Project 2

Name: Khai Dong

Part 1: Repair Project 1

The original problem is with mutual exclusion where the `DLList` breaks down upon doing concurrency. To solve this problem, we wrap each of the method in a lock to convert the original data structure into a monitor.

```
<ret-type> <method-name>(self, <arugments>){
    lock.acquire();
    ...
    lock.release();
}
```

To allow `DLList.removeHead()` to wait until the `DLList` is not empty, I added a condition variable `empty`.

```
public Object removeHead() {
    lock.acquire();
    while(this.first == null) {
        this.empty.sleep();
    }
    ...
}
```

Then, `DLList.insert()` will signal when an object is added to the list. This is the only method in `DLList` that insert an object into the list (`DLList.prepend()` calls `DLList.insert()`).

```
public void insert(Object item, Integer sortKey) {
    if(!lock.isHeldByCurrentThread())
        this.lock.acquire();

    ++ this.size; // increment size counter
    DLLElement newElem = new DLLElement(item, sortKey);
    if(this.first == null){
        this.first = this.last = newElem;
        this.empty.wake();
        this.lock.release();
        return;
    }

    ...

    this.empty.wake();
```

```
        this.lock.release();
    }
```

`DLList.insert()` check if the current thread already acquire the lock before try to acquire the lock since it may be called by `DLList.prepend()` . Return to the 2 tests from project 1, there are no longer a `NullExceptionPointer` nor an invalid input.

```
([1,1] [3,3] [4,4] [6,6] (previously, ([1,1] [4, 4] [3,3] [6,6])
([1,1]) (previously throw a NullPointerException)
```

Part 2: Bounder Buffer

I implemented `BoundedBuffer` with several `KThread.yieldIfOughtTo()` within the `read()` and `write()` code to see if constant context switching would cause problems. Inside of `read` and `write` , there are also assertion to check for underflow and overflow. I made 3 tests to test `Bounded Buffer` .

- `BoundedBuffer_selfTestUnderflow` : First fork a `write` thread, then having a thread try to `read()` when the buffer is empty. The `read` thread will block itself, allow the `write` thread to run, then the `read` thread will print out the character. Expect no `AssertionError` and the correct character get printed out (also get asserted).
- `BoundedBuffer_selfTestOverflow` : First set up a buffer with a capacity of 1, then fill it with a character. Then, fork a `read` thread and run a `write` thread. Assert if `read` gives the right character. Expect no `AssertionError` (no overflow, underflow, nor invalid output). The read character also get printed out.
- `BoundedBuffer_selfTestConcurrency` : First set `Kthread.oughtToYield` to a sufficient long list of `True` . Then, fork several `read` and `write` threads then run them. For this test, since there is no exact order, I just expect all read values and the values remaining in the buffer added up to all the added characters. I used a `synchronizedList` to accumulate all the characters, then sequentially `read()` everything out of the buffer at the end. Also check for underflow and overflow in this test since assertions are built into `BoundedBuffer` .

The tests run correctly. Since there is no interleaving that could cause problem (besides starvation), there is no interleaving diagram. The assertions should already account for potential errors that may arise.

Part 3: `Condition2` Implementation

Changing from `Condition` to `Condition2` gives the same output.

```
KThread.DLL_selfTest() ...
([-11,B1] [-10,B3] [-9,B5] [-8,B7] [-7,B9] [-6,B11] [-5,A2] [-4,A4] [-3,A6] [-2,A8] [-1,A10] [0,A12])
([-11,B1] [-10,A2] [-9,B3] [-8,A4] [-7,B5] [-6,A6] [-5,B7] [-4,A8] [-3,B9] [-2,A10] [-1,B11] [0,A12])
([-11,B1] [-10,A2] [-9,B3] [-8,B5] [-7,A4] [-6,A6] [-5,B7] [-4,A8] [-3,B9] [-2,B11] [-1,A10] [0,A12])
([1,1] [3,3] [4,4] [6,6])
([1,1])
remove (1, 1) after waiting
1
()
```

```
()
KThread.BoundedBufferTest() ...
a
[]
a
[b]
Other tests ...
```

```java
package nachos.threads;

import java.util.ArrayDeque;
import java.util.Queue;

public class BoundedBuffer {

    private final int maxsize;
    Queue<Character> buffer;

    // synchronized
    private final Lock lock;
    private final Condition2 empty; // whether the buffer is empty
    private final Condition2 full; // whether the buffer is full

    // non-default constructor with a fixed size
    public BoundedBuffer(int maxsize) {
        this.maxsize = maxsize;
        this.buffer = new ArrayDeque<>(maxsize);

        this.lock = new Lock();
        this.empty = new Condition2(this.lock);
        this.full = new Condition2(this.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() {
        this.lock.acquire();

        KThread.yieldIfOughtTo();

        if(this.buffer.isEmpty()) {
            this.empty.sleep(); // wait if there is nothing in the buffer
        }
```

```
37          Character res = this.buffer.poll();
38          assert res != null : "underflow"; // check
   for underflow
39
40          KThread.yieldIfOughtTo();
41
42          this.full.wake(); // wake a write thread up
   when remove a thing from the buffer
43          this.lock.release();
44          return res;
45      }
46
47      // Write the given character c into the buffer,
   blocking until
48      // enough space is available to satisfy the
   request.
49      public void write(char c) {
50          this.lock.acquire();
51
52          KThread.yieldIfOughtTo();
53
54          if(this.buffer.size() == this.maxsize){
55              this.full.sleep();
56          }
57
58          KThread.yieldIfOughtTo();
59
60          this.buffer.add(c);
61          assert this.buffer.size() >= this.maxsize : "
   overflow"; // check for underflow
62
63          this.empty.wake(); // wake a read() thread up
    after adding thing to the buffer
64          this.lock.release();
65      }
66
67      // Prints the contents of the buffer; for
   debugging only
68      public void print() {
69          this.lock.acquire();
70          String res = this.buffer.toString();
```

```
71          this.lock.release();
72          System.out.println(res);
73      }
74
75      public boolean isEmpty(){
76          lock.acquire();
77          boolean res = this.buffer.isEmpty();
78          lock.release();
79          return res;
80      }
81
82  }
83
```

```
 1 package nachos.threads;
 2
 3 import java.util.LinkedList;
 4 import nachos.machine.*;
 5 import nachos.threads.*;
 6
 7 /**
 8  * A synchronized queue.
 9  */
10 public class SynchList {
11     /**
12      * Allocate a new synchronized queue.
13      */
14     public SynchList() {
15     list = new LinkedList<Object>();
16     lock = new Lock();
17     listEmpty = new Condition(lock);
18     }
19
20     /**
21      * Add the specified object to the end of the
   queue. If another thread is
22      * waiting in <tt>removeFirst()</tt>, it is woken
    up.
23      *
24      * @param   o   the object to add. Must not be <
   tt>null</tt>.
25      */
26     public void add(Object o) {
27     Lib.assertTrue(o != null);
28
29     lock.acquire();
30     list.add(o);
31     listEmpty.wake();
32     lock.release();
33     }
34
35     /**
36      * Remove an object from the front of the queue,
   blocking until the queue
37      * is non-empty if necessary.
```

```
38      *
39      * @return  the element removed from the front of
    the queue.
40      */
41     public Object removeFirst() {
42     Object o;
43
44     lock.acquire();
45     while (list.isEmpty())
46         listEmpty.sleep();
47     o = list.removeFirst();
48     lock.release();
49
50     return o;
51     }
52
53     public boolean isEmpty(){
54         lock.acquire();
55         boolean res = list.isEmpty();
56         lock.release();
57         return res;
58     }
59
60     private static class PingTest implements Runnable
    {
61     PingTest(SynchList ping, SynchList pong) {
62         this.ping = ping;
63         this.pong = pong;
64     }
65
66     public void run() {
67         for (int i=0; i<10; i++)
68         pong.add(ping.removeFirst());
69     }
70
71     private SynchList ping;
72     private SynchList pong;
73     }
74
75     /**
76      * Test that this module is working.
```

```
77      */
78     public static void selfTest() {
79     SynchList ping = new SynchList();
80     SynchList pong = new SynchList();
81
82     new KThread(new PingTest(ping, pong)).setName("
   ping").fork();
83
84     for (int i=0; i<10; i++) {
85         Integer o = new Integer(i);
86         ping.add(o);
87         Lib.assertTrue(pong.removeFirst() == o);
88     }
89     }
90
91     private LinkedList<Object> list;
92     private Lock lock;
93     private Condition listEmpty;
94 }
95
96
```

```
 1 package nachos.threads;
 2
 3 import nachos.machine.Lib;
 4 import nachos.machine.Machine;
 5
 6 /**
 7  * An implementation of condition variables that
   disables interrupt()s for
 8  * synchronization.
 9  *
10  * <p>
11  * You must implement this.
12  *
13  * @see nachos.threads.Condition
14  */
15 public class Condition2 {
16     /**
17      * Allocate a new condition variable.
18      *
19      * @param   conditionLock   the lock associated
   with this condition
20      *              variable. The current thread must
    hold this
21      *              lock whenever it uses <tt>sleep()
   </tt>,
22      *              <tt>wake()</tt>, or <tt>wakeAll()
   </tt>.
23      */
24     public Condition2(Lock conditionLock) {
25         this.conditionLock = conditionLock;
26         this.waitQueue = new SynchList();
27     }
28
29     /**
30      * Atomically release the associated lock and go
   to sleep on this condition
31      * variable until another thread wakes it using <
   tt>wake()</tt>. The
32      * current thread must hold the associated lock.
   The thread will
33      * automatically reacquire the lock before <tt>
```

```
33 sleep()</tt> returns.
34      */
35     public void sleep() {
36         Lib.assertTrue(conditionLock.
   isHeldByCurrentThread());
37
38         conditionLock.release(); // release the lock
39         boolean intStatus = Machine.interrupt().
   disable(); // disable interrupts
40         waitQueue.add(KThread.currentThread()); //
   put current thread into the wait queue
41         KThread.sleep(); // block current thread
42         Machine.interrupt().restore(intStatus); //
   restore interrupts
43         conditionLock.acquire(); //  reacquire the
   lock
44     }
45
46     /**
47      * Wake up at most one thread sleeping on this
   condition variable. The
48      * current thread must hold the associated lock.
49      */
50     public void wake() {
51         Lib.assertTrue(conditionLock.
   isHeldByCurrentThread());
52         if (!waitQueue.isEmpty()) {
53             boolean intStatus = Machine.interrupt().
   disable(); // disable interrupts
54             ((KThread) waitQueue.removeFirst()).ready
   (); // dispatch thread from the wait queue
55             Machine.interrupt().restore(intStatus);
   // restore interrupts
56         }
57     }
58
59     /**
60      * Wake up all threads sleeping on this condition
    variable. The current
61      * thread must hold the associated lock.
62      */
```

```
63      public void wakeAll() {
64          Lib.assertTrue(conditionLock.
    isHeldByCurrentThread());
65          boolean intStatus = Machine.interrupt().
    disable(); // disable interrupts
66          while(!waitQueue.isEmpty()) {
67              ((KThread) waitQueue.removeFirst()).
    ready(); // dispatch thread from the wait queue
68          }
69          Machine.interrupt().restore(intStatus); //
    restore interrupts
70      }
71
72      private Lock conditionLock;
73
74      private SynchList waitQueue;
75
76  }
77
```

```
 1 package nachos.threads;
 2
 3 import nachos.machine.*;
 4
 5 import java.util.*;
 6
 7 /**
 8  * A KThread is a thread that can be used to execute
   Nachos kernel code. Nachos
 9  * allows multiple threads to run concurrently.
10  *
11  * To create a new thread of execution, first declare
    a class that implements
12  * the <tt>Runnable</tt> interface. That class then
   implements the <tt>run</tt>
13  * method. An instance of the class can then be
   allocated, passed as an
14  * argument when creating <tt>KThread</tt>, and
   forked. For example, a thread
15  * that computes pi could be written as follows:
16  *
17  * <p><blockquote><pre>
18  * class PiRun implements Runnable {
19  *     public void run() {
20  *         // compute pi
21  *         ...
22  *     }
23  * }
24  * </pre></blockquote>
25  * <p>The following code would then create a thread
   and start it running:
26  *
27  * <p><blockquote><pre>
28  * PiRun p = new PiRun();
29  * new KThread(p).fork();
30  * </pre></blockquote>
31  */
32 public class KThread {
33     /**
34      * Get the current thread.
35      *
```

```
36      * @return  the current thread.
37      */
38     public static KThread currentThread() {
39     Lib.assertTrue(currentThread != null);
40     return currentThread;
41     }
42
43     /**
44      * Allocate a new <tt>KThread</tt>. If this is
   the first <tt>KThread</tt>,
45      * create an idle thread as well.
46      */
47     public KThread() {
48     if (currentThread != null) {
49         tcb = new TCB();
50     }
51     else {
52         readyQueue = ThreadedKernel.scheduler.
   newThreadQueue(false);
53         readyQueue.acquire(this);
54
55         currentThread = this;
56         tcb = TCB.currentTCB();
57         name = "main";
58         restoreState();
59
60         createIdleThread();
61     }
62     }
63
64     /**
65      * Allocate a new KThread.
66      *
67      * @param   target  the object whose <tt>run</tt>
    method is called.
68      */
69     public KThread(Runnable target) {
70     this();
71     this.target = target;
72     }
73
```

```
 74
 75      /**
 76       * Set the target of this thread.
 77       *
 78       * @param   target  the object whose <tt>run</tt
    > method is called.
 79       * @return  this thread.
 80       */
 81      public KThread setTarget(Runnable target) {
 82      Lib.assertTrue(status == statusNew);
 83
 84      this.target = target;
 85      return this;
 86      }
 87
 88      /**
 89       * Set the name of this thread. This name is
    used for debugging purposes
 90       * only.
 91       *
 92       * @param   name    the name to give to this
    thread.
 93       * @return  this thread.
 94       */
 95      public KThread setName(String name) {
 96      this.name = name;
 97      return this;
 98      }
 99
100      /**
101       * Get the name of this thread. This name is
    used for debugging purposes
102       * only.
103       *
104       * @return  the name given to this thread.
105       */
106      public String getName() {
107      return name;
108      }
109
110      /**
```

```
111      * Get the full name of this thread. This
    includes its name along with its
112      * numerical ID. This name is used for debugging
     purposes only.
113      *
114      * @return  the full name given to this thread.
115      */
116     public String toString() {
117     return (name + " (#" + id + ")");
118     }
119
120     /**
121      * Deterministically and consistently compare
    this thread to another
122      * thread.
123      */
124     public int compareTo(Object o) {
125     KThread thread = (KThread) o;
126
127     if (id < thread.id)
128         return -1;
129     else if (id > thread.id)
130         return 1;
131     else
132         return 0;
133     }
134
135     /**
136      * Causes this thread to begin execution. The
    result is that two threads
137      * are running concurrently: the current thread
     (which returns from the
138      * call to the <tt>fork</tt> method) and the
    other thread (which executes
139      * its target's <tt>run</tt> method).
140      */
141     public void fork() {
142     Lib.assertTrue(status == statusNew);
143     Lib.assertTrue(target != null);
144
145     Lib.debug(dbgThread,
```

```
146           "Forking thread: " + toString() + "
    Runnable: " + target);
147
148     boolean intStatus = Machine.interrupt().disable
    ();
149
150     tcb.start(new Runnable() {
151         public void run() {
152             runThread();
153         }
154         });
155
156     ready();
157
158     Machine.interrupt().restore(intStatus);
159     }
160
161     private void runThread() {
162     begin();
163     target.run();
164     finish();
165     }
166
167     private void begin() {
168     Lib.debug(dbgThread, "Beginning thread: " +
    toString());
169
170     Lib.assertTrue(this == currentThread);
171
172     restoreState();
173
174     Machine.interrupt().enable();
175     }
176
177     /**
178      * Finish the current thread and schedule it to
    be destroyed when it is
179      * safe to do so. This method is automatically
    called when a thread's
180      * <tt>run</tt> method returns, but it may also
    be called directly.
```

```
181      *
182      * The current thread cannot be immediately
    destroyed because its stack and
183      * other execution state are still in use.
    Instead, this thread will be
184      * destroyed automatically by the next thread to
     run, when it is safe to
185      * delete this thread.
186      */
187     public static void finish() {
188     Lib.debug(dbgThread, "Finishing thread: " +
    currentThread.toString());
189
190     Machine.interrupt().disable();
191
192     Machine.autoGrader().finishingCurrentThread();
193
194     Lib.assertTrue(toBeDestroyed == null);
195     toBeDestroyed = currentThread;
196
197
198     currentThread.status = statusFinished;
199
200     sleep();
201     }
202
203     /**
204      * Relinquish the CPU if any other thread is
    ready to run. If so, put the
205      * current thread on the ready queue, so that it
     will eventually be
206      * rescheduled.
207      *
208      * <p>
209      * Returns immediately if no other thread is
    ready to run. Otherwise
210      * returns when the current thread is chosen to
    run again by
211      * <tt>readyQueue.nextThread()</tt>.
212      *
213      * <p>
```

```
214      * Interrupts are disabled, so that the current
    thread can atomically add
215      * itself to the ready queue and switch to the
    next thread. On return,
216      * restores interrupts to the previous state, in
     case <tt>yield()</tt> was
217      * called with interrupts disabled.
218      */
219     public static void yield() {
220     Lib.debug(dbgThread, "Yielding thread: " +
    currentThread.toString());
221
222     Lib.assertTrue(currentThread.status ==
    statusRunning);
223
224     boolean intStatus = Machine.interrupt().disable
    ();
225
226     currentThread.ready();
227
228     runNextThread();
229
230     Machine.interrupt().restore(intStatus);
231     }
232
233     /**
234      * Relinquish the CPU, because the current
    thread has either finished or it
235      * is blocked. This thread must be the current
    thread.
236      *
237      * <p>
238      * If the current thread is blocked (on a
    synchronization primitive, i.e.
239      * a <tt>Semaphore</tt>, <tt>Lock</tt>, or <tt>
    Condition</tt>), eventually
240      * some thread will wake this thread up, putting
     it back on the ready queue
241      * so that it can be rescheduled. Otherwise, <tt
    >finish()</tt> should have
242      * scheduled this thread to be destroyed by the
```

```
242 next thread to run.
243      */
244     public static void sleep() {
245     Lib.debug(dbgThread, "Sleeping thread: " +
    currentThread.toString());
246
247     Lib.assertTrue(Machine.interrupt().disabled());
248
249     if (currentThread.status != statusFinished)
250         currentThread.status = statusBlocked;
251
252     runNextThread();
253     }
254
255     /**
256      * Moves this thread to the ready state and adds
     this to the scheduler's
257      * ready queue.
258      */
259     public void ready() {
260     Lib.debug(dbgThread, "Ready thread: " + toString
    ());
261
262     Lib.assertTrue(Machine.interrupt().disabled());
263     Lib.assertTrue(status != statusReady);
264
265     status = statusReady;
266     if (this != idleThread)
267         readyQueue.waitForAccess(this);
268
269     Machine.autoGrader().readyThread(this);
270     }
271
272     /**
273      * Waits for this thread to finish. If this
    thread is already finished,
274      * return immediately. This method must only be
    called once; the second
275      * call is not guaranteed to return. This thread
     must not be the current
276      * thread.
```

```
277      */
278     public void join() {
279     Lib.debug(dbgThread, "Joining to thread: " +
    toString());
280
281     Lib.assertTrue(this != currentThread);
282
283     }
284
285     /**
286      * Create the idle thread. Whenever there are no
     threads ready to be run,
287      * and <tt>runNextThread()</tt> is called, it
    will run the idle thread. The
288      * idle thread must never block, and it will
    only be allowed to run when
289      * all other threads are blocked.
290      *
291      * <p>
292      * Note that <tt>ready()</tt> never adds the
    idle thread to the ready set.
293      */
294     private static void createIdleThread() {
295     Lib.assertTrue(idleThread == null);
296
297     idleThread = new KThread(new Runnable() {
298         public void run() { while (true) Machine.
    yield(); }
299     });
300     idleThread.setName("idle");
301
302     Machine.autoGrader().setIdleThread(idleThread);
303
304     idleThread.fork();
305     }
306
307     /**
308      * Determine the next thread to run, then
    dispatch the CPU to the thread
309      * using <tt>run()</tt>.
310      */
```

```
311     private static void runNextThread() {
312     KThread nextThread = readyQueue.nextThread();
313     if (nextThread == null)
314         nextThread = idleThread;
315
316     nextThread.run();
317     }
318
319     /**
320      * Dispatch the CPU to this thread. Save the
    state of the current thread,
321      * switch to the new thread by calling <tt>TCB.
    contextSwitch()</tt>, and
322      * load the state of the new thread. The new
    thread becomes the current
323      * thread.
324      *
325      * <p>
326      * If the new thread and the old thread are the
    same, this method must
327      * still call <tt>saveState()</tt>, <tt>
    contextSwitch()</tt>, and
328      * <tt>restoreState()</tt>.
329      *
330      * <p>
331      * The state of the previously running thread
    must already have been
332      * changed from running to blocked or ready (
    depending on whether the
333      * thread is sleeping or yielding).
334      *
335      * @param   finishing   <tt>true</tt> if the
    current thread is
336      *              finished, and should be
    destroyed by the new
337      *              thread.
338      */
339     private void run() {
340     Lib.assertTrue(Machine.interrupt().disabled());
341
342     Machine.yield();
```

```
343
344     currentThread.saveState();
345
346     Lib.debug(dbgThread, "Switching from: " +
    currentThread.toString()
347           + " to: " + toString());
348
349     currentThread = this;
350
351     tcb.contextSwitch();
352
353     currentThread.restoreState();
354     }
355
356     /**
357      * Prepare this thread to be run. Set <tt>status
    </tt> to
358      * <tt>statusRunning</tt> and check <tt>
    toBeDestroyed</tt>.
359      */
360     protected void restoreState() {
361     Lib.debug(dbgThread, "Running thread: " +
    currentThread.toString());
362
363     Lib.assertTrue(Machine.interrupt().disabled());
364     Lib.assertTrue(this == currentThread);
365     Lib.assertTrue(tcb == TCB.currentTCB());
366
367     Machine.autoGrader().runningThread(this);
368
369     status = statusRunning;
370
371     if (toBeDestroyed != null) {
372         toBeDestroyed.tcb.destroy();
373         toBeDestroyed.tcb = null;
374         toBeDestroyed = null;
375     }
376     }
377
378     /**
379      * Prepare this thread to give up the processor
```

```
379 . Kernel threads do not
380      * need to do anything here.
381      */
382     protected void saveState() {
383     Lib.assertTrue(Machine.interrupt().disabled());
384     Lib.assertTrue(this == currentThread);
385     }
386
387     private static class PingTest implements
    Runnable {
388     PingTest(int which) {
389         this.which = which;
390     }
391
392     public void run() {
393         for (int i=0; i<5; i++) {
394         System.out.println("*** thread " + which +
    " looped "
395                     + i + " times");
396         currentThread.yield();
397         }
398     }
399
400     private int which;
401     }
402
403     /**
404      * Tests whether this module is working.
405      */
406     public static void selfTest() {
407     Lib.debug(dbgThread, "Enter KThread.selfTest");
408
409     new KThread(new PingTest(1)).setName("forked
    thread").fork();
410     new PingTest(0).run();
411     }
412
413     private static final char dbgThread = 't';
414
415     /**
416      * Additional state used by schedulers.
```

```
417      *
418      * @see nachos.threads.PriorityScheduler.
    ThreadState
419      */
420     public Object schedulingState = null;
421
422     private static final int statusNew = 0;
423     private static final int statusReady = 1;
424     private static final int statusRunning = 2;
425     private static final int statusBlocked = 3;
426     private static final int statusFinished = 4;
427
428     /**
429      * The status of this thread. A thread can
    either be new (not yet forked),
430      * ready (on the ready queue but not running),
    running, or blocked (not
431      * on the ready queue and not running).
432      */
433     private int status = statusNew;
434     private String name = "(unnamed thread)";
435     private Runnable target;
436     private TCB tcb;
437
438     /**
439      * Unique identifer for this thread. Used to
    deterministically compare
440      * threads.
441      */
442     private int id = numCreated++;
443     /** Number of times the KThread constructor was
    called. */
444     private static int numCreated = 0;
445
446     private static ThreadQueue readyQueue = null;
447     private static KThread currentThread = null;
448     private static KThread toBeDestroyed = null;
449     private static KThread idleThread = null;
450
451
452
```

```
452 
    ////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////
    //////
453 
    ////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////
    //////
454 
    ////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////
    //////
455     // new stuff starts here
456 
457     public static boolean[] oughtToYield = null;
458     public static int numTimesBefore = 0;
459     public static void yieldIfOughtTo() {
460         if (numTimesBefore < oughtToYield.length &&
    oughtToYield[numTimesBefore]){
461             numTimesBefore++;
462             KThread.yield();
463         } else {
464             numTimesBefore++;
465         }
466     }
467 
468     public static boolean[][] yieldData = null;
469     public static int[] yieldCount = null;
470 
471     /**
472      * Given this unique location, yield the
473      * current thread if it ought to. It knows
474      * to do this if yieldData[i][loc] is true,
    where
475      * i is the number of times that this function
476      * has already been called from this location.
477      *
478      * @param loc unique location. Every call to
479      * yieldIfShould that you
480      * place in your DLList code should
481      * have a different loc number.
```

```
482      */
483     public static void yieldIfShould(int loc) {
484         if (yieldCount == null || yieldData == null)
485             return;
486         if (0 <= loc && loc < yieldCount.length &&
    loc < yieldData.length){
487             int curCount = yieldCount[loc];
488             if (yieldData[loc] != null && 0 <=
    curCount && curCount <= yieldData[loc].length){
489                 yieldCount[loc] += 1;
490                 if (yieldData[loc][curCount]){
491                     KThread.yield();
492                 }
493 
494             }
495         }
496     }
497 
498 
499     /**
500      * Tests the shared DLList by having two threads
     running countdown.
501      * One thread will insert even-numbered data
    from "A12" to "A2".
502      * The other thread will insert odd-numbered
    data from "B11" to "B1".
503      * Don't forget to initialize the oughtToYield
    array before forking.
504      *
505      */
506     public static void DLL_selfTest() {
507         Lib.debug(dbgThread, "Enter KThread.
    DLL_selfTest");
508 
509         // all A nodes get appended before B nodes
510         DLList.DLListTest.reset();
511         numTimesBefore = 0;
512         oughtToYield = new boolean[]{
513                 false, false, false, false, false,
    true,
514                 false, false, false, false, false,
```

```
514 false};
515         new KThread(new DLList.DLListTest("B", 11, 1
    , 2)).fork();
516         new DLList.DLListTest("A", 12, 1, 2).run();
517         System.out.println(DLList.DLListTest.
    testList);
518 
519         // A and B node alternating
520         DLList.DLListTest.reset();
521         numTimesBefore = 0;
522         oughtToYield = new boolean[]{
523                 true, true, true, true, true, true,
524                 true, true, true, true, true, false
    };
525         new KThread(new DLList.DLListTest("B", 11, 1
    , 2)).fork();
526         new DLList.DLListTest("A", 12, 1, 2).run();
527         System.out.println(DLList.DLListTest.
    testList);
528 
529         // 2 A nodes then 2 B nodes
530         DLList.DLListTest.reset();
531         numTimesBefore = 0;
532         oughtToYield = new boolean[]{
533                 false, true, false, true, false,
    true,
534                 false, true, false, true, false,
    false};
535         new KThread(new DLList.DLListTest("B", 11, 1
    , 2)).fork();
536         new DLList.DLListTest("A", 12, 1, 2).run();
537         System.out.println(DLList.DLListTest.
    testList);
538 
539     }
540 
541     public static void DLL_selfTest2(){
542         Lib.debug(dbgThread, "Enter KThread.
    DLL_selfTest2");
543         DLList testList = new DLList();
544 
```

```
545         testList.insert(1, 1);
546         testList.insert(6, 6);
547 
548         yieldCount = new int[]{0};
549         yieldData = new boolean[][]{
550                 new boolean[] {true, false}
551         };
552         new KThread(() -> {
553             testList.insert(4, 4);
554             KThread.yield();
555         }).fork();
556         testList.insert(3, 3);
557         System.out.println(testList);
558     }
559 
560     public static void DLL_selfTest3(){
561         Lib.debug(dbgThread, "Enter KThread.
    DLL_selfTest2");
562         DLList testList = new DLList();
563 
564         testList.insert(1, 1);
565 
566         yieldCount = new int[]{0, 0};
567         yieldData = new boolean[][]{
568                 null,
569                 new boolean[] {true, false}
570         };
571         new KThread(() -> {
572             testList.removeHead();
573             KThread.yield();
574         }).fork();
575         try {
576             testList.prepend(-1);
577             System.out.println(testList);
578         } catch (NullPointerException e){
579             System.out.println("Catch a
    NullPointerException, as desired");
580         }
581     }
582 
583     public static void DLL_selfTest4(){
```

```
584         Lib.debug(dbgThread, "Enter KThread.
    DLL_selfTest4");
585         DLList testList = new DLList();
586 
587         yieldCount = null;
588         yieldData = null;
589         new KThread(() -> {
590             testList.insert(1, 1);
591             KThread.yield();
592         }).fork();
593 
594         Integer i = (Integer) testList.removeHead();
595         System.out.println("remove (1, 1) after
    waiting");
596         System.out.println(i);
597         System.out.println(testList);
598 
599     }
600 
601     public static void
    BoundedBuffer_selfTestUnderflow() {
602         Lib.debug(dbgThread, "Enter KThread.
    BoundedBuffer_selfTestUnderflow");
603 
604         BoundedBuffer buffer = new BoundedBuffer(1);
605         new KThread(() -> buffer.write('a')).fork();
606         char c = buffer.read();
607         assert c == 'a';
608         System.out.println(c);
609         buffer.print();
610     }
611 
612     public static void
    BoundedBuffer_selfTestOverflow() {
613         Lib.debug(dbgThread, "Enter KThread.
    BoundedBuffer_selfTestOverflow");
614         BoundedBuffer buffer = new BoundedBuffer(1);
615         buffer.write('a');
616 
617         new KThread(() -> {
618             char c = buffer.read();
```

```
619             assert c == 'a';
620             System.out.println(c);
621         }).fork();
622         buffer.write('b');
623         buffer.print();
624 
625         assert buffer.read() == 'b';
626         assert buffer.isEmpty();
627     }
628 
629     public static void
    BoundedBuffer_selfTestConcurrency() {
630         Lib.debug(dbgThread, "Enter KThread.
    BoundedBuffer_selfTestConcurrency");
631 
632         oughtToYield = new boolean[50];
633         Arrays.fill(oughtToYield, true);
634         List<Character> accum = Collections.
    synchronizedList(new ArrayList<>());
635 
636         BoundedBuffer buffer = new BoundedBuffer(10
    );
637 
638         new KThread(() -> buffer.write('a')).fork();
639         new KThread(() -> buffer.write('c')).fork();
640         new KThread(() -> buffer.write('b')).fork();
641         new KThread(() -> buffer.write('w')).fork();
642 
643         new KThread(() -> accum.add(buffer.read())).
    fork();
644         new KThread(() -> accum.add(buffer.read())).
    fork();
645         new KThread(() -> accum.add(buffer.read())).
    fork();
646         new KThread(() -> accum.add(buffer.read())).
    fork();
647         new KThread(() -> accum.add(buffer.read())).
    fork();
648         new KThread(() -> accum.add(buffer.read())).
    fork();
649 
```

```
650         new KThread(() -> buffer.write('a')).fork();
651         new KThread(() -> buffer.write('c')).fork();
652         new KThread(() -> buffer.write('b')).fork();
653         new KThread(() -> buffer.write('w')).fork();
654         new KThread(() -> buffer.write('c')).fork();
655         new KThread(() -> buffer.write('b')).fork();
656 
657         new KThread(() -> accum.add(buffer.read())).
    fork();
658         new KThread(() -> accum.add(buffer.read())).
    fork();
659         new KThread(() -> accum.add(buffer.read())).
    fork();
660         new KThread(() -> accum.add(buffer.read())).
    fork();
661 
662 
663         buffer.write('a'); // set off all the other
    thread
664         while (!buffer.isEmpty()) accum.add(buffer.
    read());
665 
666         assert Collections.frequency(accum, 'a') ==
    3;
667         assert Collections.frequency(accum, 'b') ==
    3;
668         assert Collections.frequency(accum, 'c') ==
    3;
669         assert Collections.frequency(accum, 'w') ==
    2;
670 
671     }
672 
673 
674 }
675 
```

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

/**
 * An implementation of a doubly linked list
 * Edited by Khai Dong (dongk@union.edu)
 */
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

    // synchronized
    private Lock lock;
    private Condition2 empty;


    /**
     * Creates an empty sorted doubly-linked list.
     */
    public DLList() {
        this.first = this.last = null;
        this.size = 0;

        this.lock = new Lock();
        this.empty = new Condition2(this.lock);
    }

    /**
     * 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.
     */
```

```
35      public void prepend(Object item) {
36          lock.acquire();
37
38          int key = 0;
39          if (this.first != null){
40              KThread.yieldIfShould(1);
41              key = this.first.key - 1;
42          }
43          this.insert(item, key);
44          // lock get released by insert(...)
45      }
46
47      /**
48       * Removes the head of the list and returns the
    data item stored in
49       * it.
50       * underflow is prevented (i.e. removeHead will
    now always remove
51       * something instead of ever returning null
52       *
53       * @return the data stored at the head of the
    list, sleep until list has something
54       */
55      public Object removeHead() {
56
57          lock.acquire();
58          while(this.first == null) {
59              this.empty.sleep();
60          }
61
62          Object returnData = this.first.data;
63          -- this.size; // decrement the size
64          this.first = this.first.next; // update new
    head
65          if(this.first == null) { // if pop the last
    element
66              this.last = null;
67          } else {
68              this.first.prev = null; // update
    previous of the current head
69          }
```

```
 70
 71          lock.release();
 72          return returnData;
 73      }
 74
 75      /**
 76       * Tests whether the list is empty.
 77       *
 78       * @return true iff the list is empty.
 79       */
 80      public boolean isEmpty() {
 81          lock.acquire();
 82          boolean res = this.size == 0;
 83          lock.release();
 84          return res;
 85
 86
 87      }
 88
 89      /**
 90       * returns number of items in list
 91       * @return
 92       */
 93      public int size(){
 94          lock.acquire();
 95          int res = this.size;
 96          lock.release();
 97          return res;
 98      }
 99
100
101      /**
102       * Inserts item into the list in sorted order
    according to sortKey.
103       */
104      public void insert(Object item, Integer sortKey
    ) {
105          if(!lock.isHeldByCurrentThread())
106              this.lock.acquire();
107
108          ++ this.size; // increment size counter
```

```
109         DLLElement newElem = new DLLElement(item,
    sortKey);
110         if(this.first == null){
111             this.first = this.last = newElem;
112 
113             this.empty.wake();
114             this.lock.release();
115             return;
116         }
117         DLLElement curElem = this.first;
118         // traverse the list to find the correct
    position to put item
119         while(curElem != null && curElem.key <=
    sortKey)
120             curElem = curElem.next;
121 
122         KThread.yieldIfShould(0);
123         if(curElem == null){
124             this.last.next = newElem;
125             newElem.prev = this.last;
126             this.last = newElem;
127         } else {
128             newElem.prev = curElem.prev;
129             newElem.next = curElem;
130             curElem.prev = newElem;
131             if (newElem.prev == null) this.first =
    newElem;
132             else newElem.prev.next = newElem;
133         }
134         this.empty.wake();
135         this.lock.release();
136     }
137 
138 
139     /**
140      * returns list as a printable string. A single
    space should separate each list item,
141      * and the entire list should be enclosed in
    parentheses. Empty list should return "()"
142      * @return list elements in order
143      */
```

```
144     public String toString() {
145         StringBuilder builder = new StringBuilder();
146         builder.append('(');
147 
148         this.lock.acquire();
149         DLLElement curNode = this.first;
150         while(curNode != null){
151             builder.append(curNode.toString());
152             if(curNode != this.last) builder.append(
    ' ');
153             curNode = curNode.next;
154         }
155         this.lock.release();
156 
157         builder.append(')');
158         return builder.toString();
159     }
160 
161     /**
162      * returns list as a printable string, from the
    last node to the first.
163      * String should be formatted just like in
    toString.
164      * @return list elements in backwards order
165      */
166     public String reverseToString(){
167         StringBuilder builder = new StringBuilder();
168         builder.append('(');
169 
170         this.lock.acquire();
171         DLLElement curNode = this.last;
172         while(curNode != null){
173             builder.append(curNode.toString());
174             if(curNode != this.first) builder.append
    (' ');
175             curNode = curNode.prev;
176         }
177         this.lock.release();
178 
179         builder.append(')');
180         return builder.toString();
```

```
181     }
182 
183     /**
184      *  inner class for the node
185      */
186     private class DLLElement
187     {
188         private DLLElement next;
189         private DLLElement prev;
190         private int key;
191         private Object data;
192 
193         /**
194          * Node constructor
195          * @param item data item to store
196          * @param sortKey unique integer ID
197          */
198         public DLLElement(Object item, int sortKey)
199         {
200             key = sortKey;
201             data = item;
202             next = null;
203             prev = null;
204         }
205 
206         /**
207          * returns node contents as a printable
    string
208          * @return string of form [<key>,<data>]
    such as [3,"ham"]
209          */
210         public String toString(){
211             return "[" + key + "," + data + "]";
212         }
213     }
214 
215 
216     public static class DLListTest implements
    Runnable {
217 
218         // shared doubly linked list
```

```
219         public static DLList testList = new DLList
    ();
220 
221         private final String label;
222         private final int from, to, step;
223 
224         DLListTest(String label, int from, int to,
    int step){
225             assert from >= to;
226             assert step > 0; // make sure the func
    will end
227             assert label != null && label.length
    () != 0; // make sure label is not empty or null
228 
229             this.label = label;
230             this.from = from;
231             this.to = to;
232             this.step = step;
233         }
234 
235         /**
236          * Prepends multiple nodes to a shared
    doubly-linked list. For each
237          * integer in the range from...to (inclusive
    ), make a string
238          * concatenating label with the integer, and
     prepend a new node
239          * containing that data (that's data, not
    key). For example,
240          * countDown("A",8,6,1) means prepend three
    nodes with the data
241          * "A8", "A7", and "A6" respectively.
    countDown("X",10,2,3) will
242          * also prepend three nodes with "X10", "X7
    ", and "X4".
243          *
244          * This method should conditionally yield
    after each node is inserted.
245          * Print the list at the very end.
246          *
247          * Preconditions: from>=to and step>0
```

```
248          *
249          * @param label string that node data should
     start with
250          * @param from integer to start at
251          * @param to integer to end at
252          * @param step subtract this from the
    current integer to get to the next integer
253          */
254         public void countDown(String label, int from
    , int to, int step) {
255             for (int i = from ; i >= to ; i -= step
    ){
256                 String nodeLabel = label + i;
257                 testList.prepend(nodeLabel);
258                 KThread.yieldIfOughtTo();
259             }
260         }
261 
262         public static void reset(){
263             testList = new DLList();
264         }
265 
266         @Override
267         public void run() {
268             countDown(label, from, to, step);
269             KThread.yield();
270         }
271     }
272 
273 }
```