

# Implementation of Primitive Disk Device Driver

Vishruth Raghuraj Gollahalli

(Bonus implemented: Option 2: Interrupt handling for Interrupt 14 to avoid seeing multiple interrupt messages on console, Option 3: Possible design of thread safe system)

To avoid the busy waiting, the following modifications were made:

1. `BlockingDisk::wait_until_ready()`:  
This function first checks if the disk object is ready. If it is not, then instead of busy waiting, the disk object is added to the ready queue of the scheduler, the size of the disk ready queue is updated and finally the CPU is yielded to the next thread at the head of the ready queue.
2. `BlockingDisk::is_ready()`:  
This function simply calls the underlying `SimpleDisk::is_ready` function
3. The read and write functions are updated so as to first issue the operation through `SimpleDisk`'s issue operation function (this function was made public in the `kernel.C` file). Then the `wait_until_ready` is called which is the `BlockingDisk` implementation that does not use busy waiting. Finally, the read and the write operations are similar to the operations in `SimpleDisk` implementation.

In the `kernel.C` file, the following modifications were made:

1. `_USES_SCHEDULER` macro uncommented to make use of the scheduler
2. Included `blocking_disk.H`
3. Pointer added to `BlockingDisk` instead of `SimpleDisk`
4. Registered interrupt 14 with Interrupt handler
5. Created new `BlockingDisk` object rather than `SimpleDisk` object and added this to the scheduler queue

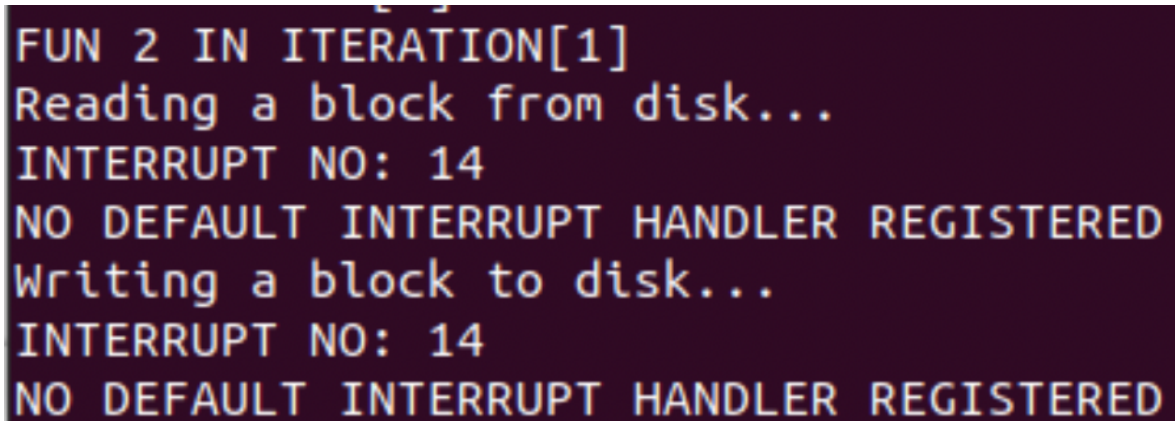
In `makefile`, changes were made to add `scheduler.o` target to enable compilation of the program with the scheduler code.

Possible design for thread safe system (Bonus 3 attempt):

In the case of the multithreaded system that we have, one way of ensuring only one thread can make read/write operation at a time is by queueing up the threads in a new ready queue, this time, the disk ready queue. The threads can be made to give up the

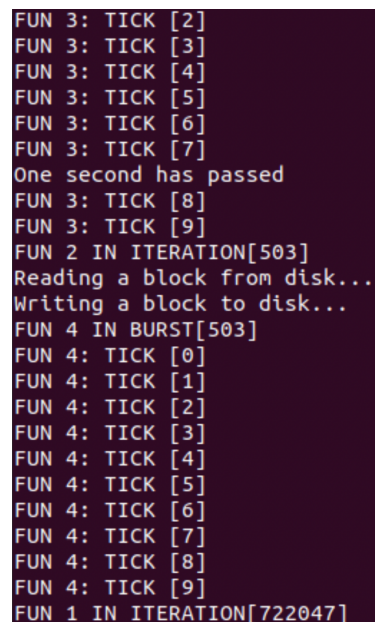
CPU after the read/write call has been added to the disk queue.  
Race conditions in the case of a multicore system can be handled by using mutex locks or test-and-set mechanisms to ensure at a time, only one particular processor can run a thread that makes changes to the queue. I have not implemented a lock based mechanism for thread-safe systems in this machine problem.

## SCREENSHOTS



```
FUN 2 IN ITERATION[1]
Reading a block from disk...
INTERRUPT NO: 14
NO DEFAULT INTERRUPT HANDLER REGISTERED
Writing a block to disk...
INTERRUPT NO: 14
NO DEFAULT INTERRUPT HANDLER REGISTERED
```

The above screenshot shows the output without registering the event handler



```
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
One second has passed
FUN 3: TICK [8]
FUN 3: TICK [9]
FUN 2 IN ITERATION[503]
Reading a block from disk...
Writing a block to disk...
FUN 4 IN BURST[503]
FUN 4: TICK [0]
FUN 4: TICK [1]
FUN 4: TICK [2]
FUN 4: TICK [3]
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 1 IN ITERATION[722047]
```

Above screenshot shows the output on registering event handler for interrupt 14 and interrupt handling is shown as “One second passed” message is showing that interrupt handling works correctly. Further no message is visible for not registering handler like in the previous screenshot