/O size must occur in the multiplies of sector size.
  - The memory being read from or written to must also be sector-byte aligned.

Direct I/O Example

```
int fd, ret;
unsigned char* buf;
ret = posix_memalign((void**)&buf, 512, size);

fd = open("/dev/sdb1", O_RDONLY | O_DIRECT);
if (fd < 0){
    perror("open /dev/sdb1 failed");
    exit(1);
}

ret = pread(fd, buf, size, 0);
```

Transferred bytes

Start reading from this byte offset

1. Since we are using **direct I/O to USB drive**, the transferred bytes and offset must be in the multiplies of 512 bytes & allocated memory space must be aligned on a 512-byte.
2. Therefore, posix_memalign is required to allocate aligned memory space for USB direct I/O.
3. The magic number is not always 512 bytes. It depends on the <u>sector size</u> of the underlying block device.

Provided I/O interface

```c
#include <unistd.h>
#include <assert.h>

unsigned int num_read_requests = 0;
unsigned int num_write_requests = 0;
size_t block_size = 512; // (bytes)

void io_read(int fd, void* buf, int index)
{
    off_t offset = index * block_size;
    ssize_t read_bytes = pread(fd, buf, block_size, offset);
    assert(read_bytes==block_size);
    num_read_requests++;
}

void io_write(int fd, void* buf, int index)
{
    off_t offset = index * block_size;
    ssize_t write_bytes = pwrite(fd, buf, block_size, offset);
    assert(write_bytes==block_size);
    num_write_requests++;
}
```

Direct I/O Example w/ my_io.h

```
int fd, ret;
unsigned char *buf;
ret = posix_memalign((void**)&buf, block_size, 512);

fd = open("/dev/sdb1", O_RDONLY | O_DIRECT);
if (fd < 0){
    perror("open /dev/sdb1 failed");
    exit(1);
}

io_read(fd, buf, which_idx);
```

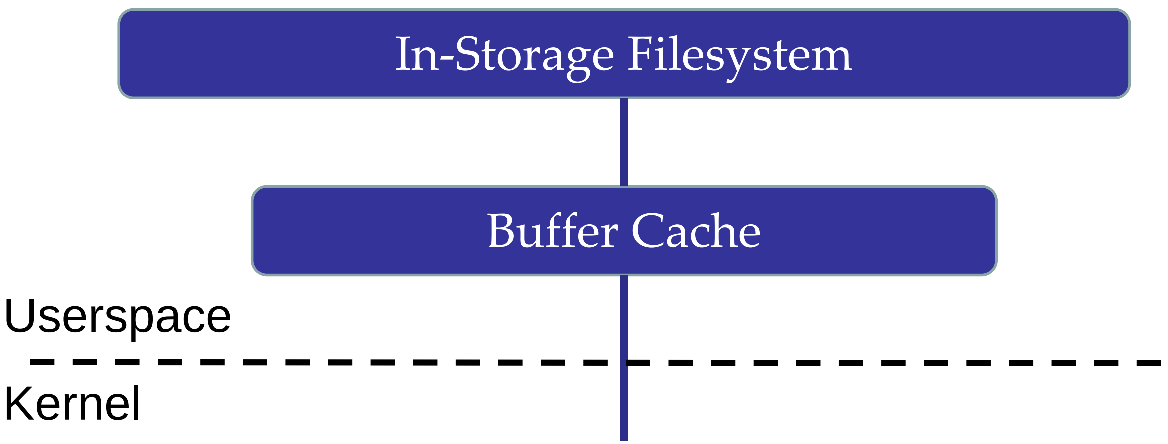
It is compulsory to call io_read (io_write) when accessing USB data.
You cannot call pread (pwrite) or modify my_io.h by yourself!

Outline

- **ISFS Introduction**
  - From IMFS to ISFS
  - Overall Structure of ISFS
- **Direct I/O to USB drive**
  - Mount a USB drive in Virtual Machine
  - Direct I/O to USB drive
- **Buffer Cache**
  - A simple buffer cache design & architecture
  - Eviction policy: LRU
  - Buffer Cache Optimization
  - Dirty Blocks Writeback
- **Grading for Programming Assignment 2**
  - Basic Part (50%) + Advanced Part (50%)
  - Bonus (10%)

Buffer Cache Design

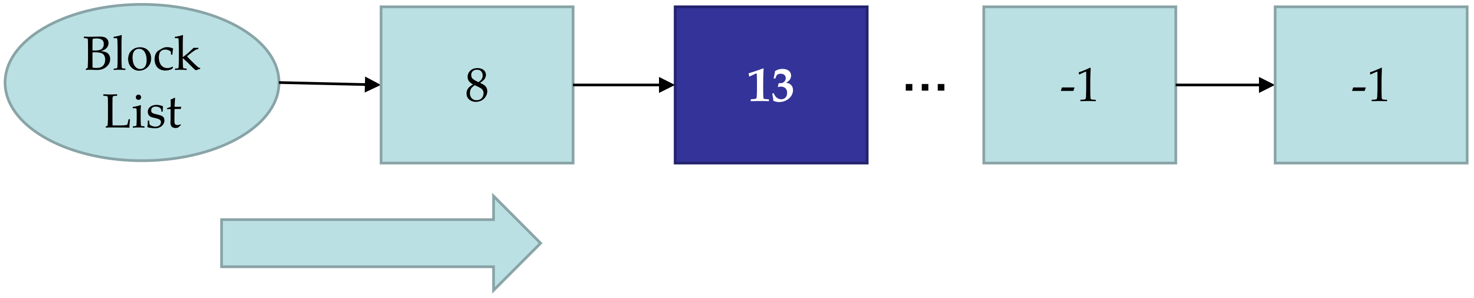
- To minimize the frequency of disk access, the kernel keeps a **buffer** to store the recently accessed files.
  - This buffer is called the **page cache**.
  - **Page cache** is a part of main memory which contains different pages of data from storage.
- In our scenario, **“Buffer Cache”** works in similar idea as page cache. However, we manage **“Buffer Cache”** in the units of **blocks** (**512B**) for ease of implementation.



Every requests from ISFS will go to Buffer Cache first.

Buffer Cache: Access Mechanism

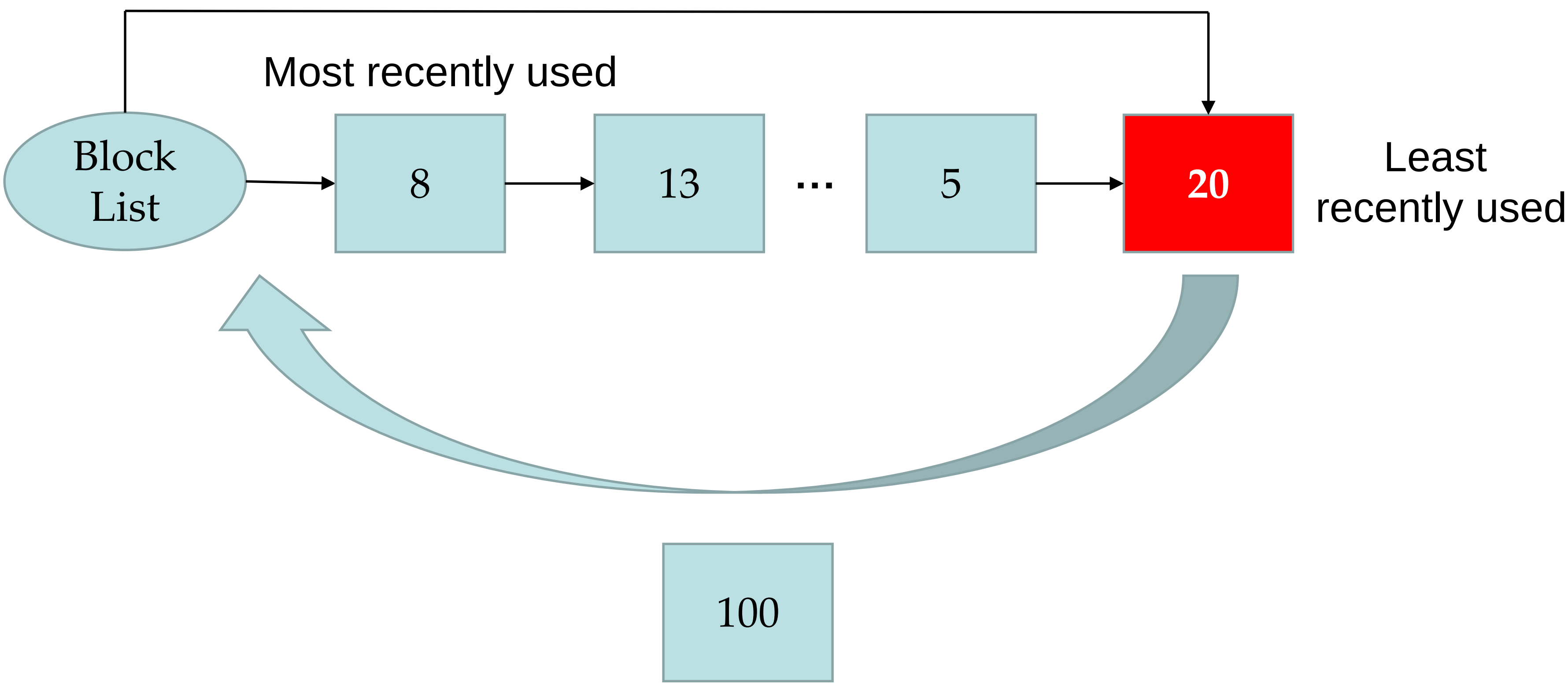
- **Every I/O** request from ISFS will go to Buffer Cache first.
  – If the requested data can be found in the Buffer Cache, directly accessing the data in the Buffer Cache.
  – If the requested data cannot be found:
    - ① Choosing an empty block or Evicting a block from the Buffer Cache;
    - ② Then, using Direct I/O to read the requested block and keeping the requested block in the Buffer Cache.



Search all blocks to check the requested block is present or not

Eviction Policy: LRU

- **Least recently used** (**LRU**): Discard the **least recently used** block first.



Buffer Cache Optimization

- The search time to find out a particular block could be quite high if only a single LRU list is used.
  - The worst case will be n times, where n is the number of blocks in Buffer Cache.

- There are many ways to optimize the search time, e.g., hash, tree, etc.

- How to optimize the search time of buffer cache while maintaining the LRU eviction policy?

Dirty Block Writeback

- When we perform a write request with Buffer Cache:
  - The write(s) won't directly go to the storage.
  - Instead, the block of memory is modified. The modified block is called "**dirty**".

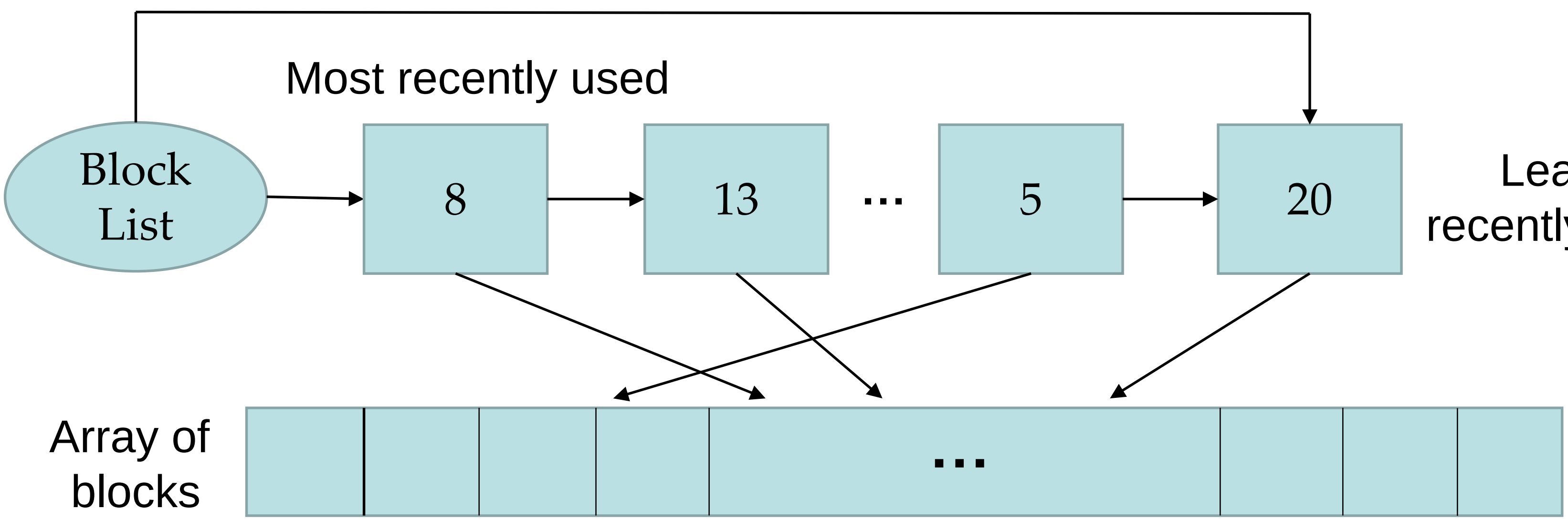
- When a block of memory is to be replaced, we need to check whether the block is dirty or not.
  - **Dirty Block** → Write the block back to the storage before being replaced.
  - **Not Dirty Block** → Simply replace the block.

- What if the filesystem un-mounts?
  - To maintain **consistency**, ISFS must write the dirty block(s) back to the storage after un-mounting.

(Suggest) Data Structures

```
char* buf_cache_head;
char* buf_cache_tail;

struct block_info{
    int block_id;
    bool dirty;
    block_info* next_block_info;
    block_info* prev_block_info;
    char* block_ptr; // point to the allocated memory address
};
block_info block[SIZE_BUF_CACHE];
```



Outline

- **ISFS Introduction**
  - From IMFS to ISFS
  - Overall Structure of ISFS
- **Direct I/O to USB drive**
  - Mount a USB drive in Virtual Machine
  - Direct I/O to USB drive
- **Buffer Cache**
  - A simple buffer cache design & architecture
  - Eviction policy: LRU
  - Buffer Cache Optimization
  - Dirty Blocks Writeback
- **Grading for Programming Assignment 2**
  - Basic Part (50%) + Advanced Part (50%)
  - Bonus (10%)

Grading – Basic Part (50%)

- Persisting both metadata and file data into a USB drive.
  - [H] Using the provided I/O interface (my_io.h)
  - [H] The file data can still be accessed after re-mounting ISFS
  - [25%] The support of cd, ls, mkdir, touch, echo "string" >> file, cat, rmdir, rm
  - [25%] The support of "big file" and "big directory"
    - ➢ To test the support of "big file", a "big file" will be copied from kernel FS to ISFS and then copied back.
    - ➢ To test the support of "big directory", a shell script will be used to create a large number of files & directories.

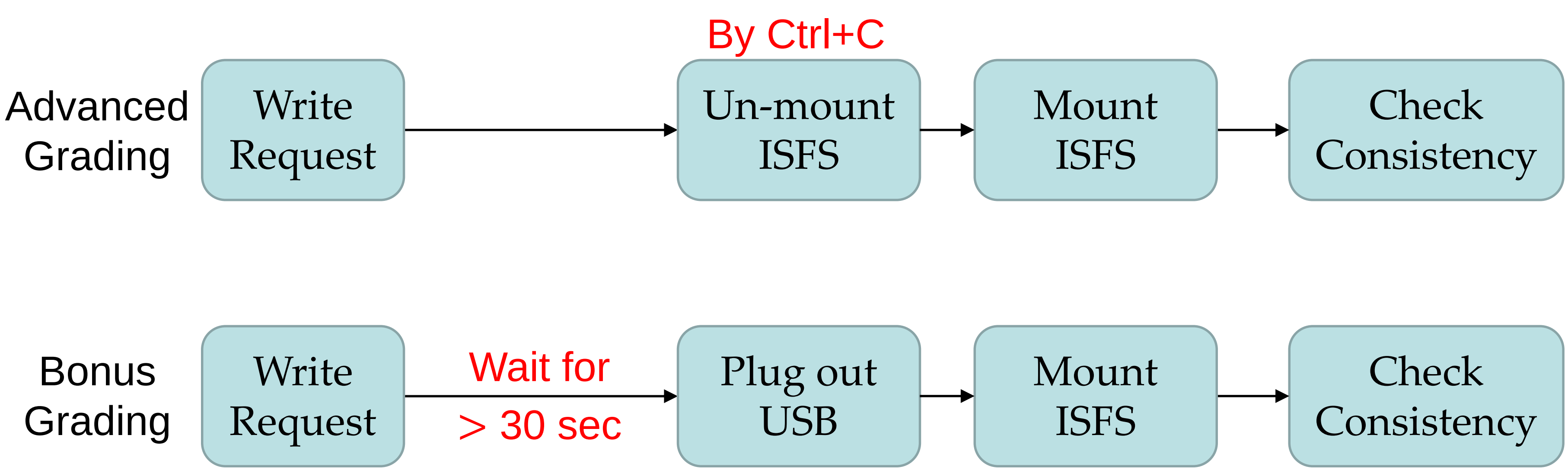
- ***Note1****: Fail to support [H] requirements will get 0 points in this part.*
- ***Note2****: Modifying the code in my_io.h is prohibited.*
- ***Note3****: Don't need to support hard link & soft link in the ISFS.*

Grading – Advanced Part (50%)

- You are required to build a Buffer Cache to minimize the number of accesses to the storage (USB drive).
  - [35%] Correctness of Buffer Cache:
    1. Implement LRU eviction policy
    2. Consistency maintenance: dirty blocks writeback when un-mounting (Ctrl+C) ISFS
    3. USB access minimization: show the numbers of reads/writes are reduced after using Buffer Cache.
  - [15%] Buffer Cache Optimization:
    1. Explain your method to reduce search times.
    2. Show the worst case of search times $\leq \frac{n}{4}$, where n is the number of blocks in Buffer Cache.

Grading – Bonus (10%)

- The dirty blocks will be wrote back after un-mounting the ISFS. There will still be a problem if the USB drive is removed without proper procedure.
  - To provide stronger consistency, invoke another thread to write the dirty blocks to USB drive every 30 seconds.
  - Be aware of the critical section!



Parameter Setting

- 1 block = 512 bytes, 1 page = 4,096 bytes = 8 blocks
- Superblock = 1 page

```
struct superblock{
    unsigned int size_ibmap; // 48 pages
    unsigned int size_dbmap; // 48 pages
    unsigned int size_inode; // 32 bytes
    unsigned int max_size_filename; // 12 bytes
    unsigned int root_inum; // = 0
    unsigned int num_disk_ptrs_per_inode; // 4
        // 2 direct ptrs, 1 indirect ptr, 1 double indirect ptr
};
```

- Inode Bitmap = 48 pages
- Data Bitmap = 48 pages
  - Every bit is used to specify the corresponding inode or data block is in use or not. (0 → not used, 1 → used)
  - $48\ (pages) \times 4{,}096\ (bytes) \times 8 = 1{,}572{,}864\ (bits)$

Parameter Setting (Conti)

- Inode Table = 12,288 pages
  - Each inode requires 8 integers = 32 bytes
  - $\frac{12{,}288\ (pages) \times 4096\ (bytes)}{32\ (bytes)} = 1{,}572,,864\ (inodes)$

```
struct inode_struct{
    int flag;
        // indicating the type of file of this node
        // (regular file or dir or something else)
    int blocks; // how many blocks have been used
    int used_size; // how many bytes have been used
    int links_count; // # hard links to this file
    int block[4]; // a set of inum points to data region
};
```

- Data Region = 196,608 pages
  - $196{,}608\ (pages) \times 8\ (blocks) = 1{,}572{,}864\ (blocks)$

Parameter Setting (Conti)

- The size of your Buffer Cache:
  - 10,446 pages = 83,568 blocks = 42,786,816 bytes

- Total size required by your ISFS:
  - $1\ (superblock) + 48 \times 2\ (bitmaps) + 12{,}288\ (inode\ table) + 196{,}608\ (data\ region) = 208{,}993 (pages) = 856{,}035{,}328\ (bytes)$
  - The size of your testing USB drive should bigger than above value.

- Please remember to initialize the superblock & two bitmaps when creating a new ISFS.

Submission

- **Submission Deadline**: ~~**11:59pm on May 12, 2020**~~
  11:59pm on May 19, 2020

- Please submit two things to CUHK Blackboard:
  ① **The whole package of your project**
    - ✓ Including the source code(s), Makefile, etc.
    - ✓ Naming the package of your ISFS project after your **student ID**

  ② **A short report**
    - ✓ Showing how to run your project
    - ✓ Providing the screen shots of the results to prove that your ISFS functions well.

- Discussion is allowed, but no **plagiarism**
  - Your code(s) will be cross-checked

Reference

- https://en.wikipedia.org/wiki/Page_replacement_algorithm
- https://www.quora.com/Why-does-O_DIRECT-require-I-O-to-be-512-byte-aligned
- https://en.wikipedia.org/wiki/Page_cache
- http://man7.org/linux/man-pages/man3/posix_memalign.3.html
- http://man7.org/linux/man-pages/man2/pwrite.2.html