Design for MP5

1. Implementation of blocking disk

The SimpleDisk class apply I/O operation in a greedy "busy-waiting" mode — it will not give up CPU until the I/O operation is finished. To optimize the CPU utilization, BlockingDisk is implemented. The general idea is enqueue the current thread when the disk is busy and give up CPU immediately. There are several approaches of how to check and complete I/O operation later, and the interrupt approach is used here. The interrupt handler will be illustrated in Bonus section.

We derive a BlockingDisk class from the base class SimpleDisk. Besides the original members, we add a blocking queue for all the BlockingDisk objects and also override read and write functions.

1.1. Constructor

The constructor is simple. We use the constructor of the base class and meanwhile initialize the blocking queue.

1.2. void read(unsigned long _block_no, unsigned char* _buf)

In this function, we enqueue the current thread and then send read request to the disk. The order must not be changed. This is because if the disk is ready it will trigger an interrupt right after we send the request; if the enqueue operation has not been applied before sending request, chances are that the interrupt will take place before the thread has been enqueued.

Then we check if the disk is ready or not; if yes, we apply the read operation and dequeue the blocking queue; otherwise, we give up the CPU — every time the thread gains CPU, it check the disk and acts as the same.

1.3. void write(unsigned long _block_no, unsigned char* _buf)

This function is basically the same with the read function. Rather than apply read operation, we apply write operation.

2. Implementation of filesystem

According to the requirement, the file should be able to be extended at any time, thus makes the implementation more difficult. Here we discuss the FAT approach and inode approach and give some explanation on why we choose the latter one.

The FAT approach is linked allocation. We need to maintain a table to record how a block goes to the next one. The inode approach, however, is indexed allocation — each file owns its inode, which contains all the indices of blocks this file owns in an ordered fashion.

One advantage of the FAT approach is that a file can be extended as large as it wants — as long as there are blocks available. However, this method is very problematic. As for a 10MB disk (the disk we use here), there are totally 10MB / 512B = 20K blocks. If we allocate one integer to record one block, 20K * 4B = 80KB is needed for FAT. This is too large to store on the memory. If we put this table on the disk, 80KB / 512B = 160 blocks are needed. The problem is that after bunch of write, append, rewrite, create, delete operations, the list goes anywhere within these 160 blocks. Therefore, to trace a file from beginning to the end, we may need to apply multiple I/O operation just in order to get the index of the blocks it owns.

The inode approach, on the other hand, avoid this problem. Here we assign 1 block for the inode of each file. It can be calculated that 160 blocks are needed to store in odes — the same as what FAT needs. However, to access a file from beginning to the end, we simply read the corresponding inode from the disk, and its blocks are all contained in that inode block — only one I/O operation is enough. This becomes the main reason why we choose to use inode for filesystem. Admittedly, this approach has some drawbacks. The size of the file is bounded by 128 integers/block * 512B = 64KB and the number of files is bounded by 10MB / 64KB = 160. Although this brings more limitation, it also brings fairness. Also, we

can use multi-level inode to enlarge the size of the file. (Because the disk is small and this is a simple operating system, only single-level inode is used here.)

Also, we use a bitmap to provide better block management. Since 10MB / 512B = 20K bits are needed and one block has 512B * 8 = 4K bits, 20K / 4K = 5 blocks are needed to store bitmap.

2.1. Constructor

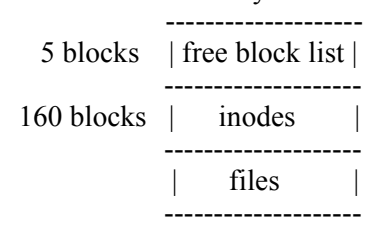
No members need to be initialized here.

2.2. get_inode(), get_block(), free_inode(), free_block()

We create these four functions for management of disk blocks. These functions are nothing but use bitmap to allocate/free blocks.

2.3. BOOLEAN Format(SimpleDisk* _disk, unsigned int _size)

In this function, we wipe the existing filesystem on the disk and write a new, clean filesystem. A standard vanilla filesystem is describe as follows:



2.4. BOOLEAN Mount(SimpleDisk* _disk)

In this function, we load the filesystem from the disk to the filesystem object. Besides initialization of member variables, we read the bitmap into the memory to save bunch of I/O operations. The size of the bitmap is 5 blocks * 512B = 2560B, which is acceptable for the memory.

2.5. BOOLEAN LookupFile(int _file_id)

Here, we simply check if the corresponding bit of the inode bitmap is 0 or 1. If it is 0, it indicates that the inode is allocated, i.e., the file exists; vice versa.

Since the first 5 bits are used for blocks storing bitmap, the inode bitmap starts at the 6th bit, while the file ID starts at 0. Therefore, every time we manipulate the inode bitmap by file ID, we add an offset.

A little change is done here, we do not initialize any file object in this function, because a pure look-up function could be used in more scenarios.

2.6. BOOLEAN CreateFile(int _file_id)

In this function, we first call Lookup function to check if the file exists; if not, allocate a free inode corresponding to the given file ID. Here, we assign a free block for the file we just created and record this block on its inode.

2.7. BOOLEAN DeleteFile(int _file_id)

Similarly, we check if the file exists; if yes, free all the blocks it owns and then free its inode block.

3. Implementation of file

3.1. Constructor

Whenever we create a new File object, we create a new file. Here we set all the variables and meanwhile call the CreateFile function.

3.2. unsigned int Read(unsigned int _n, unsigned char* _buf)

This function reads the file of _n bytes starting from the current position. In this function, we do several things. We read the inode block, and calculate which block to start and to stop. Since we can only read an entire block of data once, we need to allocate a temporary buffer to store the data and cut the data to where it needs to start and stop, and then copy them onto the given buffer. There are different cases that needs to be handled, which is very trivial and will not be discussed here.

3.3. unsigned int Write(unsigned int _n, unsigned char* _buf)

The write function is similar to the read function but even more complicated. Since we start to write at the current position, in order not to overwrite the previous data, we need to read onto a buffer the block where the current position is and then append what we are going to write on the buffer. Also, if we run out of block, we need to allocate a new block for the file, modify the inode, and then write the inode onto disk.

3.4. void Reset()

Assign current position to be 0.

3.5. void Rewrite()

In this function, we free all the blocks the file owns except the first block (note that every file must have at least one block). Then we need to clean the first block — we write all 0s onto that block — and then reset the current position and size.

3.6. BOOLEAN EoF()

Since we record the size of the file, this should be easy to implement — simply compare if the current position is larger than the size or not.

4. Implementation of exercise_file_system()

This function is plugged in thread3 to test the filesystem, in this function we do the following.

When this function is called the first time, we format the disk and mount the filesystem. Then it goes in an infinite loop; when all the contents in this loop are executed once, thread3 gives up CPU. Therefore we guaranteed that every time thread3 gains CPU it executes the contents in the loop — thus ensures the filesystem is continuously tested. In the infinite loop, we do the following.

Create file: create two files — one for small file and the other for large file (by large file, we mean that the size of larger than one block).

Write the small file the string “the quick brown fox jumps over the lazy dog.\n”.

Append to the small file the string “the lazy dog wakes up and gets really angry.\n”.

Reset the small file.

Read the small file.

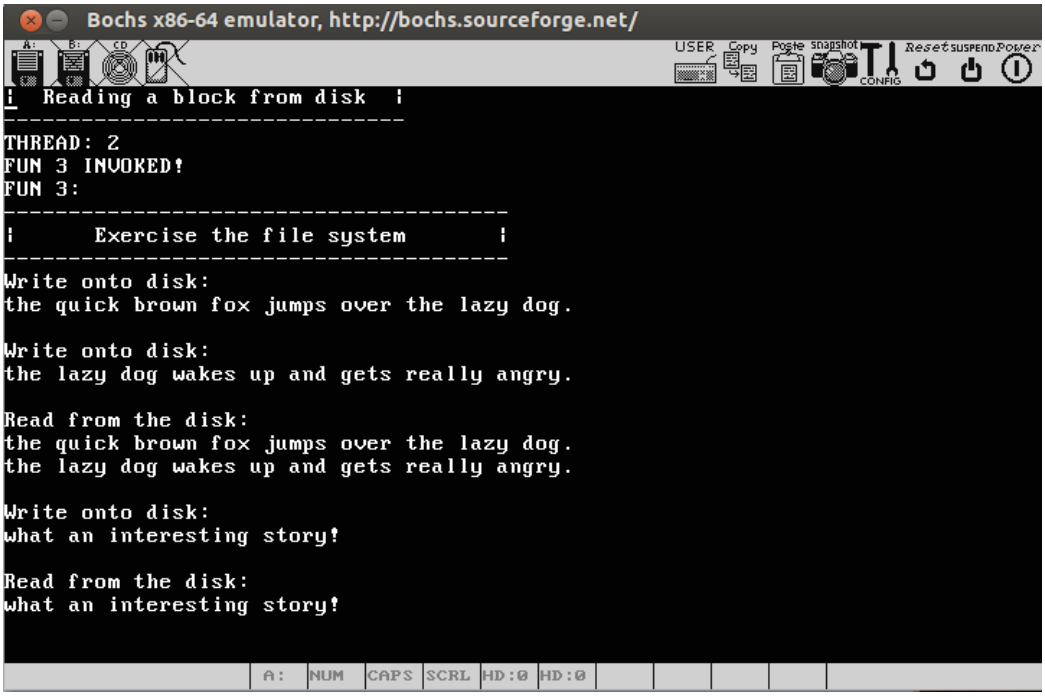
Rewrite the small file.

Write the small file.

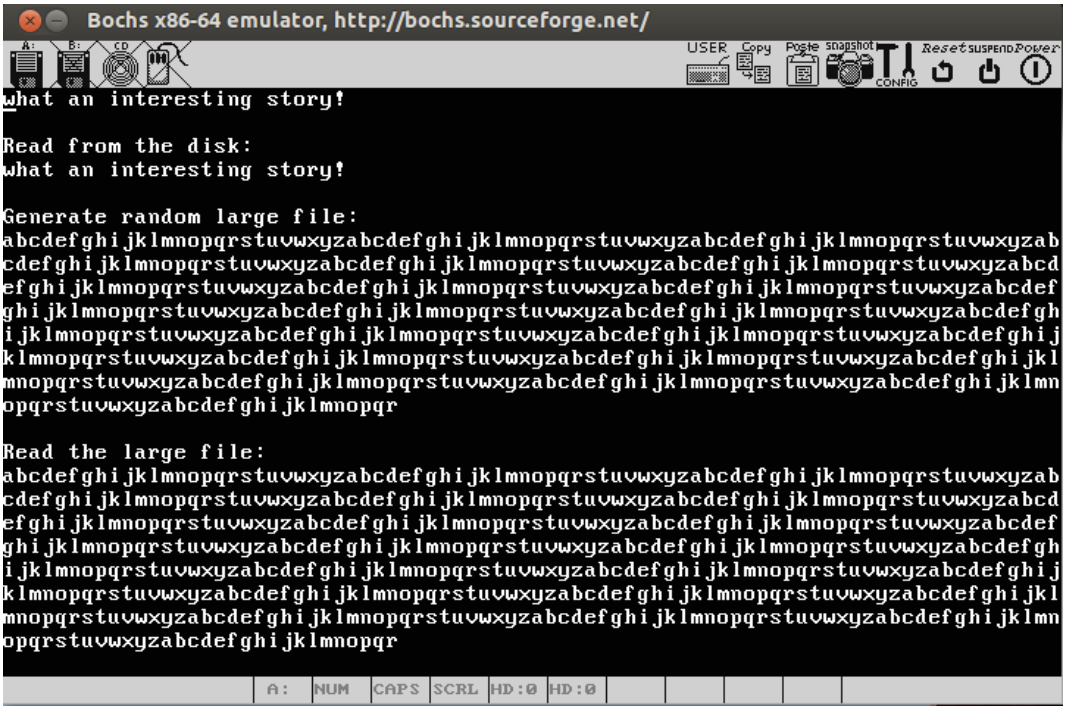
Reset the small file.

Read the small file.

Until now, we test all the necessary file operations. The result is as follows.



To make sure the filesystem does work functionally, we test the large file. We do the following.
Write the large file.
Reset the large file.
Read the large file.
Until now, we test all the necessary file operations. The result is as follows.



We set two macros to facilitate the testing: if uncomment #define _EXERCISE_SMALL_FILE_, it will stop when the small file finishes testing; if uncomment #define _EXERCISE_LARGE_FILE_, it will

stop when the large file finishes testing. Note that only one of the two macros can be uncommented at one time and the two macros are commented in the code handed in.

5. Bonus Options

5.1. Option 0: Using Interrupt for Concurrency

We always need to trade off between quick return time and utilization of CPU. The simple disk approach, even though wasting a lot of CPU resources, guarantees to apply I/O operation at the first moment the disk is ready. The blocking disk approach, on the other hand, saves CPU resources, but its problem is that we could only apply I/O operation at some specific event, like resume a thread, even though the disk has already been prepared for a long time.

Here we use interrupt to solve this problem. Whenever the disk gets ready, the interrupt 14 will be triggered. If we register an interrupt handler function beforehand, that function will be called and I/O operation will be executed at the moment the disk gets ready.

In the interrupt handler, we preempt the current thread — put the current thread in the head of the ready queue and yield to the thread from blocking queue. When the blocked thread completes its I/O operation, the preempted thread will gain the CPU. However, the problem is that the disk could get ready during the period from when the thread send request to when it yield to another thread. If an interrupt happens during this period, the thread will preempt itself and sabotage everything. Hence, in the handler, we first check if the current thread is the blocked thread — if not, preempt; otherwise, simply return.

The interrupt handler is not complete yet. Other cases are illustrated in 5.4.

5.2. Option 1: Support for Disk Mirroring

We derive the MirroredDisk class from SimpleDisk. The mirrored disk records two disks: master disk and slave disk, while the original member “disk_id” in the base class indicates the current disk.

The general idea of implementing the mirrored disk is the following: when try to read from the disk, we send the request to both of the disks and read from the one that gets ready first; when try to write from the disk, we send the request to both of the disks and write the data to both of them. To guarantee that, some changes are needed.

Rather than using one block queue, we use three queues — one for blocked thread, one for disk type (master/slave), and one for status — whether the task is finished or not.

The write function is relatively simple: similarly with the write function in blocking disk, we just need to write onto both the master and the slave disk. We enqueue the thread, check if the disk is ready — if not, gives up CPU, otherwise write onto the disk. And we do the same to the other disk.

As for the read function, first of all, we enqueue [current thread, master, not completed] and send the request to the master disk; then enqueue [current thread, slave, not completed] and send the request to the slave disk. Then we check the status of the disks. If either one of them is ready, we read from that disk; otherwise give up the CPU. When either of the disks is ready, besides applying the read operation, we search for the corresponding task issued to its mirrored disk. Since one disk and one thread can only maps to one I/O operation and every I/O operation of the same thread is sent to both of the disks, [T, disk, S] must be an identical task of [T, mirrored disk, S’]. When [T, disk, S] is completed, [T, mirrored disk, S’] becomes redundant — we do not want apply the I/O operation to that task again. Therefore, when we complete an unfinished task, we search for the corresponding task in the queue. If we find the task, we mark the task as “completed”. When next time that task is dequeued, we discard that task and keep dequeueing until we find the task that has not been finished, then we finish that task.

5.3. Option 2: Design of a thread-safe disk and file system

When the disk, file, and filesystem are accessed by multiple threads, race conditions need to be considered. We use lock-based solution to avoid these problems. However, operations on lock needs to be atomic (like using test_and_set()) and achieving an atomic function is very

complicated (changes on low level are needed). To simplify this scenario, we replace the atomic function with assignment statements. We consider 2-3 lines of assignment statement is very fast and the probability that interrupt comes in this period is very low.

Disk access: We forbid different threads to write on the disk at one time. Therefore, we set a lock — any write operation on disk needs to acquire the lock first and only the thread gets the lock can apply lock operation.

File access: We forbid different threads to apply write-associated operations (write, rewrite) on the file at one time. Similarly, we set a lock for all File class. Other operations like read, reset, EoF, however, do not acquire the lock and can be applied by multiple threads synchronously.

Filesystem access: We forbid different threads to apply write-associated operations on the filesystem at one time. The write-associated operations includes create file, delete file, format file, and all the bitmap operations (get inode, free inode, get block, free block). Again, we use lock-based approach. Other operations, like loop-up and mount, are allowed to be apply by multiple threads synchronously.

5.4. Option 3: Implementation of a thread-safe disk and file system

As mentioned above, we set one lock for disk, one lock for file, and two locks for filesystem. The reason why two locks are needed for filesystem is that the bitmap operations and other write-associated operations (create/delete file, format file) are not mutually exclusive. Actually, they must be allowed to happen synchronously, because some bitmap operations are needed during other operations. For example, when we try to create a file, we need to get inode and get block operations. Also, permitting these operations to happen synchronously is harmless: they are trying to modify different regions of the disk, let alone the disk lock provides further protection.

Here we complete the interrupt handler. The interrupt could happen at any moment, even within the critical section of the scheduler. We never want the interrupt handler to do a context switch during the critical section of the scheduler.

However, we cannot simply disable the interrupt when we are in these sections. If we disable the interrupt and an interrupt comes in during this period, it will not do something like wait and remind us when the interrupt is enabled later — it will just disappear and we will lose all the information of that interrupt.

The problem is solved by adding a lock scheduler. Since there will be only one scheduler per single machine, the scheduler operation does not need to acquire the lock since it has the highest priority. Before critical section, we set the lock; after critical section, we free the lock. When an interrupt happens, it first check if the lock is set — if not, execute the handler code; otherwise, simply return. If a I/O task is unlucky enough, the disk get ready when the lock is set, the thread has to wait for its next turn of CPU to apply the I/O operation. However, the probability is very low — in most cases, the interrupt happens outside the critical section and the I/O operation gets to be applied right after the disk is ready. In other cases, we must sacrifices the fast return time to the security considerations.

Similarly from MP4 - bonus2, since there are context switch in an interrupt, we send EOI before the interrupt handler. The reason is explained in detail in the design document of MP4.

6. Other changes of the source code

Even though the bug has not been found out, the original frame pool does not work correctly. When trying to write on the memory allocated by frame pool, the content of the stack is overwritten. Roughly speaking, the filesystem class starts at 0x120000, the scheduler class starts at 0x130000 and the disk class starts at 0x140000. When set up a chunk of memory for an temporary buffer that starts 0x210000, any write operation on that buffer writes directly on the scheduler class, which certainly falls within the stack arrange. When using the frame pool from MP2, the problem is gone. Therefore, the frame pool of MP2 is used here.