Machine Problem 6: Primitive Disk Device Driver

Gaurangi Sinha
UIN: 734004353
CPSC611: Operating System

Assigned Task

Main Task: Completed.
Bonus Option 1: Not Completed.
Bonus Option 2: Not Completed.
Bonus Option 3: Completed.
Bonus Option 4: Completed

System Design

The goal of Machine Problem 6 is to develop and investigate kernel-level device drivers built on top of a simple programmed I/O block device.

1. Implement Non-Busy Waiting Disk Operations:

Develop a layer on top of the existing simple disk device that supports blocking read and write operations, but without busy waiting. This means the I/O operations should not tie up the entire system or return prematurely, ensuring the disk is ready to transfer the data before completing the call.

2. Enhance System Efficiency:

Optimize how the system handles disk operations to reduce CPU resource waste. This involves finding a balance between quick return times from disk operations and minimizing the CPU's idle time while it waits for these operations to complete.

Main Task: Introduce BlockingDisk:
Create a new device called BlockingDisk, derived from the existing SimpleDisk, designed to handle disk operations without blocking the CPU. The BlockingDisk will enable the system to continue performing other tasks while waiting for disk operations to complete by implementing non-looping read and write functions.

Bonus Option 3 & 4:

To ensure thread safety and concurrency in a system with a single processor handling disk I/O operations, a carefully designed scheduler and locking mechanism are key:

(a) Thread Yielding: Threads should yield periodically to prevent any single thread from monopolizing the CPU. This ensures fair CPU time distribution among all threads, particularly when they are awaiting I/O operations.

(b) Modified Scheduler: The existing scheduler should be enhanced to manage access to the CPU efficiently. It should rotate which thread has CPU access, ensuring that no thread is starved of processing time.
In our single-processor system, the previously developed scheduler class from the previous MP, with a few adjustments, efficiently manages the rotation of CPU access among threads.

(c) Exclusive I/O Operation: Implement a system where only one thread can perform an I/O operation at any given time. This control is crucial to avoid data corruption and ensure the integrity of I/O operations.

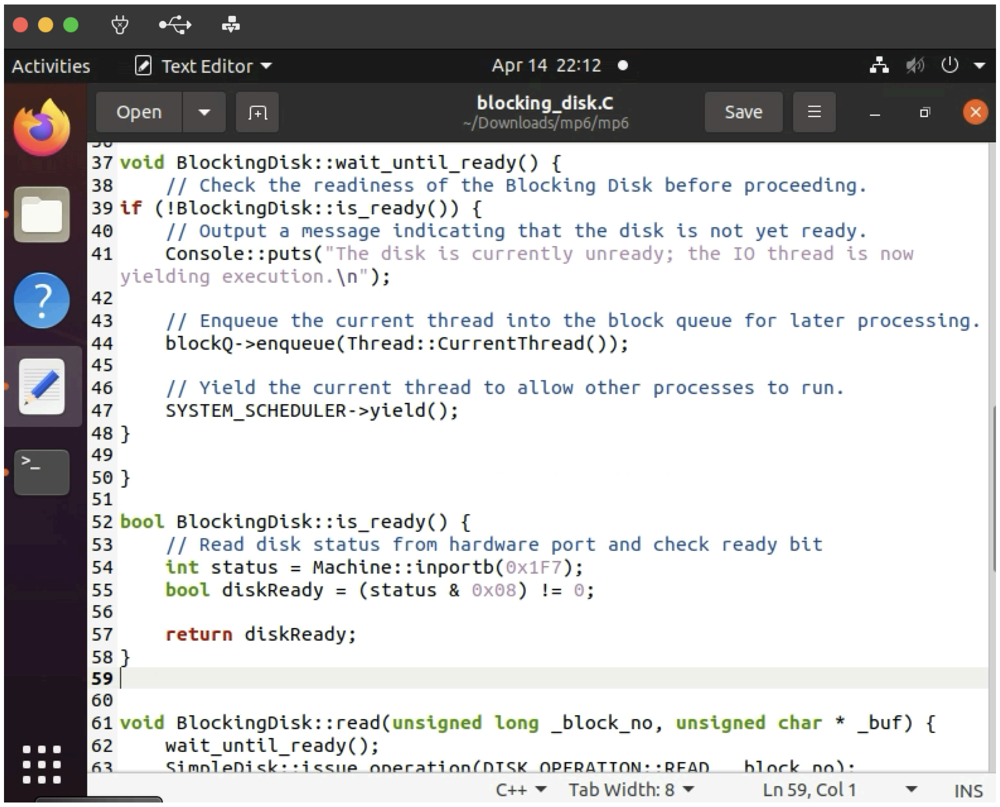
(d) Queue Management: Utilize a queue system where threads that require I/O operations are queued. Only the thread at the front of the queue is allowed to perform its I/O operation when the disk is ready. This setup prevents simultaneous I/O requests, which can lead to race conditions.
Given the presence of a queue in my implementation, we ensure that only the frontmost thread is dequeued when the disk is prepared to execute an operation. This approach prevents concurrent I/O operations from being initiated by multiple threads on the disk.

(e) Locks for Atomic Operations: Implement software locks to ensure that I/O operations (read and write) are atomic. These locks can help prevent race conditions by ensuring that no two threads can perform read and write operations on the disk at the same time.

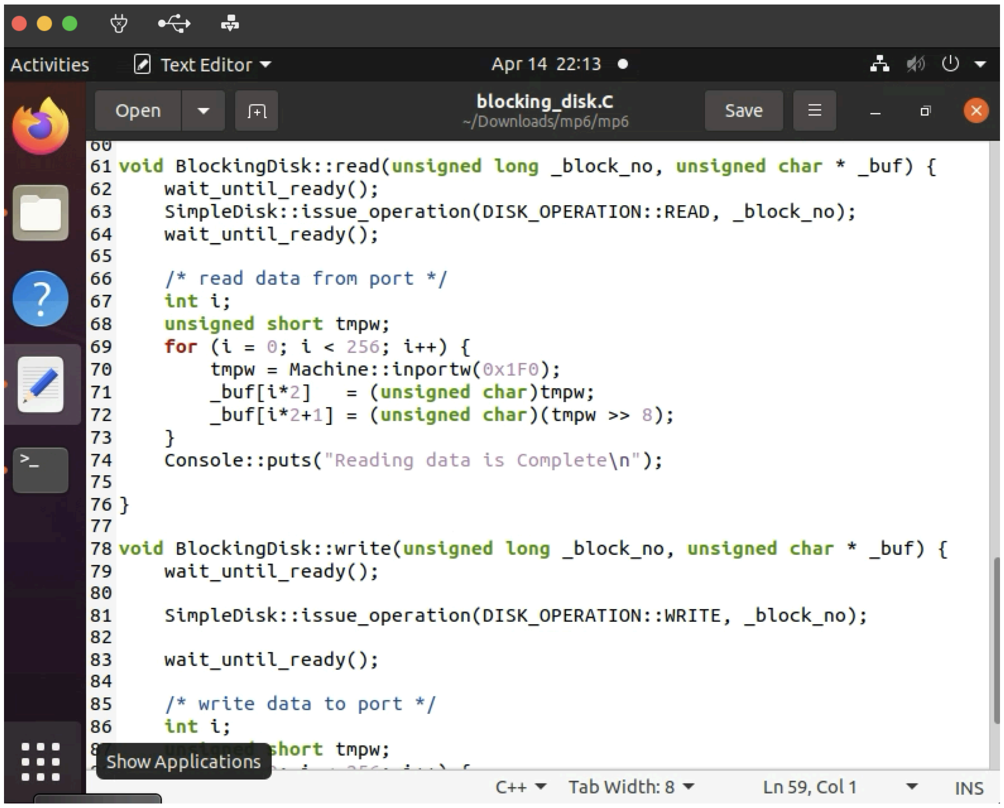
Code Description

I mainly made several changes in the files "scheduler.C", "scheduler.H", "blocking_disk.C", and “blocking_disk.H”. I also made the header file, “thread_node.h”.

Class BlockingDisk:



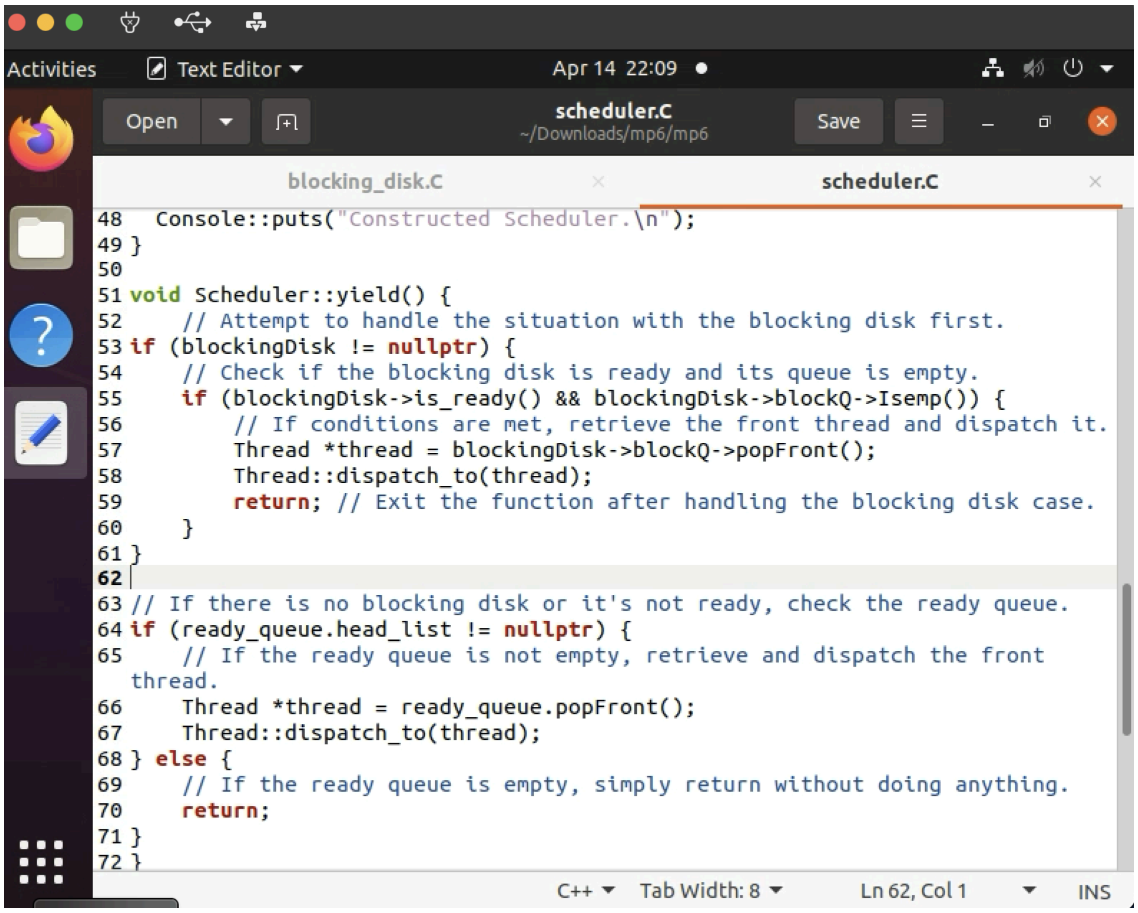
```
void BlockingDisk::wait_until_ready() {
    // Check the readiness of the Blocking Disk before proceeding.
if (!BlockingDisk::is_ready()) {
    // Output a message indicating that the disk is not yet ready.
    Console::puts("The disk is currently unready; the IO thread is now
yielding execution.\n");

    // Enqueue the current thread into the block queue for later processing.
    blockQ->enqueue(Thread::CurrentThread());

    // Yield the current thread to allow other processes to run.
    SYSTEM_SCHEDULER->yield();
}

}

bool BlockingDisk::is_ready() {
    // Read disk status from hardware port and check ready bit
    int status = Machine::inportb(0x1F7);
    bool diskReady = (status & 0x08) != 0;

    return diskReady;
}



void BlockingDisk::read(unsigned long _block_no, unsigned char * _buf) {
    wait_until_ready();
```



```
void BlockingDisk::read(unsigned long _block_no, unsigned char * _buf) {
    wait_until_ready();
    SimpleDisk::issue_operation(DISK_OPERATION::READ, _block_no);
    wait_until_ready();

    /* 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);
    }
    Console::puts("Reading data is Complete\n");

}

void BlockingDisk::write(unsigned long _block_no, unsigned char * _buf) {
    wait_until_ready();

    SimpleDisk::issue_operation(DISK_OPERATION::WRITE, _block_no);

    wait_until_ready();

    /* write data to port */
    int i;
            short tmpw;
```

Figure above: BlockingDisk class

The BlockingDisk class handles disk operations in a non-blocking manner. This means it manages read and write operations without causing the executing thread to busy-wait, allowing the system to perform other tasks while waiting for the disk to become ready. It does this by checking disk readiness and, if the disk isn't ready, queuing the current thread for later execution and yielding control to allow other processes to run. The class includes methods to ensure readiness before performing any operations, and it handles actual data transfers by interacting directly with hardware ports. This setup improves system responsiveness and resource utilization by avoiding busy waiting during disk operations.
`wait_until_ready()`: This checks if the disk is ready for operations. If not ready, it enqueues the current thread into a block queue and yields execution to allow other processes to run, effectively freeing up the CPU until the disk is ready.
`is_ready()`: Checks the disk's status from a hardware port to determine if it's ready for operations. It returns a boolean indicating the readiness based on the disk's status register.



```
  Console::puts("Constructed Scheduler.\n");
}

void Scheduler::yield() {
    // Attempt to handle the situation with the blocking disk first.
if (blockingDisk != nullptr) {
    // Check if the blocking disk is ready and its queue is empty.
    if (blockingDisk->is_ready() && blockingDisk->blockQ->Isemp()) {
        // If conditions are met, retrieve the front thread and dispatch it.
        Thread *thread = blockingDisk->blockQ->popFront();
        Thread::dispatch_to(thread);
        return; // Exit the function after handling the blocking disk case.
    }
}

// If there is no blocking disk or it's not ready, check the ready queue.
if (ready_queue.head_list != nullptr) {
    // If the ready queue is not empty, retrieve and dispatch the front
thread.
    Thread *thread = ready_queue.popFront();
    Thread::dispatch_to(thread);
} else {
    // If the ready queue is empty, simply return without doing anything.
    return;
}
}
```

Figure: yield() in scheduler class.

The `yield()` function in the `Scheduler` class checks if there's a `blockingDisk` object and if it's ready for operations with an empty queue. If both conditions are met, it retrieves the first thread from the disk's queue, dispatches it to continue execution, and then exits the function early. This is typically used to manage CPU resources efficiently, allowing the system to handle threads waiting on I/O operations without stalling other processes.

Compilation

To compile the code, we run the following commands:

```
make clean
make
```

A simple script to copy the kernel onto the floppy image. The script mounts the floppy image, copies the kernel image onto it, and then unmounts the floppy image again. So we use the below command.

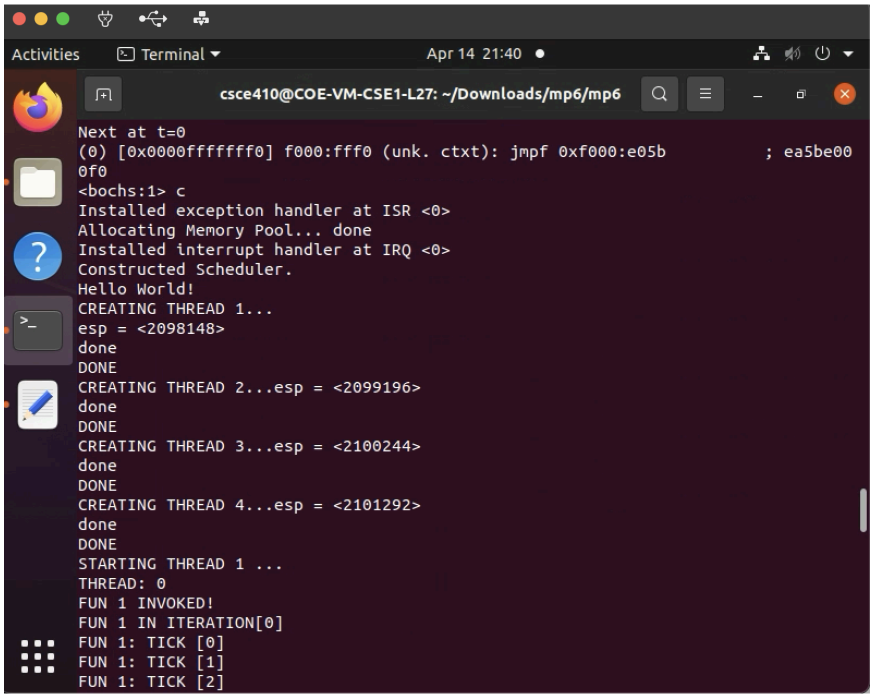
```
./copykernel.sh
```

and then to start the Bochs simulator we run the following command:

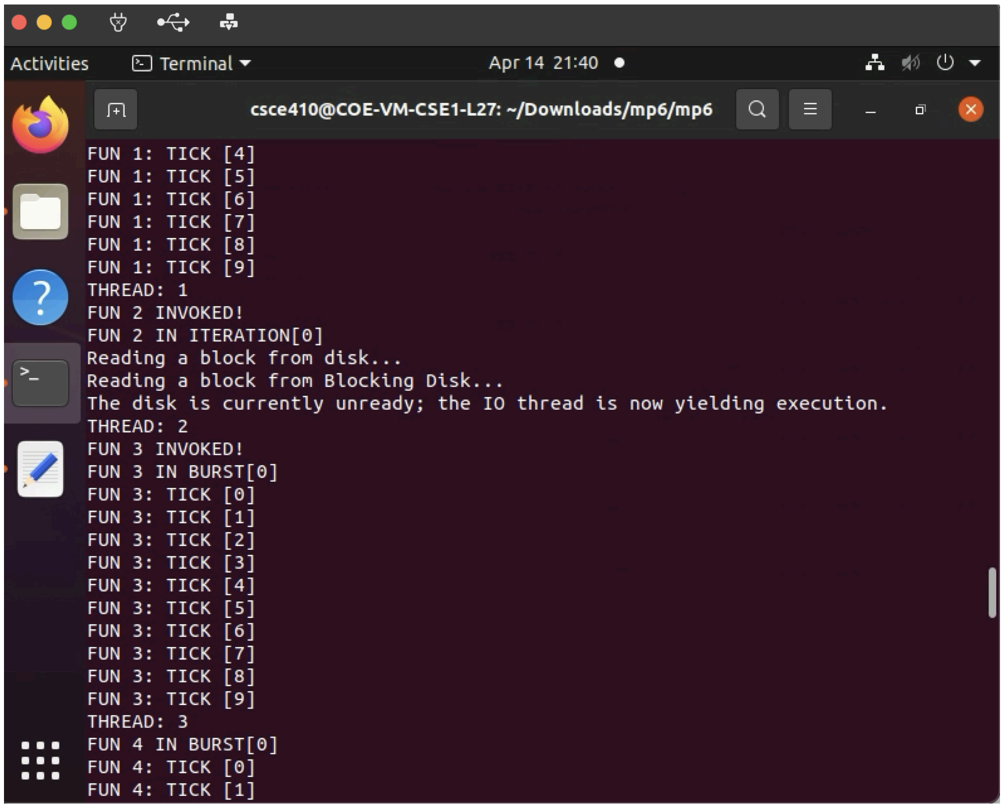
```
bochs –f bochsrc.bxrc
```

Testing

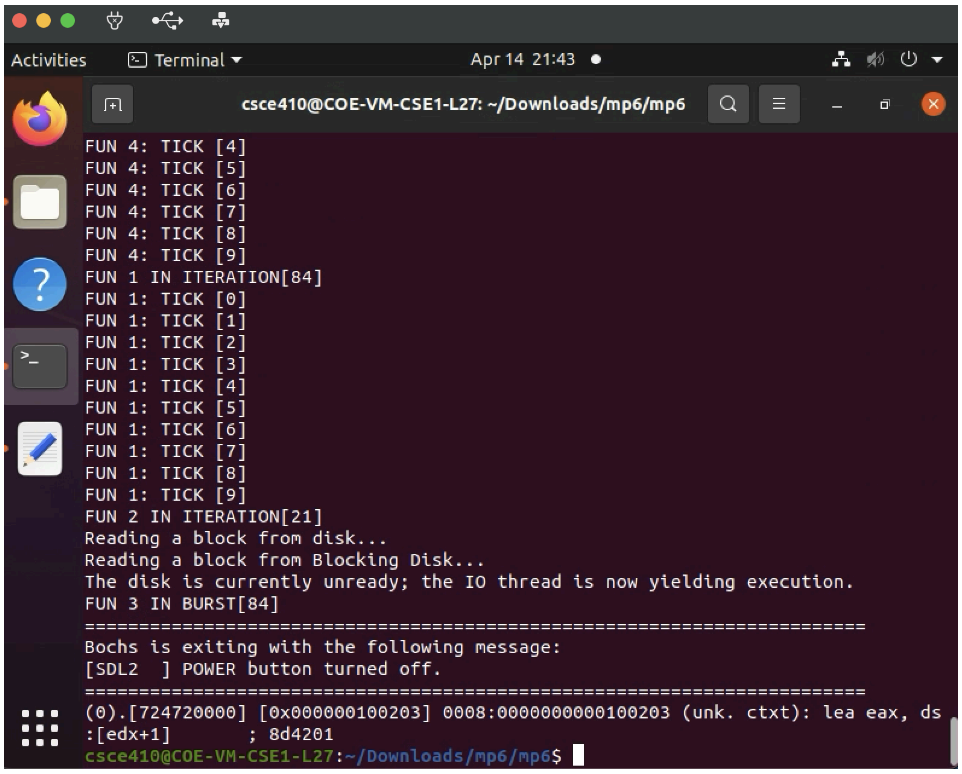
For testing, the machine problem, I ran the above command and, I used the given test function from kernel.C. I ran two cases or scenarios where the _USES_SCHEDULER_ and _BLOCKING DISK_ macros are set and unset. All the tests are passing, for given and additional scenarios.



```
Next at t=0
(0) [0x0000fffffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b          ; ea5be00
0f0
<bochs:1> c
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Constructed Scheduler.
Hello World!
CREATING THREAD 1...
esp = <2098148>
done
DONE
CREATING THREAD 2...esp = <2099196>
done
DONE
CREATING THREAD 3...esp = <2100244>
done
DONE
CREATING THREAD 4...esp = <2101292>
done
DONE
STARTING THREAD 1 ...
THREAD: 0
FUN 1 INVOKED!
FUN 1 IN ITERATION[0]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
```

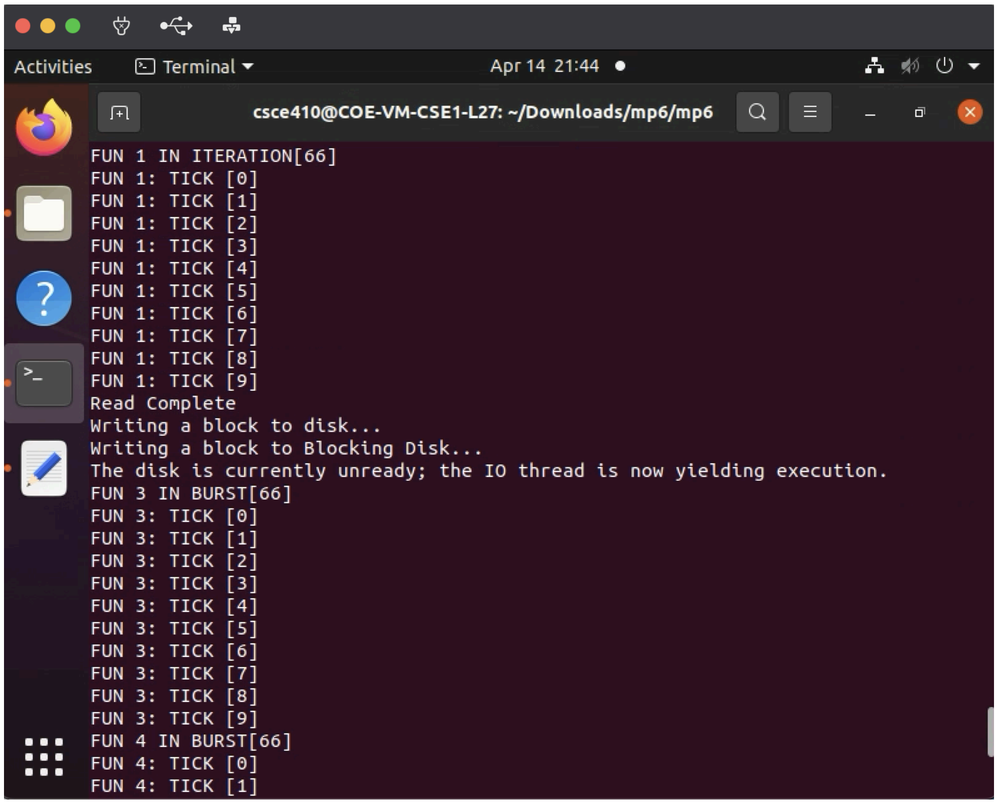


```
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
THREAD: 1
FUN 2 INVOKED!
FUN 2 IN ITERATION[0]
Reading a block from disk...
Reading a block from Blocking Disk...
The disk is currently unready; the IO thread is now yielding execution.
THREAD: 2
FUN 3 INVOKED!
FUN 3 IN BURST[0]
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]
THREAD: 3
FUN 4 IN BURST[0]
FUN 4: TICK [0]
FUN 4: TICK [1]
```

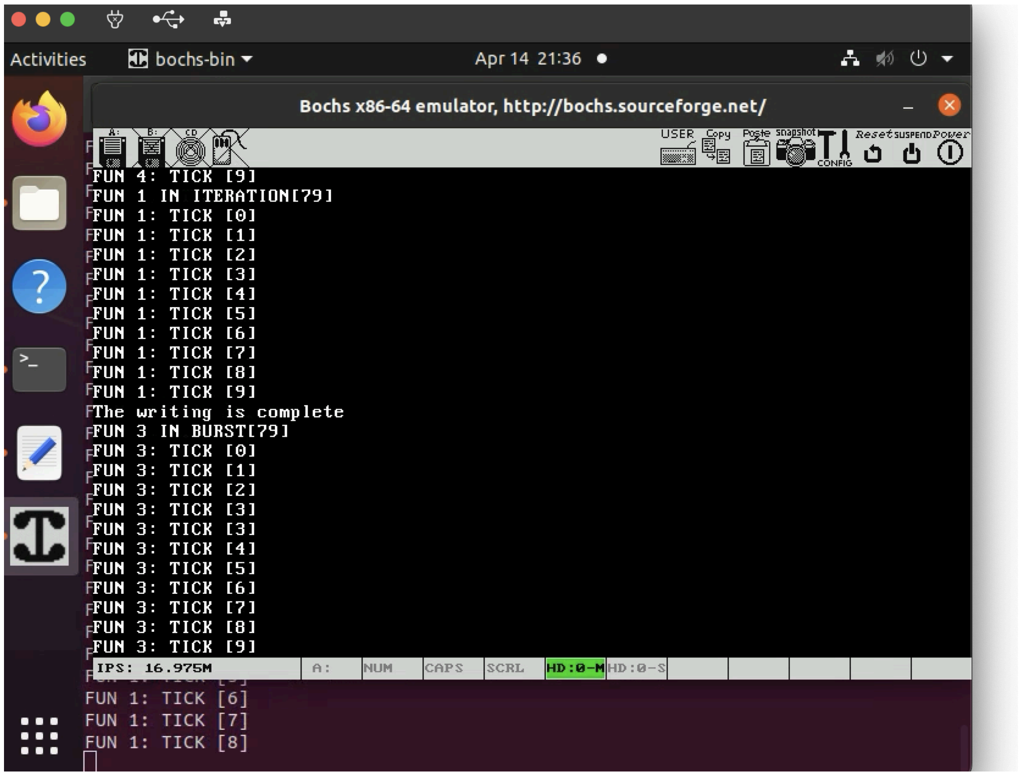


```
FUN 4: TICK [4]
FUN 4: TICK [5]
FUN 4: TICK [6]
FUN 4: TICK [7]
FUN 4: TICK [8]
FUN 4: TICK [9]
FUN 1 IN ITERATION[84]
FUN 1: TICK [0]
FUN 1: TICK [1]
FUN 1: TICK [2]
FUN 1: TICK [3]
FUN 1: TICK [4]
FUN 1: TICK [5]
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
FUN 1: TICK [9]
FUN 2 IN ITERATION[21]
Reading a block from disk...
Reading a block from Blocking Disk...
The disk is currently unready; the IO thread is now yielding execution.
FUN 3 IN BURST[84]
========================================================================
Bochs is exiting with the following message:
[SDL2  ] POWER button turned off.
========================================================================
(0).[724720000] [0x000000100203] 0008:0000000000100203 (unk. ctxt): lea eax, ds
:[edx+1]        ; 8d4201
csce410@COE-VM-CSE1-L27:~/Downloads/mp6/mp6$
```

The figure above: Thread yields during read before thread 3 assumes control.


Activities   Terminal   Apr 14 21:44
csce410@COE-VM-CSE1-L27: ~/Downloads/mp6/mp6

```
FUN 1 IN ITERATION[66]
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]
Read Complete
Writing a block to disk...
Writing a block to Blocking Disk...
The disk is currently unready; the IO thread is now yielding execution.
FUN 3 IN BURST[66]
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[66]
FUN 4: TICK [0]
FUN 4: TICK [1]
```


Activities   bochs-bin   Apr 14 21:36
Bochs x86-64 emulator, http://bochs.sourceforge.net/

```
FUN 4: TICK [9]
FUN 1 IN ITERATION[79]
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]
The writing is complete
FUN 3 IN BURST[79]
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]
IPS: 16.975M   A:   NUM   CAPS   SCRL   HD:0-M   HD:0-S
FUN 1: TICK [6]
FUN 1: TICK [7]
FUN 1: TICK [8]
```