MP5 Design Documentation

Name: Shiyang Chen UIN: 623009273

BlockingDisk

BlockingDisk eliminates “busy-waiting” entailed by SimpleDisk by putting the current thread on the blocking queue when the disk hasn’t been prepared for data thransfer and giving up CPU immediately. Besides, we use interrupt approach to check and complete I/O operation later. The interrupt handler will be discussed in more detail in Bonus section.
The BlockingDisk class is derived from the class SimpleDisk.We add a blocking queue and a write lock for all the BlockingDisk objects and override read and write functions. Here are some rough explanation on read and write function. More details can be found in comments
void read(unsigned long _block_no, unsigned char* _buf)
In this function, we put the current thread on the blocking queue and then send read request to the disk. Then we check if the disk is ready or not; if yes, we apply the read operation and remove the current thread off the blocking queue; otherwise, we give up the CPU. Every time the thread gains CPU, it checks the disk
void write(unsigned long _block_no, unsigned char* _buf)
This function is basically the same with the read function. One slight difference is we check if the write lock is released before we do anything: if yes, we let the current thread hold the write lock and proceed; if not, it means another thread is currently holding the write lock and doing write operation, in this case we let the thread wait until the write lock is freed

Filesystem

Here we use indexed allocation simply because this approach is similar to what we have done with page table in MP3. With indexed allocation scheme, each file owns its inode, which contains all the indices of blocks this file owns in the ascending order of block number.
With a 10MB disk, there will be 10MB/512B=20K blocks in total. If each block use one integer number to record, then 20K*4B=80KB is needed. Considering one block can fit 512 bytes, 80KB/512B=160 blocks are needed to store all inodes. Therefore the max size of a file is 128 integers/block*512B=64KB and there would be 10MB/64KB=160 files in total.
Similar to the bitmap implementation in memory management part, we use bitmap to mark whether a block is free or has been allocated. Since 10MB/512B=20K bits are needed and one block has 512B*8=4K bits, so 20K/4K=5 blocks are needed to store bitmap.
Besides functions have been declared in file_system.H, we add four more auxiliary functions: get_inode(), get_block(), free_inode(), free_block(); these functions are

used to allocate and free blocks and set corresponding bits in bitmap to 0 or 1.
Here are rough explanations on some functions within file_system.H

BOOLEAN Format(SimpleDisk* _disk, unsigned int _size)

In the function, we wipe off the existing filesystem on the disk and write a new, clean filesystem.
We format the disk as follows: first 5 blocks are used for store bitmap, the next 160 blocks are for inodes, the rest of blocks are all reserved for data file.

BOOLEAN Mount(SimpleDisk* _disk)

In this function, we initialize member variables and read bitmap into the memory to save number of I/O operations in the future. The size of the bitmap is 5 blocks*512B= 2560B, which is acceptable for the memory.

BOOLEAN LookupFile(int _file_id)

First we need to check the corresponding bit of the inode bitmap: If it is 0, it means the inode is allocated, i.e. the file exists; otherwise the file doesn’t exist.
Since the first 5 bits are used for blocks storing bitmap, the inode bitmap starts at the 6th bit. Given the file ID starts at 0, we add an offset to a given file ID whenever we make modification on the inode bitmap
Other functions are self-explanatory and more details can be found in comments.

File

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

Thes function reads the file of _n bytes starting from the current position. First we read the inode block, and calculate which block to start and to stop. Since only sequential read is supported and we can only read an entire block of data once, we set aside a temporary buffer to store the data and cut the data to where it needs to start and stop according to current_position variable, and then copy them onto the given buffer.

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

The write operation is a bit tricky. Since we can only start writting at the current position, to prevent overwritting the previous data we first read onto a buffer the block in which the current position is and then append what we are going to write onto the buffer.
Besides, if a file runs out of its current block, we need to allocate a new block for the file, modify the inode (in memory) accordingly and then write the modified inode onto disk.

void Rewrite()

This function frees all the blocks a file owns except the first block (because every file must have at least one block). Then the content on the first block is wiped off, i.e. we write all 0s into that block and reset the current position and file size.

Execute the test

In the exercise_file_system() we do the following.
When this function is called the first time, we format the disk and mount the filesystem. Then we run an infinite loop; only after all statements in the loop are

executed once can thread3 yields CPU. Therefore whenever thread3 gains CPU it tests the filesystem, which means the filesystem will be continuously tested.
In the infinite loop, we do the following.
Create two files: one is for small file test and the other is for large file test (large file means its size is larger than one block).
Write the small file the string “Operating System is very hard!\n”.
Append to the small file the string “But Shiyang Chen is smart enough!\n”.
Reset the small file.
Read the small file.
Rewrite the small file.
Write the small file the string “Sorry I lie about the second half.\n”.
Reset the small file.
Read the small file.

After passing small file test, we test the large file. We do the following.
Write the large file 700 random characters
Reset the large file.
Read the large file.

Note that we set two macros for testing: if uncomment #define _SMALL_FILE_TEST, the program will stop after testing the small file; if uncomment #define _LARGE_FILE_TEST, the program will stop after testing the large file; if uncomment neither of them, the program won’t stop and switch between 4 threads and you will see testing results flash by on screen; if uncomment both of them, the program will stop after testing the small file. More details can be found in comments

Bonus Options

Option 0: Using Interrupt for Concurrency
Whenever the disk gets ready, the interrupt 14 will be triggered. We register an interrupt handler function beforehand, so that function will be called and then I/O operation will be executed.
In the interrupt handler, we preempt the current thread and yield the CPU to the thread taken out from blocking queue. But instead of putting the current thread in the tail of the ready queue, we put it in the head of the ready queue so that after interrupt is over the previously preempted thread can continue to its job.
However, there is a chance that the disk could already be ready during the period from when the thread send request to when it yield to another thread. That is to say, the thread we are about to take out from the blocking queue happen to be the current thread. If so, there is no point letting a thread preempt itself. Rather, we simply return (no context swith) and let the current thread continue to do his job (which is finishing I/O operation)
Option 2: Design of a thread-safe disk and file system
In order to prevent race conditions, we use mutex lock here. Given that operations on mutex lock has to be atomic and writing an atomic function needs changes on low

level stuff thus complicated, we replace the atomic function with assignment statements to simplify the situation. We assume 2-3 lines of assignment statements is very fast to execute thus there is an odd chance that interrupt comes in this period.

Disk access: simultaneous writing operations on the disk by multiple threads are forbidden. Therefore, we set a write lock and prescribe that any write operation on the disk needs to acquire the lock first and only then can the thread do write-related operation.
File access: we forbid different threads to do write-associated operations (write, rewrite) on the file at one time. Similarly, we set a write lock. Other operations like read, reset, EoF does not need the lock and can be done by multiple threads simultaneously.
Filesystem access: we forbid different threads to do 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 set a write lock. Other operations, like loop-up and mount, are not restricted.
Option 3: Implementation of a thread-safe disk and file system
There are several things we should notice when it comes to implementation.
One thing is while class disk and class file only need one lock, class filesystem needs two. The reason 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 first.
Another consideration concerns interrupt handler. Given the interrupt could happen at any moment, it might happen 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 so an interrupt coming during this period will just disappear and we will lose all the information of that interrupt.
Our approach is to give scheduler a lock. Before critical section, we set the lock; after critical section, we free the lock. When an interrupt happens, it first checks if the lock is set: if not, execute the handler code; otherwise the handler function simply return. If an 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 finish the I/O operation. Luckily, such probability is quite low; in most cases, the interrupt happens outside the critical section and the I/O operation gets executed right after the disk is ready.
The last consideration concerns EOI signal. When the interrupt handler is ready to deal with an interrupt, in dispatch_interrupt function, it looks at the handler table, finds the right handler, and then calls the corresponding handler function. However, this function will not be returned in a short time. If the new thread does not pend from here, it will return from somewhere else so the EOI signal will not be sent to PIC

soon. Also, if this is the case, when this thread regains CPU later, it will return from here, causing an unreansonable EOI signal sent. To avoid such case, we send EOI signal before the handler function (context switch) is called.

Other notices

To accommodate the EOI discussed above, I modify the dispatch_interrupt function in interrupts.C
I fail bonus operation 0. So even though you will see a macro called _MIRROR_DISK_ in kernel.C, don’t try to uncomment it
The original frame pool does not function well. When trying to write on the memory allocated by frame pool, the content of the stack is overwritten. Roughly speaking, the filesystem object starts at 0x120000, the scheduler object starts at 0x130000 and the disk object starts at 0x140000. When set up a chunk of memory for a temporary buffer that starts at 0x210000, any write operation on that buffer writes directly on the scheduler object. So I switch back to frame pool design in MP2, the problem is gone.
Makefile.linux64 is also tailored accordingly.