Design Document
Primitive Disk Device driver

CSCE611
Operating Systems
MP6- Fall 2022

by

Tanu Shree

DEPARTMENT OF

COMPUTER SCIENCE AND ENGINEERING

TEXAS A&M UNIVERSITY, COLLEGE STATION

This Machine problem is mainly about investigating the kernel-level device drivers on top of a simple programmed-I/O block device.
Following things are achieved as part of this problem:

1. Complete I/O operations without busy waiting (basic MP)
2. Mirroring the Disk (bonus option 1)
3. Design of a thread safe disk system (bonus option 3)
4. Implement thread safe disk system (bonus option 4)
5. Use interrupt for concurrency (bonus 2)

The source file was provided with the simple disk implementation, in which the given block device uses busy waiting to wait for I/O operations to complete. Following implementation were done:

1. **Complete I/O operations without busy waiting (basic MP)**

   In this, the expectation is to support same blocking read and write operations as the basic implementation, but without busy waiting. To achieve this, a device called BlockingDisk is implemented. This class is derived from simpleDisk class (existing low-level device). The idea is that, when a thread calls the read/write operation, it should not block the CPU while waiting for the disk to complete the operation. Instead, it should yield the CPU.
   Changes are done in blocking_disk.H and blocking_disk.C files:
   Added, new method wait_until_ready (), in which it checks that device is ready or not. If not, the scheduler first puts the thread in the queue and then yields the CPU. Sometime later, when the thread returns it again checks if the device is ready and the process repeats. Resume () and yield () are called here.
   Method is_ready_blocking () returns if the device is ready or not. The functions read () and write () are same as previous.

2. **Mirroring the Disk (bonus option 1):**

   Mirroring disk is implemented by inheriting the Blocking disk. In this, two instances of blocking disk are created (MASTER and DEPENDENT) in the constructor. Data is read from either of the two disks, whichever is ready first. With the write operation, data is written in both the master and dependent disks.

   Changes are done in blocking_disk.H and blocking_disk.C files.
   MirroringDisk class is derived from public blockingDisk class. Two blocking disk instances MASTER_DISK and DEPENDENT_DISK are created in the constructor. Method wait_until_ready () is created, in which it checks if either of the two disks (MASTER and DEPENDENT) is ready. If not, the scheduler yields the thread.

Method issue_operation () is defined with Disk ID passed (this method was taken from simpleDisk).
In function read (), two read operations are issued, both for MASTER and DEPENDENT and then wait_until_ready is called. The data is read from either of the disk which is ready first.
In the write () method, blockingDisk write is called for both instances MASTER_DISK and DEPENDENT_DISK and the data is written to both.

3. **Design of a thread safe disk system (bonus option 3)**

To avoid the race condition, one approach can be to use lock in the critical section. Here the critical sections are the read and write operation. So, for that 'test and set' lock implementation can be done. The lock is acquired before the read /write operation is issued and is released once the read/write operation is complete. This ensures that race condition among the concurrent threads is avoided.

4. **Implement thread safe disk system (bonus option 4):**

The above-mentioned design is implemented in the blocking disk basic MP implementation. The changes are done in blockingDisk.C file.
TestAndSet () is defined in which the lock variable is passed, which is stored in temp variable and lock is set to 1. This function returns temp variable.
Init_lock() initializes the lock with value 0. It means there is no lock.
Lock_acquire (): It checks the status of the lock. If lock set to 0, i.e., while condition is false, it comes out of the loop and acquires the lock and enters critical section. If the lock is 1, the while loop continues until it is false. It does not let the other thread to enter the critical section.
Lock_release(): It sets the lock back to 0.
In blockingDisk::read(), first lock_aquire() is called before the simpleDisk::read() is called. And then, lock_release() is called.
Similary in blockingDisk::write, first lock_aquire() is called before the simpleDisk::write() is called. And then, lock_release() is called.

5. **Use interrupts for concurrency (bonus option 2):**

The interrupts indicate that disk needs attention and using interrupts polling time can be reduced i.e., instead of checking the status disk regularly, if the interrupt handlers are registered, the thread in waiting blocking queue gets popped out and completes the I/O.
To implement this, changes were done in blockingdisk.H, blokcingDisk.C and kernel.C.
In kernel.C, the interrupt handler is registered. In blockingDisk files, a waiting queue named blocking_Q is created and is used to store the blocked threads.
Wait_until_ready (): In this when read/write operation is issued, and interrupts enabled, the thread gets pushed in the blocking queue and yield. If not enabled, the regular resume and yield ().

handle_interrupt (): In this method, when interrupt is handled, the thread from the blocking queue is popped out and then is resumed.

Testing:

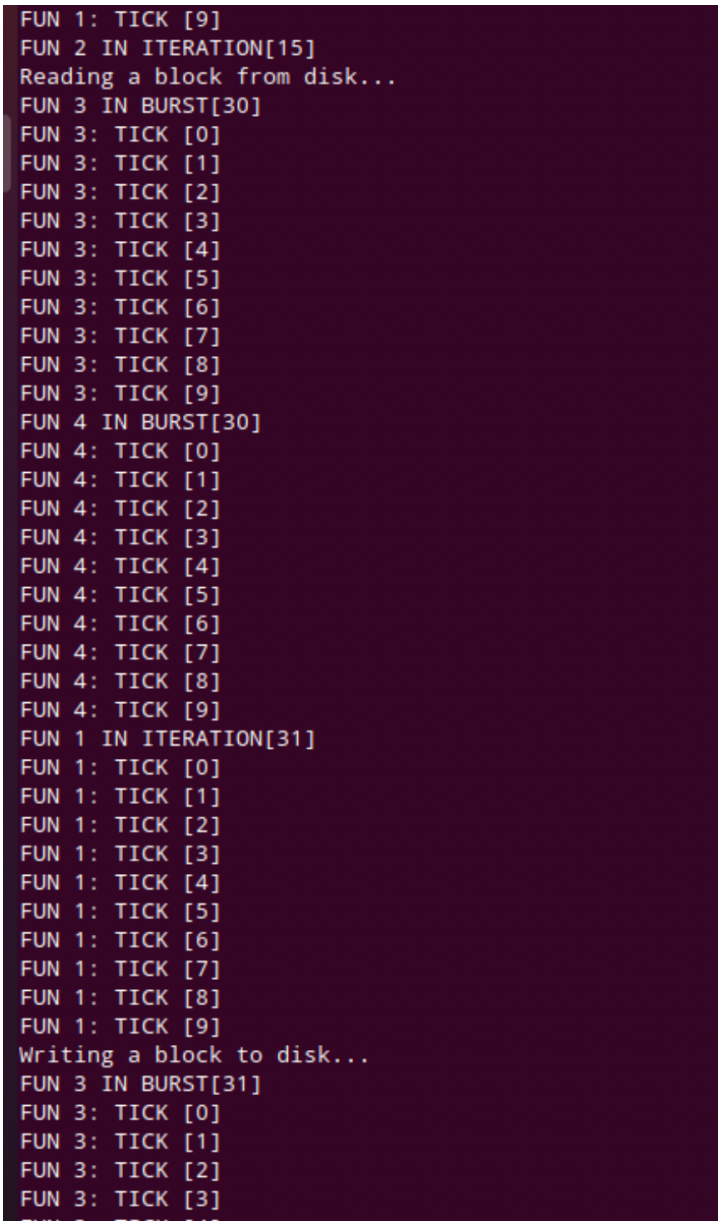
To test the basic MP implementation, changes are done in kernel.C file. Blocking_disk.H is included and an instance of blocking_disk is created and simple_disk is removed.
For mirroring, thread synchronization and interrupt handling, uncomment _USES_MIRRORING_ macro in kernel.C and uncomment INTERRUPT_ENABLE and ENABLE_THREAD_SYNC macro in blockingDisk.H file. Thread sync and interrupt are implemented in mirroringDisk.

Screenshots of Results:

With busy wait: Initially without any change, both read/write operation were complete and then next thread came in.

```
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 1 IN ITERATION[685]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
FUN 2 IN ITERATION[685]
Reading a block from disk...
Writing a block to disk...
FUN 3 IN BURST[685]
FUN 3: TICK [0]
FUN 3: TICK [1]
FUN 3: TICK [2]
FUN 3: TICK [3]
FUN 3: TICK [4]
FUN 3: TICK [5]
FUN 3: TICK [6]
FUN 3: TICK [7]
FUN 3: TICK [8]
FUN 3: TICK [9]
FUN 4 IN BURST[685]
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[686]
FUN 1: TICK [0]
```

With no busy wait: After read operation was issued by fun2, thread1 yielded and thread 2 took control.



With thread sync and interrupt handling