MP6: Primitive Device Driver

Ashutosh Punyani
UIN: 834006613
CSCE611: Operating System

Assigned Tasks

Main: Completed (Implementation of a basic blocking disk system where threads do not pause until I/O is finish)
Bonus Option 1: Completed (Support for Disk Mirroring)
Bonus Option 2: Not completed (Using Interrupts for Concurrency)
Bonus Option 3: Completed (Design of a thread-safe disk system)
Bonus Option 4: Completed (Implementation of a thread-safe disk system)

System Design

The main objective of machine problem 6 primarily involves exploring kernel-level device drivers for a basic programmed-I/O block device. For bonus option 1, we need to enable the interrupts while we start the thread. For the bonus option 2, we need to implement the round robin scheduling which generates a timer based interrupt every 50 ms to make the running thread yield.

Code Description

During the implementation of this machine problem, I made changes to the following six files:

1. **scheduler.H**
2. **scheduler.C**
3. **blocking_disk.H**
4. **blocking_disk.C**
5. **mirroring_disk.H**
6. **mirroring_disk.C**
7. **kernel.C**
8. **makefile**

I have used the same FIFO Scheduler code for `scheduler.H` and `scheduler.C` that I implemented for MP5. I removed the round-robin scheduler code as it was not required for this machine problem.

1. **blocking_disk.H**

In `blocking_disk.H`, I have added two new functions (`wait_until_ready()` and `is_ready_blocked()`) for doing the disk block operations.

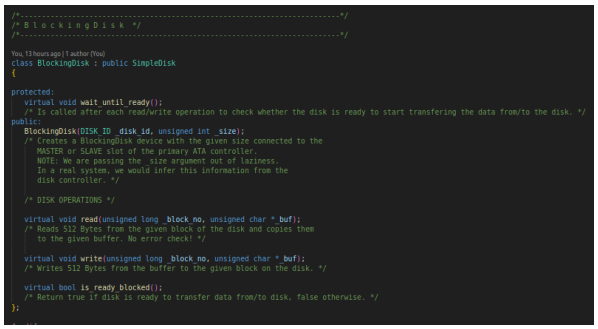


Figure 1: **blocking_disk.H**

2. **blocking_disk.C**

   (a) **TestAndSet related functions ()** : To ensure that threads do not simultaneously execute read and write operations, a mutex lock is employed. Locks are acquired and released before and after the I/O operation. The mutex lock is implemented using the Test And Set algorithm. The current implementation utilizes a simple Test And Set algorithm with an integer variable isLocked to represent the lock state. The `acquire_lock` function spins in a busy-wait loop until it successfully acquires the lock, and the `release_lock` function resets the lock state. To make sure our lock works well with multiple threads, we need to set it up correctly. When a thread wants to grab the lock, we should change its state using `acquire_lock` to avoid conflicts between threads. When we're done with the lock and want to release it, we use `release_lock` to safely reset its state.

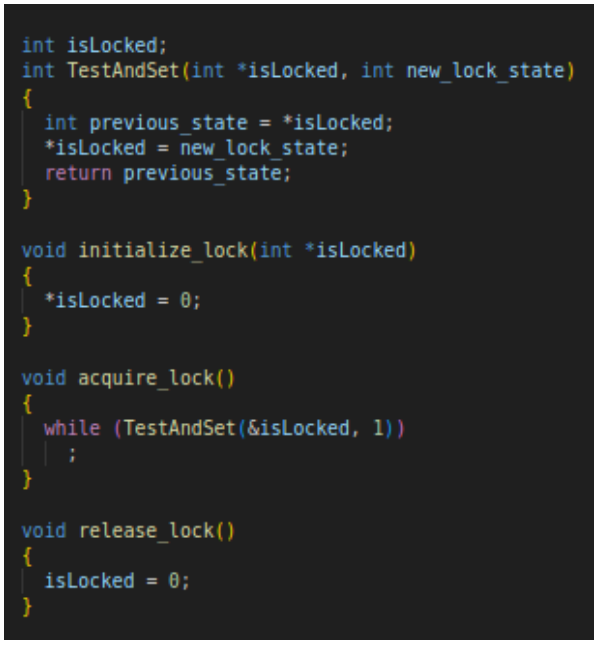


Figure 2: **TestAndSet related functions**

   (b) **BlockingDisk::BlockingDisk((Class Constructor)** : The constructor is used to construct the blocking disk object. This initializes the base class (SimpleDisk) with the provided disk ID and size. It also initializes the TestAndSet lock for thread safe implementation . It uses the `initialize_lock` function to set up a lock mechanism, for ensuring thread safety when accessing shared resources within the BlockingDisk class.

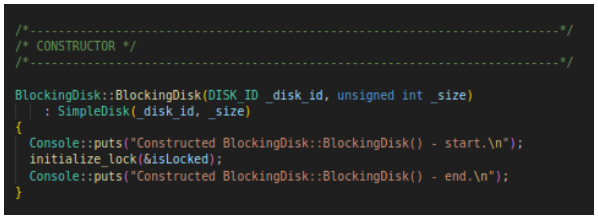


Figure 3: **BlockingDisk::BlockingDisk((Class Constructor)**

(c) **BlockingDisk::read** : This function uses a lock (`acquire_lock` and `release_lock`) to ensure exclusive access to the critical section where the disk read operation takes place. This function uses the simple disk read function. Simple disk read function initiates a read operation on a disk by first calling the `wait_until_ready()` function. If the disk is not ready, the thread yields the CPU and waits for next turn. Otherwise, it reads 512 bytes data from disk to a buffer.

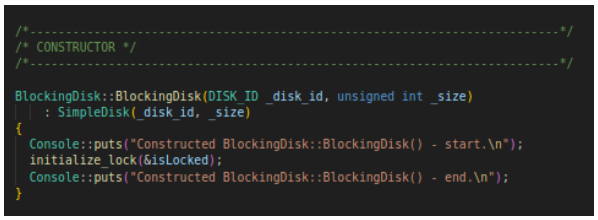


Figure 4: **BlockingDisk::read**

(d) **BlockingDisk::write)** : This function also uses a lock (`acquire_lock` and `release_lock`) to ensure exclusive access to the critical section where the disk write operation takes place. This function uses the simple disk write function. Simple disk write function initiates a write operation on a disk by first calling the `wait_until_ready()` function. If the disk is not ready, the thread yields the CPU and waits for next turn. Otherwise, it writes 512 bytes data from buffer to the disk.

Figure 5: **BlockingDisk::write)**

(e) **BlockingDisk::is_ready_blocked** : This function returns if the device is ready or not.

```
bool BlockingDisk::is_ready_blocked()
{
    return SimpleDisk::is_ready();
}
```

figs/Screenshot from 2023-11-05 21-19-36.png

Figure 6: **BlockingDisk::wait_until_ready**

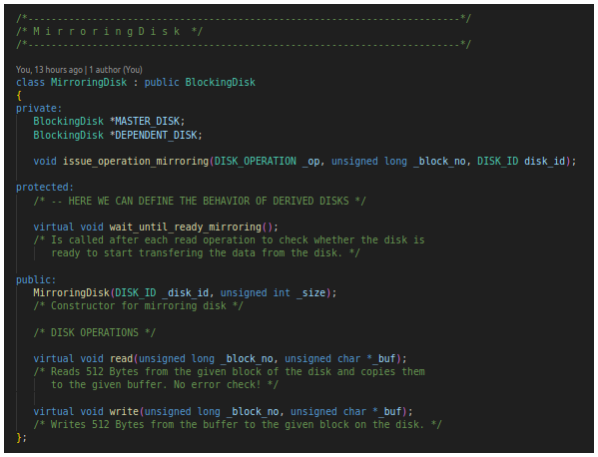
(f) **BlockingDisk::wait_until_ready** : This function verifies the readiness of a device. If the device is not ready, the scheduler places the current thread in a queue and yields the CPU. Later, when the thread is resumed, it rechecks the device's readiness, and this cycle may repeat. The functions resume() and yield() are invoked as part of this process.

```
void BlockingDisk::wait_until_ready()
{
  while (!SimpleDisk::is_ready())
  {
    SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
    SYSTEM_SCHEDULER->yield();
  }
}
```

Figure 7: **BlockingDisk::wait_until_ready**

3. **mirrorring_disk.H**

The MirroringDisk class is a representation of a mirrored disk system, inheriting from BlockingDisk. It manages two disks (MASTER_DISK and DEPENDENT_DISK) and includes methods for handling disk operations and checking readiness. The constructor sets up the mirroring disk, and the class overrides the read and write methods to implement mirroring behavior between the disks.



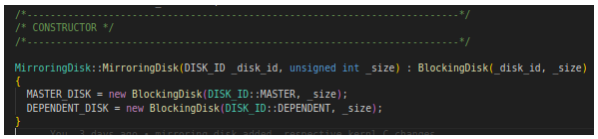
```
/*--------------------------------------------------------------------------*/
/* M i r r o r i n g D i s k */
/*--------------------------------------------------------------------------*/

You, 13 hours ago | 1 author (You)
class MirroringDisk : public BlockingDisk
{
private:
   BlockingDisk *MASTER_DISK;
   BlockingDisk *DEPENDENT_DISK;

   void issue_operation_mirroring(DISK_OPERATION _op, unsigned long _block_no, DISK_ID disk_id);

protected:
   /* -- HERE WE CAN DEFINE THE BEHAVIOR OF DERIVED DISKS */

   virtual void wait_until_ready_mirroring();
   /* Is called after each read operation to check whether the disk is
      ready to start transfering the data from the disk. */

public:
   MirroringDisk(DISK_ID _disk_id, unsigned int _size);
   /* Constructor for mirroring disk */

   /* DISK OPERATIONS */

   virtual void read(unsigned long _block_no, unsigned char *_buf);
   /* Reads 512 Bytes from the given block of the disk and copies them
      to the given buffer. No error check! */

   virtual void write(unsigned long _block_no, unsigned char *_buf);
   /* Writes 512 Bytes from the buffer to the given block on the disk. */
};
```

Figure 8: **mirrorring_disk.H**

4. **mirrorring_disk.C**

   (a) **MirroringDisk::MirroringDisk**

   This is the MirroredDisk class constructor. As this driver is responsible for handling two disks, it includes two BlockingDisk objects—one for the master disk and another for the dependent disk.

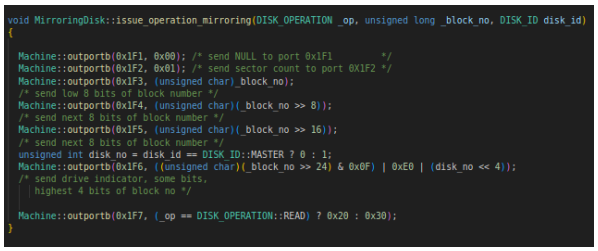


```
/*--------------------------------------------------------------------------*/
/* CONSTRUCTOR */
/*--------------------------------------------------------------------------*/

MirroringDisk::MirroringDisk(DISK_ID _disk_id, unsigned int _size) : BlockingDisk(_disk_id, _size)
{
  MASTER_DISK = new BlockingDisk(DISK_ID::MASTER, _size);
  DEPENDENT_DISK = new BlockingDisk(DISK_ID::DEPENDENT, _size);
}
```

Figure 9: **MirroringDisk::MirroringDisk**

   (b) **MirroringDisk::issue**$_{o}$*peration*$_{m}$*irroring*

   This function incorporates the disk number as a parameter. This enables it to request read/write operations from the controller, which oversees both the master disk and dependent disk based on the specified disk number—either MASTER or DEPENDENT.



```
void MirroringDisk::issue_operation_mirroring(DISK_OPERATION _op, unsigned long _block_no, DISK_ID disk_id)
{

  Machine::outportb(0x1F1, 0x00); /* send NULL to port 0x1F1        */
  Machine::outportb(0x1F2, 0x01); /* send sector count to port 0X1F2 */
  Machine::outportb(0x1F3, (unsigned char)_block_no);
  /* send low 8 bits of block number */
  Machine::outportb(0x1F4, (unsigned char)(_block_no >> 8));
  /* send next 8 bits of block number */
  Machine::outportb(0x1F5, (unsigned char)(_block_no >> 16));
  /* send next 8 bits of block number */
  unsigned int disk_no = disk_id == DISK_ID::MASTER ? 0 : 1;
  Machine::outportb(0x1F6, ((unsigned char)(_block_no >> 24) & 0x0F) | 0xE0 | (disk_no << 4));
  /* send drive indicator, some bits,
     highest 4 bits of block no */

  Machine::outportb(0x1F7, (_op == DISK_OPERATION::READ) ? 0x20 : 0x30);
}
```

Figure 10: **MirroringDisk::issue**$_{o}$*peration*$_{m}$*irroring*

   (c) **MirroringDisk::read**

   In this function, the initial step involves calling the issuemirroredoperation() function twice. This is done to inform the controller that it intends to perform a read operation on both the master disk and the dependent disk. If neither of the disks is ready, the thread willingly yields the CPU, allowing other processes to execute. However, if both disks are

ready, the function proceeds to read data from the port without specifying which disk is the source—it is indifferent to whether the data comes from the master disk or the dependent disk.

```
void MirroringDisk::read(unsigned long _block_no, unsigned char *_buf)
{
  /* Reads 512 Bytes in the given block of the given disk drive and copies them
     to the given buffer. No error check! */

  issue_operation_mirroring(DISK_OPERATION::READ, _block_no, DISK_ID::MASTER);
  issue_operation_mirroring(DISK_OPERATION::READ, _block_no, DISK_ID::DEPENDENT);

  wait_until_ready_mirroring();

  /* read data from port */
  int i;
  unsigned short tmpw;
  for (i = 0; i < 256; i++)
  {
    tmpw = Machine::inportw(0x1F0);
    _buf[i * 2] = (unsigned char)tmpw;
    _buf[i * 2 + 1] = (unsigned char)(tmpw >> 8);
  }
}
```

Figure 11: **MirroringDisk::read**

(d) **MirroringDisk::write**

This is responsible for writing data to both the master and dependent disks.

```
void MirroringDisk::write(unsigned long _block_no, unsigned char *_buf)
{
  MASTER_DISK->write(_block_no, _buf);
  DEPENDENT_DISK->write(_block_no, _buf);
}
```

Figure 12: **MirroringDisk::write**

Testing

During the development of the code, I wrote several `Console:puts()` and `Console:putui()` statements to pinpoint where my code was breaking and to understand if the logic was incorrect or not performing as expected. I removed some Console statements.

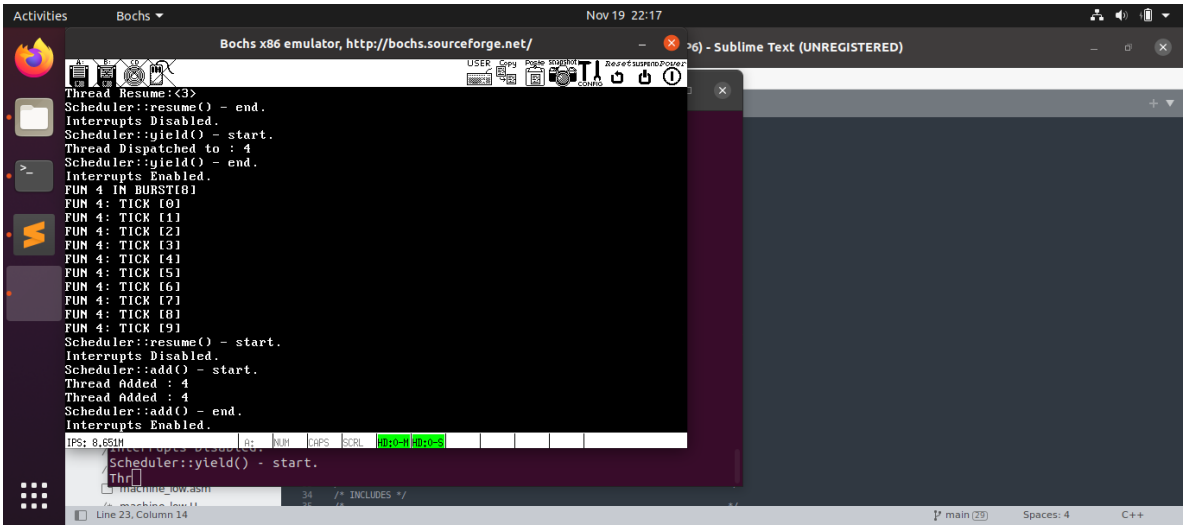


Figure 13: **Testing (with and_MIRRORING_DISK_ uncommented)**

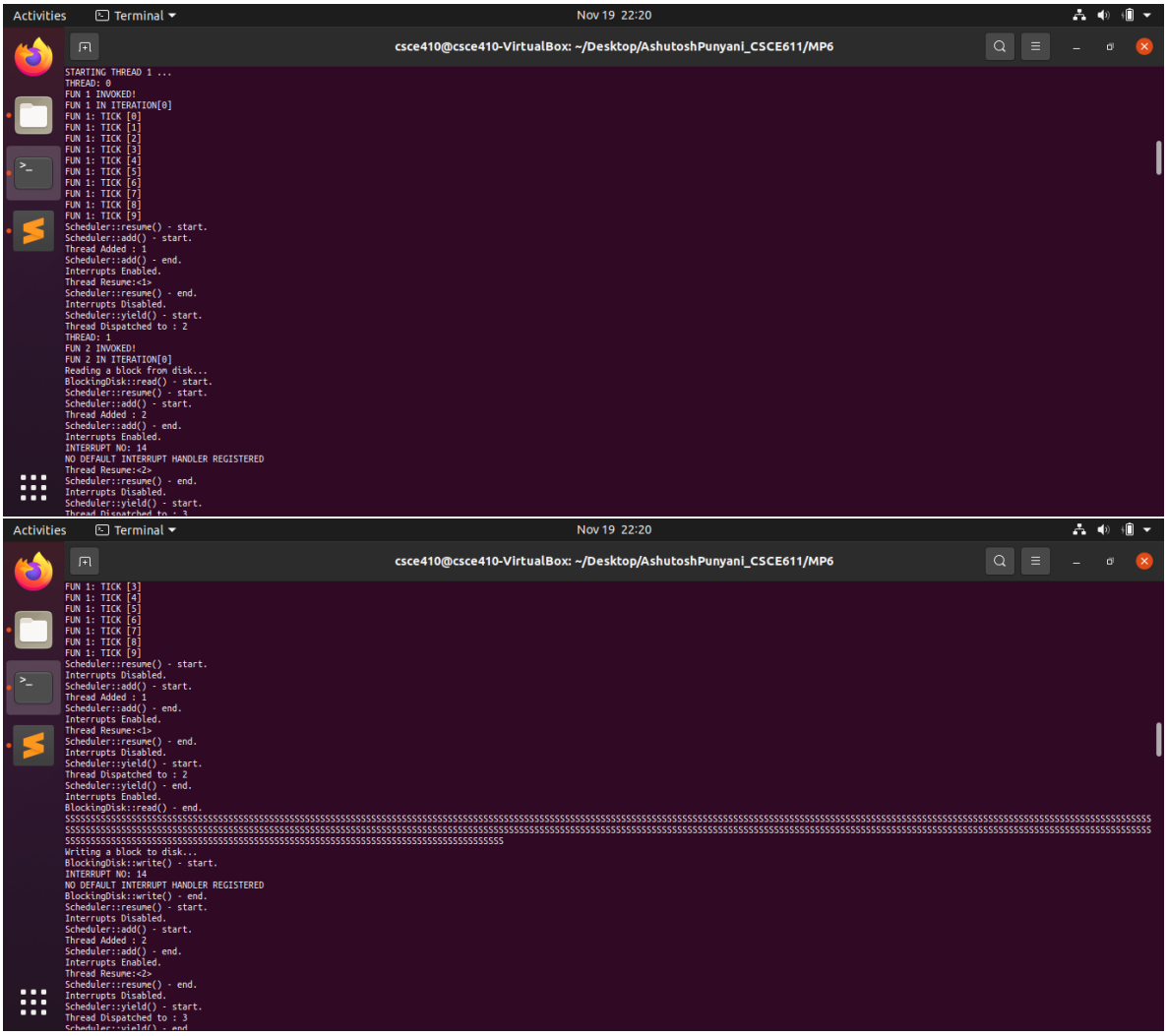


Figure 14: **Testing (with and_MIRRORING_DISK_ commented)**