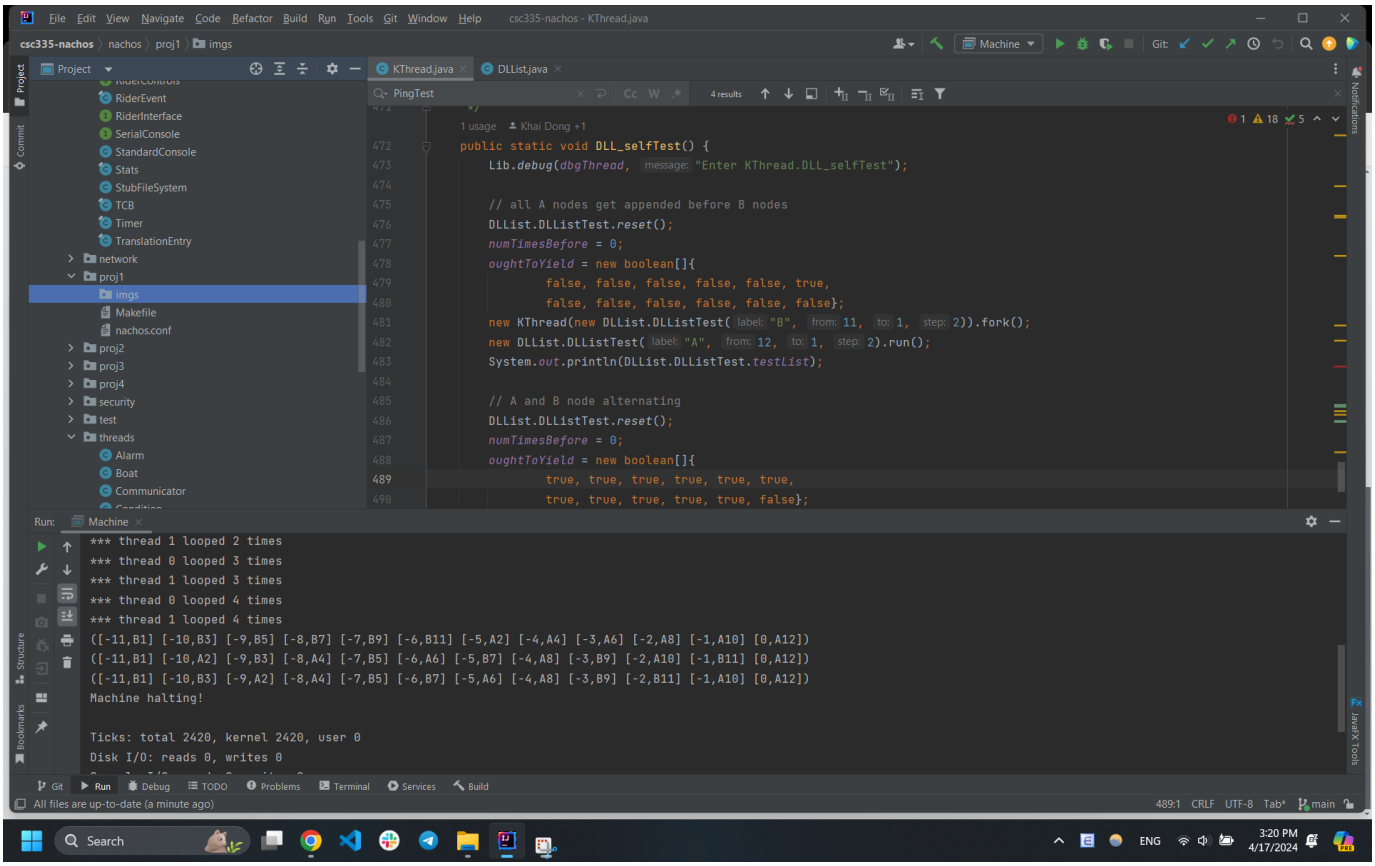
Project 1

Name: Khai Dong

Part 1

The source code for the interleaving is in `KThread.DLL_selfTest`.



Part 2.1 (Invalid Data)

This interleaving is in `KThread.DLL_selfTest2`. Initially, `testList` contains 2 nodes `(1, 1)` and `(6, 6)`.

Thread 1: testList.insert(3, 3)	Thread 2: testList.insert(4, 4)

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

```
    while(curElem != null
            && curElem.key <= sortKey)
        curElem = curElem.next;
```

```
    ++ this.size; // increment size counter
    DLLElement newElem =
            new DLLElement(item, sortKey);
    if(this.first == null){
        this.first = this.last = newElem;
        return;
    }
    DLLElement curElem = this.first;
    while(curElem != null
            && curElem.key <= sortKey)
        curElem = curElem.next;

    if(curElem == null){
        this.last.next = newElem;
        newElem.prev = this.last;
        this.last = newElem;
    } else {
        newElem.prev = curElem.prev;
        newElem.next = curElem;
        curElem.prev = newElem;
        if (newElem.prev == null)
            this.first = newElem;
        else newElem.prev.next = newElem;
    }
```

```
    if(curElem == null){
        this.last.next = newElem;
        newElem.prev = this.last;
        this.last = newElem;
    } else {
        newElem.prev = curElem.prev;
        newElem.next = curElem;
        curElem.prev = newElem;
        if (newElem.prev == null)
            this.first = newElem;
        else newElem.prev.next = newElem;
    }
```

In Thread 1, after the while loop, `curElem` will be the node `(6, 6)`. Then, a context switch happens, Threads 2 starts and finishes. Here, since Thread 1 has made no change to `testList`, so after Thread 2 (inserting `(4, 4)` into the list), the list will become

```
(1, 1), (4, 4), (6, 6)
```

Context switching back to Thread 1, since `curElem` is still the node `(6, 6)`, `(3, 3)` is inserted just before `(6, 6)`, yielding the final list:

```
(1, 1), (4, 4), (3, 3), (6, 6)
```

This is invalid data since the list supposed to be sorted by sortKey.

Part 2.1 (Null Pointer Exception)

This interleaving is in `KThread.DLL_selfTest3`. Initially, `testList` contains a single node `(1, 1)`.

Thread 1: testList.prepend(0)	Thread 2: testList.removeHead()

`int key = 0;`
`if (this.first != null){`
 	
`    key = this.first.key - 1;`
`}`
`this.insert(item, key);`
	`if(this.first == null) {`
	`    return null;`
	`}`
	`Object returnData = this.first.data;`
	`-- this.size; // decrement the size`
	`this.first = this.first.next; // update new head`
	`if(this.first == null) { // if pop the last element`
	`    this.last = null;`
	`} else {`
	`    this.first.prev = null;`
	`}`
	`return returnData;`

Since `testList` initially is not empty, Thread 1 will do the if branch. We do a context switch just after Thread 1 done evaluating `this.first != null`. Then, we do the entirety of `testList.removeHead()`. Since the Thread 1 has not made any change to `testList`, `testList` will end up empty with `testList.first` become `null`. Then, when Thread 1 tries to access `this.first.key` (equivalent to `null.key`), it will result in a `NullPointerException`.

```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

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

    /**
     * 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) {
        int key = 0;
        if (this.first != null){
            KThread.yieldIfShould(1);
            key = this.first.key - 1;
        }
        this.insert(item, key);
    }
```

```
35
36      /**
37       * Removes the head of the list and returns the
    data item stored in
38       * it.  Returns null if no nodes exist.
39       *
40       * @return the data stored at the head of the
    list or null if list empty
41       */
42      public Object removeHead() {
43          if(this.first == null) {
44              return null;
45          }
46          Object returnData = this.first.data;
47          -- this.size; // decrement the size
48          this.first = this.first.next; // update new
    head
49          if(this.first == null) { // if pop the last
    element
50              this.last = null;
51          } else {
52              this.first.prev = null; // update
    previous of the current head
53          }
54          return returnData;
55      }
56
57      /**
58       * Tests whether the list is empty.
59       *
60       * @return true iff the list is empty.
61       */
62      public boolean isEmpty() {
63          return this.size == 0;
64      }
65
66      /**
67       * returns number of items in list
68       * @return
69       */
70      public int size(){
```

```
 71          return this.size;
 72      }
 73
 74
 75      /**
 76       * Inserts item into the list in sorted order
    according to sortKey.
 77       */
 78      public void insert(Object item, Integer sortKey
    ) {
 79          ++ this.size; // increment size counter
 80          DLLElement newElem = new DLLElement(item,
    sortKey);
 81          if(this.first == null){
 82              this.first = this.last = newElem;
 83              return;
 84          }
 85          DLLElement curElem = this.first;
 86          // traverse the list to find the correct
    position to put item
 87          while(curElem != null && curElem.key <=
    sortKey)
 88              curElem = curElem.next;
 89
 90          KThread.yieldIfShould(0);
 91          if(curElem == null){
 92              this.last.next = newElem;
 93              newElem.prev = this.last;
 94              this.last = newElem;
 95          } else {
 96              newElem.prev = curElem.prev;
 97              newElem.next = curElem;
 98              curElem.prev = newElem;
 99              if (newElem.prev == null) this.first =
    newElem;
100              else newElem.prev.next = newElem;
101          }
102      }
103
104
105      /**
```

```
106      * returns list as a printable string. A single
    space should separate each list item,
107      * and the entire list should be enclosed in
    parentheses. Empty list should return "()"
108      * @return list elements in order
109      */
110     public String toString() {
111         StringBuilder builder = new StringBuilder();
112         builder.append('(');
113         DLLElement curNode = this.first;
114         while(curNode != null){
115             builder.append(curNode.toString());
116             if(curNode != this.last) builder.append(
    ' ');
117             curNode = curNode.next;
118         }
119         builder.append(')');
120         return builder.toString();
121     }
122 
123     /**
124      * returns list as a printable string, from the
    last node to the first.
125      * String should be formatted just like in
    toString.
126      * @return list elements in backwards order
127      */
128     public String reverseToString(){
129         StringBuilder builder = new StringBuilder();
130         builder.append('(');
131         DLLElement curNode = this.last;
132         while(curNode != null){
133             builder.append(curNode.toString());
134             if(curNode != this.first) builder.append
    (' ');
135             curNode = curNode.prev;
136         }
137         builder.append(')');
138         return builder.toString();
139     }
140 
```

```
141     /**
142      *  inner class for the node
143      */
144     private class DLLElement
145     {
146         private DLLElement next;
147         private DLLElement prev;
148         private int key;
149         private Object data;
150 
151         /**
152          * Node constructor
153          * @param item data item to store
154          * @param sortKey unique integer ID
155          */
156         public DLLElement(Object item, int sortKey)
157         {
158             key = sortKey;
159             data = item;
160             next = null;
161             prev = null;
162         }
163 
164         /**
165          * returns node contents as a printable
    string
166          * @return string of form [<key>,<data>]
    such as [3,"ham"]
167          */
168         public String toString(){
169             return "[" + key + "," + data + "]";
170         }
171     }
172 
173 
174     public static class DLListTest implements
    Runnable {
175 
176         // shared doubly linked list
177         public static DLList testList = new DLList
    ();
```

```
178
179         private final String label;
180         private final int from, to, step;
181
182         DLListTest(String label, int from, int to,
    int step){
183             assert from >= to;
184             assert step > 0; // make sure the func
    will end
185             assert label != null && label.length
    () != 0; // make sure label is not empty or null
186
187             this.label = label;
188             this.from = from;
189             this.to = to;
190             this.step = step;
191         }
192
193         /**
194          * Prepends multiple nodes to a shared
    doubly-linked list. For each
195          * integer in the range from...to (inclusive
    ), make a string
196          * concatenating label with the integer, and
     prepend a new node
197          * containing that data (that's data, not
    key). For example,
198          * countDown("A",8,6,1) means prepend three
    nodes with the data
199          * "A8", "A7", and "A6" respectively.
    countDown("X",10,2,3) will
200          * also prepend three nodes with "X10", "X7
    ", and "X4".
201          *
202          * This method should conditionally yield
    after each node is inserted.
203          * Print the list at the very end.
204          *
205          * Preconditions: from>=to and step>0
206          *
207          * @param label string that node data should
```

```
207  start with
208           * @param from integer to start at
209           * @param to integer to end at
210           * @param step subtract this from the
    current integer to get to the next integer
211           */
212          public void countDown(String label, int from
    , int to, int step) {
213              for (int i = from ; i >= to ; i -= step
    ){
214                  String nodeLabel = label + i;
215                  testList.prepend(nodeLabel);
216                  KThread.yieldIfOughtTo();
217              }
218          }
219
220          public static void reset(){
221              testList = new DLList();
222          }
223
224          @Override
225          public void run() {
226              countDown(label, from, to, step);
227              KThread.yield();
228          }
229      }
230
231 }
```

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

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

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

```
109  purposes only.
110      *
111      * @return  the full name given to this thread.
112      */
113     public String toString() {
114     return (name + " (#" + id + ")");
115     }
116
117     /**
118      * Deterministically and consistently compare
    this thread to another
119      * thread.
120      */
121     public int compareTo(Object o) {
122     KThread thread = (KThread) o;
123
124     if (id < thread.id)
125         return -1;
126     else if (id > thread.id)
127         return 1;
128     else
129         return 0;
130     }
131
132     /**
133      * Causes this thread to begin execution. The
    result is that two threads
134      * are running concurrently: the current thread
     (which returns from the
135      * call to the <tt>fork</tt> method) and the
    other thread (which executes
136      * its target's <tt>run</tt> method).
137      */
138     public void fork() {
139     Lib.assertTrue(status == statusNew);
140     Lib.assertTrue(target != null);
141
142     Lib.debug(dbgThread,
143           "Forking thread: " + toString() + "
    Runnable: " + target);
144
```

```
145     boolean intStatus = Machine.interrupt().disable
    ();
146
147     tcb.start(new Runnable() {
148         public void run() {
149             runThread();
150         }
151         });
152
153     ready();
154
155     Machine.interrupt().restore(intStatus);
156     }
157
158     private void runThread() {
159     begin();
160     target.run();
161     finish();
162     }
163
164     private void begin() {
165     Lib.debug(dbgThread, "Beginning thread: " +
    toString());
166
167     Lib.assertTrue(this == currentThread);
168
169     restoreState();
170
171     Machine.interrupt().enable();
172     }
173
174     /**
175      * Finish the current thread and schedule it to
    be destroyed when it is
176      * safe to do so. This method is automatically
    called when a thread's
177      * <tt>run</tt> method returns, but it may also
    be called directly.
178      *
179      * The current thread cannot be immediately
    destroyed because its stack and
```

```
180      * other execution state are still in use.
    Instead, this thread will be
181      * destroyed automatically by the next thread to
     run, when it is safe to
182      * delete this thread.
183      */
184     public static void finish() {
185     Lib.debug(dbgThread, "Finishing thread: " +
    currentThread.toString());
186
187     Machine.interrupt().disable();
188
189     Machine.autoGrader().finishingCurrentThread();
190
191     Lib.assertTrue(toBeDestroyed == null);
192     toBeDestroyed = currentThread;
193
194
195     currentThread.status = statusFinished;
196
197     sleep();
198     }
199
200     /**
201      * Relinquish the CPU if any other thread is
    ready to run. If so, put the
202      * current thread on the ready queue, so that it
     will eventually be
203      * rescheduled.
204      *
205      * <p>
206      * Returns immediately if no other thread is
    ready to run. Otherwise
207      * returns when the current thread is chosen to
    run again by
208      * <tt>readyQueue.nextThread()</tt>.
209      *
210      * <p>
211      * Interrupts are disabled, so that the current
    thread can atomically add
212      * itself to the ready queue and switch to the
```

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

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

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

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

```
343     Lib.debug(dbgThread, "Switching from: " +
    currentThread.toString()
344           + " to: " + toString());
345
346     currentThread = this;
347
348     tcb.contextSwitch();
349
350     currentThread.restoreState();
351     }
352
353     /**
354      * Prepare this thread to be run. Set <tt>status
    </tt> to
355      * <tt>statusRunning</tt> and check <tt>
    toBeDestroyed</tt>.
356      */
357     protected void restoreState() {
358     Lib.debug(dbgThread, "Running thread: " +
    currentThread.toString());
359
360     Lib.assertTrue(Machine.interrupt().disabled());
361     Lib.assertTrue(this == currentThread);
362     Lib.assertTrue(tcb == TCB.currentTCB());
363
364     Machine.autoGrader().runningThread(this);
365
366     status = statusRunning;
367
368     if (toBeDestroyed != null) {
369         toBeDestroyed.tcb.destroy();
370         toBeDestroyed.tcb = null;
371         toBeDestroyed = null;
372     }
373     }
374
375     /**
376      * Prepare this thread to give up the processor
    . Kernel threads do not
377      * need to do anything here.
378      */
```

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

```
416      */
417     public Object schedulingState = null;
418 
419     private static final int statusNew = 0;
420     private static final int statusReady = 1;
421     private static final int statusRunning = 2;
422     private static final int statusBlocked = 3;
423     private static final int statusFinished = 4;
424 
425     /**
426      * The status of this thread. A thread can
    either be new (not yet forked),
427      * ready (on the ready queue but not running),
    running, or blocked (not
428      * on the ready queue and not running).
429      */
430     private int status = statusNew;
431     private String name = "(unnamed thread)";
432     private Runnable target;
433     private TCB tcb;
434 
435     /**
436      * Unique identifer for this thread. Used to
    deterministically compare
437      * threads.
438      */
439     private int id = numCreated++;
440     /** Number of times the KThread constructor was
    called. */
441     private static int numCreated = 0;
442 
443     private static ThreadQueue readyQueue = null;
444     private static KThread currentThread = null;
445     private static KThread toBeDestroyed = null;
446     private static KThread idleThread = null;
447 
448 
449 
    ////////////////////////////////////////////////////
    ////////////////////////////////////////////////////
    //////
```

```
450 
    ////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////
    //////
451 
    ////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////
    //////
452     // new stuff starts here
453 
454     public static boolean[] oughtToYield = null;
455     public static int numTimesBefore = 0;
456     public static void yieldIfOughtTo() {
457         if (numTimesBefore < oughtToYield.length &&
    oughtToYield[numTimesBefore]){
458             numTimesBefore++;
459             KThread.yield();
460         } else {
461             numTimesBefore++;
462         }
463     }
464 
465     public static boolean[][] yieldData = null;
466     public static int[] yieldCount = null;
467 
468     /**
469      * Given this unique location, yield the
470      * current thread if it ought to. It knows
471      * to do this if yieldData[i][loc] is true,
    where
472      * i is the number of times that this function
473      * has already been called from this location.
474      *
475      * @param loc unique location. Every call to
476      * yieldIfShould that you
477      * place in your DLList code should
478      * have a different loc number.
479      */
480     public static void yieldIfShould(int loc) {
481         if (yieldCount == null || yieldData == null)
482             return;
```

```
483         if (0 <= loc && loc < yieldCount.length &&
    loc < yieldData.length){
484             int curCount = yieldCount[loc];
485             if (yieldData[loc] != null && 0 <=
    curCount && curCount <= yieldData[loc].length){
486                 yieldCount[loc] += 1;
487                 if (yieldData[loc][curCount]){
488                     KThread.yield();
489                 }
490 
491             }
492         }
493     }
494 
495 
496     /**
497      * Tests the shared DLList by having two threads
     running countdown.
498      * One thread will insert even-numbered data
    from "A12" to "A2".
499      * The other thread will insert odd-numbered
    data from "B11" to "B1".
500      * Don't forget to initialize the oughtToYield
    array before forking.
501      *
502      */
503     public static void DLL_selfTest() {
504         Lib.debug(dbgThread, "Enter KThread.
    DLL_selfTest");
505 
506         // all A nodes get appended before B nodes
507         DLList.DLListTest.reset();
508         numTimesBefore = 0;
509         oughtToYield = new boolean[]{
510                 false, false, false, false, false,
    true,
511                 false, false, false, false, false,
    false};
512         new KThread(new DLList.DLListTest("B", 11, 1
    , 2)).fork();
513         new DLList.DLListTest("A", 12, 1, 2).run();
```

```
514         System.out.println(DLList.DLListTest.
    testList);
515 
516         // A and B node alternating
517         DLList.DLListTest.reset();
518         numTimesBefore = 0;
519         oughtToYield = new boolean[]{
520                 true, true, true, true, true, true,
521                 true, true, true, true, true, false
    };
522         new KThread(new DLList.DLListTest("B", 11, 1
    , 2)).fork();
523         new DLList.DLListTest("A", 12, 1, 2).run();
524         System.out.println(DLList.DLListTest.
    testList);
525 
526         // 2 A nodes then 2 B nodes
527         DLList.DLListTest.reset();
528         numTimesBefore = 0;
529         oughtToYield = new boolean[]{
530                 false, true, false, true, false,
    true,
531                 false, true, false, true, false,
    false};
532         new KThread(new DLList.DLListTest("B", 11, 1
    , 2)).fork();
533         new DLList.DLListTest("A", 12, 1, 2).run();
534         System.out.println(DLList.DLListTest.
    testList);
535 
536     }
537 
538     public static void DLL_selfTest2(){
539         Lib.debug(dbgThread, "Enter KThread.
    DLL_selfTest2");
540         DLList testList = new DLList();
541 
542         testList.insert(1, 1);
543         testList.insert(6, 6);
544 
545         yieldCount = new int[]{0};
```

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

```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);
```

```
36
37      alarm  = new Alarm();
38
39      Machine.interrupt().enable();
40      }
41
42      /**
43       * Test this kernel. Test the <tt>KThread</tt>, <
   tt>Semaphore</tt>,
44       * <tt>SynchList</tt>, and <tt>ElevatorBank</tt>
   classes. Note that the
45       * autograder never calls this method, so it is
   safe to put additional
46       * tests here.
47       */
48      public void selfTest() {
49      KThread.selfTest();
50
51      KThread.DLL_selfTest();
52      KThread.DLL_selfTest2();
53      KThread.DLL_selfTest3();
54
55      Semaphore.selfTest();
56      SynchList.selfTest();
57      if (Machine.bank() != null) {
58          ElevatorBank.selfTest();
59      }
60      }
61
62      /**
63       * A threaded kernel does not run user programs,
   so this method does
64       * nothing.
65       */
66      public void run() {
67      }
68
69      /**
70       * Terminate this kernel. Never returns.
71       */
72      public void terminate() {
```

```
73      Machine.halt();
74      }
75
76      /** Globally accessible reference to the
    scheduler. */
77      public static Scheduler scheduler = null;
78      /** Globally accessible reference to the alarm
    . */
79      public static Alarm alarm = null;
80      /** Globally accessible reference to the file
    system. */
81      public static FileSystem fileSystem = null;
82
83      // dummy variables to make javac smarter
84      private static RoundRobinScheduler dummy1 = null
    ;
85      private static PriorityScheduler dummy2 = null;
86      private static LotteryScheduler dummy3 = null;
87      private static Condition2 dummy4 = null;
88      private static Communicator dummy5 = null;
89      private static Rider dummy6 = null;
90      private static ElevatorController dummy7 = null;
91  }
92
```