MP6: Simple Kernel Threads

Sathvik Kote Revanasiddappa
UIN: 634005706
CSCE611: Operating System

Assigned Tasks

Main: Add a layer on top of busy waiting device to support the same blocking read and write operations as the basic implementation, but without busy waiting in the device driver code. : Completed.

Bonus:

- Support for Disk Mirroring: Did not attempt
- Using Interrupts for Concurrency: Did not attempt.
- Design of a thread-safe disk system: Completed
- Implementation of a thread-safe disk system: Complete.

System Design

Objective

The goal of this machine problem is to add a layer on top of a simple device driver to support blocking read and write operations as the basic implementation, but without busy waiting in the device driver code. The user should be able to call 'read' and 'write' operations without worrying that the call may either return prematurely or tie up the entire system waiting for the device to return.

Design of Blocking queue:

The idea for the design is simillar to the scheduler design. Each disk operation is held in a queue if the disk is not ready to perform the operation. Below are the steps exaplaining the design.

1. When a thread perfroms a disk operation, issue the operation.
2. Check whether the disk is ready to transfer the data.
3. If the disk is ready continue the operation.
4. If the disk is not ready, add the thread to the blocking queue and yield.
5. Expose a method that can be utilised in the scheduler to chech whether the blocked thread performaing the disk operating is ready to execute.
6. Scheduler before performaing any operation checks whether the thread is ready. If the thread is ready, dispatch it. Else dispatch other thread in the scheduler pool.

Design of Bonus Questions:

Design of a thread-safe disk system:

The assumption is only one thread can use the disk for any operation. In such a case, the requirement is to make sure a thread which is accessing the disk for an operation should be given priority until it completes the given operation. This can be implemented using a lock.

Below are the steps to implement a thread safe disk system.

- Before performaing any operation, check whether the disk is occupied by any other threads. In other words, check whether the lock is free or not.
- If the lock is free, acquire the lock and continue the operation.
- If the lock is not free, add the thread to the end of the blocking queue and wait for the turn.
- While continuing the operation, if the disk is not ready to transfer the data, yeild the thread. In this case, add the thread to the beginning of the queue, such that when the disk is ready to continue, the thread at the beginning of the queue can continue the operation.

The above steps ensures no two threads acquire the lock on the disk at the same time. This enables thread safe disk operations. If multiple threads request for a disk operation, the first thread wins the scheduling process and rest of them wait until first thread finishes the operation.

Implementation Details

Mac Support

I have modified the below files to support the functioning of MP3 problem on Mac M2.

- copykernel.sh
- bochsrc.bxrc
- kernel.C
- scheduler.C
- blockingDisk.H
- blockingDisk.C
- thread.C

blockingDisk.H/C

The below explanation includes both the Machine problem and bonus questions 3 and 4.

1. Below are the variables used to implement the blocking disk and their explanation.
2. Below are the private methods declared within the blockingDisk which are explained further.
3. Below are the public methods declared within the blockingDisk which are explained further.
4. Below is the constructor used to initialize the class variables.
5. bool BlockingDisk::is_ready(): This function is used by the scheduler to check whether the thread in the blocking disk queue is ready to execute. If there is a thread in the queue and the lock is free, the function returns true. If the disk is ready to transfer data for the waiting thread, it returns true. Additional dummy code is added to induce delay.
6. void BlockingDisk::add_disk_thread_to_front(Thread *_thread): Add the thread to the front of the queue.

```
private:
   /* A variable will hold the starting node of the linked list.*/
   DiskThreadList *head;

   /* The variable will hold the last node of the linked list. This is will
      enable to append new threads on to the ready queue in O(n) */
   DiskThreadList *tail;

   /* Variable to hold the number of threads requested for the disk operation.*/
   unsigned int no_of_disk_threads;

   /* Variable to get the lock for disk operation.*/
   unsigned int current_thread;

   /* Variable to induce delay in disk operation.*/
   unsigned int delay;
```

Figure 1: Blocking Disk Variables

```
/* Function to add  a disk to a list of blocked threads.*/
void add_disk_thread(Thread *thread);

/* Function to add a disk to a list of blocked threads to front.*/
void add_disk_thread_to_front(Thread *thread);

/* Function to get the lock current_thread. The method will set the
   current_thread variable to thread Id of current thread.*/
void set_current_thread();

/* Release the lock for other threads to perform the disk operation.*/
void clear_current_thread();

/* Before the disk operation, check whether other threads are accessing the disk.
   If yes, wait in the queue for the turn.*/
void check_other_disk_operation();

/* Function to yeild the thread, if disk is not ready to continue the operation.*/
virtual void wait_until_ready();
```

Figure 2: Blocking Disk Private Methods

```
public:
   BlockingDisk(DISK_ID _disk_id, unsigned int _size);
   /* Creates a BlockingDisk device with the given size connected to the
      MASTER or SLAVE 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. */

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

   /* Get the latest thread in the queue. Used by the scheduler*/
   Thread *get_disk_thread();
```

Figure 3: Blocking Disk Public Methods

7. void BlockingDisk::add_disk_thread(Thread *_thread): Similarly add the thread to the end of the queue.

8. Thread *BlockingDisk::get_disk_thread(): Function is used by scheduler. It will remove thread at the front of the queue and return it.

9. void BlockingDisk::check_other_disk_operation(): Function checks if the lock is required by other threads and waits at the end of the queue. If not it continues.

```
/* Constructor to initialiaze the variables and other data structures*/
BlockingDisk::BlockingDisk(DISK_ID _disk_id, unsigned int _size)
    : SimpleDisk(_disk_id, _size)
{
  /* Initially the blocking queue is empty.*/
  head = nullptr;
  tail = nullptr;

  /* The queue is empty.*/
  no_of_disk_threads = 0;

  /* set the lock to -1 to indicate the thread is important..*/
  current_thread = -1;

  /* Variable to control the delay*/
  delay = 0;
}
```

Figure 4: Blocking Disk Constructor

```
/*  This function is used by the scheduler to check whether there are any disc threads
    ready to execute. */
bool BlockingDisk::is_ready()
{
  /*  If there are threads in the queue and no threads holds the lock.
      Return true to execute the threads in the waiting queue.*/
  if (no_of_disk_threads > 0 && current_thread == -1)
  {
    /* Code to induce delay.*/
    if (delay == 0)
    {
      delay = 1;
      return false;
    }
    delay = 0;
    Console::puts("No Current operation. Next Operation is ready.\n");
    return true;
  }
  /* If disc is ready for disk to transfer data, execute the blocked thread.*/
  else if (no_of_disk_threads > 0 && SimpleDisk::is_ready())
  {
    /* Code to induce delay.*/
    if (delay == 0)
    {
      delay = 1;
      return false;
    }
    delay = 0;
    Console::puts("Disk operation is ready to continue.\n");
    return true;
  }

  /* In all other cases return false.*/
  return false;
}
```

Figure 5: Blocking Disk IS_READY

10. void BlockingDisk::wait_until_ready(): Function checks if the disk is ready to perform the operation. If not, it will add the thread to the front of the waiting queue.

11. void BlockingDisk::set_current_thread(): It will update the value of lock to the value of current thread ID.

12. void BlockingDisk::clear_current_thread(): This function will release the lock acquired by the operating thread.

13. void BlockingDisk::read(unsigned long _block_no, unsigned char *_buf): The read operation is performed as explained in the image.

14. void BlockingDisk::write(unsigned long _block_no, unsigned char *_buf): Similarly, write operation is performed.

```
void BlockingDisk::add_disk_thread_to_front(Thread *_thread)
{
  DiskThreadList *_node = new DiskThreadList;
  _node->thread = _thread;
  this->no_of_disk_threads++;

  if (head == nullptr)
  {
    _node->next = nullptr;
    head = tail = _node;
    return;
  }

  _node->next = head;
  head = _node;
}
```

Figure 6: Blocking Disk Add thread to front

```
/* Get the latest thread in the queue.*/
Thread *BlockingDisk::get_disk_thread()
{
  DiskThreadList *node = head;
  head = head->next;

  Thread *thread = node->thread;
  delete node;

  this->no_of_disk_threads--;
  return thread;
}
```

Figure 7: Blocking Disk Get Disk Thread

```
/*  Verifies if other threads are operating on the disk.
    If yes, waits by yeilding the thread. */
void BlockingDisk::check_other_disk_operation()
{
  if (current_thread != -1)
  {
    Console::puts("Yeilding the thread as other disk operation in progress.\n");
    add_disk_thread(Thread::CurrentThread());
    SYSTEM_SCHEDULER->yield();
  }
  Console::puts("No other disk operation.\n");
}
```

Figure 8: Blocking Disk Check Other Disk Operation

scheduler.C

There is one change to the yeild function of the scheduler. Before finding an thread in the scheduler queue, the scheduler checkes whether any disk thread is ready to execute.

```
/*  Yeilds a thread, if the disk is not ready to transfer the data.*/
void BlockingDisk::wait_until_ready()
{
  if (!SimpleDisk::is_ready())
  {
    Console::puts("Yeilding the thread as disk is not ready.\n");
    add_disk_thread_to_front(Thread::CurrentThread());
    SYSTEM_SCHEDULER->yield();
  }
  Console::puts("Continuing operation.\n");
}
```

Figure 9: Blocking Disk Wait until ready

```
/* Sets the lock varible to current thread ID*/
void BlockingDisk::set_current_thread()
{
  // if (Machine::interrupts_enabled())
  //   Machine::disable_interrupts();

  current_thread = Thread::CurrentThread()->ThreadId();

  // if (!Machine::interrupts_enabled())
  //   Machine::enable_interrupts();
}

/* Releases the lock.*/
void BlockingDisk::clear_current_thread()
{
  // if (Machine::interrupts_enabled())
  //   Machine::disable_interrupts();

  current_thread = -1;

  // if (!Machine::interrupts_enabled())
  //   Machine::enable_interrupts();
}
```

Figure 10: Blocking Disk Set and Clear Current Thread

thread.C

There are no additional changes from the last MP.

kernel.C

Added Macros which controls the functionality.

- _USES_SCHEDULER_: Macro to use scheduler.
- _BLOCKING_DISK_: Macro to include blocking disk.
- _THREAD_SAFE_DISK : Macro to test thread safe fucntionalit

```
/*  The execution is as follows:
    1. Check if other threads are performing an operation.
    2. Get the lock by updating the current thread value.
    3. Issue operation.
    4. Waiting until the disk is ready.
    5. Perform the operation.
    6. Release the lock, by setting the variable to -1.*/
void BlockingDisk::read(unsigned long _block_no, unsigned char *_buf)
{
  // SimpleDisk::read(_block_no, _buf);

  check_other_disk_operation();

  set_current_thread();

  issue_operation(DISK_OPERATION::READ, _block_no);

  wait_until_ready();

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

  clear_current_thread();

  Console::puts("read::Read operation is complete.\n");
}
```

Figure 11: Blocking Disk Read

```
/* Functionality similar to read.*/
void BlockingDisk::write(unsigned long _block_no, unsigned char *_buf)
{
  check_other_disk_operation();

  set_current_thread();

  issue_operation(DISK_OPERATION::WRITE, _block_no);

  wait_until_ready();

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

  clear_current_thread();
  Console::puts("write::Write operation is complete.\n");
}
```

Figure 12: Blocking Disk write

Testing

You can find logs in the output.txt.

```
/* Yeild CPU to the next thread.*/
void Scheduler::yield()
{
    /* Disable Interrupts until you dispatch a new thread.*/
    // if (Machine::interrupts_enabled())
    // {
    //     Machine::disable_interrupts();
    // }

    /* Check if there are any blocked threads waiting for the disk operation.*/
    if (SYSTEM_DISK->is_ready())
    {
        /* Get  the latest threadn and yield.*/
        Thread *next_thread = SYSTEM_DISK->get_disk_thread();
        Console::puts("Yeilding Thread To Blocking Disk Thread [");
        Console::puti(next_thread->ThreadId());
        Console::puts("]\n");

        // /* Enable the interrupts before dispatching the thread.*/
        // if (!Machine::interrupts_enabled())
        // {
        //     Machine::enable_interrupts();
        // }

        Thread::dispatch_to(next_thread);
        return;
    }
```

Figure 13: Scheduler yeild

- Single thread (Thread 2) issues disk I/O operations; while other threads run regular non-terminating functions. Thread 2 yielded CPU to another thread on initiating Disk I/O operation. Busy waiting was avoided. Disk I/O operation was completed.
- Two threads (Thread 2, 3) issues disk I/O: Thread 3 waits until thread 2 finishes the operation.

All tests are passing.

Figure 14: Test 1

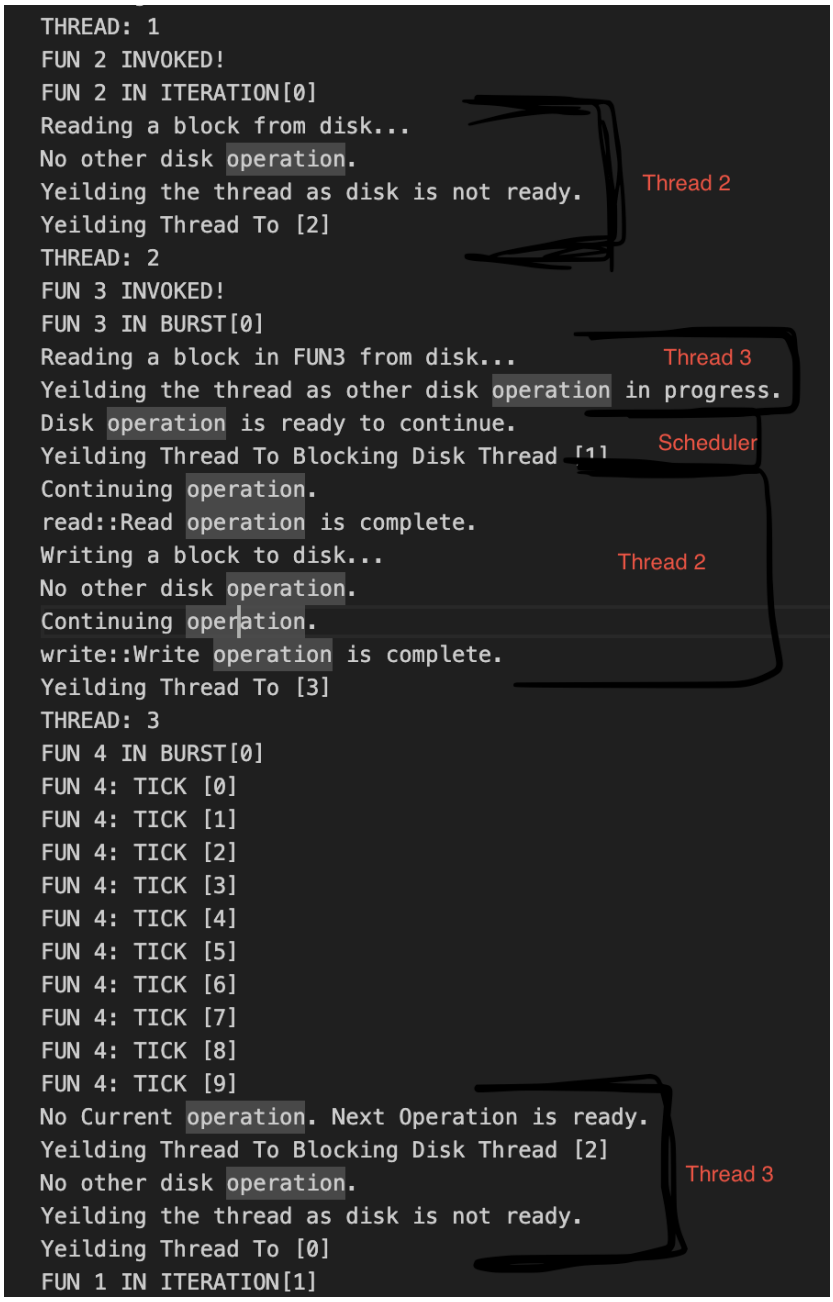

Figure 15: Test 2