Design Document
Simple Disk Device Driver

CSCE 611
Operating Systems
MP6 - Fall 2018
By
Himanshu Gupta

Design for device driver
Bonus Attempted – **Option 3 and Option 4**
Note: makefile is modified for a new queue class.

I have implemented a basic device driver which is done according to the design for option 3. First, we need to update the scheduler class with a blocking disk object, which is added from the kernel file. We write an api to add the disk object in the scheduler class. The yield function will check the disk object first, verify if the disk has anything to read or not and then schedule that thread.

Next, we implement the blocking disk class, where it will contain a queue of blocked threads on the disk. We add a queue size member as well which will track the total number of threads in the queue. The extra functions which we add to the blocking disk class are:

- Disk enqueue - this will add the thread to the blocked thread queue and update the counter for the number of threads the blocked thread queue.
- Is ready- this function call the is_ready function from the simple disk class.
- Wait until ready- this is the main api which will first check if the disk is ready or not. If not we simply take this current thread and add it to the blocked thread queue. After this we yield this current thread and now it is not on the scheduler queue.

The thread added to the blocked queue thread will only be given CPU when the the disk returns true for is_ready call. Otherwise, we simply loop throught the threads which are added in the scheduler queue, the thread waiting on the disk I/O are added in a separate queue so they will not be given any cpu. This way we have avoided the busy waiting loop in the original simple disk implementation. The threads waiting on disk I/O will not get any cpu because the scheduler will only dequeue from the scheduler queue.

When the yield function finds the disk ready, it will check from the size of the queue that if we have any thread on the blocked thread queue. If yes, then it dequeues the thread from that queue, decreases the size of the blocked thread queue and then dispatches the cpu to that thread. This thread then performs the read or write operation for what it was blocked and then yields. This way we remove the busy waiting of the threads.

I have also updated the kernel file to make blocking disk object instead of simple disk.

Design for option 3: This design of device drivers will safely run multiple thread for a single processor. Since in this assignment we only have one processor, multiprogramming is achieved only by implementing multiple threads. As several threads call the read/write function we queue the read write requests to a blocked queue and make the thread give up the cpu. This way only a single read/write request is handled at a time.
For multiprocessor system, we will race condition between several cpu to access the I/O queue and will need some form of protection. For this we will need to implement a thread safe queue and a lock on the I/O device. The thread safe queue will make sure that different processors do not update the queue in wrong manner (if they are in a race condition) and a lock on the I/O

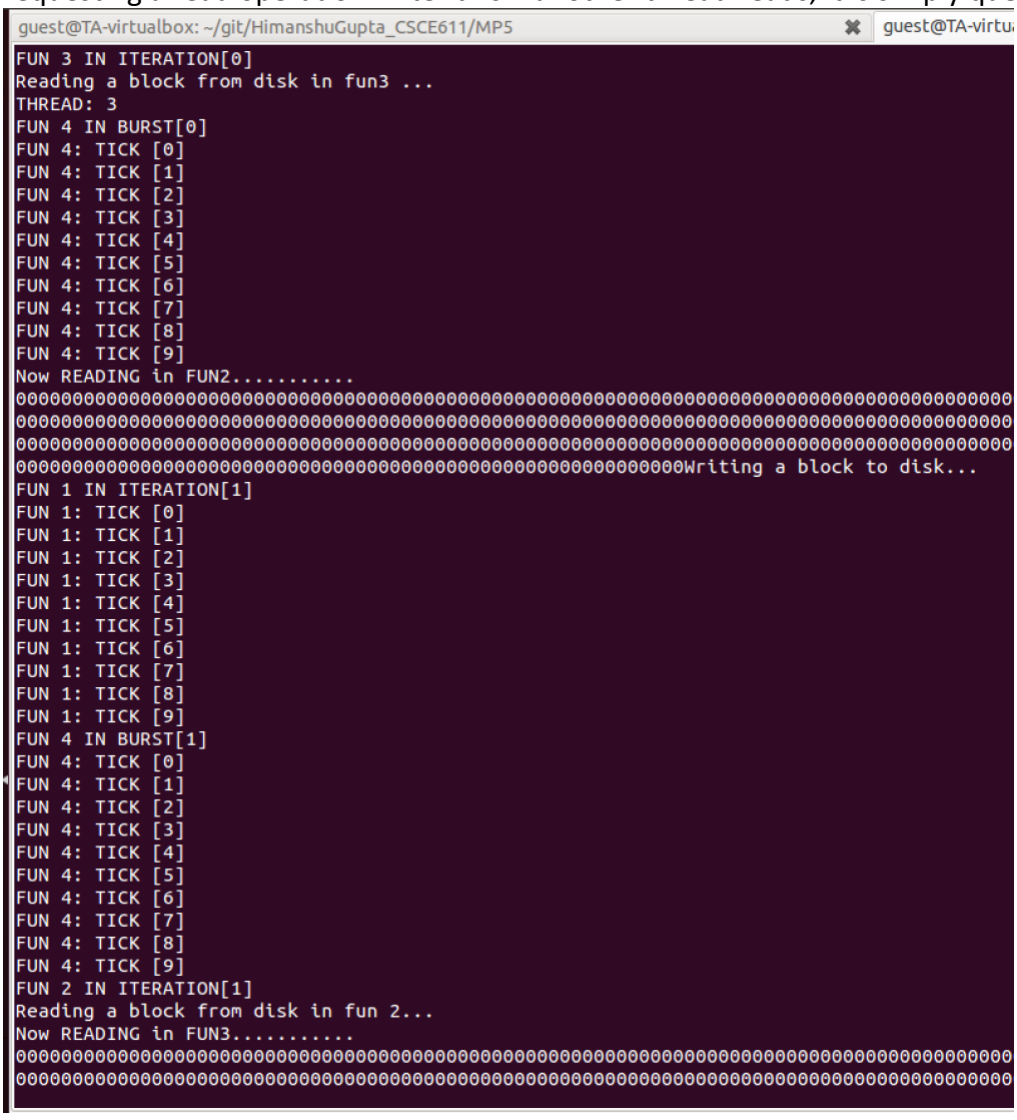
device will ensure that we do not give up the resource if it is busy. This implementation cannot be tested in our current setup as we only have one processor.

Testing

For Normal testing purpose, I was able to avoid the busy waiting loop and make the cpu yield once it checks that the device is not ready, However for testing multiple threads read/write, I have updated the fun3 function also where it does read write operation.

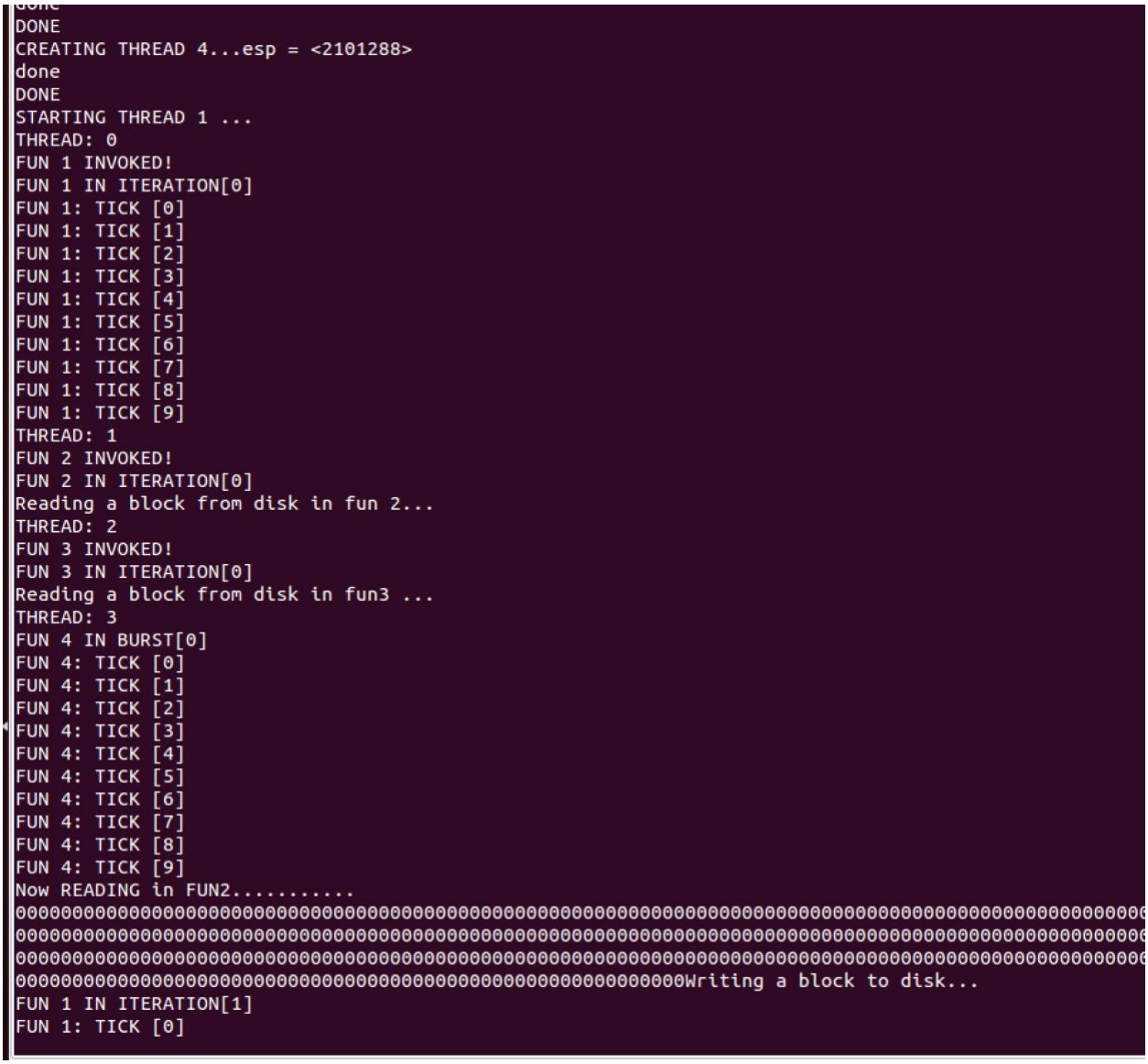
The order of sequence is maintained by the queue as the thread is made to give up the cpu after requesting the read write operation

The line where we see “Reading a block from disk in fun3” in the screen shot below is where the read request is made. THIS IS NOT the actual read operation but the debug that the thread is requesting a read operation. After this if another thread reads, it is simply queued.



The next line later which says “Now READING in FUN2” is when the actual read is happeneing from the device. This shows that the FIFO order is guaranteed.

The condition of multiple threads accessing from option 3 is also tested in this case. As there are 2 threads requesting read operation, we made them queue up in the blocked thread. The actual operation is sequential and in FIFO order and therefore the thread safe case is tested.



The following files are used in this MP6, if any of them is missing please contact me.
Makefile
Queue.H
Scheduler.C
Scheduler.H
blocking_disk.C

blocking_disk.H
kernel.C