MP6: Primitive Disk Device Driver

Vinay Balamurali
UIN: 936002032
CSCE611: Operating System

Assigned Tasks

Main: Completed.
Bonus 1: Completed.
Bonus 2: Completed.
Bonus 3: Idea is given in design doc but did not implement the code.

System Design

This machine problem primarily aims to to implement a device driver for I/O by implementing `read` and `write` methods that do not employ busy waiting.

Main

To prevent busy waiting, the `read` and `write` methods have been overridden in the derived class `NonBlockingDisk`. Every time these methods issue a read/write operation, they are added to a blocked queue and yield the CPU. This design choice was chosen as it was noticed that disks were ready immediately after the operation was issued. Due to this, the code path to exercise Blocked threads did not occur.

Bonus 1 and 2: Design and Implementation of a thread-safe disk system

The problem statement requires implementation of a thread-safe system but we do not have access to any synchronization libraries. The following algorithm will work provided it runs on a uni-processor system and instructions are not executed out-of-order. The previous statement does not hold true for the most part in the modern world, but the solution has been implemented considering hypothetical conditions. Peterson's algorithm works for 2 threads only. The expanded version of this is called the Filter Lock algorithm. In this algorithm, all threads pass through multiple stages, each getting filtered out at every stage. At the highest stage, there is only thread that can acquire the lock. It follows 2 important properties:

- At least one of the threads attempting enter a higher level is successful.
- If more than one thread attempts the scenario mentioned in the previous point, then one of them gets blocked and remains waiting at that level.

This machine problem uses the Filter Lock algorithm to synchronize accesses to the disk.

Bonus 3: Using Interrupts for Concurrency

Currently class `NonBlockingDisk` derives from `SimpleDisk`. We need to make sure it derives from class `InterruptHandler` as well. When the object of this class is instantiated, we also register it with the Interrupt Handler with **IRQ code 14**. The Interrupt handler for class `NonBlockingDisk` must override the base class one. In this method. At the time a read/write operation is issued, they must also be queued up in the blocked queue. Again, the issue is that interrupts are generated instantaneously so it might be best to first queue the request before issuing the operation. In the interrupt handler, the request is dequeued from the blocked queue and passed on to the scheduler by calling the method `resume`, which adds it to the tail of the Ready Queue.

Code Description

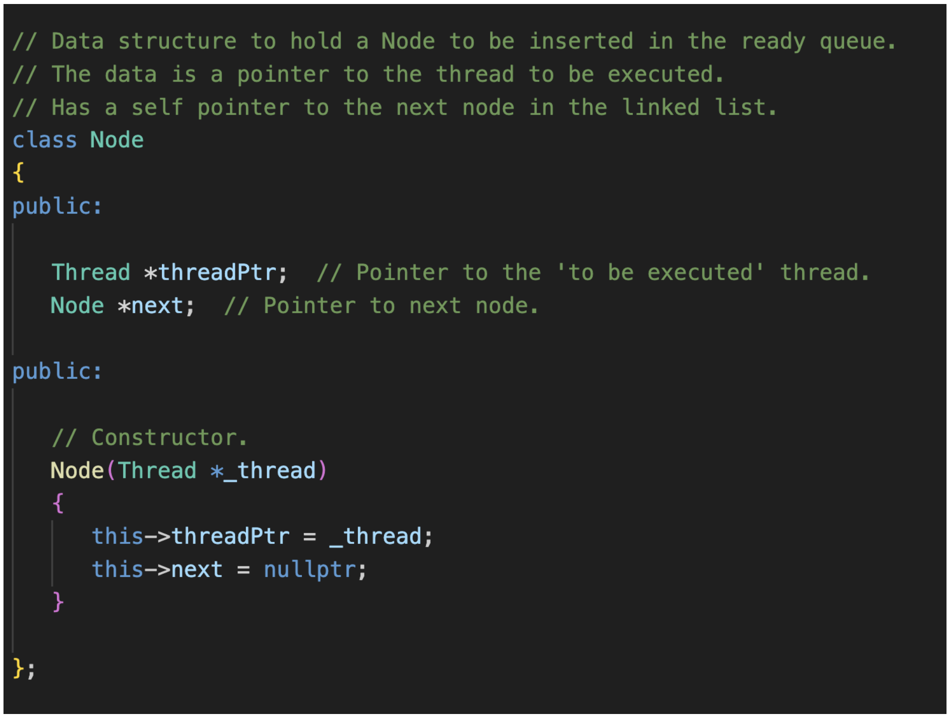
0.1 Files Added/Modified

- scheduler.H
- scheduler.C
- nonblocking_disk.H
- nonblocking_disk.C
- thread.C
- simple_disk.H
- kernel.C
- makefile

scheduler.H : This header file is largely unchanged from the previous MP. This header file provides method declarations for the class FIFO scheduler. Additionally, it provides an implementation for a Queue Data structure as a linked list which holds elements having the pointer to a thread.

Implementation of Class `Node`:

- `Thread *threadPtr` : Pointer to the thread to be executed.
- `Node *next` : Pointer to the next node in the linked list.

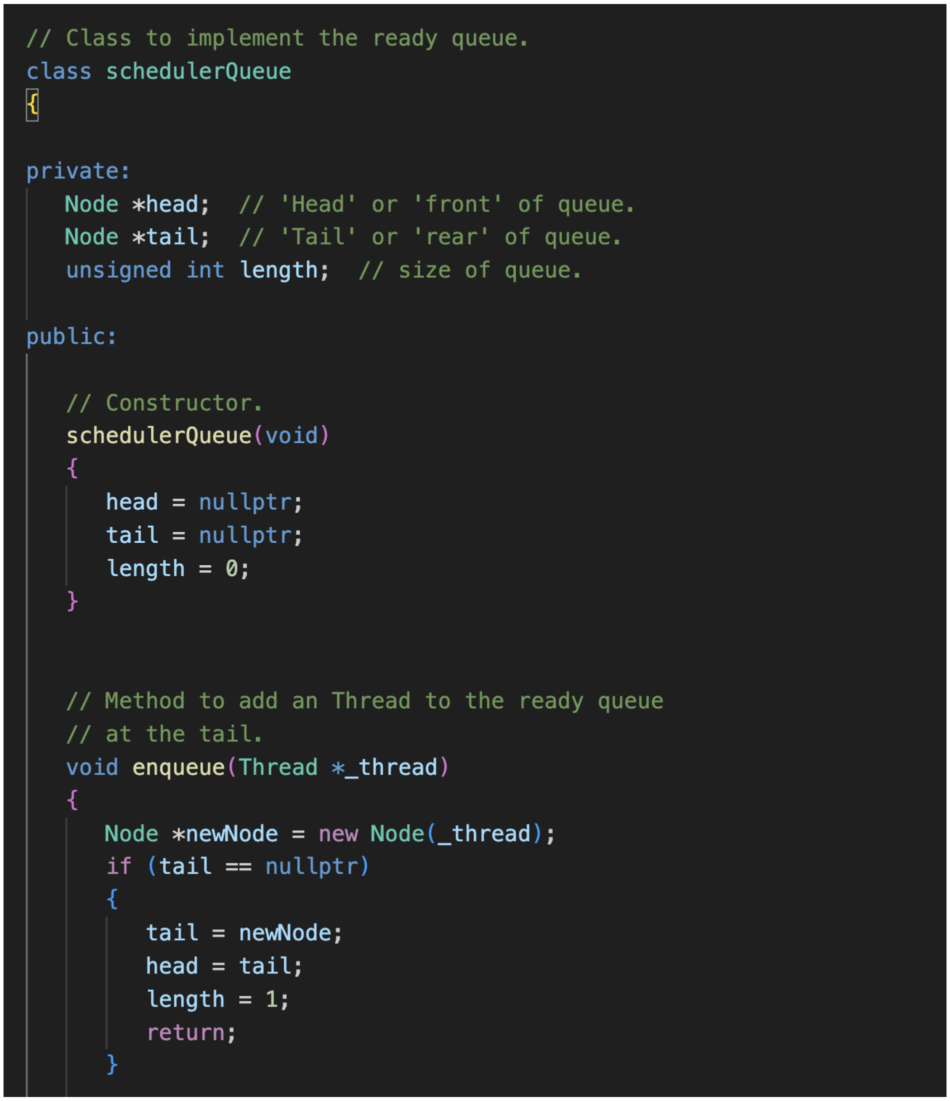


```
// Data structure to hold a Node to be inserted in the ready queue.
// The data is a pointer to the thread to be executed.
// Has a self pointer to the next node in the linked list.
class Node
{
public:

    Thread *threadPtr;  // Pointer to the 'to be executed' thread.
    Node *next;  // Pointer to next node.

public:

    // Constructor.
    Node(Thread *_thread)
    {
        this->threadPtr = _thread;
        this->next = nullptr;
    }

};
```

Figure 1: Class `Node`

The class `schedulerQueue` implements the Ready Queue and provides methods to add and remove items. Implementation of Class `schedulerQueue`:

- `Node *head` : Pointer to head of queue.
- `Node *tail` : Pointer to tail of queue.
- `unsigned int length` : Number of elements in the queue.
- `schedulerQueue(void)` : Constructor to initialize pointers to null and number of elements to 0.
- `void enqueue(Thread *_thread)` : Method to add an element at the tail of the queue.
- `Thread* dequeue(void)` : Method to remove an element from the beginning of the queue.
- `void removeThreadById(Thread* _thread)` : A special method to remove an element based on the thread ID.
- `unsigned int fetchSize(void)` : Method to fetch the size of the queue.

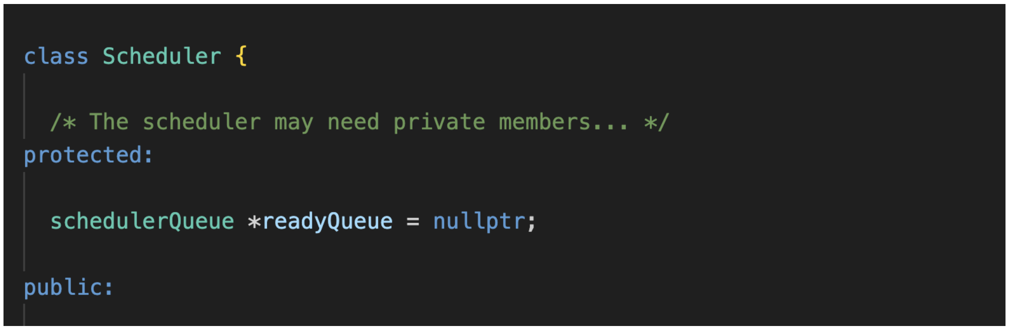


```
// Class to implement the ready queue.
class schedulerQueue
{

private:
   Node *head;  // 'Head' or 'front' of queue.
   Node *tail;  // 'Tail' or 'rear' of queue.
   unsigned int length;  // size of queue.

public:

   // Constructor.
   schedulerQueue(void)
   {
      head = nullptr;
      tail = nullptr;
      length = 0;
   }


   // Method to add an Thread to the ready queue
   // at the tail.
   void enqueue(Thread *_thread)
   {
      Node *newNode = new Node(_thread);
      if (tail == nullptr)
      {
         tail = newNode;
         head = tail;
         length = 1;
         return;
      }
```

Figure 2: Class `schedulerQueue`

The following items have been modified in class `Scheduler`:

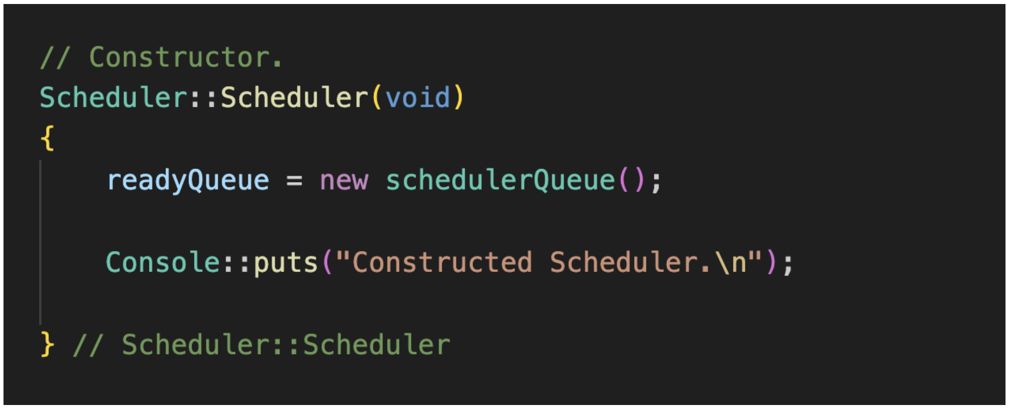
- `schedulerQueue *readyQueue` : protected member to hold a reference to the Ready Queue. This pointer has been made protected so that derived class can access the member variable directly.



```
class Scheduler {

  /* The scheduler may need private members... */
protected:

  schedulerQueue *readyQueue = nullptr;

public:
```

Figure 3: Class `Scheduler`

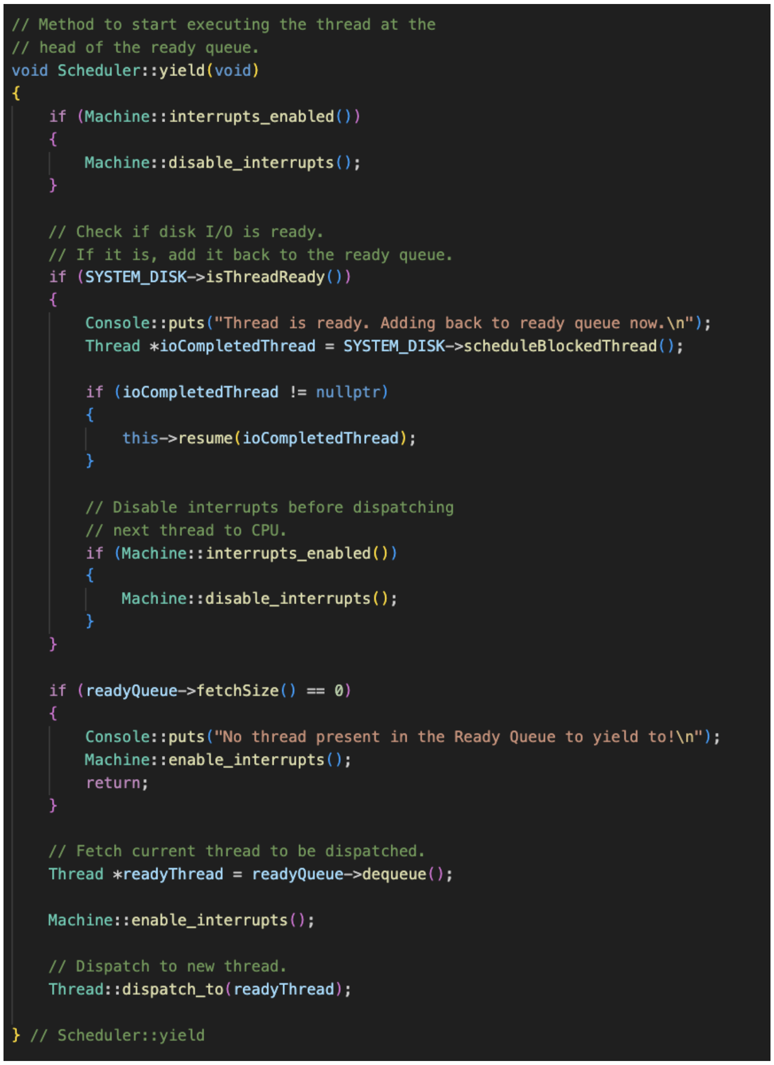
scheduler.C: Scheduler::Scheduler() : This is the constructor of this class. The Ready Queue is initialized within this method.



```
// Constructor.
Scheduler::Scheduler(void)
{
    readyQueue = new schedulerQueue();

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

} // Scheduler::Scheduler
```

Figure 4: Constructor of class `Scheduler`

scheduler.C: Scheduler::yield(void) : This method first disables all interrupts and pops the element from the head of the Ready Queue. It then utilizes the thread management library to dispatch to the popped thread. It enables all interrupts before returning. In MP6, this method has an additional responsibility. It is required to check if any threads present in the I/O blocked queue have gotten a response and now need to be added back to the Ready Queue.

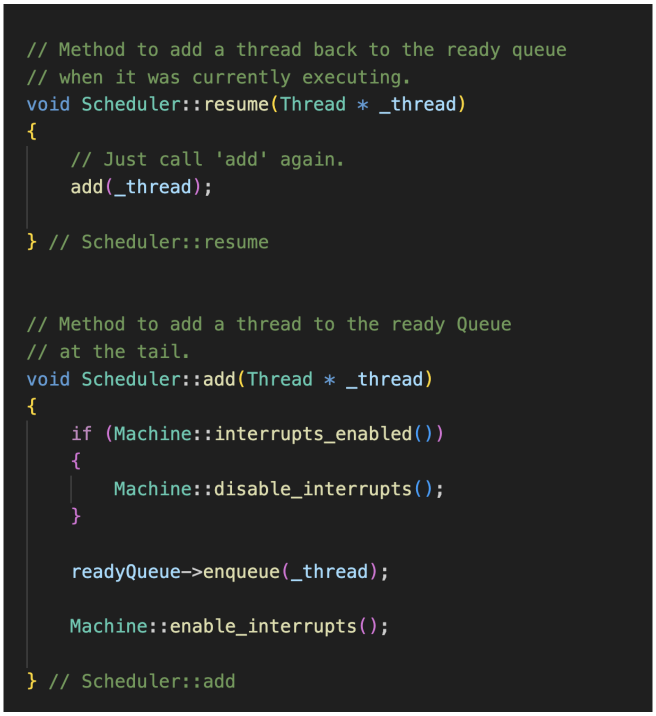


```
// Method to start executing the thread at the
// head of the ready queue.
void Scheduler::yield(void)
{
    if (Machine::interrupts_enabled())
    {
        Machine::disable_interrupts();
    }

    // Check if disk I/O is ready.
    // If it is, add it back to the ready queue.
    if (SYSTEM_DISK->isThreadReady())
    {
        Console::puts("Thread is ready. Adding back to ready queue now.\n");
        Thread *ioCompletedThread = SYSTEM_DISK->scheduleBlockedThread();

        if (ioCompletedThread != nullptr)
        {
            this->resume(ioCompletedThread);
        }

        // Disable interrupts before dispatching
        // next thread to CPU.
        if (Machine::interrupts_enabled())
        {
            Machine::disable_interrupts();
        }
    }

    if (readyQueue->fetchSize() == 0)
    {
        Console::puts("No thread present in the Ready Queue to yield to!\n");
        Machine::enable_interrupts();
        return;
    }

    // Fetch current thread to be dispatched.
    Thread *readyThread = readyQueue->dequeue();

    Machine::enable_interrupts();

    // Dispatch to new thread.
    Thread::dispatch_to(readyThread);

} // Scheduler::yield
```

Figure 5: `yield` method of class `Scheduler`

scheduler.C: Scheduler::add(Thread * _thread) : This method first disables all interrupts and adds a new thread to be executed at the tail of the ready queue. It enables all interrupts before returning.

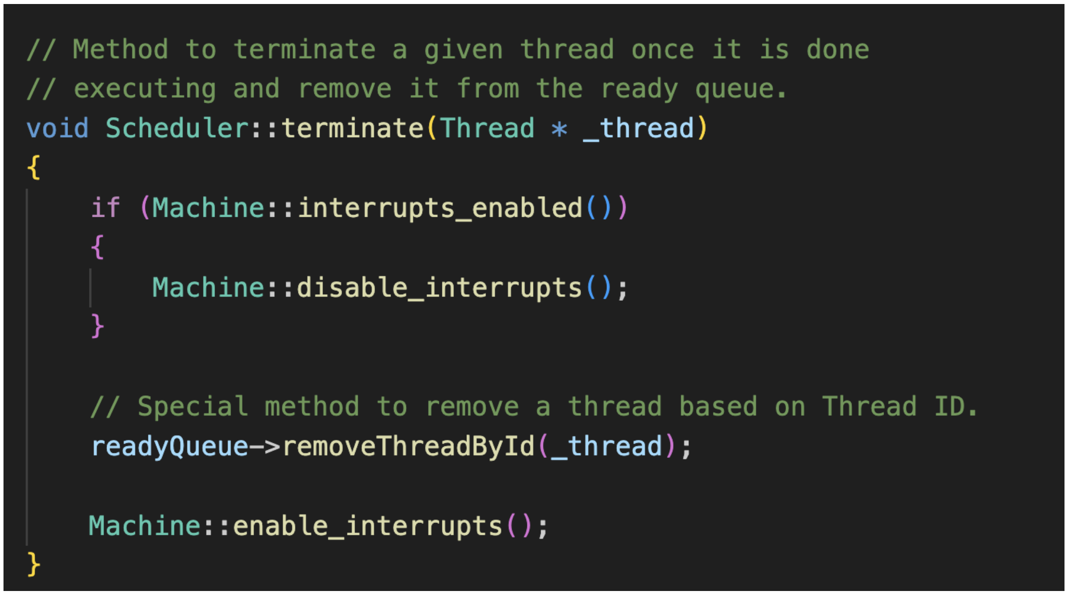
scheduler.C: Scheduler::resume(Thread * _thread) : This method is the same as `add` for all purposes. Hence, we just call add internally from here. It is primarily meant for those threads waiting on events. This method is called when a thread needs to yield execution. But before that it needs to be added back the end of the Ready Queue, for which it calls this method.



```
// Method to add a thread back to the ready queue
// when it was currently executing.
void Scheduler::resume(Thread * _thread)
{
    // Just call 'add' again.
    add(_thread);

} // Scheduler::resume


// Method to add a thread to the ready Queue
// at the tail.
void Scheduler::add(Thread * _thread)
{
    if (Machine::interrupts_enabled())
    {
        Machine::disable_interrupts();
    }

    readyQueue->enqueue(_thread);

    Machine::enable_interrupts();

} // Scheduler::add
```

Figure 6: `add` and `resume` methods of class `Scheduler`

scheduler.C: Scheduler::terminate(Thread * _thread) : This method is called when a terminating thread is required to release its resources. But the thread could have been added back to the ready queue and a search is required to be performed to delete it. This necessitates traversing the entire linked list and finding a match by Thread id.



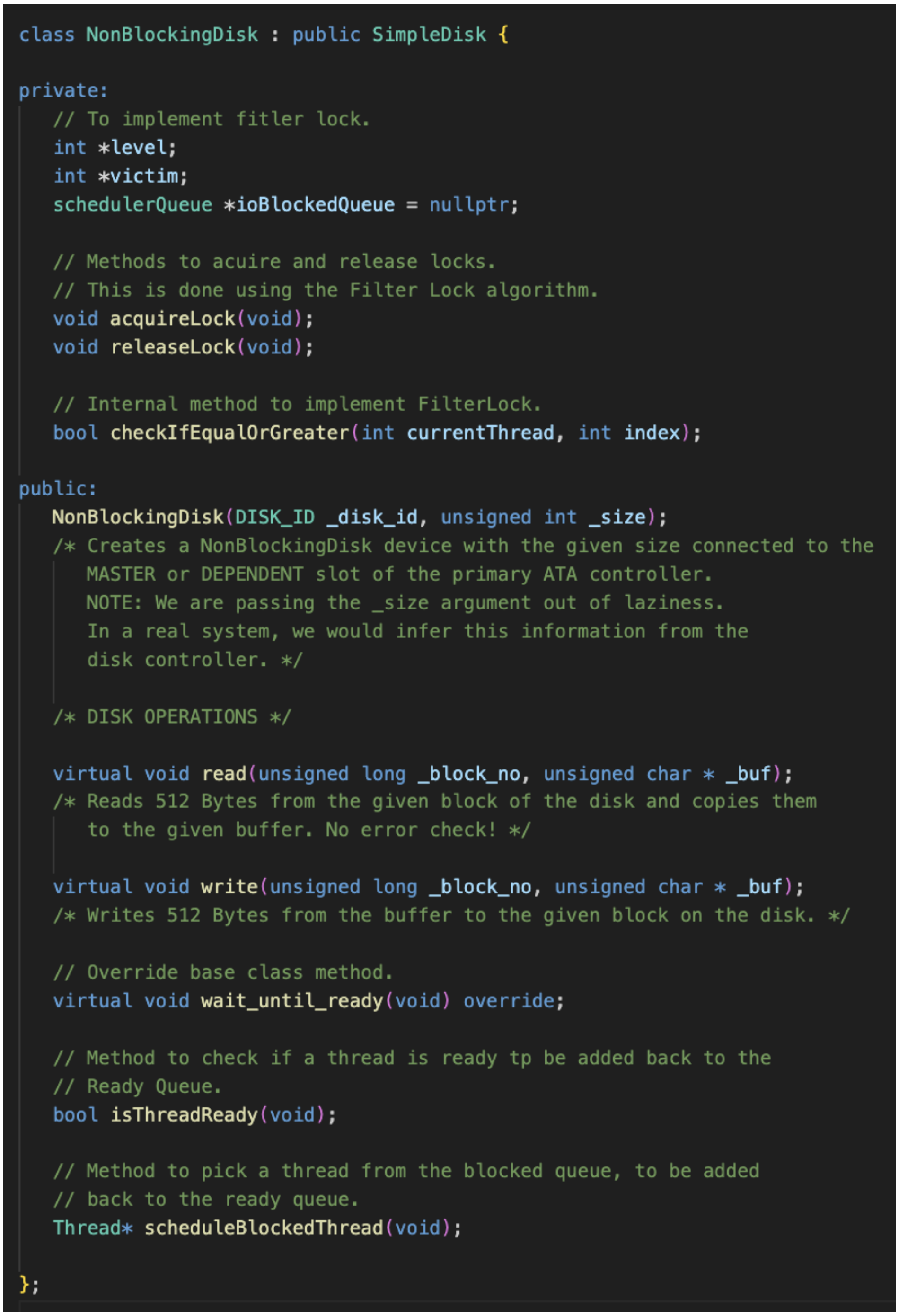
```
// Method to terminate a given thread once it is done
// executing and remove it from the ready queue.
void Scheduler::terminate(Thread * _thread)
{
    if (Machine::interrupts_enabled())
    {
        Machine::disable_interrupts();
    }

    // Special method to remove a thread based on Thread ID.
    readyQueue->removeThreadById(_thread);

    Machine::enable_interrupts();
}
```

Figure 7: `terminate` method of class `Scheduler`

nonblocking_disk.H : This header file has been modified to accommodate the members to handle the filter lock. Few methods have been added to support the blocked queue.

- `int *level` : Integer array to handler Filter Lock.

- `int *victim` : Integer array to handler Filter Lock.
- `schedulerQueue *ioBlockedQueue` : Pointer to the I/O blocked queue.
- `acquireLock` : Method to acquire a lock to a critical section.
- `releaseLock` : Method to release a after going through a critical section.
- `bool checkIfEqualOrGreater(int currentThread, int index)` : Method used in handling the Filter Lock algorithm.

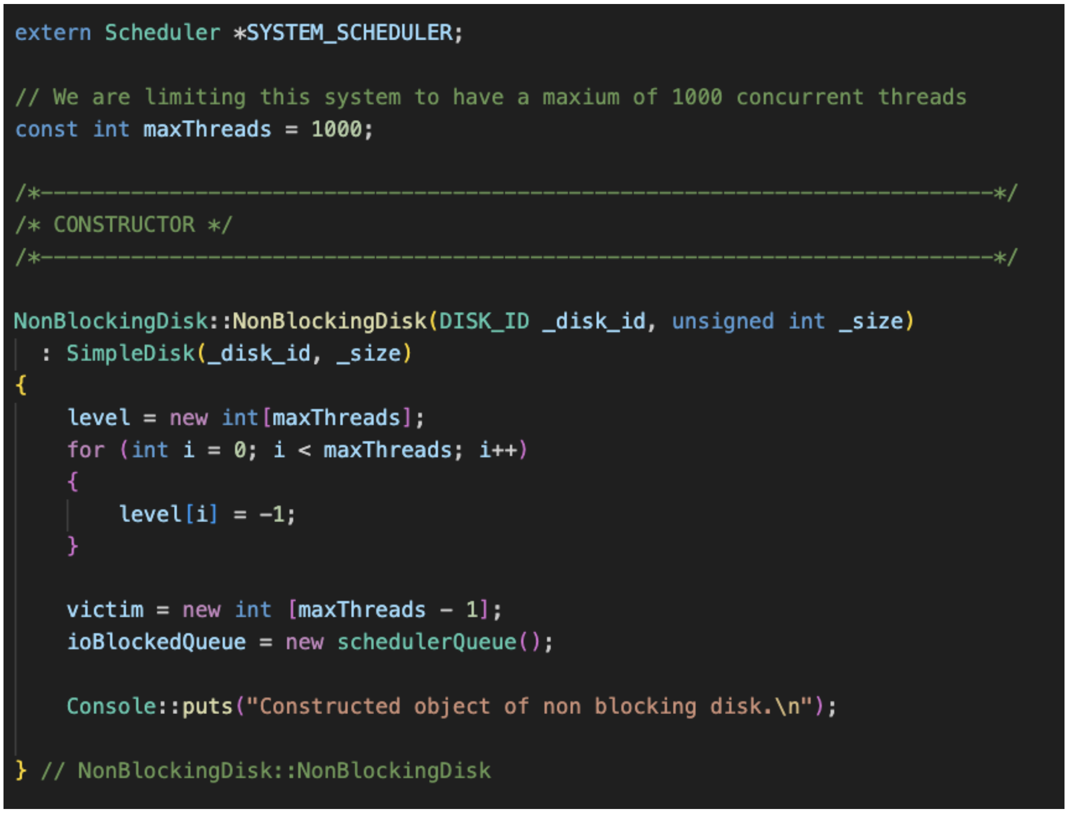


```
class NonBlockingDisk : public SimpleDisk {

private:
   // To implement fitler lock.
   int *level;
   int *victim;
   schedulerQueue *ioBlockedQueue = nullptr;

   // Methods to acuire and release locks.
   // This is done using the Filter Lock algorithm.
   void acquireLock(void);
   void releaseLock(void);

   // Internal method to implement FilterLock.
   bool checkIfEqualOrGreater(int currentThread, int index);

public:
   NonBlockingDisk(DISK_ID _disk_id, unsigned int _size);
   /* Creates a NonBlockingDisk device with the given size connected to the
      MASTER or DEPENDENT slot of the primary ATA controller.
      NOTE: We are passing the _size argument out of laziness.
      In a real system, we would infer this information from the
      disk controller. */

   /* 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. */

   // Override base class method.
   virtual void wait_until_ready(void) override;

   // Method to check if a thread is ready tp be added back to the
   // Ready Queue.
   bool isThreadReady(void);

   // Method to pick a thread from the blocked queue, to be added
   // back to the ready queue.
   Thread* scheduleBlockedThread(void);

};
```

Figure 8: `NonBlockingDisk` class Header file

nonblocking_disk.C: NonBlockingDisk::NonBlockingDisk : Parameterized constructor that does the following:

- Calling the base class `SimpleDisk` constructor with disk ID and size.
- Allocating and initializing an array `level` of size `maxThreads (1000)` to store lock levels for each thread, initialized to `-1`.
- Allocating a `victim` array for lock contention management.
- Creating an `ioBlockedQueue` for managing threads waiting on I/O operations.

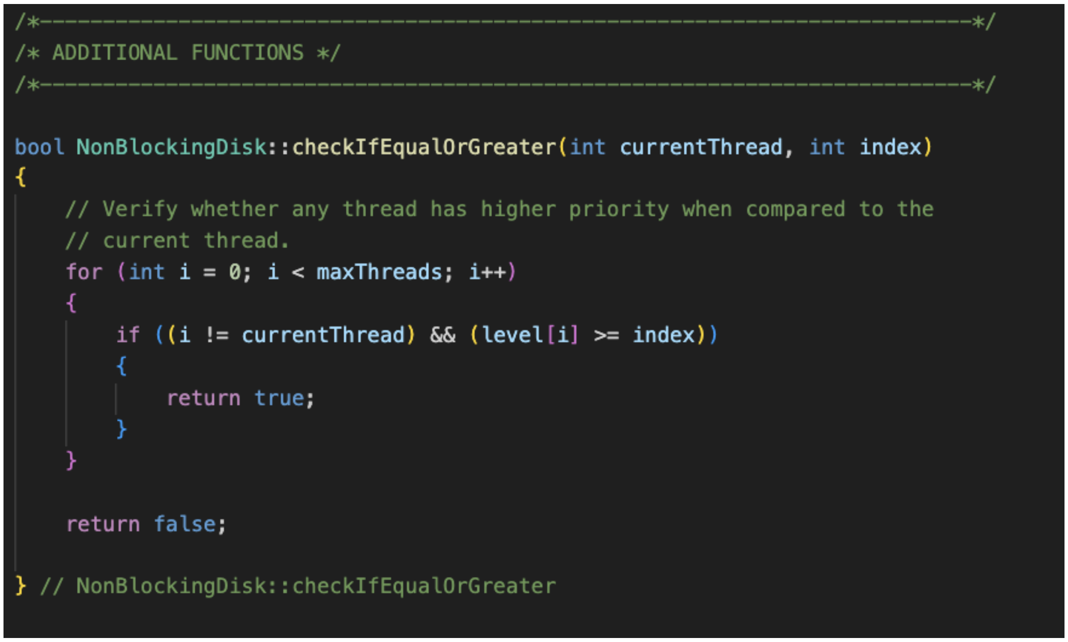


```
extern Scheduler *SYSTEM_SCHEDULER;

// We are limiting this system to have a maxium of 1000 concurrent threads
const int maxThreads = 1000;

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

NonBlockingDisk::NonBlockingDisk(DISK_ID _disk_id, unsigned int _size)
  : SimpleDisk(_disk_id, _size)
{
    level = new int[maxThreads];
    for (int i = 0; i < maxThreads; i++)
    {
        level[i] = -1;
    }

    victim = new int [maxThreads - 1];
    ioBlockedQueue = new schedulerQueue();

    Console::puts("Constructed object of non blocking disk.\n");

} // NonBlockingDisk::NonBlockingDisk
```

Figure 9: **class** `NonBlockingDisk` constructor

nonblocking_disk.C: NonBlockingDisk::checkIfEqualOrGreater

- Checks if any other thread has a lock level greater than or equal to `index`, excluding the `currentThread`.
- Returns `true` if such a thread exists; otherwise, `false`.
- Used in `acquireLock()` to ensure mutual exclusion.

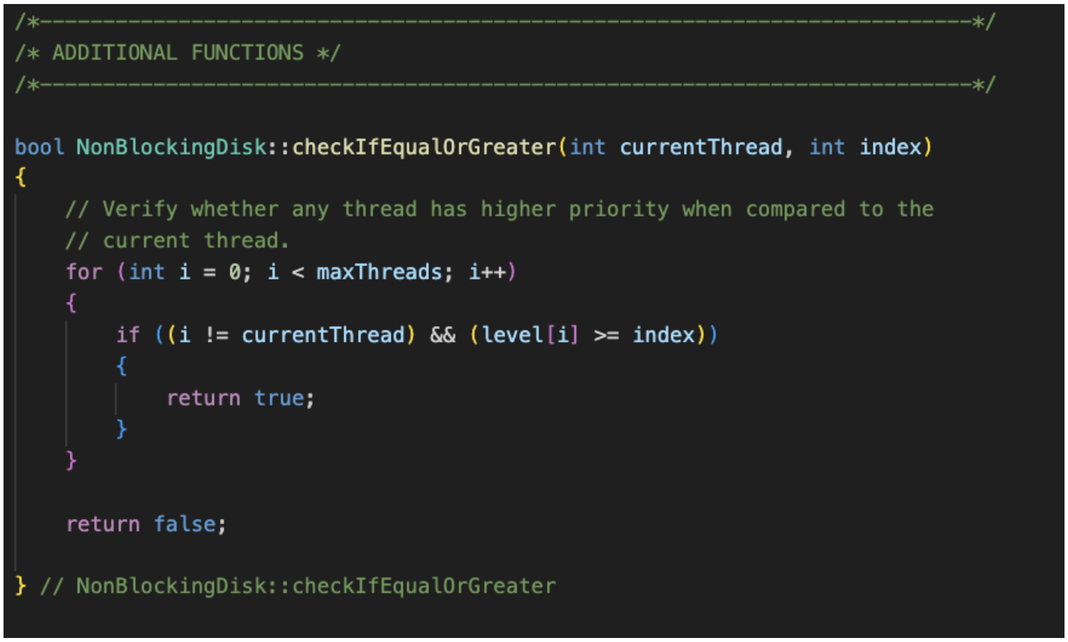


```cpp
/*-------------------------------------------------------------------------*/
/* ADDITIONAL FUNCTIONS */
/*-------------------------------------------------------------------------*/

bool NonBlockingDisk::checkIfEqualOrGreater(int currentThread, int index)
{
    // Verify whether any thread has higher priority when compared to the
    // current thread.
    for (int i = 0; i < maxThreads; i++)
    {
        if ((i != currentThread) && (level[i] >= index))
        {
            return true;
        }
    }

    return false;

} // NonBlockingDisk::checkIfEqualOrGreater
```

Figure 10: `NonBlockingDisk::checkIfEqualOrGreater`

nonblocking_disk.C: NonBlockingDisk::acquireLock

- Implements Peterson's algorithm for mutual exclusion across `maxThreads - 1` levels.
- Sets the current thread's lock level (`level[threadID]`) from `0` to `maxThreads - 2`.
- Marks the current thread as the "victim" for each level (`victim[i] = threadID`).
- Waits while another thread has an equal or higher level and is the victim.

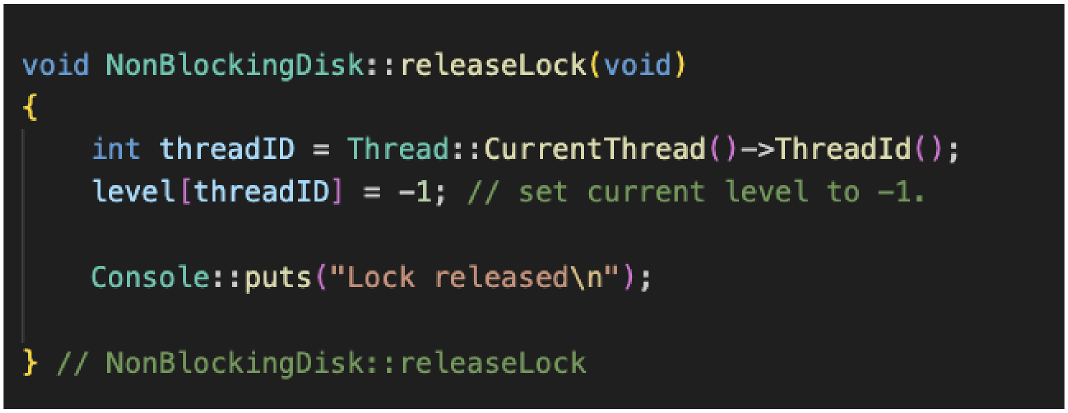


```cpp
/*-------------------------------------------------------------------------*/
/* ADDITIONAL FUNCTIONS */
/*-------------------------------------------------------------------------*/

bool NonBlockingDisk::checkIfEqualOrGreater(int currentThread, int index)
{
    // Verify whether any thread has higher priority when compared to the
    // current thread.
    for (int i = 0; i < maxThreads; i++)
    {
        if ((i != currentThread) && (level[i] >= index))
        {
            return true;
        }
    }

    return false;

} // NonBlockingDisk::checkIfEqualOrGreater
```

Figure 11: **class** `NonBlockingDisk::acquireLock`

nonblocking_disk.C: NonBlockingDisk::releaseLock

- Resets the lock level of the current thread to `-1`, indicating that the thread has released the lock.

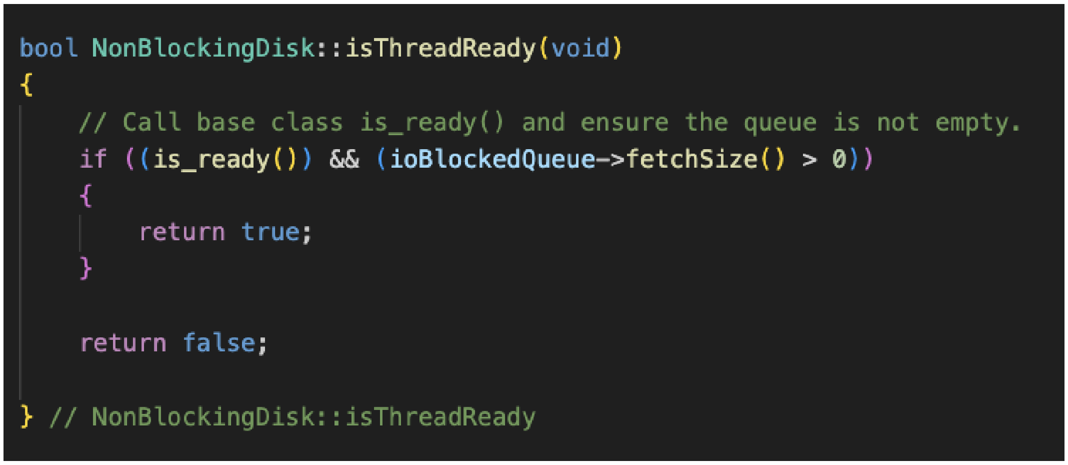


```
void NonBlockingDisk::releaseLock(void)
{
    int threadID = Thread::CurrentThread()->ThreadId();
    level[threadID] = -1; // set current level to -1.

    Console::puts("Lock released\n");

} // NonBlockingDisk::releaseLock
```

Figure 12: **class** `NonBlockingDisk::releaseLock`

nonblocking_disk.C: NonBlockingDisk::isThreadReady

- Checks if the disk is ready for I/O operations using `is_ready()` and if there are threads in the `ioBlockedQueue`.
- Returns `true` if both conditions are met; otherwise, `false`.

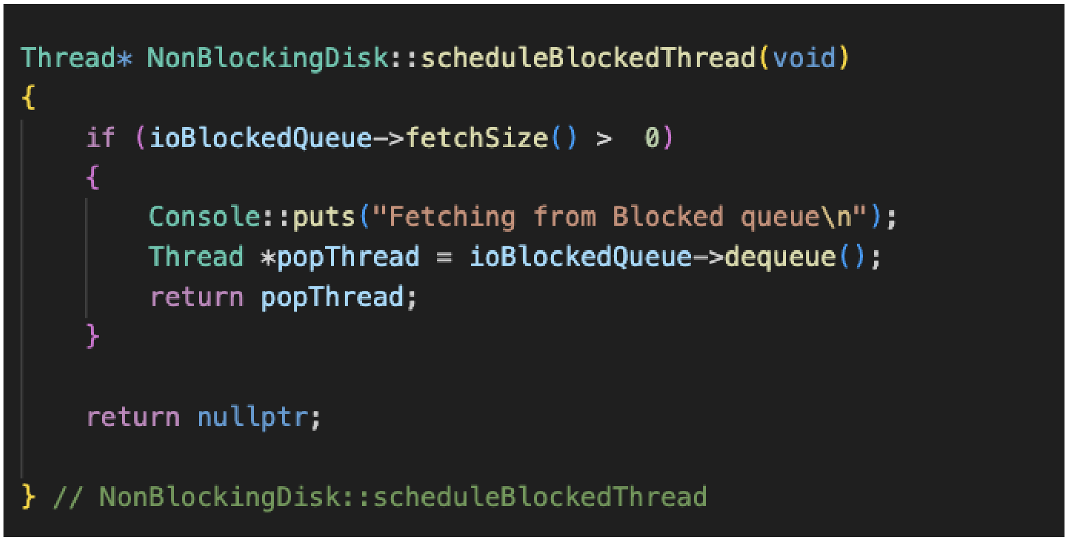


```
bool NonBlockingDisk::isThreadReady(void)
{
    // Call base class is_ready() and ensure the queue is not empty.
    if ((is_ready()) && (ioBlockedQueue->fetchSize() > 0))
    {
        return true;
    }

    return false;

} // NonBlockingDisk::isThreadReady
```

Figure 13: **class** `NonBlockingDisk::isThreadReady`

nonblocking_disk.C: NonBlockingDisk::scheduleBlockedThread

- Retrieves and removes the next thread from the `ioBlockedQueue` if it is not empty.
- Returns the dequeued thread or `nullptr` if the queue is empty.

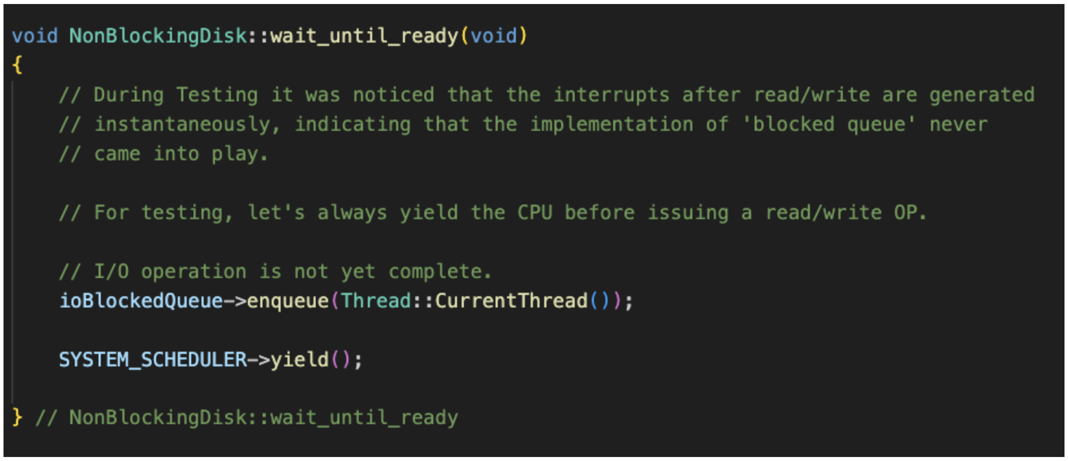


```
Thread* NonBlockingDisk::scheduleBlockedThread(void)
{
    if (ioBlockedQueue->fetchSize() >  0)
    {
        Console::puts("Fetching from Blocked queue\n");
        Thread *popThread = ioBlockedQueue->dequeue();
        return popThread;
    }

    return nullptr;

} // NonBlockingDisk::scheduleBlockedThread
```

Figure 14: **class** `NonBlockingDisk::scheduleBlockedThread`

nonblocking_disk.C: NonBlockingDisk::wait_until_ready

- Enqueues the current thread in the `ioBlockedQueue` using `ioBlockedQueue->enqueue(Thread::CurrentThread())`
- Yields the CPU using `SYSTEM_SCHEDULER->yield()`, allowing another thread to run while the I/O operation completes.
- This method helps handle non-blocking behavior by putting the current thread in a waiting state.

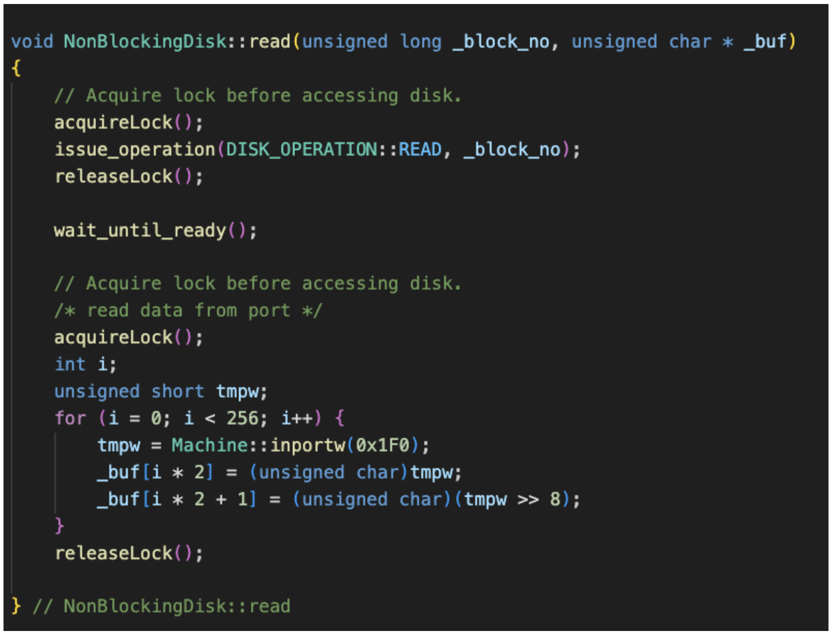


```
void NonBlockingDisk::wait_until_ready(void)
{
    // During Testing it was noticed that the interrupts after read/write are generated
    // instantaneously, indicating that the implementation of 'blocked queue' never
    // came into play.

    // For testing, let's always yield the CPU before issuing a read/write OP.

    // I/O operation is not yet complete.
    ioBlockedQueue->enqueue(Thread::CurrentThread());

    SYSTEM_SCHEDULER->yield();

} // NonBlockingDisk::wait_until_ready
```

Figure 15: **class** `NonBlockingDisk::wait_until_ready`

nonblocking_disk.C: NonBlockingDisk::read

- Acquires a lock before initiating a read operation using `acquireLock()`.
- Issues a read request to the specified block using `issue_operation(DISK_OPERATION::READ, _block_no)`.
- Releases the lock and calls `wait_until_ready()` to wait for I/O completion.
- Acquires the lock again to read data from the disk port.

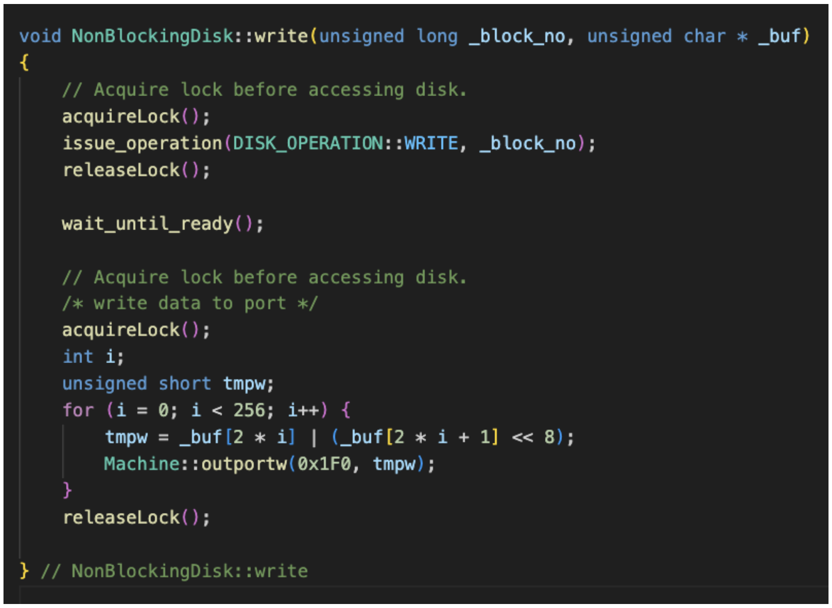


```
void NonBlockingDisk::read(unsigned long _block_no, unsigned char * _buf)
{
    // Acquire lock before accessing disk.
    acquireLock();
    issue_operation(DISK_OPERATION::READ, _block_no);
    releaseLock();

    wait_until_ready();

    // Acquire lock before accessing disk.
    /* read data from port */
    acquireLock();
    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);
    }
    releaseLock();

} // NonBlockingDisk::read
```

Figure 16: **class** `NonBlockingDisk::read`

nonblocking_disk.C: NonBlockingDisk::write

- Acquires a lock before initiating a write operation using `acquireLock()`.
- Issues a write request to the specified block using `issue_operation(DISK_OPERATION::WRITE, _block_no)`.
- Releases the lock and calls `wait_until_ready()` to wait for I/O completion.
- Acquires the lock again to write data to the disk port.

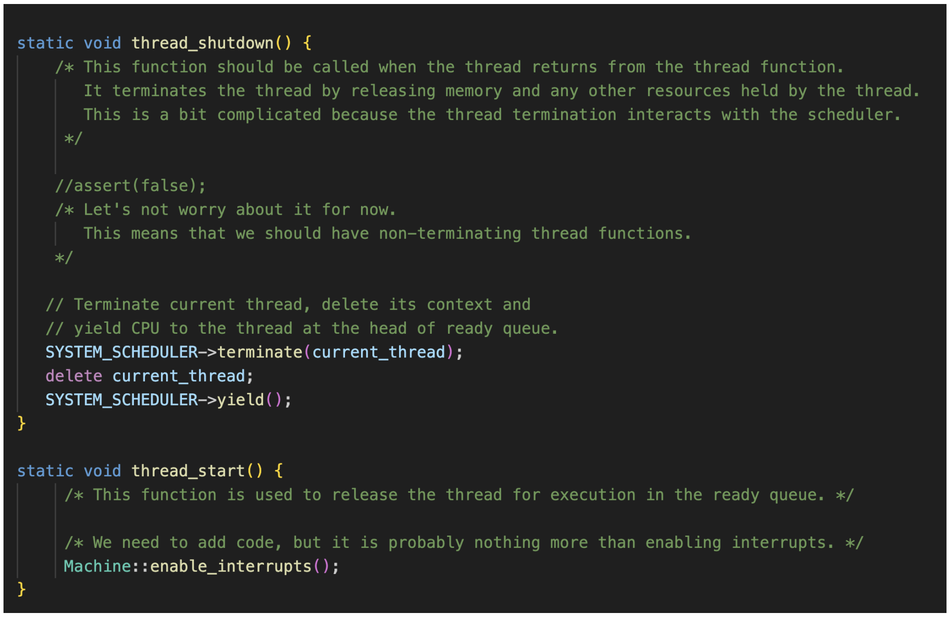


```
void NonBlockingDisk::write(unsigned long _block_no, unsigned char * _buf)
{
    // Acquire lock before accessing disk.
    acquireLock();
    issue_operation(DISK_OPERATION::WRITE, _block_no);
    releaseLock();

    wait_until_ready();

    // Acquire lock before accessing disk.
    /* write data to port */
    acquireLock();
    int i;
    unsigned short tmpw;
    for (i = 0; i < 256; i++) {
        tmpw = _buf[2 * i] | (_buf[2 * i + 1] << 8);
        Machine::outportw(0x1F0, tmpw);
    }
    releaseLock();

} // NonBlockingDisk::write
```

Figure 17: **class** `NonBlockingDisk::write`

thread.C: Thread::thread_start() : This method enables all interrupts as part of the thread management library.

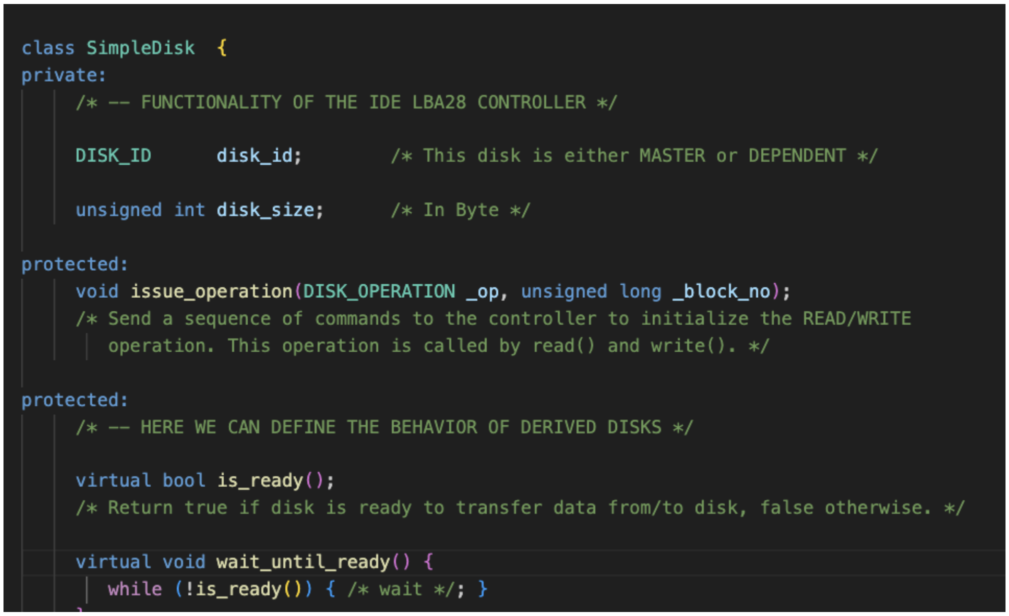
thread.C: Thread::thread_shutdown() : This method is called when a thread returns from a thread function. It releases all resources held by it by calling the method `terminate` and yields CPU to the head of queue.



```
static void thread_shutdown() {
    /* This function should be called when the thread returns from the thread function.
       It terminates the thread by releasing memory and any other resources held by the thread.
       This is a bit complicated because the thread termination interacts with the scheduler.
     */

    //assert(false);
    /* Let's not worry about it for now.
       This means that we should have non-terminating thread functions.
    */

   // Terminate current thread, delete its context and
   // yield CPU to the thread at the head of ready queue.
   SYSTEM_SCHEDULER->terminate(current_thread);
   delete current_thread;
   SYSTEM_SCHEDULER->yield();
}

static void thread_start() {
     /* This function is used to release the thread for execution in the ready queue. */

     /* We need to add code, but it is probably nothing more than enabling interrupts. */
     Machine::enable_interrupts();
}
```

Figure 18: `start` and `shutdown` methods of the Thread Management Library

simple_disk.H : The method `issue_operation` has been made protected so that derived classes can access them without making it a public interface.

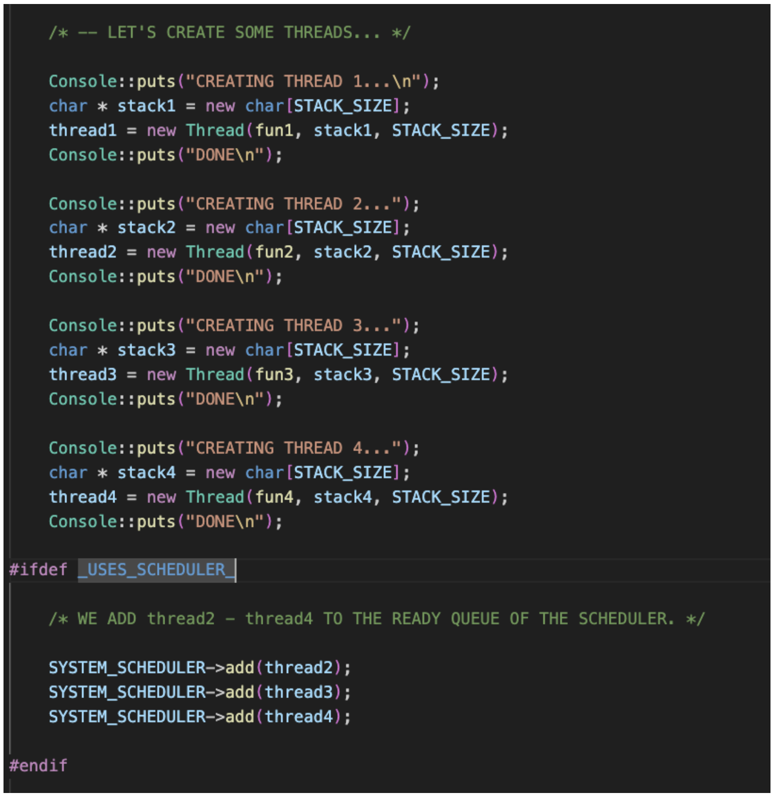


```
class SimpleDisk  {
private:
     /* -- FUNCTIONALITY OF THE IDE LBA28 CONTROLLER */

     DISK_ID      disk_id;        /* This disk is either MASTER or DEPENDENT */

     unsigned int disk_size;      /* In Byte */

protected:
     void issue_operation(DISK_OPERATION _op, unsigned long _block_no);
     /* Send a sequence of commands to the controller to initialize the READ/WRITE
        operation. This operation is called by read() and write(). */

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

     virtual bool is_ready();
     /* Return true if disk is ready to transfer data from/to disk, false otherwise. */

     virtual void wait_until_ready() {
        while (!is_ready()) { /* wait */; }
```

Figure 19: **Changes to class** `SimpleDisk`

kernel.C :

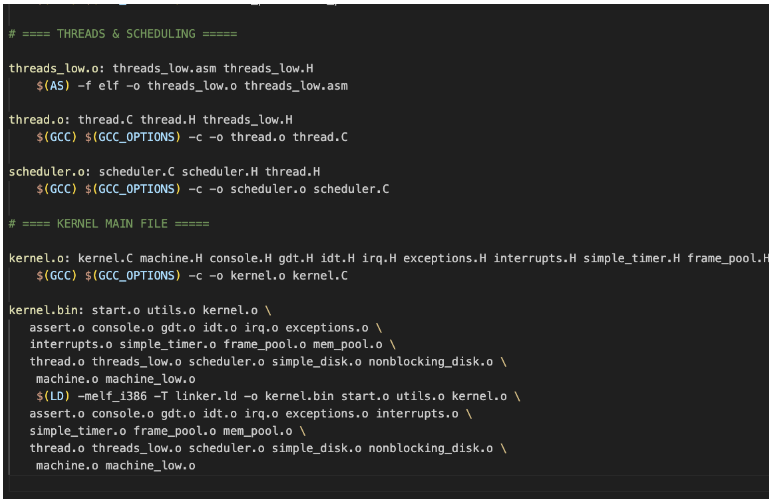
- The stack has been increased to 4KB.
- Un-commented macros to use the scheduler.
- Created object of class `NonBlockingDisk` instead of `SimpleDisk`.



```
    /* -- LET'S CREATE SOME THREADS... */

    Console::puts("CREATING THREAD 1...\n");
    char * stack1 = new char[STACK_SIZE];
    thread1 = new Thread(fun1, stack1, STACK_SIZE);
    Console::puts("DONE\n");

    Console::puts("CREATING THREAD 2...");
    char * stack2 = new char[STACK_SIZE];
    thread2 = new Thread(fun2, stack2, STACK_SIZE);
    Console::puts("DONE\n");

    Console::puts("CREATING THREAD 3...");
    char * stack3 = new char[STACK_SIZE];
    thread3 = new Thread(fun3, stack3, STACK_SIZE);
    Console::puts("DONE\n");

    Console::puts("CREATING THREAD 4...");
    char * stack4 = new char[STACK_SIZE];
    thread4 = new Thread(fun4, stack4, STACK_SIZE);
    Console::puts("DONE\n");

#ifdef _USES_SCHEDULER_

    /* WE ADD thread2 - thread4 TO THE READY QUEUE OF THE SCHEDULER. */

    SYSTEM_SCHEDULER->add(thread2);
    SYSTEM_SCHEDULER->add(thread3);
    SYSTEM_SCHEDULER->add(thread4);

#endif
```

Figure 20: **Modification to the Kernel file**

makefile : Chnages have been made to compile and link files `scheduler.H` and `scheduler.C`.

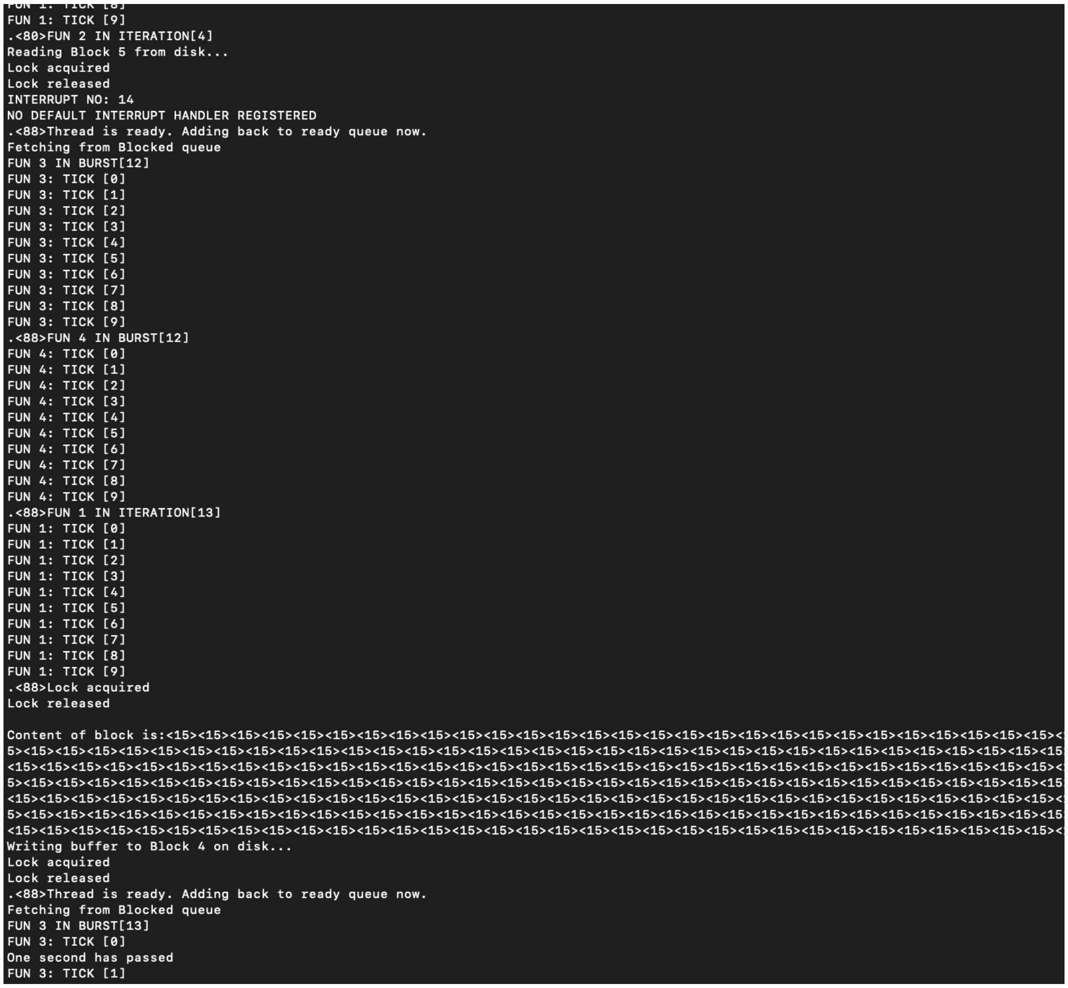


```
# ==== THREADS & SCHEDULING =====

threads_low.o: threads_low.asm threads_low.H
    $(AS) -f elf -o threads_low.o threads_low.asm

thread.o: thread.C thread.H threads_low.H
    $(GCC) $(GCC_OPTIONS) -c -o thread.o thread.C

scheduler.o: scheduler.C scheduler.H thread.H
    $(GCC) $(GCC_OPTIONS) -c -o scheduler.o scheduler.C

# ==== KERNEL MAIN FILE =====

kernel.o: kernel.C machine.H console.H gdt.H idt.H irq.H exceptions.H interrupts.H simple_timer.H frame_pool.H
    $(GCC) $(GCC_OPTIONS) -c -o kernel.o kernel.C

kernel.bin: start.o utils.o kernel.o \
   assert.o console.o gdt.o idt.o irq.o exceptions.o \
   interrupts.o simple_timer.o frame_pool.o mem_pool.o \
   thread.o threads_low.o scheduler.o simple_disk.o nonblocking_disk.o \
    machine.o machine_low.o
    $(LD) -melf_i386 -T linker.ld -o kernel.bin start.o utils.o kernel.o \
   assert.o console.o gdt.o idt.o irq.o exceptions.o interrupts.o \
   simple_timer.o frame_pool.o mem_pool.o \
   thread.o threads_low.o scheduler.o simple_disk.o nonblocking_disk.o \
    machine.o machine_low.o
```

Figure 21: **Changes to the `makefile`**

Testing

In the test-suite, `func2()` issues the I/O operations. Every time it issues an operation, it gets context switched out as shown in the diagram. Logging relating to acquiring locks is also seen. Detailed logs can be found under *../outputs/non-blocking-run.log*



```
FUN 1: TICK [8]
FUN 1: TICK [9]
.<80>FUN 2 IN ITERATION[4]
Reading Block 5 from disk...
Lock acquired
Lock released
INTERRUPT NO: 14
NO DEFAULT INTERRUPT HANDLER REGISTERED
.<88>Thread is ready. Adding back to ready queue now.
Fetching from Blocked queue
FUN 3 IN BURST[12]
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]
.<88>FUN 4 IN BURST[12]
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]
.<88>FUN 1 IN ITERATION[13]
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]
.<88>Lock acquired
Lock released

Content of block is:<15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><
5><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15
<15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><
5><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15
<15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><
5><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15
<15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><15><
Writing buffer to Block 4 on disk...
Lock acquired
Lock released
.<88>Thread is ready. Adding back to ready queue now.
Fetching from Blocked queue
FUN 3 IN BURST[13]
FUN 3: TICK [0]
One second has passed
FUN 3: TICK [1]
```

Figure 22: Test run output