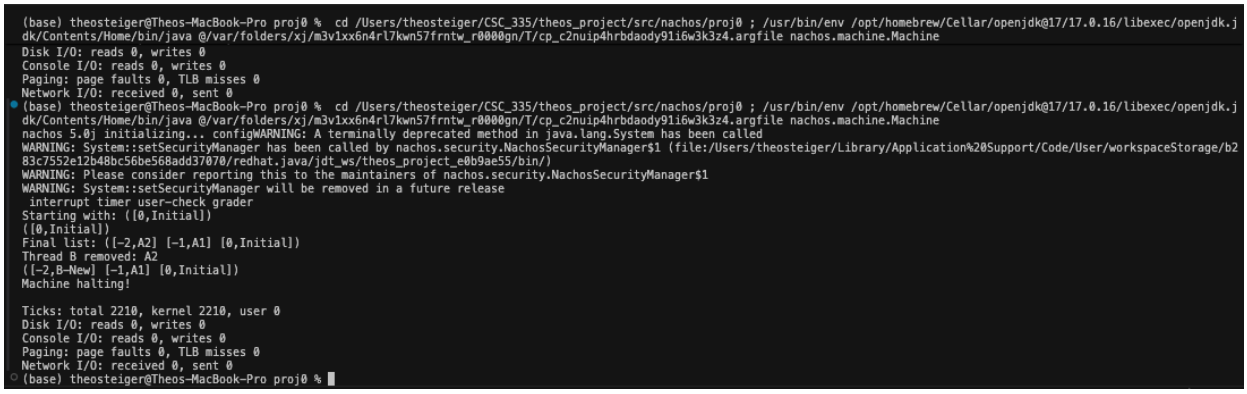
Rerun the two failing interleavings from Project 1 to make sure you have fixed all of your concurrency problems. Document your fixes in your write-up by summarizing each original problem and then showing how your code now prevents the errors (pasting in the corrected output for this part is fine).

1) a) Fatal Error: The original Problem here was that when we yielded before updating first after prepending our new element, then we would get a NullPointerException when we tried to removeHead because first was modified by another thread and thus not the correct current value of first (because of the time of the context switch).

To fix this, we implemented locks and condition variables in order to make it so that prepend and removeHead are mutually exclusive (we can't context switch in the middle of the method because it is still holding the lock).

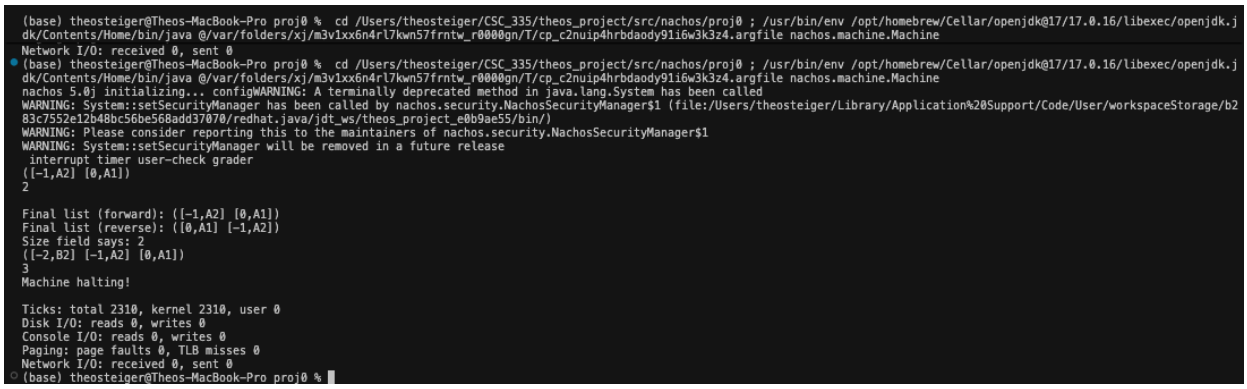
This mutual exclusivity fixes the original problem that we had, leading to no errors and a correct final state of our list as seen in the screenshot below (Note that the final state of the list is shown in the line below "Thread B removed: A2")



b) Corruption Error: In the case of the corruption test, our fix is basically the same as in the fatal error test. In this case, we originally got a corrupted list because during the calculation of the value of the key in thread A, we context switch to thread B, which does the same calculation for the value of the key. Both Thread A and Thread B read first == null and calculated newKey = 0 because they both thought the list was empty at the time of reading.

To fix this, we use locks and condition variables again to make it so that only one method can hold the lock at a time. Because of this, we got one of the expected results that I originally described in project 1. This expected result is the result we get if we imagine that we let thread A run completely, and then let thread B run completely. We get this result because in essence that's what's happening, since thread A continues to hold the lock until it finishes running after which thread B acquires the lock.

Here's a look at the new and correct output that we get after running the corruption test (Note that the final state of the DLL is seen in the 2 lines above "Machine halting!")



Document these new tests (and what the errors would have been) in your write-up. Be sure to test your wait and signal calls by forcing what would normally be underflow and overflow without the protection!

2) **BB_underflowTest()**

- Creates a small buffer (size 3) that starts empty
- 2 Reader threads that each try to read 2 characters (4 reads total)
- 1 Writer thread that writes 4 characters, but only after readers have started

Underflow Condition:

- Readers attempt to read from an empty buffer immediately (before anything has been written)
- Without protection, this would cause underflow (reading non-existent data)
- The Writer intentionally delays with yield() calls to ensure readers hit the empty buffer first

Errors without Monitor Protection:

- Readers would return uninitialized array values or stale data
- Reading at position first when size=0 could access invalid indices and cause an error
- Multiple readers could read the same position before first is updated
- The size, first, and last variables could become inconsistent throughout different threads

How Monitor Protection Prevents Errors:

- Readers are blocked using bufferEmpty.sleep() when buffer is empty
- When a we write() bufferEmpty.wake() gets called after the adding data
- Blocked Readers wake up and successfully read valid data that has been written by the writer
- All shared variables are protected by locks

BB_overflowTest()

- Creates a very small buffer (size 2) to quickly hit overflow

- 2 Writer threads that each try to write 3 characters (6 writes total)
- 1 SlowReader that reads slowly with yield() delays

Overflow Condition:
- Writers try to write 6 characters into a size-2 buffer and without protection, this would cause overflow (overwriting unread data)
- The SlowReader intentionally delays to ensure buffer fills up

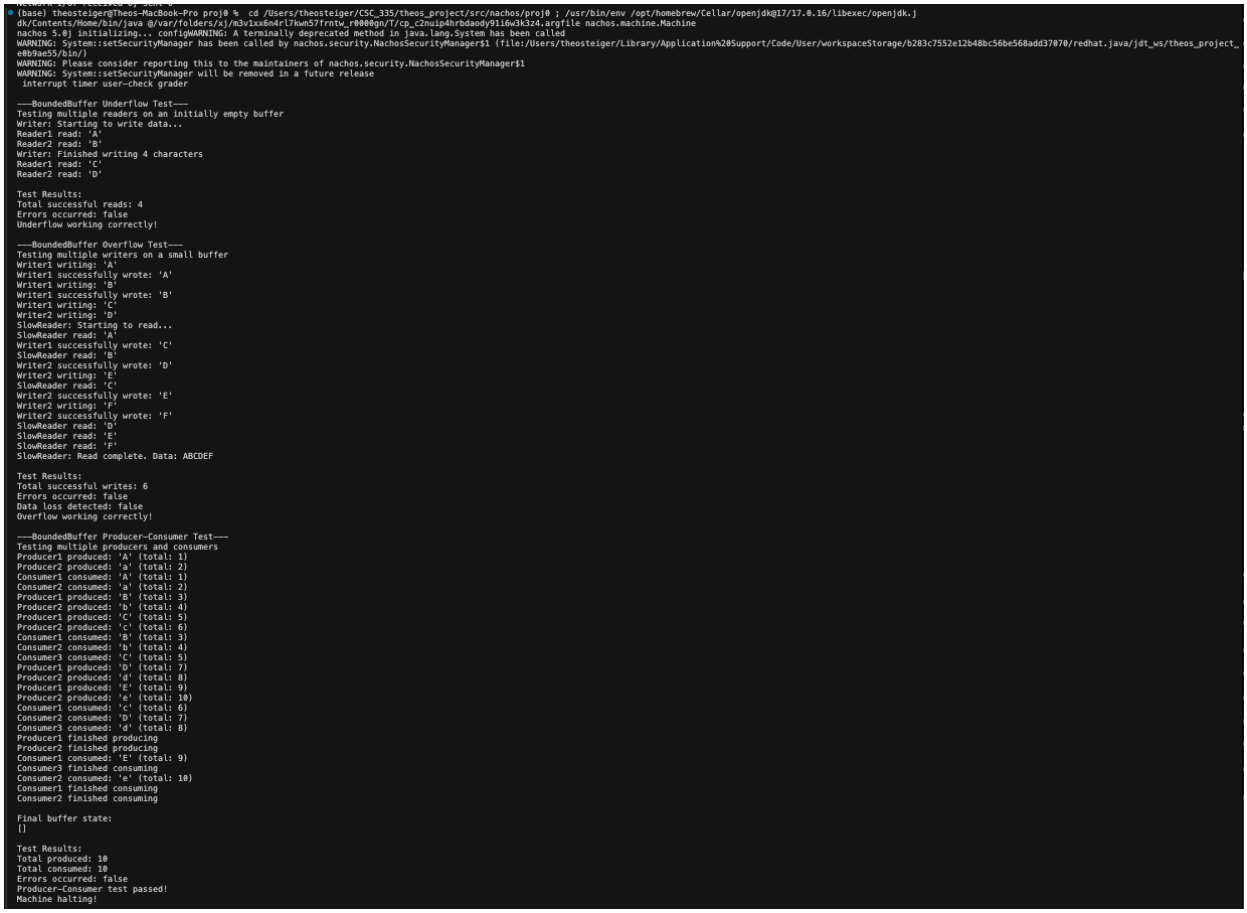
Issues without Monitor Protection:
- Writers would overwrite unread data in the circular buffer that previous writers have written
- The circular wraparound (last = (last + 1) % maxSize) would overwrite data at first
- size could exceed maxSize or become negative
- Multiple writers could write to the same position

How Monitor Protection Prevents Errors:
- Writers block on bufferFull.sleep() when buffer is full
- Reader's read() calls bufferFull.wake() after reading data (and making room on buffer)
- Writers wake up and successfully write without data loss because there is now space on the buffer

BB_producerConsumerTest()
- Buffer size 5
- 2 Producer threads writing 5 characters each (10 total)
- 3 Consumer threads reading 4, 4, and 2 characters (10 total)
- Consumers read slower than producers write (simulating more inconsistent real world use cases)

Testing Complete Synchronization:
- Tests both underflow and overflow
- Multiple threads compete for the same lock
- Tests the efficacy of wake() calls

without Monitor Protection:
- Multiple producers/consumers modifying first, last, size simultaneously
- Threads could get stuck waiting with no way to signal each other
- Total produced ≠ total consumed due to some updates being lost
- Forward and backward progress could conflict (consumption and production)



3)

As you can see, all of the tests pass with the same final output as when they used the condition.

```
--- DLList Tests (using Condition2) ---
Starting with: ([0,Initial])
([0,Initial])
Final list: ([-2,A2] [-1,A1] [0,Initial])
([-1,A2] [0,A1])
2

Final list (forward): ([-1,A2] [0,A1])
Final list (reverse): ([0,A1] [-1,A2])
Size field says: 2

--- BoundedBuffer Tests (using Condition2) ---

---BoundedBuffer Underflow Test---
Testing multiple readers on an initially empty buffer
Thread B removed: A2
([-1,B-New] [0,A1])
([-2,B2] [-1,B-New] [0,A1])
3
Writer: Starting to write data...
Reader1 read: 'A'
Reader2 read: 'B'
Writer: Finished writing 4 characters
Reader1 read: 'C'
Reader2 read: 'D'

Test Results:
Total successful reads: 4
Errors occurred: false
Underflow working correctly!

---BoundedBuffer Overflow Test---
Testing multiple writers on a small buffer
Writer1 writing: 'A'
Writer1 successfully wrote: 'A'
Writer1 writing: 'B'
Writer1 successfully wrote: 'B'
Writer1 writing: 'C'
Writer2 writing: 'D'
SlowReader: Starting to read...
SlowReader read: 'A'
Writer1 successfully wrote: 'C'
SlowReader read: 'B'
Writer2 successfully wrote: 'D'
Writer2 writing: 'E'
Writer2 successfully wrote: 'E'
Writer2 writing: 'F'
SlowReader read: 'C'
SlowReader read: 'D'
Writer2 successfully wrote: 'F'
SlowReader read: 'E'
SlowReader read: 'F'
SlowReader: Read complete. Data: ABCDEF

Test Results:
Total successful writes: 6
Errors occurred: false
Data loss detected: false
Overflow working correctly!
```

```
---BoundedBuffer Producer-Consumer Test---
Testing multiple producers and consumers
Producer1 produced: 'A' (total: 1)
Producer2 produced: 'a' (total: 2)
Consumer1 consumed: 'A' (total: 1)
Consumer2 consumed: 'a' (total: 2)
Producer1 produced: 'B' (total: 3)
Producer2 produced: 'b' (total: 4)
Consumer1 consumed: 'B' (total: 3)
Consumer2 consumed: 'b' (total: 4)
Producer1 produced: 'C' (total: 5)
Producer2 produced: 'c' (total: 6)
Consumer1 consumed: 'C' (total: 5)
Consumer2 consumed: 'c' (total: 6)
Producer1 produced: 'D' (total: 7)
Producer2 produced: 'd' (total: 8)
Consumer1 consumed: 'D' (total: 7)
Consumer2 consumed: 'd' (total: 8)
Consumer1 finished consuming
Producer1 produced: 'E' (total: 9)
Consumer2 finished consuming
Producer1 finished producing
Producer2 produced: 'e' (total: 10)
Consumer3 consumed: 'E' (total: 9)
Producer2 finished producing
Consumer3 consumed: 'e' (total: 10)
Consumer3 finished consuming

Final buffer state:
[]

Test Results:
Total produced: 10
Total consumed: 10
Errors occurred: false
Producer-Consumer test passed!

--- Direct Condition2 Implementation Test ---

---Condition2 Implementation Test---
Testing new condition variables

-- Testing wake() (single wake) --
Waiter1 waiting on condition...
Waiter2 waiting on condition...
Waiter3 waiting on condition...
Signaler: Setting value to 1 and waking one thread
Waiter1 woke up! Shared value: 1
Signaler: 1 thread(s) woke up after wake()

-- Testing wakeAll() (broadcast) --
WaiterA waiting for value 2
WaiterB waiting for value 3
WaiterC waiting for value 5
Broadcaster: Setting value to 5 and waking all threads
Waiter2 woke up! Shared value: 5
Waiter3 woke up! Shared value: 5
WaiterA proceeding with value: 5
WaiterB proceeding with value: 5
WaiterC proceeding with value: 5

Condition2 test completed
If wake() woke exactly one thread and wakeAll() woke all threads,
then the interrupt-based implementation is working correctly!
Machine halting!

Ticks: total 6140, kernel 6140, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: page faults 0, TLB misses 0
Network I/O: received 0, sent 0
(base) theosteiger@Theos-MacBook-Pro proj0 %
```

Modified Files:

Java

```java
package nachos.threads;

public class BoundedBuffer {

   private char[] buffer;
   private int maxSize;
   private int size;
   private int first;
   private int last;
   private Lock lock;
   private Condition bufferEmpty;
   private Condition bufferFull;

   public BoundedBuffer(int maxsize) {
       this.size = 0;
       this.first = 0;
       this.last = 0;
       this.maxSize = maxsize;
       this.buffer = new char[maxSize];
       lock = new Lock();
       bufferEmpty = new Condition(lock);
       bufferFull = new Condition(lock);
   }
   // Read a character from the buffer, blocking until there is a char
   // in the buffer to satisfy the request. Return the char read.
   public char read() {
       lock.acquire();
       // wait while buffer empty
       while (this.size == 0) {
           bufferEmpty.sleep();
       }
```

```
        // remove from head
        char c = this.buffer[first];
        first = (first + 1) % maxSize;
        this.size--;

        // wake a writer waiting for space
        bufferFull.wake();
        lock.release();

        return c;
    }
    // Write the given character c into the buffer, blocking until
    // enough space is available to satisfy the request.
    public void write(char c) {
        lock.acquire();
        // wait while buffer full
        while (this.size == this.maxSize) {
            bufferFull.sleep();
        }

        // insert at tail
        buffer[last] = c;
        last = (last + 1) % maxSize;
        this.size++;

        // wake a reader waiting for data
        bufferEmpty.wake();
        lock.release();
    }
    // Prints the contents of the buffer; for debugging only
    public void print() {
        lock.acquire();
        StringBuilder sb = new StringBuilder();
```

```
        sb.append("[");
        for (int i = 0; i < this.size; i++) {
            if (i > 0) sb.append(", ");
            int idx = (first + i) % maxSize;
            sb.append(buffer[idx]);
        }
        sb.append("]");
        System.out.println(sb.toString());
        lock.release();
    }
}
```

Java

```
package nachos.threads;  // don't change this. Gradescope needs it.

public class DLList
{
    private DLLElement first;  // pointer to first node
    private DLLElement last;   // pointer to last node
    private int size;          // number of nodes in list
    private Lock lock;
    private Condition listEmpty;

    /**
     * Creates an empty sorted doubly-linked list.
     */
    public DLList() {
        lock = new Lock();
        listEmpty = new Condition(lock);
```

```
        this.size = 0;
        this.first = null;  // pointer to first node
        this.last = null;
    }

    /**
     * Add item to the head of the list, setting the key for the new
     * head element to min_key - 1, where min_key is the smallest key
     * in the list (which should be located in the first node).
     * If no nodes exist yet, the key will be 0.
     */
    public void prepend(Object item) {
        lock.acquire();
        int newKey = 0;
        if (this.first != null) {
            newKey = first.key - 1;
        }

        DLLElement newElement = new DLLElement(item, newKey);

        if (this.first == null) {  // Empty list
            this.first = newElement;
            this.last = newElement;
            listEmpty.wake();
        } else {  // Non-empty list
            newElement.next = this.first;
            this.first.prev = newElement;
            this.first = newElement;
        }
        this.size++;
        lock.release();
    }

    /**
```

```
 * Removes the head of the list and returns the data item stored in
 * it.  Returns null if no nodes exist.
 *
 * @return the data stored at the head of the list or null if list empty
 */
public Object removeHead() {
    lock.acquire();
    while (this.size == 0) {
        listEmpty.sleep();
    } // else {
    Object dataItem = first.data;
    this.first = first.next;

    if (first != null) {
        first.prev = null;
    } else {
        this.last = null;
    }
    this.size--;
    lock.release();

    return dataItem;
}

/**
 * Tests whether the list is empty.
 *
 * @return true iff the list is empty.
 */
public boolean isEmpty() {
    lock.acquire();
    boolean isEmpty = this.size == 0;
    lock.release();
    return isEmpty;
```

```
}

/**
 * returns number of items in list
 * @return
 */
public int size(){
    lock.acquire();
    int howBig = this.size;
    lock.release();
    return howBig;
}



/**
 * Inserts item into the list in sorted order according to sortKey.
 */
public void insert(Object item, Integer sortKey) {
    lock.acquire();
    DLLElement elementToInsert = new DLLElement(item, sortKey);

    if (first == null) { // empty list
        this.first = elementToInsert;
        this.last = elementToInsert;
        this.size++;
        listEmpty.wake();
    } else if (sortKey <= first.key) { // insert at head
        elementToInsert.next = this.first;
        this.first.prev = elementToInsert;
        this.first = elementToInsert;
        this.size++;
    } else if (sortKey >= last.key) { // insert at tail
        elementToInsert.prev = this.last;
        this.last.next = elementToInsert;
```

```
            this.last = elementToInsert;
            this.size++;
        } else { // insert somewhere in the middle
            DLLElement current = this.first;
            while (current.next != null && current.next.key < sortKey) {
                current = current.next;
            }

            elementToInsert.next = current.next;
            elementToInsert.prev = current;
            if (current.next != null) {
                current.next.prev = elementToInsert;
            }
            current.next = elementToInsert;
            this.size++;
        }
        lock.release();
    }


    /**
     * returns list as a printable string. A single space should separate each
list item,
     * and the entire list should be enclosed in parentheses. Empty list should
return "()"
     * @return list elements in order
     */
    private String toStringUnsafe() {
        if (this.size == 0) {
            return "()";
        } else {
            String toRet = "(";
            DLLElement current = this.first;
            boolean isFirst = true;
```

```
        while (current != null) {
            if (!isFirst) {
                toRet = toRet + " ";
            }
            toRet = toRet + current.toString();
            isFirst = false;
            current = current.next;
        }
        toRet = toRet + ")";
        return toRet;
    }
}

public String toString() {
    lock.acquire();
    String toRet = this.toStringUnsafe();
    lock.release();
    return toRet;
}

/**
 * returns list as a printable string, from the last node to the first.
 * String should be formatted just like in toString.
 * @return list elements in backwards order
 */
private String reverseToStringUnsafe(){
    if (this.size == 0) {
        return "()";
    } else {
        String toRet = "(";
        DLLElement current = this.last;
        boolean isFirst = true;
        while (current != null) {
            if (!isFirst) {
```

```
                toRet = toRet + " ";
            }
            toRet = toRet + current.toString();
            isFirst = false;
            current = current.prev;
        }
        toRet = toRet + ")";
        return toRet;
    }
}

public String reverseToString() {
    lock.acquire();
    String toRet = this.reverseToStringUnsafe();
    lock.release();
    return toRet;
}

/**
 *  inner class for the node
 */
private class DLLElement
{
    private DLLElement next;
    private DLLElement prev;
    private int key;
    private Object data;

    /**
     * Node constructor
     * @param item data item to store
     * @param sortKey unique integer ID
     */
    public DLLElement(Object item, int sortKey)
```

```
        {
            key = sortKey;
            data = item;
            next = null;
            prev = null;
        }

        /**
         * returns node contents as a printable string
         * @return string of form [<key>,<data>] such as [3,"ham"]
         */
        public String toString(){
            return "[" + key + "," + data + "]";
        }
    }
}
```

Java

```
package nachos.threads;

import nachos.machine.*;

/**
* A multi-threaded OS kernel.
*/
public class ThreadedKernel extends Kernel {
    /**
     * Allocate a new multi-threaded kernel.
     */
    public ThreadedKernel() {
    super();
```

```
}

/**
 * Initialize this kernel. Creates a scheduler, the first thread, and an
 * alarm, and enables interrupts. Creates a file system if necessary.
 */
public void initialize(String[] args) {
// set scheduler
String schedulerName = Config.getString("ThreadedKernel.scheduler");
scheduler = (Scheduler) Lib.constructObject(schedulerName);

// set fileSystem
String fileSystemName = Config.getString("ThreadedKernel.fileSystem");
if (fileSystemName != null)
    fileSystem = (FileSystem) Lib.constructObject(fileSystemName);
else if (Machine.stubFileSystem() != null)
    fileSystem = Machine.stubFileSystem();
else
    fileSystem = null;

// start threading
new KThread(null);

alarm  = new Alarm();

Machine.interrupt().enable();
}

/**
 * Test this kernel. Test the <tt>KThread</tt>, <tt>Semaphore</tt>,
 * <tt>SynchList</tt>, and <tt>ElevatorBank</tt> classes. Note that the
 * autograder never calls this method, so it is safe to put additional
 * tests here.
 */
```

```
public void selfTest() {
// KThread.selfTest();
// KThread.DLL_selfTest();  // Run our DLL test instead

// KThread.DLL_fatalErrorTest();     // Will cause NullPointerException
// KThread.DLL_corruptionTest();   // Will corrupt the list structure

// BoundedBuffer tests
KThread.BB_underflowTest();     // Test underflow protection
KThread.BB_overflowTest();       // Test overflow protection
KThread.BB_producerConsumerTest(); // Test producer-consumer scenario

Semaphore.selfTest();
SynchList.selfTest();
if (Machine.bank() != null) {
    ElevatorBank.selfTest();
}
}

/**
 * A threaded kernel does not run user programs, so this method does
 * nothing.
 */
public void run() {
}

/**
 * Terminate this kernel. Never returns.
 */
public void terminate() {
Machine.halt();
}

/** Globally accessible reference to the scheduler. */
```

```
    public static Scheduler scheduler = null;
    /** Globally accessible reference to the alarm. */
    public static Alarm alarm = null;
    /** Globally accessible reference to the file system. */
    public static FileSystem fileSystem = null;

    // dummy variables to make javac smarter
    private static RoundRobinScheduler dummy1 = null;
    private static PriorityScheduler dummy2 = null;
    private static LotteryScheduler dummy3 = null;
    private static Condition2 dummy4 = null;
    private static Communicator dummy5 = null;
    private static Rider dummy6 = null;
    private static ElevatorController dummy7 = null;
}
```

Java

```
package nachos.threads;

import nachos.machine.*;

/**
* A KThread is a thread that can be used to execute Nachos kernel code. Nachos
* allows multiple threads to run concurrently.
*
* To create a new thread of execution, first declare a class that implements
* the <tt>Runnable</tt> interface. That class then implements the <tt>run</tt>
* method. An instance of the class can then be allocated, passed as an
* argument when creating <tt>KThread</tt>, and forked. For example, a thread
* that computes pi could be written as follows:
*
```

```
* <p><blockquote><pre>
* class PiRun implements Runnable {
*     public void run() {
*         // compute pi
*         ...
*     }
* }
* </pre></blockquote>
* <p>The following code would then create a thread and start it running:
*
* <p><blockquote><pre>
* PiRun p = new PiRun();
* new KThread(p).fork();
* </pre></blockquote>
*/
public class KThread {
    /**
     * Get the current thread.
     *
     * @return  the current thread.
     */
    public static KThread currentThread() {
    Lib.assertTrue(currentThread != null);
    return currentThread;
    }

    /**
     * Allocate a new <tt>KThread</tt>. If this is the first <tt>KThread</tt>,
     * create an idle thread as well.
     */
    public KThread() {
    if (currentThread != null) {
        tcb = new TCB();
    }
```

```
else {
    readyQueue = ThreadedKernel.scheduler.newThreadQueue(false);
    readyQueue.acquire(this);

    currentThread = this;
    tcb = TCB.currentTCB();
    name = "main";
    restoreState();

    createIdleThread();
}
}

/**
 * Allocate a new KThread.
 *
 * @param   target  the object whose <tt>run</tt> method is called.
 */
public KThread(Runnable target) {
this();
this.target = target;
}

/**
 * Set the target of this thread.
 *
 * @param   target  the object whose <tt>run</tt> method is called.
 * @return  this thread.
 */
public KThread setTarget(Runnable target) {
Lib.assertTrue(status == statusNew);

this.target = target;
return this;
```

```
}

/**
 * Set the name of this thread. This name is used for debugging purposes
 * only.
 *
 * @param   name    the name to give to this thread.
 * @return  this thread.
 */
public KThread setName(String name) {
this.name = name;
return this;
}

/**
 * Get the name of this thread. This name is used for debugging purposes
 * only.
 *
 * @return  the name given to this thread.
 */
public String getName() {
return name;
}

/**
 * Get the full name of this thread. This includes its name along with its
 * numerical ID. This name is used for debugging purposes only.
 *
 * @return  the full name given to this thread.
 */
public String toString() {
return (name + " (#" + id + ")");
}
```

```
/**
 * Deterministically and consistently compare this thread to another
 * thread.
 */
public int compareTo(Object o) {
KThread thread = (KThread) o;

if (id < thread.id)
    return -1;
else if (id > thread.id)
    return 1;
else
    return 0;
}

/**
 * Causes this thread to begin execution. The result is that two threads
 * are running concurrently: the current thread (which returns from the
 * call to the <tt>fork</tt> method) and the other thread (which executes
 * its target's <tt>run</tt> method).
 */
public void fork() {
Lib.assertTrue(status == statusNew);
Lib.assertTrue(target != null);

Lib.debug(dbgThread,
      "Forking thread: " + toString() + " Runnable: " + target);

boolean intStatus = Machine.interrupt().disable();

tcb.start(new Runnable() {
    public void run() {
        runThread();
    }
```

```
    });

ready();

Machine.interrupt().restore(intStatus);
}

private void runThread() {
begin();
target.run();
finish();
}

private void begin() {
Lib.debug(dbgThread, "Beginning thread: " + toString());

Lib.assertTrue(this == currentThread);

restoreState();

Machine.interrupt().enable();
}

/**
 * Finish the current thread and schedule it to be destroyed when it is
 * safe to do so. This method is automatically called when a thread's
 * <tt>run</tt> method returns, but it may also be called directly.
 *
 * The current thread cannot be immediately destroyed because its stack and
 * other execution state are still in use. Instead, this thread will be
 * destroyed automatically by the next thread to run, when it is safe to
 * delete this thread.
 */
public static void finish() {
```

```
Lib.debug(dbgThread, "Finishing thread: " + currentThread.toString());

Machine.interrupt().disable();

Machine.autoGrader().finishingCurrentThread();

Lib.assertTrue(toBeDestroyed == null);
toBeDestroyed = currentThread;


currentThread.status = statusFinished;

sleep();
}

/**
 * Relinquish the CPU if any other thread is ready to run. If so, put the
 * current thread on the ready queue, so that it will eventually be
 * rescheuled.
 *
 * <p>
 * Returns immediately if no other thread is ready to run. Otherwise
 * returns when the current thread is chosen to run again by
 * <tt>readyQueue.nextThread()</tt>.
 *
 * <p>
 * Interrupts are disabled, so that the current thread can atomically add
 * itself to the ready queue and switch to the next thread. On return,
 * restores interrupts to the previous state, in case <tt>yield()</tt> was
 * called with interrupts disabled.
 */
public static void yield() {
Lib.debug(dbgThread, "Yielding thread: " + currentThread.toString());
```

```
    Lib.assertTrue(currentThread.status == statusRunning);

    boolean intStatus = Machine.interrupt().disable();

    currentThread.ready();

    runNextThread();

    Machine.interrupt().restore(intStatus);
    }

    /**
     * Conditionally yield based on the oughtToYield array and execution count.
     * This method tracks how many times it has been executed across all threads
     * and yields if oughtToYield[numTimesBefore] is true.
     */
    public static void yieldIfOughtTo() {
        if (numTimesBefore < oughtToYield.length &&
oughtToYield[numTimesBefore]) {
            numTimesBefore++;
            KThread.yield();
        } else {
            numTimesBefore++;
        }
    }

    /**
     * Given this unique location, yield the
     * current thread if it ought to. It knows
     * to do this if yieldData[loc][i] is true, where
     * i is the number of times that this function
     * has already been called from this location.
     *
     * @param loc unique location. Every call to
```

```
     *            yieldIfShould that you
     *            place in your DLList code should
     *            have a different loc number.
     */
    public static void yieldIfShould(int loc) {
        if (loc < yieldData.length && yieldCount[loc] < yieldData[loc].length) {
            if (yieldData[loc][yieldCount[loc]]) {
                yieldCount[loc]++;
                KThread.yield();
            } else {
                yieldCount[loc]++;
            }
        }
    }

    /**
     * Relinquish the CPU, because the current thread has either finished or it
     * is blocked. This thread must be the current thread.
     *
     * <p>
     * If the current thread is blocked (on a synchronization primitive, i.e.
     * a <tt>Semaphore</tt>, <tt>Lock</tt>, or <tt>Condition</tt>), eventually
     * some thread will wake this thread up, putting it back on the ready queue
     * so that it can be rescheduled. Otherwise, <tt>finish()</tt> should have
     * scheduled this thread to be destroyed by the next thread to run.
     */
    public static void sleep() {
    Lib.debug(dbgThread, "Sleeping thread: " + currentThread.toString());

    Lib.assertTrue(Machine.interrupt().disabled());

    if (currentThread.status != statusFinished)
        currentThread.status = statusBlocked;
```

```
runNextThread();
}

/**
 * Moves this thread to the ready state and adds this to the scheduler's
 * ready queue.
 */
public void ready() {
Lib.debug(dbgThread, "Ready thread: " + toString());

Lib.assertTrue(Machine.interrupt().disabled());
Lib.assertTrue(status != statusReady);

status = statusReady;
if (this != idleThread)
    readyQueue.waitForAccess(this);

Machine.autoGrader().readyThread(this);
}

/**
 * Waits for this thread to finish. If this thread is already finished,
 * return immediately. This method must only be called once; the second
 * call is not guaranteed to return. This thread must not be the current
 * thread.
 */
public void join() {
Lib.debug(dbgThread, "Joining to thread: " + toString());

Lib.assertTrue(this != currentThread);

}

/**
```

```
     * Create the idle thread. Whenever there are no threads ready to be run,
     * and <tt>runNextThread()</tt> is called, it will run the idle thread. The
     * idle thread must never block, and it will only be allowed to run when
     * all other threads are blocked.
     *
     * <p>
     * Note that <tt>ready()</tt> never adds the idle thread to the ready set.
     */
    private static void createIdleThread() {
    Lib.assertTrue(idleThread == null);

    idleThread = new KThread(new Runnable() {
        public void run() { while (true) KThread.yield(); }
    });
    idleThread.setName("idle");

    Machine.autoGrader().setIdleThread(idleThread);

    idleThread.fork();
    }

    /**
     * Determine the next thread to run, then dispatch the CPU to the thread
     * using <tt>run()</tt>.
     */
    private static void runNextThread() {
    KThread nextThread = readyQueue.nextThread();
    if (nextThread == null)
        nextThread = idleThread;

    nextThread.run();
    }

    /**
```

```
     * Dispatch the CPU to this thread. Save the state of the current thread,
     * switch to the new thread by calling <tt>TCB.contextSwitch()</tt>, and
     * load the state of the new thread. The new thread becomes the current
     * thread.
     *
     * <p>
     * If the new thread and the old thread are the same, this method must
     * still call <tt>saveState()</tt>, <tt>contextSwitch()</tt>, and
     * <tt>restoreState()</tt>.
     *
     * <p>
     * The state of the previously running thread must already have been
     * changed from running to blocked or ready (depending on whether the
     * thread is sleeping or yielding).
     *
     * @param   finishing   <tt>true</tt> if the current thread is
     *              finished, and should be destroyed by the new
     *              thread.
     */
    private void run() {
    Lib.assertTrue(Machine.interrupt().disabled());

    Machine.yield();

    currentThread.saveState();

    Lib.debug(dbgThread, "Switching from: " + currentThread.toString()
          + " to: " + toString());

    currentThread = this;

    tcb.contextSwitch();

    currentThread.restoreState();
```

```
}

/**
 * Prepare this thread to be run. Set <tt>status</tt> to
 * <tt>statusRunning</tt> and check <tt>toBeDestroyed</tt>.
 */
protected void restoreState() {
Lib.debug(dbgThread, "Running thread: " + currentThread.toString());

Lib.assertTrue(Machine.interrupt().disabled());
Lib.assertTrue(this == currentThread);
Lib.assertTrue(tcb == TCB.currentTCB());

Machine.autoGrader().runningThread(this);

status = statusRunning;

if (toBeDestroyed != null) {
    toBeDestroyed.tcb.destroy();
    toBeDestroyed.tcb = null;
    toBeDestroyed = null;
}
}

/**
 * Prepare this thread to give up the processor. Kernel threads do not
 * need to do anything here.
 */
protected void saveState() {
Lib.assertTrue(Machine.interrupt().disabled());
Lib.assertTrue(this == currentThread);
}

private static class PingTest implements Runnable {
```

```
PingTest(int which) {
    this.which = which;
}

public void run() {
    for (int i=0; i<5; i++) {
    System.out.println("*** thread " + which + " looped "
                + i + " times");
    currentThread.yield();
    }
}

private int which;
}

private static class DLListTest implements Runnable {
private static DLList sharedList = new DLList();
private String label;
private int from;
private int to;
private int step;

DLListTest(String label, int from, int to, int step) {
    this.label = label;
    this.from = from;
    this.to = to;
    this.step = step;
}

/**
 * Prepends multiple nodes to a shared doubly-linked list. For each
 * integer in the range from...to (inclusive), make a string
 * concatenating label with the integer, and prepend a new node
 * containing that data (that's data, not key). For example,
```

```
    * countDown("A",8,6,1) means prepend three nodes with the data
    * "A8", "A7", and "A6" respectively. countDown("X",10,2,3) will
    * also prepend three nodes with "X10", "X7", and "X4".
    *
    * This method should conditionally yield after each node is inserted.
    * Print the list at the very end.
    *
    * Preconditions: from>=to and step>0
    *
    * @param label string that node data should start with
    * @param from integer to start at
    * @param to integer to end at
    * @param step subtract this from the current integer to get to the next
integer
    */
   public void countDown(String label, int from, int to, int step) {
       for (int i = from; i >= to; i -= step) {
       String data = label + i;
       sharedList.prepend(data);
       yieldIfOughtTo();
       }
       System.out.println(sharedList.toString());
   }

   public void run() {
       countDown(label, from, to, step);
   }
   }

   /**
    * Tests whether this module is working.
    */
   public static void selfTest() {
   Lib.debug(dbgThread, "Enter KThread.selfTest");
```

```
  new KThread(new PingTest(1)).setName("forked thread").fork();
  new PingTest(0).run();
  }

  /**
   * Demonstrates a FATAL concurrency error (NullPointerException).
   * Creates an interleaving where prepend overwrites the first pointer
   * while another thread is in the middle of modifying it.
   */
  public static void DLL_fatalErrorTest() {
  // Reset yield mechanism
  yieldData = new boolean[10][100];
  yieldCount = new int[10];

  // Test class that removes and adds
  class RemoveAndAdd implements Runnable {
      public void run() {
      // Remove head
      Object removed = DLListTest.sharedList.removeHead();
      System.out.println("Thread B removed: " + removed);
      // Add something back
      DLListTest.sharedList.prepend("B-New");
      }
  }

  // Test class that prepends multiple times
  class MultiplePrepend implements Runnable {
      public void run() {
      DLListTest.sharedList.prepend("A1");
      // This second prepend will access first.prev which might be in bad
state
      DLListTest.sharedList.prepend("A2");
      }
```

```
}

// Initialize with one element
DLListTest.sharedList = new DLList();
DLListTest.sharedList.prepend("Initial");

System.out.println("Starting with: " + DLListTest.sharedList.toString());

// Location 4: Thread A yields after setting next but before setting prev
yieldData[4][0] = true;  // First prepend of A yields here

// Location 8: Thread B yields after updating first in removeHead
yieldData[8][0] = true;  // RemoveHead yields after changing first

// Location 5: Thread A continues and will crash trying first.prev
// because first has been changed by Thread B

new KThread(new RemoveAndAdd()).setName("Thread-B").fork();

try {
    new MultiplePrepend().run();
    System.out.println("Final list: " + DLListTest.sharedList.toString());
} catch (NullPointerException e) {
    System.out.println("FATAL ERROR: NullPointerException occurred!");
    System.out.println("Stack trace:");
    e.printStackTrace();
    throw e;  // Re-throw to actually crash
}
}

/**
 * Demonstrates a NON-FATAL concurrency error that corrupts the list.
 * The interleaving causes the size counter to be inconsistent and
 * creates duplicate keys due to race conditions.
```

```
 */
public static void DLL_corruptionTest() {
// Reset yield mechanism
yieldData = new boolean[10][100];
yieldCount = new int[10];

// Test class that does both prepend and removeHead
class MixedOperations implements Runnable {
    private String label;
    private boolean doRemove;

    MixedOperations(String label, boolean doRemove) {
    this.label = label;
    this.doRemove = doRemove;
    }

    public void run() {
    DLListTest.sharedList.prepend(label + "1");
    if (doRemove && DLListTest.sharedList.size() > 0) {
        DLListTest.sharedList.removeHead();
    }
    DLListTest.sharedList.prepend(label + "2");
    }
}

// Initialize empty list
DLListTest.sharedList = new DLList();

// Location 0: After entering prepend - Thread A yields
yieldData[0][0] = true;

// Location 1: After reading key - Thread B yields
yieldData[1][1] = true;
```

```
    // Location 2: After creating element - Thread A yields again
    yieldData[2][2] = true;

    // Location 4: In non-empty branch - causes wrong pointer updates
    yieldData[4][3] = true;

    // Location 5: After setting prev pointer
    yieldData[5][4] = true;

    new KThread(new MixedOperations("B", true)).setName("Thread-B").fork();
    new MixedOperations("A", false).run();

    System.out.println("\nFinal list (forward): " +
DLListTest.sharedList.toString());
    System.out.println("Final list (reverse): " +
DLListTest.sharedList.reverseToString());
    System.out.println("Size field says: " + DLListTest.sharedList.size());

    // Check for duplicate keys
    String listStr = DLListTest.sharedList.toString();
    if (listStr.contains("[0,") && listStr.indexOf("[0,") !=
listStr.lastIndexOf("[0,")) {
        System.out.println("CORRUPTION: Duplicate key 0 detected!");
    }
    }

    /**
     * Tests the shared DLList by having two threads running countdown.
     * One thread will insert even-numbered data from "A12" to "A2".
     * The other thread will insert odd-numbered data from "B11" to "B1".
     * Don't forget to initialize the oughtToYield array before forking.
     *
     */
    public static void DLL_selfTest() {
```

```
    oughtToYield = new boolean[100];
    numTimesBefore = 0;
    DLListTest.sharedList = new DLList();  // Reset the shared list

    // // Thread B nodes at head, thread A at the tail
    // oughtToYield[0] = false;
    // oughtToYield[1] = false;
    // oughtToYield[2] = false;
    // oughtToYield[3] = false;
    // oughtToYield[4] = false;
    // oughtToYield[5] = true;
    // oughtToYield[6] = false;
    // oughtToYield[7] = false;
    // oughtToYield[8] = false;
    // oughtToYield[9] = false;
    // oughtToYield[10] = false;
    // oughtToYield[11] = false;



    // Alternate between threads A and B to get sorted order
    // oughtToYield[0] = true;   // A yields after A12
    // oughtToYield[1] = true;   // B yields after B11
    // oughtToYield[2] = true;   // A yields after A10
    // oughtToYield[3] = true;   // B yields after B9
    // oughtToYield[4] = true;   // A yields after A8
    // oughtToYield[5] = true;   // B yields after B7
    // oughtToYield[6] = true;   // A yields after A6
    // oughtToYield[7] = true;   // B yields after B5
    // oughtToYield[8] = true;   // A yields after A4
    // oughtToYield[9] = true;   // B yields after B3
    // oughtToYield[10] = false; // A doesn't yield after A2 (last A insertion)
    // oughtToYield[11] = false; // B doesn't yield after B1 (last B insertion)

    // two A nodes and 2 B nodes alternating
```

```
    oughtToYield[0] = false;  // A doesn't yield after A12
    oughtToYield[1] = true;   // A yields after A10 (2 A's done)
    oughtToYield[2] = false;  // B doesn't yield after B11
    oughtToYield[3] = true;   // B yields after B9 (2 B's done)
    oughtToYield[4] = false;  // A doesn't yield after A8
    oughtToYield[5] = true;   // A yields after A6 (2 A's done)
    oughtToYield[6] = false;  // B doesn't yield after B7
    oughtToYield[7] = true;   // B yields after B5 (2 B's done)
    oughtToYield[8] = false;  // A doesn't yield after A4
    oughtToYield[9] = false;  // A doesn't yield after A2 (last 2 A's)
    oughtToYield[10] = false; // B doesn't yield after B3
    oughtToYield[11] = false; // B doesn't yield after B1 (last 2 B's)

    new KThread(new DLListTest("B", 11, 1, 2)).setName("odd thread").fork();
    DLListTest testA = new DLListTest("A", 12, 2, 2);
    testA.run();

    // Print the final list after both threads complete
    // System.out.println("Final list: " + DLListTest.sharedList.toString());
    }

    private static final char dbgThread = 't';

    /**
     * Additional state used by schedulers.
     *
     * @see nachos.threads.PriorityScheduler.ThreadState
     */
    public Object schedulingState = null;

    private static final int statusNew = 0;
    private static final int statusReady = 1;
    private static final int statusRunning = 2;
    private static final int statusBlocked = 3;
```

```
    private static final int statusFinished = 4;

    /**
     * The status of this thread. A thread can either be new (not yet forked),
     * ready (on the ready queue but not running), running, or blocked (not
     * on the ready queue and not running).
     */
    private int status = statusNew;
    private String name = "(unnamed thread)";
    private Runnable target;
    private TCB tcb;

    /**
     * Unique identifer for this thread. Used to deterministically compare
     * threads.
     */
    private int id = numCreated++;
    /** Number of times the KThread constructor was called. */
    private static int numCreated = 0;

    private static ThreadQueue readyQueue = null;
    private static KThread currentThread = null;
    private static KThread toBeDestroyed = null;
    private static KThread idleThread = null;

    private static boolean[] oughtToYield = new boolean[100];
    private static int numTimesBefore = 0;

    // New 2D yielding mechanism for fine-grained control
    private static boolean[][] yieldData = new boolean[10][100];  //
[location][count]
    private static int[] yieldCount = new int[10];  // count for each location

    /**
```

```java
    * Tests the BoundedBuffer for underflow protection.
    * Without proper synchronization (condition variables), multiple readers
    * on an empty buffer would cause:
    * - Reads of uninitialized/stale data
    * - Array index out of bounds errors
    * - Race conditions on the size/first/last variables
    */
   public static void BB_underflowTest() {
       System.out.println("\n---BoundedBuffer Underflow Test---");
       System.out.println("Testing multiple readers on an initially empty
buffer");

       final BoundedBuffer buffer = new BoundedBuffer(3);
       final int[] readCount = new int[1];  // Track successful reads
       final boolean[] errors = new boolean[1];  // Track any errors

       // Reader thread class - tries to read from buffer
       class Reader implements Runnable {
           private String name;

           Reader(String name) {
               this.name = name;
           }

           public void run() {
               try {
                   for (int i = 0; i < 2; i++) {
                       char c = buffer.read();
                       System.out.println(name + " read: '" + c + "'");
                       readCount[0]++;
                       KThread.yield();
                   }
               } catch (Exception e) {
                   System.out.println(name + " ERROR: " + e.getMessage());
```

```
                errors[0] = true;
            }
        }
    }

    // Writer thread - writes data after a delay
    class DelayedWriter implements Runnable {
        public void run() {
            // Let readers start first and block
            KThread.yield();
            KThread.yield();

            System.out.println("Writer: Starting to write data...");
            buffer.write('A');
            buffer.write('B');
            buffer.write('C');
            buffer.write('D');
            System.out.println("Writer: Finished writing 4 characters");
        }
    }

    // Start multiple readers before any data is available
    new KThread(new Reader("Reader1")).setName("Reader1").fork();
    new KThread(new Reader("Reader2")).setName("Reader2").fork();

    // Start writer after readers
    new KThread(new DelayedWriter()).setName("Writer").fork();

    // Let threads run
    for (int i = 0; i < 10; i++) {
        KThread.yield();
    }

    System.out.println("\nTest Results:");
```

```
        System.out.println("Total successful reads: " + readCount[0]);
        System.out.println("Errors occurred: " + errors[0]);

        if (!errors[0] && readCount[0] == 4) { // what we expect
            System.out.println("Underflow working correctly!");
        } else {                               // errors
            System.out.println("Test failed");
        }
    }

    /**
     * Tests the BoundedBuffer for overflow protection.
     * Without proper synchronization, multiple writers on a full buffer would
cause:
     * - Data loss (overwriting unread data)
     * - Buffer corruption
     * - Incorrect size tracking
     */
    public static void BB_overflowTest() {
        System.out.println("\n---BoundedBuffer Overflow Test---");
        System.out.println("Testing multiple writers on a small buffer");

        // Very small buffer to quickly hit overflow condition
        final BoundedBuffer buffer = new BoundedBuffer(2);
        final int[] writeCount = new int[1];
        final boolean[] errors = new boolean[1];
        final boolean[] dataLoss = new boolean[1];

        // Writer thread class
        class Writer implements Runnable {
            private String name;
            private char startChar;

            Writer(String name, char startChar) {
```

```
            this.name = name;
            this.startChar = startChar;
        }

        public void run() {
            try {
                for (int i = 0; i < 3; i++) {
                    char c = (char)(startChar + i);
                    System.out.println(name + " writing: '" + c + "'");
                    buffer.write(c);
                    writeCount[0]++;
                    System.out.println(name + " successfully wrote: '" + c +
"'");
                }
            } catch (Exception e) {
                System.out.println(name + " ERROR: " + e.getMessage());
                errors[0] = true;
            }
        }
    }

    // Slow reader - reads data slowly
    class SlowReader implements Runnable {
        public void run() {
            // Let writers fill the buffer first
            KThread.yield();
            KThread.yield();
            KThread.yield();

            System.out.println("SlowReader: Starting to read...");
            StringBuilder readData = new StringBuilder();

            for (int i = 0; i < 6; i++) {
                char c = buffer.read();
```

```
                readData.append(c);
                System.out.println("SlowReader read: '" + c + "'");
                KThread.yield();  // Read slowly
            }

            String result = readData.toString();
            System.out.println("SlowReader: Read complete. Data: " +
result);

            // Make sure we got all expected data
            if (!result.contains("A") || !result.contains("D")) {
                dataLoss[0] = true;
            }
        }
    }

    // Start multiple writers that will overflow the buffer
    new KThread(new Writer("Writer1", 'A')).setName("Writer1").fork();
    new KThread(new Writer("Writer2", 'D')).setName("Writer2").fork();

    // Start slow reader
    new KThread(new SlowReader()).setName("SlowReader").fork();

    // Let threads run
    for (int i = 0; i < 15; i++) {
        KThread.yield();
    }

    System.out.println("\nTest Results:");
    System.out.println("Total successful writes: " + writeCount[0]);
    System.out.println("Errors occurred: " + errors[0]);
    System.out.println("Data loss detected: " + dataLoss[0]);

    if (!errors[0] && !dataLoss[0] && writeCount[0] == 6) {
```

```
            System.out.println("Overflow working correctly!");
        } else {
            System.out.println("Test failed");
        }
    }

/**
 * Complex producer-consumer test with multiple threads.
 * Tests the complete synchronization of the BoundedBuffer.
 * Without proper synchronization, this would cause:
 * - Deadlocks
 * - Data corruption
 * - Lost updates
 */
public static void BB_producerConsumerTest() {
    System.out.println("\n---BoundedBuffer Producer-Consumer Test---");
    System.out.println("Testing multiple producers and consumers");

    final BoundedBuffer buffer = new BoundedBuffer(5);
    final int[] totalProduced = new int[1];
    final int[] totalConsumed = new int[1];
    final boolean[] errors = new boolean[1];

    // Producer thread
    class Producer implements Runnable {
        private String name;
        private char baseChar;
        private int count;

        Producer(String name, char baseChar, int count) {
            this.name = name;
            this.baseChar = baseChar;
            this.count = count;
        }
```

```
        public void run() {
            try {
                for (int i = 0; i < count; i++) {
                    char c = (char)(baseChar + (i % 26));
                    buffer.write(c);
                    totalProduced[0]++;
                    System.out.println(name + " produced: '" + c + "'
(total: " + totalProduced[0] + ")");
                    KThread.yield();
                }
                System.out.println(name + " finished producing");
            } catch (Exception e) {
                System.out.println(name + " ERROR: " + e.getMessage());
                errors[0] = true;
            }
        }
    }

    // Consumer thread
    class Consumer implements Runnable {
        private String name;
        private int count;

        Consumer(String name, int count) {
            this.name = name;
            this.count = count;
        }

        public void run() {
            try {
                for (int i = 0; i < count; i++) {
                    char c = buffer.read();
                    totalConsumed[0]++;
```

```
                    System.out.println(name + " consumed: '" + c + "'
(total: " + totalConsumed[0] + ")");
                    KThread.yield();
                    KThread.yield(); // Consume slower than produce to test
buffering
                }
                System.out.println(name + " finished consuming");
            } catch (Exception e) {
                System.out.println(name + " ERROR: " + e.getMessage());
                errors[0] = true;
            }
        }
    }

    // Start multiple producers and consumers
    new KThread(new Producer("Producer1", 'A',
5)).setName("Producer1").fork();
    new KThread(new Producer("Producer2", 'a',
5)).setName("Producer2").fork();
    new KThread(new Consumer("Consumer1", 4)).setName("Consumer1").fork();
    new KThread(new Consumer("Consumer2", 4)).setName("Consumer2").fork();
    new KThread(new Consumer("Consumer3", 2)).setName("Consumer3").fork();

    // Let threads run
    for (int i = 0; i < 30; i++) {
        KThread.yield();
    }

    // Final buffer state
    System.out.println("\nFinal buffer state:");
    buffer.print();

    System.out.println("\nTest Results:");
    System.out.println("Total produced: " + totalProduced[0]);
```

```
        System.out.println("Total consumed: " + totalConsumed[0]);
        System.out.println("Errors occurred: " + errors[0]);

        if (!errors[0] && totalProduced[0] == 10 && totalConsumed[0] == 10) {
            System.out.println("Producer-Consumer test passed!");
        } else {
            System.out.println("Test failed");
        }
    }
}
```