OS611_MP6_Report

Name: Wei-Che Hsu
UIN: 826001725

In this MP, I implemented the basic required disk function. Besides, I implemented some bonus options, including OPTION 1: Support for Disk Mirroring, OPTION 3: Design of a thread-safe disk system, and OPTION 4: Implementation of a thread-safe disk system.

When grader tests my submission, please put the files from my mp6 folder into the testing folder. These files includes: **blocking_disk.C, blocking_disk.H, disk_ints_handler.C, disk_ints_handler.H, mirrored_disk.C, mirrored_disk.H, scheduler.C, scheduler.H, thread.C, bochsrc.bxrc, and makefile.** I will describe what I do in these 11 files in the later part of my report. Also, kernel.C in my mp6 folder works well under my simulation environment, and I just attach it in my submission.

In my submission, the interrupts is disable, and the disk operation is mainly according to the status register from disk controller. I only use the FIFO scheduler from my MP5, and I don't take time on implementing new scheduler with interrupts currency disk system. In addition, I would not like to reply interrupts and check busy status to control disk operation in very short duration. I afraid of that sending EOI to disk controller after accepting irq 14 / irq 15 would cause disk status changing although there is not any data incoherent problem happening when I send EOI to irq14 / irq 15 requests. I don't verify this case very clearly. Therefore, I comment the below macro to disable interrupts for stable consideration. (Default setting in my submission)
thread.C: Line26 //#define _EN_INTERRUPTS_TH_
kernel.C: Line29 //#define _EN_INTERRUPTS_

First, I would like to describe my work for basic feature of disk function. When the status of disk is not ready for reading or writing a block, my blocking_disk uses the functions of scheduler, resume and yield, to make the current thread go back into ready queue waiting and to make the next thread obtain CPU. Therefore, the CPU usage ratio under this method becomes higher than that under busy waiting scheme.

Second, for my mirrored disk, I connect the mirrored disk to ATA1 such that I can read busy status of mirrored disk form disk controller while the main disk reports its status. Now, my disk system uses two ATA controllers: ATA 0 is for blocking_disk(main disk), and ATA1 is for mirrored_disk(sub disk). Users in kernel.C only access blocking_disk,

and in blocking_disk.C my blocking_disk will set commands to mirrored_disk.

In read function in blocking_disk.C, when user sets read command to blocking_disk.C, the read function will use issue_operation to send main disk read signal through ATA0, and then it will call read function of mirrored_disk to send mirrored disk read signal through ATA1. After issuing read signal to both disks, program will check the busy status of main disk and the busy status of mirrored disk. If both of them are not ready for read, the current thread will resume itself to thread ready queue and yield CPU. Otherwise, program will read data from one ready disk. By the way, if both of them are ready to read, the program will only read main disk.

In write function in blocking.C, the write function will use issue_operation to send main disk write signal through ATA0, and then it will call write function of mirrored_disk to send mirrored disk write signal through ATA1. After issuing write signal to both disks, the overall write function completes until writing task in main disk and writing task in mirrored are done. Before finishing each disk's task, program will check whether each disk's status is ready for writing. If it is ready, just do write action. Otherwise, skip this disk at this moment. Then, the write function in blocking_disk.C will resume the current thread and yield CPU at the check point if any disk has not finished the write operation.

I use the below form to describe which functions can control flow and which functions only do pure read/write operation.

How to issue commands:

For main Disk, BlockingDisk::read/write ➔ BlockingDisk::issue_operation.

For sub Disk, BlockingDisk::read/write ➔ MirroredDisk::read/write ➔ MirroredDisk::issue_operation

Function Name	Control flow	Read/Write operation	Issue Read/Write
BlockingDisk::write	V	V	V
BlockingDisk::read	V	V	V
BlockingDisk::issue_operation			V
MirroredDisk::read			V
MirroredDisk::read_action		V	
MirroredDisk::write			V
MirroredDisk::write_action		V	
MirroredDisk::issue_operation			V

Next, I will describe my design regarding thread-safe disk system. Because I am a beginner for thread-safe design, I choose the simple strategy, coarse grained lock, in the BlockingDisk object. I assume this system can only create one BlockingDisk object

because there is only one main disk (mirrored disk is hidden from normal user). Besides, mirrored disk is controlled by the functions in blocking_disk.C. Therefore, if I set up lock into these functions, BlockingDisk::write and BlockingDisk::read, I can confirm the main disk and the mirrored disk resource are thread-safety. I draw the system picture in Figure 1, and the critical sections include control accessing block, main disk block, and mirrored disk block.

How to design N-thread lock? I found the description and pseudo code in the website ref [1] and ref [2]. I take filter-lock, which is generalized version of Peterson Algorithm, to build up my lock. There are 4 threads in my disk system. If one thread would like to obtain the lock, it should pass through 3 levels check. At each level, it sets itself as victim leading to that the other thread in higher level or the same level has higher priority than this thread. The only one thread in the highest level can obtain the lock to use the main disk and the mirrored disk. How it works? For example, now I set read disk operation in thread 2 and 3. If read operation is not finished in thread 2 because none of disks returns ready signal, thread 2 will yield CPU. Then, thread 3 is the next thread, and it would like to obtain the lock in BlockingDisk object. However, thread 2 owns the lock, so thread 3 can't obtain the lock. Here, I set that if any thread finds that itself can't own lock at the current moment, this thread should yield CPU instead of busy waiting in the lock function. I implemented the FilterLock class in blocking_disk.C. The main functions are "acquire" and "release". By the way, there is maximum thread number can be defined in blocking_Disk.H: Line19 #define Thread_N 4.

Now, I set it as 4.

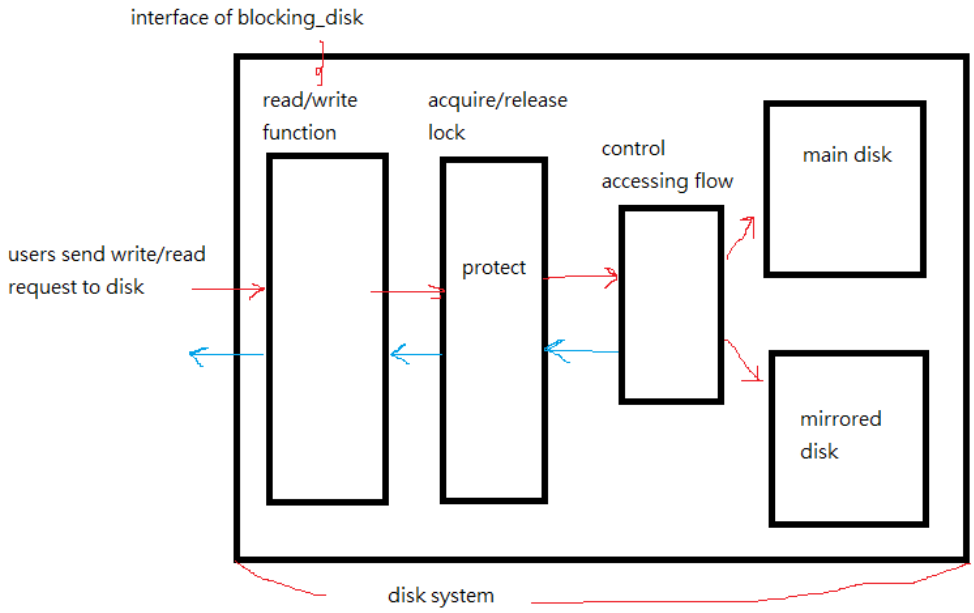

Figure 1. Disk System with Lock protection

Last, I will briefly describe what I do in each function in my submission.

blocking_disk.C and blocking_disk.H:

1. Access main disk through ports of ATA0.
2. Modify read/write function with additional control flow to mirrored disk, and when the disk operation is blocked, the current thread will yield CPU.
3. One BlokingDisk Object create MirroredDisk Object and FilterLock Object.

disk_ints_handler.C and disk_ints_handler.H:

It is dummy function now, which is used for registering the interrupt handler of irq 14 and irq15. Thus, interrupts handler will send EOI signal for replying irq14 and irq15 events. However, this object is not created in kernel.C by my default macro setting.

mirrored_disk.C and mirrored_disk.H:

1. Access Mirrored disk through ports of ATA1.
2. Read/write function just calls issue_operation to send command.
3. Read_action/write_action is real disk operation.

scheduler.C and scheduler.H:

FIFO scheduler from MP5, the interface is the same as that in my MP5 submission.

thread.C:

It is almost the same as that in my MP5 submission, except one macro which makes interrupts not enable after thread creating.

bochsrc.bxrc:

To implement mirrored disk, I moved d.img to ATA1 controller. Thus, I change some part in #hard disk region.

makefile:

Because I add some files, I have to modify this file to compile them correctly.

kernel.C:

1. I enable _USES_SCHEDULER_
2. Change SYSTEM_DISK is the pointer to BlockingDisk object instead of the pointer to SimpleDisk Object ;
3. I disable interrupts by commenting macro _EN_INTERRUPTS_. If it is enabled, the buf used in thread 2 will allocate in heap to avoid stack overflow problem. Besides, I have use disk_ints_handler object to register interrupts handler for irq

14 and irq 15.

Reference:

[1] https://www.cs.rice.edu/~vs3/comp422/lecture-notes/comp422-lec19-s08-v1.pdf

[2] http://courses.csail.mit.edu/6.852/03/lectures/Nir-mutex-slides.pdf