

MP6: Primitive Device Driver

Sushant Vijay Shelar
UIN: 733001479
CSCE611: Operating System

Assigned Tasks

Main Part: Completed.

Bonus Option 1: Completed

Bonus Option 2: Completed

Bonus Option 3: Completed

Bonus Option 4: Completed

Disk Device Driver:

The document outlines the process for designing and developing a basic disk device driver. The primary objective is to create a disk driver that operates in a blocking mode without engaging in busy waiting during I/O operations. This means that when a thread initiates a read/write operation, it won't keep the CPU occupied. Instead, the thread will relinquish control of the CPU and remain inactive until the I/O operation is finished. Additionally, the document includes explanations for optional features that enhance the functionality of the device driver.

Code Description

Files changed:

• blocking_disk.C • blocking_disk.H • kernel.C • scheduler.C • scheduler.H

BlockingDisk is a subclass derived from SimpleDisk. For the fundamental multiprocessing (MP) implementation, modifications were made in the blockingDisk.C file. The read() and write() methods in BlockingDisk are identical to those in the SimpleDisk class. The key distinctions between these methods are as follows:

Blocking is ready(): Returns the status of the disk.

```
bool BlockingDisk::block_ready()
{
    if(Machine::inportb(0x1F7) & 0x08 ==0 )
    {
        return false;
    }
    else
        return true;
}
```

Figure 1: Blocking is ready function

Wait_until_ready():

- This function checks if the disk is ready for operations. If the disk is not ready, this function triggers the 'resume' and 'yield' processes.
- This approach guarantees that the CPU is not blocked when a thread initiates a read/write operation. Instead, the function allows the CPU to be used by the next thread in line, effectively managing CPU resources during disk operations.

```
void BlockingDisk::wait_until_ready()
{
    while(!block_ready())
    {
        #ifdef _HANDLE_INTERRUPTS_

            thread_queue.enqueue(Thread::CurrentThread());

        #else
            SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
        #endif

        SYSTEM_SCHEDULER->yield();
    }
}
```

Figure 2: Wait until ready function

Bonus 1: Support for disk mirroring:

As per handout, implemented MirroredDisk class, inheriting from BlockingDisk implement for disk mirroring functionality. Here's how it operates:

Reading Operations: When a read request is initiated, MirroredDisk simultaneously sends this request to both its primary and secondary (dependent) disks. It then waits until either disk is ready to provide the requested data.

Writing Operations: For write operations, the MirroredDisk class sends the write command to both the primary and secondary disks to ensure data is mirrored accurately on both.

The key methods of the MirroredDisk class include:

- **executeTask():** This method functions similarly to the SimpleDisk class's method but includes an additional parameter, diskId, to specify the target disk for the operation.
- **waitUntilReady():** This method checks the readiness of either the primary or secondary disk. If neither is ready, it places the current thread in a queue and relinquishes control of the CPU, effectively pausing the operation until one of the disks is ready.
- **read():** This method initiates a read command to both the primary and secondary disks. Whichever disk is ready first will read and provide the data to the user.
- **write():** In this method, the write command is issued to both disks, ensuring that data is consistently mirrored across both the primary and secondary storage devices.

```

void MirroredDisk::executeTask(DISK_OPERATION _op, unsigned long _block_no, DISK_ID disk) {
    Machine::outportb(0x1F1, 0x00);
    Machine::outportb(0x1F2, 0x01);
    Machine::outportb(0x1F3, (unsigned char)_block_no);
    Machine::outportb(0x1F4, (unsigned char)(_block_no >> 8));
    Machine::outportb(0x1F5, (unsigned char)(_block_no >> 16));
    unsigned int disk_no = disk == DISK_ID::MASTER ? 0 : 1;
    Machine::outportb(0x1F6, (((unsigned char)(_block_no >> 24) & 0x0F) | 0xE0 | (disk_no << 4)));

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

void MirroredDisk::read(unsigned long _block_no, unsigned char * _buf) {
    executeTask(DISK_OPERATION::READ, _block_no, DISK_ID::MASTER);
    executeTask(DISK_OPERATION::READ, _block_no, DISK_ID::DEPENDENT);
    wait_until_ready();

    unsigned short temp;
    for (int i = 0; i < 256; i++) {
        temp = Machine::inportw(0x1F8);
        _buf[i*2] = (unsigned char)temp;
        _buf[i*2+1] = (unsigned char)(temp >> 8);
    }
}

//implemented bonus point

void MirroredDisk::write(unsigned long _block_no, unsigned char * _buf)
{
    primaryDisk->write(_block_no, _buf);
    linkedDisk->write(_block_no, _buf);
}

void MirroredDisk::wait_until_ready()
{
    while ((!primaryDisk->block_ready()) || (!linkedDisk->block_ready()))
    {
        SYSTEM_SCHEDULER->resume(Thread::CurrentThread());
        SYSTEM_SCHEDULER->yield();
    }
}

```

Figure 3: Code for MirroredDisk class

Bonus Option 2 Using Interrupts for Concurrency:

In the BlockingDisk class, a distinct blocking queue has been implemented to manage read/write operations more efficiently. Here's how it functions:

Operation Handling: Whenever a read or write operation is initiated, the thread executing this operation is placed into the blocking queue. This queue effectively manages threads that are awaiting the completion of disk operations.

CPU Utilization: After adding the thread to the waiting queue, control of the CPU is immediately passed to the next available thread. This approach optimizes CPU usage by allowing other threads to run while the current thread awaits disk operation completion.

The implementation of this functionality involves modifications to two files:

blockingDisk.C: This source file contains the implementation details of the BlockingDisk class, including the logic for managing the blocking queue and handling thread operations.

blockingDisk.H: This header file defines the BlockingDisk class, including the declaration of the blocking queue and the methods for read/write operations and thread management.

```

#ifdef _HANDLE_INTERRUPTS_
void BlockingDisk::handle_interrupt(REGS *_r)
{
    Thread* t = thread_queue.rem();
    SYSTEM_SCHEDULER->resume(t);
}
#endif

```

Figure 4: handle_interrupts

Bonus Option 3 & 4:

To manage concurrent access to a critical section by multiple threads and prevent race conditions, it's essential to employ proper locking mechanisms. This approach is particularly crucial when threads are performing read/write operations on shared resources, as in the case of bonus 4.

To address this, a simple test-and-set lock mechanism is implemented. The primary purpose of this lock is to secure exclusive access to the critical section – in this context, the disk read/write operations. Here's how it works:

Acquiring the Lock: Before a thread enters the critical section, it acquires the lock. This ensures that no other thread can enter the critical section and modify the shared resource simultaneously, thereby preventing a race condition.

Releasing the Lock: Once the thread completes its operation within the critical section, it releases the lock. This action allows another thread to acquire the lock and safely access the disk.

```
struct concurrencyControl
{
    bool flag_control;
    concurrencyControl()
    {
        flag_control=false;
    }
    bool isSetValid()
    {
        if(flag_control)
        {
            return true;
        }
        else
        {
            flag_control=true;
            return false;
        }
    }
    void acquireLock()
    {
        while(isSetValid())
        {
        }
    }
    void releaseLock()
    {
        flag_control = false;
    }
};
```

Figure 5: Concurrency Control Implementation

Testing

I used the 'make' command to compile all the files in the "mp6" folder. Then, I ran a script called 'copy kernel' to copy the bin file to the disk. After that, I executed the code using the "bochs -f bochsrc.bxrc" command.

The core testing of the disk driver's functionality is primarily conducted through test suites located in the kernel.C file. These test suites have been specifically modified to evaluate various aspects of the disk driver. To facilitate this testing, several macros have been introduced:

HANDLE_INTERRUPTS Macro: This macro is defined in blockingDisk.H. Its purpose is to test the disk driver's ability to handle interrupts effectively. By including this macro, the test suites can assess how the disk driver manages interrupt signals during operations.

THREAD_SYNCHRONIZATION Macro: Added to blockingDisk.H, this macro is designed to test the implementation of thread synchronization in the disk driver. It allows the test suites to verify that the driver maintains thread safety and ensures that multiple threads can interact with the disk driver without causing race conditions or data corruption.

USE_MIRRORING_DISK Macro: This macro is defined in kernel.C and is used to test the mirroring functionality of the disk. By enabling this macro, the test suites can evaluate how the disk driver handles disk mirroring, ensuring that data is correctly mirrored across multiple disks for redundancy and reliability.

1. Commented all the macros mentioned above:

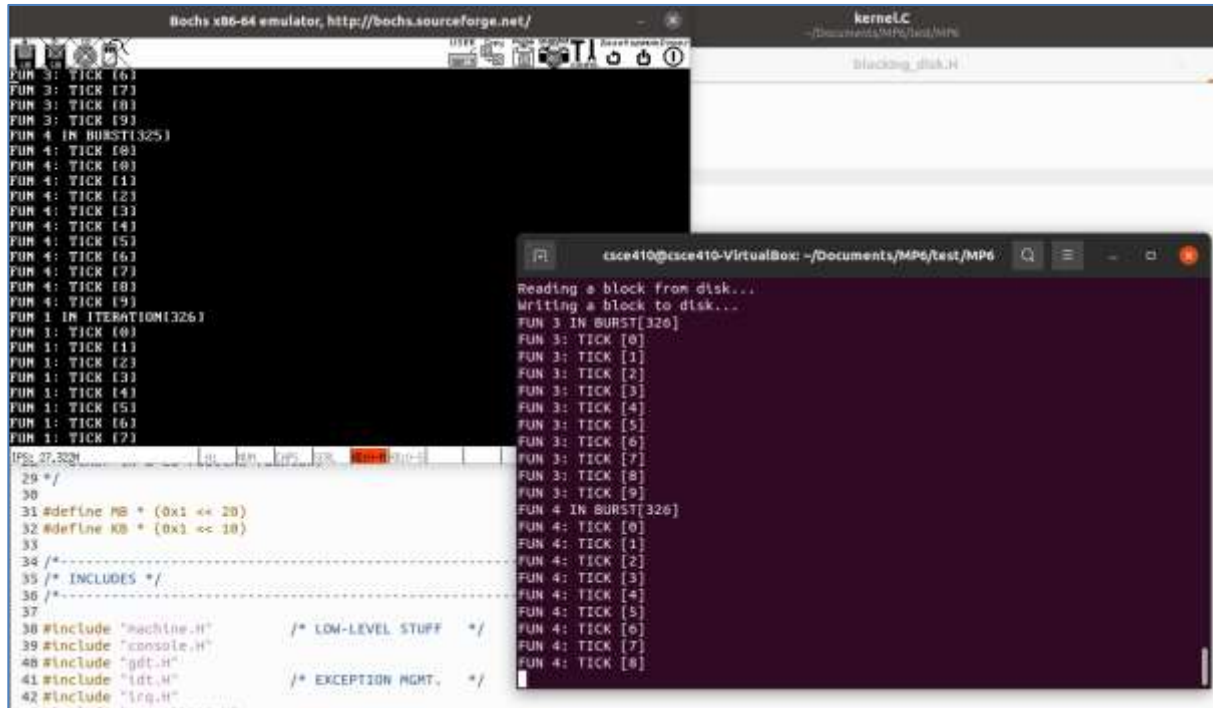


Figure 6: Commented Macros mentioned above

2. Uncommented USE_MIRRORING_DISK macro only:

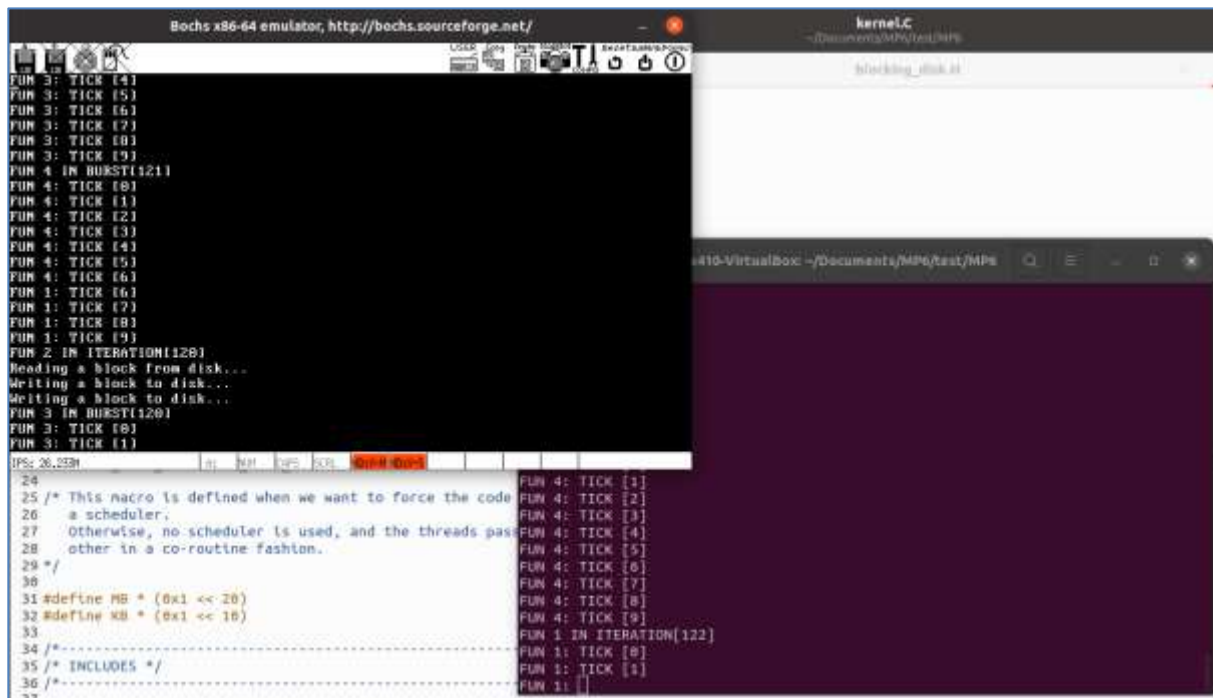


Figure 7: Uncommented USE_MIRRORING_DISK macro only

3. Uncommented THREAD_SYNCHRONIZATION Macro only:

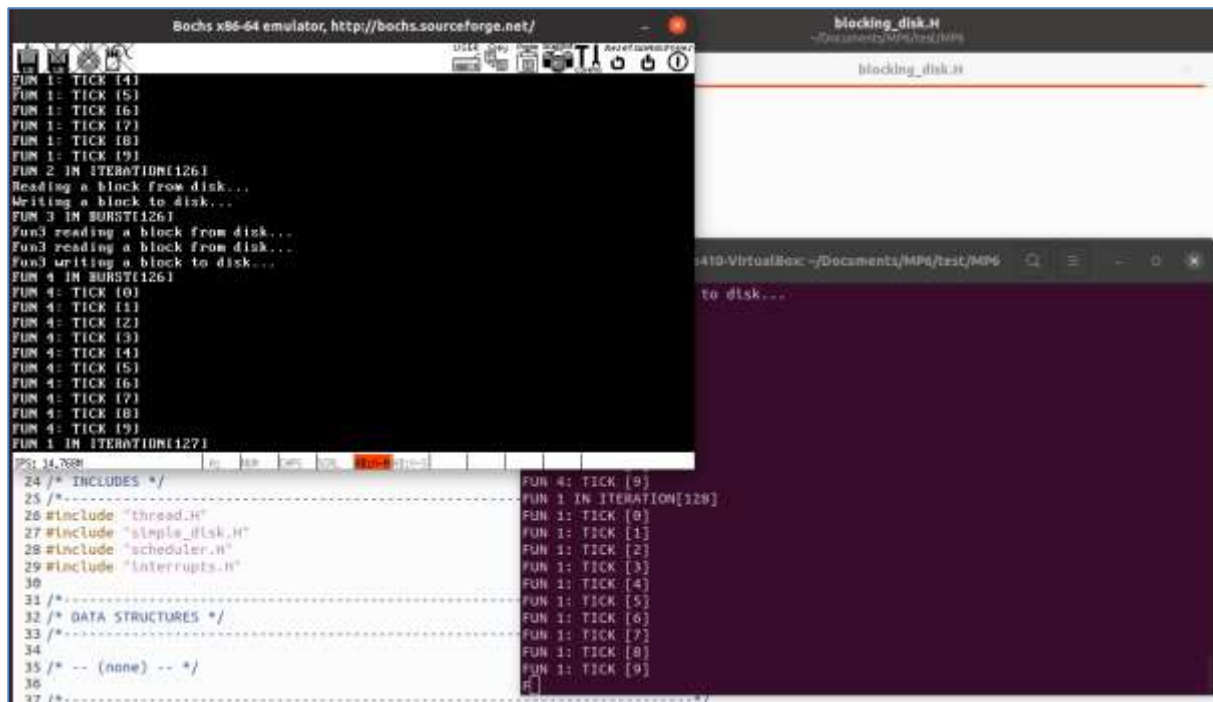


Figure 8: Uncommented THREAD_SYNCHRONIZATION macro only

4. Uncommented HANDLE_INTERRUPTS Macro:

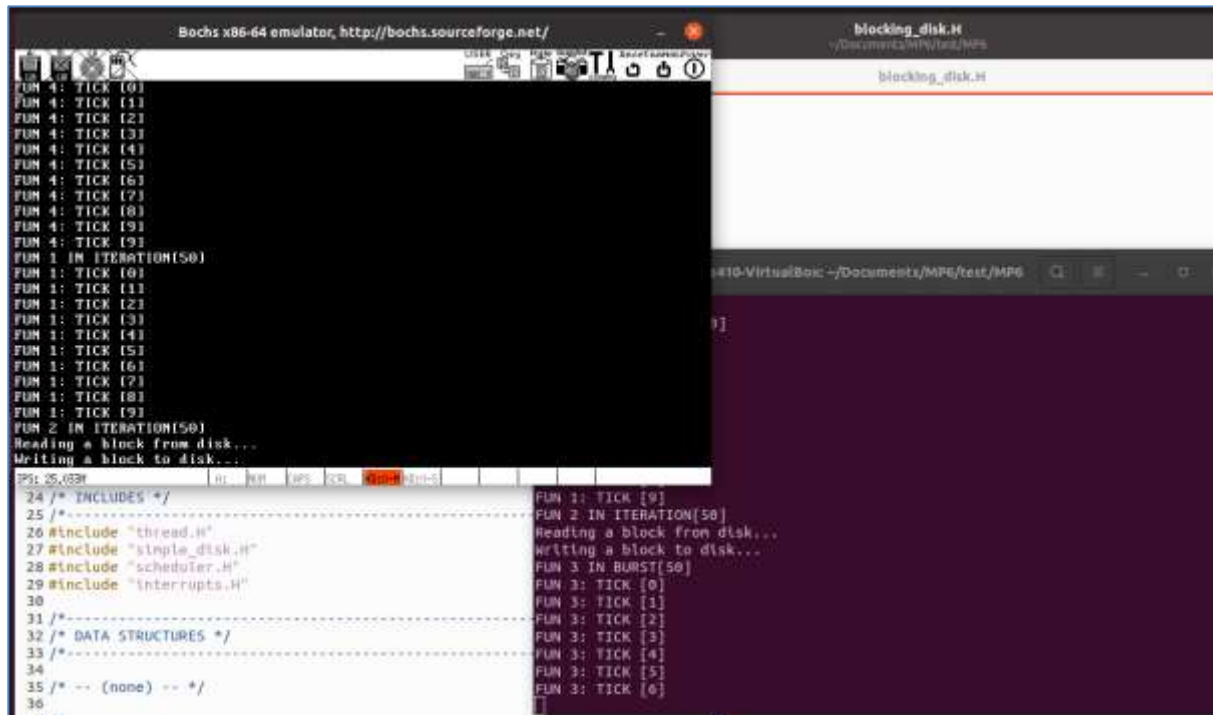
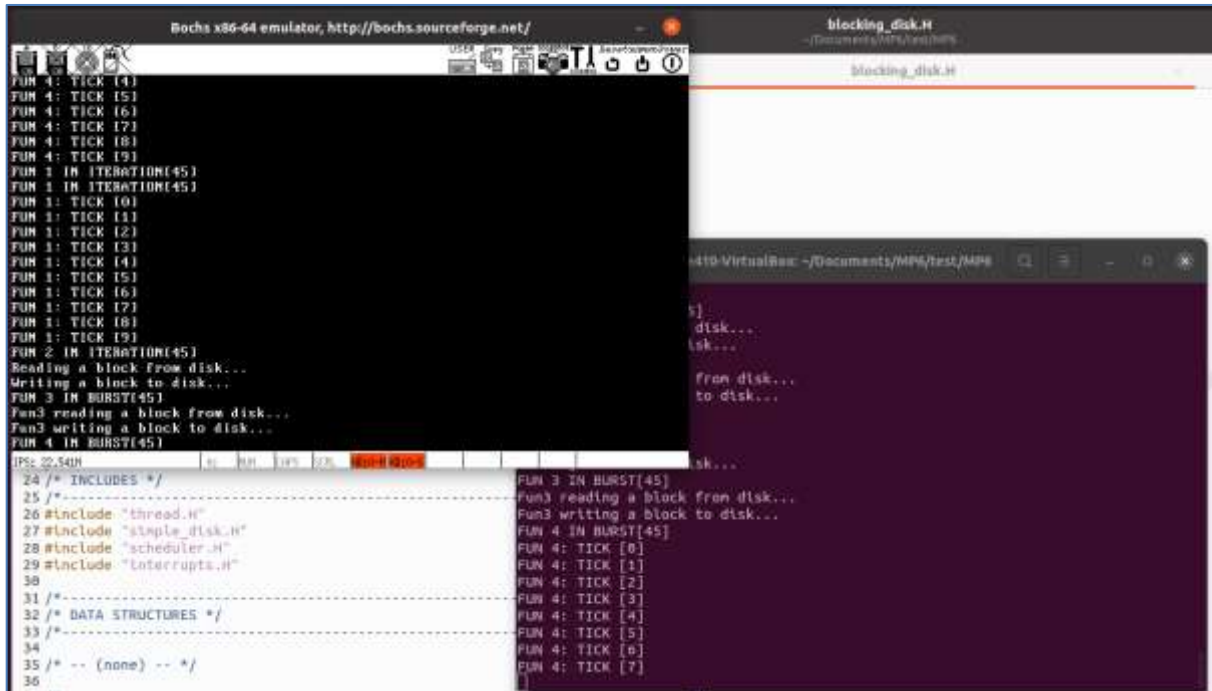


Figure 9: Uncommented HANDLE_INTERRUPTS macro only

5. Uncommented all the macros mentioned above:



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
FUN 4: TICK [1]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 1 IN ITERATION[45]
FUN 1 IN ITERATION[45]
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[45]
Reading a block from disk...
Writing a block to disk...
FUN 3 IN BURST[45]
Fun3 reading a block from disk...
Fun3 writing a block to disk...
FUN 4 IN BURST[45]
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]

24 /* INCLUDES */
25 /*-----
26 #include "thread.h"
27 #include "simple_disk.h"
28 #include "scheduler.h"
29 #include "lnterrupts.h"
30
31 /*-----
32 /* DATA STRUCTURES */
33 /*-----
34
35 /* -- (none) -- */
36
```

Figure 10: Uncommented all the macros mentioned above