MP6: Primitive Disk Device Driver

Harsh Wadhawe
UIN: 936001477
CSCE 611: Operating Systems

Assigned Tasks

Main: Completed.
Bonus Option 1: Completed (design only).
Bonus Option 2: Not implemented (covered by Option 1 design).
Bonus Option 3: Completed (interrupt-driven I/O).

System Design

The goal of MP6 is to eliminate busy-waiting in disk operations by implementing a **NonBlockingDisk** driver and integrating it with a simple **FIFO scheduler**. The system ensures that when a thread waits for disk I/O, it yields the CPU so that other threads can run.

The overall architecture has three main components:

- **FIFO Scheduler**: Implements a ready queue of runnable threads. When a thread yields, it is enqueued; the scheduler then chooses the next thread in FIFO order.
- **NonBlockingDisk**: Extends `SimpleDisk` to replace polling-based waiting with blocking and yielding. Threads that encounter a busy disk are placed on a blocked queue.
- **Interrupt-driven I/O (Bonus 3)**: `NonBlockingDisk` also implements `InterruptHandler` and registers for IRQ14. When the disk signals completion, an interrupt wakes blocked threads immediately.

Logically, the design follows the standard top-half/bottom-half device driver model:

- **Top-half:** Synchronous methods like `read()` and `write()` that issue commands and call `wait_while_busy()`.
- **Bottom-half:** `handle_interrupt()` that runs on IRQ14 and wakes the appropriate waiting threads.

Code Description

The following files were modified:

- `scheduler.H`, `scheduler.C`: FIFO scheduler implementation.
- `nonblocking_disk.H`, `nonblocking_disk.C`: Non-blocking, interrupt-driven disk driver.
- `kernel.C`: System initialization and device wiring.
- `makefile`: Added appropriate dependencies.

To build from the `MP6_Sources` directory:

```
make clean
make
```

Scheduler

```
Scheduler::Scheduler() {
  /* Initialize the FIFO ready queue to empty state */
  ready_queue_head = nullptr;
  ready_queue_tail = nullptr;
  queue_size = 0;

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

Figure 1: Scheduler constructor and queue data structure (code snippet)

scheduler.C: Scheduler() Initializes an empty FIFO ready queue implemented as a singly linked list. The scheduler stores pointers to the `head` and `tail` nodes, where each node contains a pointer to a `Thread` and a pointer to the next node. This supports O(1) enqueue at the tail.

scheduler.C: yield() Implements the core scheduling logic. The current thread (if still runnable) is appended to the ready queue. The scheduler then removes the next thread from the head of the queue and calls `Thread::dispatch_to()` to switch context. If the ready queue is empty, the current thread continues execution.

scheduler.C: resume() Adds a runnable thread to the tail of the ready queue. This is used when threads become ready again after waiting on events (e.g., disk I/O completion). The method allocates a new node, updates the tail pointer, and handles the empty-queue case correctly.

scheduler.C: terminate() Removes a thread from the ready queue in preparation for destruction or self-termination. It searches the queue for the given thread and fixes `head` and `tail` as needed. Corner cases such as the thread being at the head, tail, or not present are handled gracefully.

NonBlockingDisk

nonblocking_disk.H: Class Definition `NonBlockingDisk` inherits from `SimpleDisk` and `InterruptHandler`. It reuses the ATA protocol implementation from `SimpleDisk` and adds:

- A blocked-thread queue (linked list) for threads waiting on disk readiness.
- An overridden `wait_while_busy()` method.
- A `handle_interrupt()` method for IRQ14.

```
void Scheduler::yield() {

  Thread* current_thread = Thread::CurrentThread();

  /* If there's a current thread, add it back to the ready queue */
  if (current_thread != nullptr) {
    resume(current_thread);
  }

  /* Select the next thread from the ready queue */
  Thread* next_thread = nullptr;

  if (ready_queue_head != nullptr) {
    /* Dequeue the head of the queue (FIFO) */
    QueueNode* node_to_remove = ready_queue_head;
    next_thread = node_to_remove->thread;

    /* Update queue pointers */
    ready_queue_head = ready_queue_head->next;
    if (ready_queue_head == nullptr) {
      /* Queue is now empty */
      ready_queue_tail = nullptr;
    }

    /* Delete the node and update size */
    delete node_to_remove;
    queue_size--;

    /* Dispatch to the next thread */
    Thread::dispatch_to(next_thread);
  }
  /* If no thread in queue, the current thread continues running */
}
```

Figure 2: `yield()` implementation (code snippet)

```
void Scheduler::resume(Thread * _thread) {

  if (_thread == nullptr) {
    return;  /* Safety check: don't add null threads */
  }

  /* Create a new node for this thread */
  QueueNode* new_node = new QueueNode(_thread);

  if (ready_queue_tail == nullptr) {
    /* Queue is empty - this becomes both head and tail */
    ready_queue_head = new_node;
    ready_queue_tail = new_node;
  } else {
    /* Add to the tail of the queue (FIFO enqueue) */
    ready_queue_tail->next = new_node;
    ready_queue_tail = new_node;
  }

  queue_size++;
}
```

Figure 3: `resume()` implementation (code snippet)

nonblocking_disk.C: NonBlockingDisk() Initializes the blocked thread queue and registers the object as the IRQ14 handler. IRQ14 is the primary IDE disk interrupt. This registration enables the interrupt-driven I/O path: when the controller is ready, the interrupt subsystem calls `handle_interrupt()`.

nonblocking_disk.C: wait_while_busy() Overrides the busy-wait loop from `SimpleDisk`. Instead of polling in a tight loop:

```cpp
void Scheduler::terminate(Thread * _thread) {

  if (_thread == nullptr) {
    return;  /* Safety check */
  }

  /* Search for the thread in the ready queue */
  QueueNode* current = ready_queue_head;
  QueueNode* previous = nullptr;

  while (current != nullptr) {
    if (current->thread == _thread) {
      /* Found the thread - remove it from the queue */

      if (previous == nullptr) {
        /* Thread is at the head of the queue */
        ready_queue_head = current->next;
        if (ready_queue_head == nullptr) {
          /* Queue is now empty */
          ready_queue_tail = nullptr;
        }
      } else {
        /* Thread is in the middle or tail */
        previous->next = current->next;
        if (current == ready_queue_tail) {
          /* Thread was at the tail */
          ready_queue_tail = previous;
        }
      }

      /* Delete the node and update size */
      delete current;
      queue_size--;
      return;  /* Thread found and removed */
    }

    previous = current;
    current = current->next;
  }
```

Figure 4: `terminate()` implementation (code snippet)

```cpp
class NonBlockingDisk : public SimpleDisk, public InterruptHandler {
private:
  /* Blocked Thread Queue
   * This queue holds threads that are waiting for the disk to become ready.
   * Since only one thread accesses the disk at a time (base implementation),
   * this queue will typically have at most one thread.
   */
  struct BlockedThreadNode {
    Thread* thread;                 /* Pointer to the blocked thread */
    BlockedThreadNode* next;        /* Pointer to the next node */

    BlockedThreadNode(Thread* t) : thread(t), next(nullptr) {}
  };

  BlockedThreadNode* blocked_queue_head;  /* Head of the blocked thread queue */
  BlockedThreadNode* blocked_queue_tail; /* Tail for efficient enqueue */

  /* Flag to track if we're waiting for an interrupt */
  bool waiting_for_interrupt;  /* True when a thread is blocked waiting for disk */

protected:
  /* Override wait_while_busy() to use interrupt-driven approach.
   * Threads block and wait for IRQ14 interrupt instead of polling.
   */
  virtual void wait_while_busy();

  /* Helper function to wake up the next waiting thread (if any) */
  void wake_next_blocked_thread();

public:
  /* Interrupt Handler Implementation (BONUS OPTION 3) */
  virtual void handle_interrupt(REGS * _regs);
  /* This is called when IRQ14 (disk interrupt) occurs.
   * When the disk becomes ready, this function:
   * 1. Checks if disk is ready
   * 2. Wakes up the waiting thread (if any)
   * 3. Allows the thread to continue with data transfer
   */
```

Figure 5: Class structure of `NonBlockingDisk` (code snippet)

1. Check if the disk is busy.
2. If busy, add the current thread to the blocked queue and call the scheduler to yield the CPU.

```
NonBlockingDisk::NonBlockingDisk(unsigned int _size)
  : SimpleDisk(_size) {
  /* Initialize the blocked thread queue to empty state */
  blocked_queue_head = nullptr;
  blocked_queue_tail = nullptr;
  waiting_for_interrupt = false;

  /* Ensure scheduler is available (it should be created before the disk) */
  assert(System::SCHEDULER != nullptr);

  /* BONUS OPTION 3: Register as IRQ14 interrupt handler for disk interrupts
   * IRQ14 is the IDE primary channel interrupt, which fires when the disk
   * is ready for data transfer.
   */
  const unsigned int IRQ_DISK = 14;  /* IRQ14 is the disk interrupt */
  InterruptHandler::register_handler(IRQ_DISK, this);

  Console::puts("NonBlockingDisk: Registered as IRQ14 (disk) interrupt handler\n");
}
```

Figure 6: `NonBlockingDisk` constructor and IRQ registration

```
void NonBlockingDisk::wait_while_busy() {
  Thread* current_thread = Thread::CurrentThread();

  if (current_thread == nullptr || System::SCHEDULER == nullptr) {
    /* Fallback: if no scheduler or thread, use busy-wait (shouldn't happen) */
    while (is_busy()) {
      /* Minimal busy-wait as fallback */
    }
    return;
  }

  /* Loop until disk is ready */
  while (is_busy()) {
    /* Disk is busy - block thread and wait for interrupt */

    /* Check if thread is already in blocked queue (avoid duplicates) */
    bool thread_in_queue = false;
    BlockedThreadNode* node = blocked_queue_head;
    while (node != nullptr) {
      if (node->thread == current_thread) {
        thread_in_queue = true;
        break;
      }
      node = node->next;
    }

    /* Add current thread to blocked queue if not already there */
    if (!thread_in_queue) {
      BlockedThreadNode* new_node = new BlockedThreadNode(current_thread);

      if (blocked_queue_tail == nullptr) {
        /* Queue is empty - this becomes both head and tail */
        blocked_queue_head = new_node;
        blocked_queue_tail = new_node;
      } else {
        /* Add to the tail of the queue */
        blocked_queue_tail->next = new_node;
        blocked_queue_tail = new_node;
      }

      /* Set flag to indicate we're waiting for an interrupt */
      waiting_for_interrupt = true;
    }
```

Figure 7: `wait_while_busy()` logic (code snippet)

3. When the thread is resumed (after an interrupt), re-check the disk status.
4. Exit once the disk is ready.

This removes CPU spinning and lets other threads run during disk waits.

nonblocking_disk.C: handle_interrupt() Implements the bottom-half of the device driver. When IRQ14 fires:

- Confirm that the disk is ready.
- Remove the next thread from the blocked queue.

```
void NonBlockingDisk::handle_interrupt(REGS * _regs) {
  /* BONUS OPTION 3: Interrupt-Driven I/O Bottom Half
   *
   * This function is called when IRQ14 (disk interrupt) fires, indicating
   * that the disk has completed its operation and is ready for data transfer.
   *
   * Architecture:
   * - Top-half: read()/write() issue commands via ide_ata_issue_command()
   * - Bottom-half: This function wakes waiting threads when disk is ready
   *
   * Algorithm:
   * 1. Check if disk is ready (not busy)
   * 2. If ready and there are blocked threads, wake the next one
   * 3. The woken thread will check disk status and continue with data transfer
   *
   * Note: The actual data transfer (reading/writing bytes) happens in the
   * top-half (read()/write() functions) after the thread wakes up.
   */

  /* Check if disk is ready for data transfer */
  if (!is_busy()) {
    /* Disk is ready - wake up the next waiting thread */
    wake_next_blocked_thread();
  }
```

Figure 8: `handle_interrupt()` bottom-half implementation

- Place that thread back on the scheduler's ready queue via `resume()`.

The woken thread then continues its original disk operation.

Kernel Integration

kernel.C: main()

- `#define _USES_SCHEDULER_` is enabled to activate the scheduler.
- The scheduler is created *before* the disk so that `NonBlockingDisk` can call it.
- When the scheduler is enabled, `System::DISK` is initialized as `NonBlockingDisk`; otherwise, the fallback is `SimpleDisk`.

Bonus Option 1: Thread-Safe Disk Design

Bonus Option 1 is a design-only task. I created a separate document (`design.md`) that analyzes how to make the disk system thread-safe if multiple threads could access it concurrently.

The design identifies five major categories of race conditions:

1. **Concurrent read/write operations** (register and command corruption).
2. **Blocked queue races** (corrupted linked list, lost wake-ups, duplicates).
3. **Interrupt handler races** (handler and threads modifying shared state concurrently).
4. **Disk status races** (TOCTOU issues and spurious wake-ups).
5. **Request processing races** (request/owner mismatch, lost requests).

For each category, the design proposes concrete solutions:

- A **disk operation lock** using `Machine::disable_interrupts()` and `Machine::enable_interrupts()` around critical sections.
- A **FIFO request queue** structure to serialize disk commands.
- Atomic, interrupt-safe operations for both the blocked queue and the request queue.
- Explicit request structures that track the owning thread and completion status.
- Clear rules for what the interrupt handler is allowed to read and modify.

This design serves as a complete roadmap for a future implementation (Bonus Option 2), without changing the visible interface of `NonBlockingDisk`.

Bonus Option 2: Not Implemented

Bonus Option 2 (full thread-safe implementation) was not coded. Based on the Option 1 design, the implementation would require:

- Introducing a global or per-disk lock around all low-level disk operations.
- Implementing the request queue and wiring it into `read()`, `write()`, and the interrupt handler.
- Protecting all accesses to shared queues and state with short critical sections.
- Ensuring the interrupt handler only touches data under well-defined locking rules.

The design document estimates roughly 100–150 lines of additional code across 2–3 files to complete this option.

Bonus Option 3: Interrupt-Driven I/O

Bonus Option 3 was implemented as part of the `NonBlockingDisk` design.

- **Multiple inheritance:** `NonBlockingDisk` inherits from `SimpleDisk` and `InterruptHandler`.
- **IRQ registration:** The constructor registers the disk as the IRQ14 handler; IRQ14 is triggered when the IDE disk controller is ready.
- **Blocking model:** `wait_while_busy()` blocks the calling thread, placing it on a blocked queue instead of spinning.
- **Wake-up model:** `handle_interrupt()` runs when the disk is ready, removes a thread from the blocked queue, and resumes it via the scheduler.

This yields a clean top-half/bottom-half separation and removes polling overhead entirely.

Testing

Testing Performed:

- **Build and integration:** Verified that the project compiles without errors and links correctly after modifying the makefile and headers.
- **Basic kernel run:** Booted the kernel with the FIFO scheduler and `NonBlockingDisk` enabled. Checked that threads are created, scheduled, and terminated as expected.
- **Disk behavior:** Confirmed that disk operations complete and that the system no longer busy-waits in `wait_while_busy()`. While one thread waits for disk I/O, other threads can run.
- **Interrupt path:** Verified that IRQ14 is registered and that the interrupt handler is invoked (e.g., via console/log messages). Confirmed that waiting threads are woken and re-enter the ready queue.
- **Scheduler edge cases:** Tested scenarios with an empty ready queue, a single runnable thread, and threads calling `terminate()` on themselves.

Not Tested / Out of Scope:

- Stress tests with many threads issuing concurrent disk operations.
- Full thread-safe behavior under true parallel disk access (belongs to Bonus Option 2).
- Error handling for disk hardware failures or spurious interrupts.
- Performance benchmarks or precise timing measurements.

The testing assumes correct usage of the provided interfaces by an intelligent caller, in line with the MP instructions. Within this scope, the implementation behaves correctly and eliminates busy-waiting for disk I/O.

```
[~/Code/Fall 25/OS/Machine Problems/MP6/MP6_Sources (main ×) make run
Darwin
qemu-system-x86_64 -kernel kernel.bin -serial stdio \
-device piix3-ide,id=ide -drive id=disk,file=c.img,format=raw,if=none -device ide-hd,drive=disk,bus=ide.0
Installed exception handler at ISR <0>
Allocating Memory Pool... done
Installed interrupt handler at IRQ <0>
Constructed Scheduler.
Installed interrupt handler at IRQ <14>
NonBlockingDisk: Registered as IRQ14 (disk) interrupt handler
Hello World!
CREATING THREAD 1...
esp = <2098136>
done
DONE
CREATING THREAD 2...esp = <2099184>
done
DONE
CREATING THREAD 3...esp = <2100232>
done
DONE
CREATING THREAD 4...esp = <2101280>
done
DONE
STARTING THREAD 1 ...
```

Figure 9: Initial kernel run with NonBlockingDisk

```
Reading Block 1 from disk...

Content of block is:<1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><
1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><
1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><
1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><
1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><
1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><
1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><
1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><
1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1><1>
Writing buffer to Block 0 on disk...
```

Figure 10: Reading from a disk block using NonBlockingDisk