# CSCE 611

## Machine Problem 6: Simple Disk Device Driver

Submitted By – Vaibhav Rawat
UIN: 626008171

I completed this assignment including all the bonus options of 1, 2, 3 and 4. Code for each option is provided in a separate folder.
Main task in this assignment is to work on top of existing source code for simple kernel level device driver. Implementation in simple_disk.C is naïve and uses busy wait to check whether disk is available or not in the while loop. As a result, CPU resource is blocked by read and write operations to ata0-master disk which needs to be correct. To tackle this, I added code to wait_until_ready function of blocking_disk.C which inherits simple disk class to avoid blocking read and write. For this, I created a blocking queue associated with disk object. Whenever a disk read/write request is raised, wait_until_ready function is called which places the current thread in blocking queue and yields execution to next thread in ready queue. Current thread is only present in blocking queue and not in ready queue till disk is ready to serve the next request. Since yield function of scheduler is called regularly, I leverage the call to yield to check if the disk is ready to serve a request. If it gets ready, front thread from the blocking queue is removed and is placed in the ready queue to execute when it turns arrive in a FIFO manner. For all this to work, I have included scheduler.C and thread.C code from previous MP.

By changing putch to puti and outputting disk contents I was able to verify when disk operation is performed after a disk read/write request. I wrote values to disk and outputted other than zero. I also verified that master disk is being read/written through following output and the green/red blinking of master disk written on Bochs interface for read/write operations. Output file is shared.

Function 2 is invoked, puts thread2 in blocking queue and yields to function 3

```
FUN 2 INVOKED!
FUN 2 IN ITERATION[0]

 Reading a block from disk...
THREAD: 2
FUN 3 INVOKED!
FUN 3 IN BURST[0]
FUN 3: TICK [0]
FUN 3: TICK [1]
```

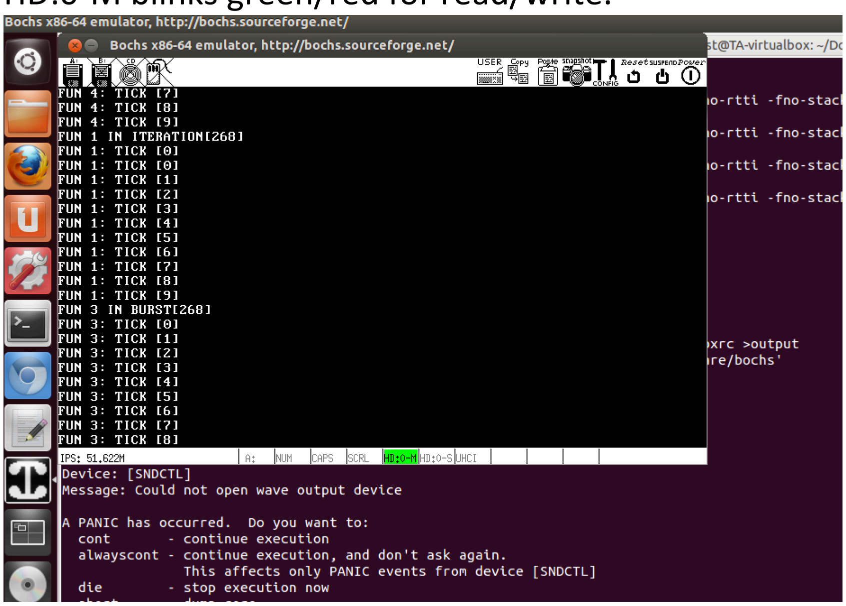
Once func2 executes after disk becomes ready, read operation is performed and For write request thread is enqueued in blocking queue.

```
FUN 3: TICK [9]
Reading Operation
000000000000000000000000000000000000000(
000000000000000000000000000000000000000(
000000000000000000000000000000000000000(
 Writing a block to disk...
FUN 4 IN BURST[1]
```

Write operation is performed by thread after disk becomes ready, in fifo order.

```
FUN 3: TICK [9]
Writing Operation
FUN 4 IN BURST[2]
```

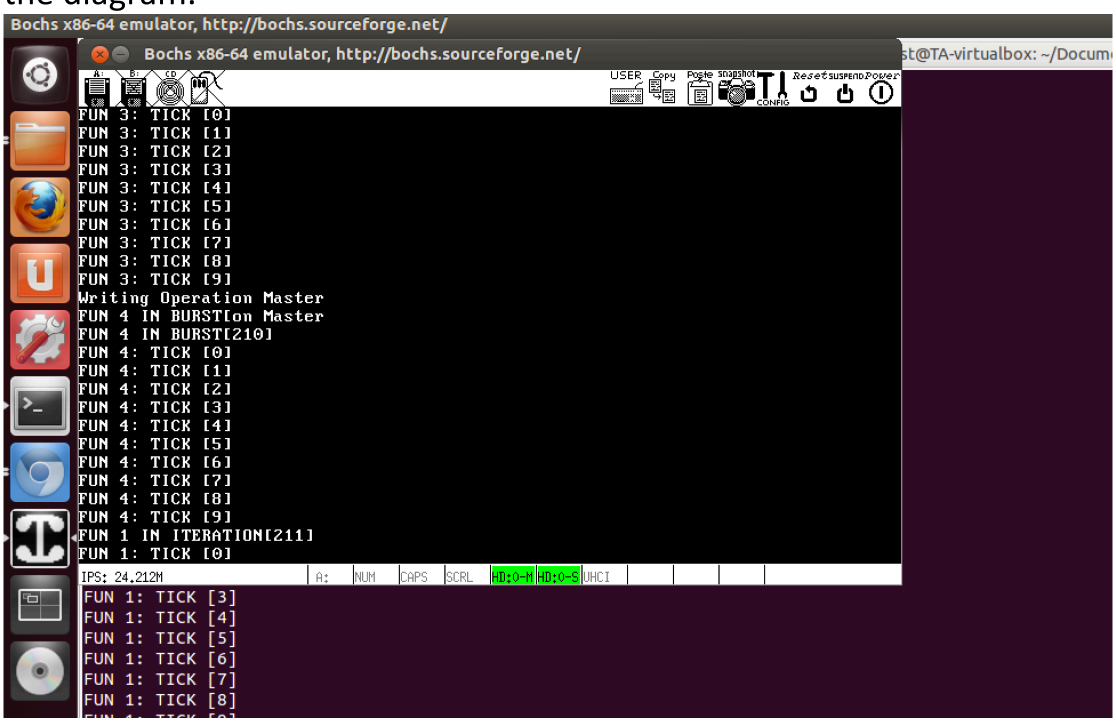
HD:0-M blinks green/red for read/write.



For Option1, I implemented my changes in mirrored_disk.C. MirroredDisk class inherits simple disk class and overrides read and write operations. For write, it performs write to both the ata0 slave and master disks. It raises a write operation for both master and slave disks by sending master and slave id to 0x1F6 port and places corresponding threads in the blocking queue. Yield keeps checking when master and slave disks become ready and once they turn to ready, it dequeues from blocking queue and places the corresponding thread to ready queue to be executed in a fifo manner. For read operation, it raises read request from both master and slave disks by sending master and slave ids to port 0x1F6 and places a thread in blocking queue. Yield checks when either of the disk becomes ready and performs the read operations using the disk that is ready first.

I verified that master and slave disk is being read and written through following output and the green/red blinking of master and slave disk written on Bochs interface for read/write operations.

Both slave and master disks are read and write as HD:0 -M and HD:0-S blinking in the diagram.




Similar outputs are seen for reading and writing operations. Output file is shared.

// SINGLE READ

```
FUN 3: TICK [9]
Reading Operation
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
 Writing a block to disk...
FUN 4 IN BURST[1]
```

// WRITE TO MASTER

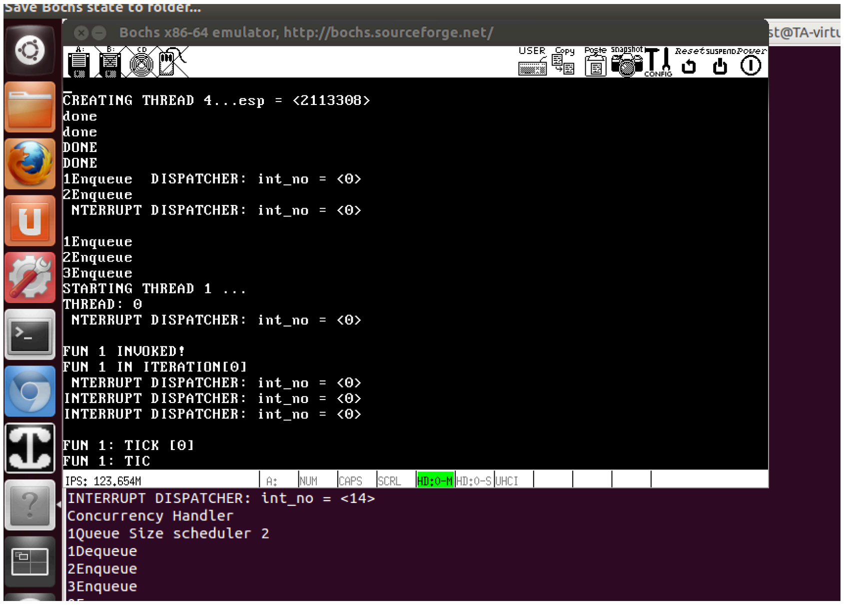
```
FUN 3: TICK [9]
Writing Operation Master
FUN 4 IN BURST[2]
FUN 4: TICK [0]
```

// WRITE TO SLAVE

```
FUN 3: TICK [9]
Writing Operation Slave
FUN 4 IN BURST[3]
FUN 4: TICK [0]
```

For Option2, I see that when disk becomes ready - irq 14 is raised, set through bochsrc.bxrc file. To handle irq 14, I implemented a Concurrency handler in kernel.C file inheriting from InterruptHandler. I registered this concurrency handler to irq 14 using register_handler API. For this option, read and write operations places the corresponding thread in blocking queue like before but I am checking for disk ready whenever this interrupt is raised. And when the disk becomes ready, I leverage this interrupt to deque the corresponding blocked thread in blocked queue and enqueue it to ready to be executed in a fifo manner. I verified my changes by outputting the interrupt number. Through the logs I can see that whenever interrupt 14 is raised blocked thread is placed into ready queue and after some iterations the read operation is performed as shared in the snapshot. Output file is shared.

// HD:0-M is in operation with interrupt 14.




// DISK READY – INTERRUPT 14 HANDLED BY CONCURRENCY HANDLER

```
FUN 2 INVOKED!
FUN 2 IN ITERATION[0]

 Reading a block from disk...
Put in queue
INTERRUPT DISPATCHER: int_no = <14>
Concurrency Handler
```

// READING OPERTATION PERFORMED

```
Reading Operation
0000000000000000000000000000000000
0000000000000000000000000000000000
0000000000000000000000000000000000
0000000000000000000000000000000000
```

For Option3 and Option4, I designed a thread safe system which ensures concurrent operations to the disk are handled safely. Using a separate blocking queue for threads waiting for input/output operations, ensures that requests are handled correctly in fifo manner by single thread. I augmented thread safety for

multi-threaded disk access or round robin scheduling, by locking access to disk's blocking queue. For this, I implemented a crude locking class with functionality to lock and unlock in blocking_disk.H. During read and write operations I first lock and then proceed to call issue_operation to disk controller for read/write. Then I proceed to add the corresponding thread to blocking queue still holding the lock. Above operations are performed atomically, to ensure the fifo ordering of issue_operation and corresponding thread in queue. I then exit the lock using unlock(). Mutual exclusion is also achieved using locks when thread is deque from blocking queue and enqueued in ready queue to avoid race conditions.
I have verified my changes by having two functions function2 and function3, reading and writing to disk. I found that their reading and writing threads are properly shifted between blocking queue and ready queue. As seen from the snapshot, first thread2 comes to read, places itself with read request in queue and when thread3 comes it does the same thing. Following that when disk becomes ready thread2 starts reading, following which thread3 resumes and performs reading.

// Fun2 invoked – puts thread for read to block queue and yields

```
FUN 2 INVOKED!
FUN 2 IN ITERATION[0]

 Reading a block from disk...
THREAD: 3
```

// Fun 3 invoked – puts thread for read to block queue and yields

```
FUN 3 INVOKED!
FUN 3 IN ITERATION[0]
Reading a block from disk...
FUN 1 IN ITERATION[1]
```

// Reading by Function 2

```
Reading Operation
0000000000000000000000000000000
0000000000000000000000000000000
0000000000000000000000000000000
```

// Reading by Function 3

```
Reading Operation
0000000000000000000000000000000000000
0000000000000000000000000000000000000
0000000000000000000000000000000000000
```