

Write-up

Part 1-Locking

The first concurrency issue I had from Project 1 was an issue where the first node added to the list would end up being written over due to “unfortunate” time slicing.

Before:

```
Broken Interleaving: Overwrites first node

Results in: B1, A2, B3, ..., A10, B11, A12
([-10,B1] [-9,A2] [-8,B3] [-7,A4] [-6,B5] [-5,A6] [-4,B7] [-3,A8] [-2,B9] [-1,A10] [0,B11])
```

The solution was fairly simple: add a lock to the beginning and end of the prepend() function, forcing only one thread to be able to write at one time.

In doing this, I did move some things around, making a private method to run the prepending, and having a public shell method that handles the locking and waking of removeAtHead.

```
Broken Interleaving: Overwrites first node

Results in: B1, A2, B3, ..., A10, B11, 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])
```

The second concurrency method causes a fatal error by trying to remove a node when there is none. This happens because of some more unfortunate timing, where the size of the DLList is increased, however the first and last pointers have not been set, and are null.

```
Broken Interleaving: Results in Null Pointer

java.lang.NullPointerException: Cannot read field "data" because "this.first" is null
    at nachos.threads.DLList.removeHead(DLList.java:75)
    at nachos.threads.KThread.SelfTestBad_2(KThread.java:697)
    at nachos.threads.KThread.selfTest(KThread.java:574)
    at nachos.threads.ThreadedKernel.selfTest(ThreadedKernel.java:49)
    at nachos.ag.AutoGrader.run(AutoGrader.java:152)
    at nachos.ag.AutoGrader.start(AutoGrader.java:50)
    at nachos.machine.Machine$1.run(Machine.java:63)
    at nachos.machine.TCB.threadroot(TCB.java:235)
    at nachos.machine.TCB.start(TCB.java:118)
    at nachos.machine.Machine.main(Machine.java:62)
```

The solution to this was the same, however my implementation of insert() does have a return within an if statement halfway through, and so I needed to make sure that the lock is also released there as well.

```
Broken Interleaving: Results in Null Pointer
()
```

Part 2-Bounded Buffer

The bounded buffer simply uses a `char[]` array to store the data put onto the buffer, and two pointers, one pointing at the next in slot, the other pointing at the next out slot.

There is another variable that keeps track of the size of the buffer. This helps prevent issues where the `nextIN` and `nextOUT` pointers might overtake each other. Because we can't add or subtract unused slots in the array as it is currently set up, `nextIN` and `nextOUT` wrap around the array as they are used. If we are to try and write to the buffer while the `size == maxsize`, then the `sleep` function will be called, making the thread wait for an open slot to write to. Alternatively if we were to read from the buffer when the `size == 0`, then we the thread sleep until a filled slot wakes it up.

Here are the tests I implemented to test both overflow and underflow.

```
Bounded Buffer Tests:
```

```
Check overflow, buffer size = 2.  
writing: a  
writing: b  
writing: c  
Overflow - removing char:a  
writing: d  
Overflow - removing char:b  
writing: e  
Overflow - removing char:c  
Final buffer state:  
ed
```

```
Check underflow, buffer size = 2.  
Reading char  
Underflow - writing char: a  
read char: a  
Reading char  
Underflow - writing char: b  
read char: b  
Reading char  
Underflow - writing char: c  
read char: c  
Final buffer state:  
cb
```

As the code is set up now, there wouldn't be any "errors" caused by overflow or underflow other than incorrect outputs caused by the nextIN pointer overtaking the nextOUT pointer, or returning null from the buffer instead of a char.

I call the tests from ThreadedKernel

Part 3- condition2, electric boogaloo

I really just went line-by-line through the pseudo code, combining the different methods used in lock(), synchList(), and given by KThread to do this.

Proof Condition2 works:

```
Bounded Buffer Tests:

Check overflow, buffer size = 2.
writing: a
writing: b
writing: c
Overflow - removing char:a
writing: d
Overflow - removing char:b
writing: e
Overflow - removing char:c
Final buffer state:
ed

Check underflow, buffer size = 2.
Reading char
Underflow - writing char: a
read char: a
Reading char
Underflow - writing char: b
read char: b
Reading char
Underflow - writing char: c
read char: c
Final buffer state:
cb
Machine halting!
```

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

```
Results in: B1, A2, B3, ..., A10, B11, 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])
```

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

```
Check if removeAtHead sleeps until a node has been added.  
Node removed first, then node added:      ()  
  
Broken Interleaving: Overwrites first node  
  
Results in: B1, A2, B3, ..., A10, B11, 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])  
  
Broken Interleaving: Results in Null Pointer  
()
```

The Code

DLLList:

```
Java

/*
 * Made by Jack Lynch, with base provided by Chris Fernandez
 * CSC 335 - Operating Systems, 9/11/2025
 */

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

public class DLLList
{
    private DLLElement first; // pointer to first node
    private DLLElement last; // pointer to last node
    private int size; // number of nodes in list
    private Lock DLLock = new Lock();
    private Condition2 nodeToRemove= new Condition2(DLLock);

    /**
     * Creates an empty sorted doubly-linked list.
     */
    public DLLList() {
        this.first = null;
        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) {  
    // System.out.println(item + " has initialized");  
    DLLock.acquire();  
    // System.out.println(item + " has entered lock");  
    privatePrepend(item);  
    nodeToRemove.wake();  
    DLLock.release();  
    KThread.yieldIfShould(2);  
    // System.out.println(item +" is yielding if should");  
    // System.out.println("kept lock? : "  
+DLLock.isHeldByCurrentThread());  
}  
  
  
private void privatePrepend(Object item) {  
    // If empty, start the list with the key = 0  
    if (this.isEmpty()) {  
        KThread.yieldIfShould(0);  
        DLLElement newNode = new DLLElement(item, 0);  
        first = newNode;  
        last = newNode;  
    }  
    // not empty, prepend with key = first.key - 1  
    else {  
  
        DLLElement newNode = new DLLElement(item, first.key-1);  
        newNode.next = first;  
        first.prev = newNode;  
        first = newNode;  
    }  
    size +=1;  
}
```

```
}

/**
 * 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() {
    this.getDLLock();
    if (this.isEmpty()) {
        nodeToRemove.sleep();
    };
    Object toReturn = first.data;
    first = first.next;
    if (first == null) last = null;
    else first.prev = null;
    size -= 1;
    DLLock.release();
    return toReturn;
}

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

```
        else return false;
    }

    /**
     * returns number of items in list
     * @return
     */
    public int size(){
        return size;
    }

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

        this.getDLLock();

        // If list is empty, set first and last to newnode and finish
        if (this.isEmpty()) {
            size += 1;
            last = newNode;
            KThread.yieldIfShould(3);

            first = newNode;
            nodeToRemove.wake();
            DLLock.release();
            return;
        }
    }
}
```

```
DLLElement runner = first;
while (!(runner == null) && runner.key < newNode.key) {
    runner = runner.next;
}
// When runner == null, the included sortKey is greater than
all other sortkeys, and should be last.
if (runner ==null) {
    newNode.prev = last;
    last.next = newNode;
    last = newNode;
}
// if runner is first, the included sortkey is less than all
other sortkeys,should be set to first.
else if (runner.equals(first)){
    newNode.next = first;
    first.prev = newNode;
    first = newNode;
}
// else runner.key >= newnode.key
else {
    newNode.next = runner;
    newNode.prev = runner.prev;
    if (newNode.prev != null) newNode.prev.next = newNode;
    newNode.next.prev = newNode;
}
size += 1;
nodeToRemove.wake();
DLLock.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  
 */  
  
public String toString() {  
    DLLElement runner = first;  
    String toReturn = "(";  
  
    this.getDLLock();  
  
    while (!(runner == null)) {  
        if (!toReturn.equals("(")) toReturn += " ";  
        toReturn += runner.toString();  
        runner = runner.next;  
    }  
    DLLock.release();  
    return toReturn + ")";  
}  
  
/**  
 * 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  
 */  
  
public String reverseToString(){  
    DLLElement runner = last;  
    String toReturn = "(";
```

```
DLLock.acquire();

    while (!(runner == null)) {
        if (!toReturn.equals("(")) toReturn += " ";
        toReturn += runner.toString();
        runner = runner.prev;
    }
    DLLock.release();
    return toReturn + ")";
}

/**
 * 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 + "]";
}

private void getDLLock() {
    DLLock.acquire();
}
}
```

KThread:

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  
 * rescheduled.  
 *  
 * <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());  
    //System.out.println("Yielding thread: " +  
currentThread.toString());  
  
    Lib.assertTrue(currentThread.status == statusRunning);  
  
    boolean intStatus = Machine.interrupt().disable();
```

```
currentThread.ready();

runNextThread();

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

<**
 * Relinquish the CPU, because the current thread has either
finished or it
 *
 * <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() {  
    //System.out.print("Thread Queue: "); readyQueue.print();  
    System.out.println("");  
  
    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");
        KThread.yield();
    }
}

private int which;
}

private static class DLLListTest implements Runnable {
    public DLLListTest(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  
*/  
private void countDown(String label, int from, int to, int  
step) {  
    for (int i = from; i >= to; i-=step){  
        myList.prepend(label+i);  
    }  
}  
  
public void run() {  
    // Countdown working output  
    this.countDown(this.label, this.from, this.to,  
this.step);  
}  
  
private static DLLList myList = new DLLList();  
private String label;  
private int from, to;  
private int step;  
}
```

```
private static class DLLListBadTest implements Runnable {  
    public DLLListBadTest(int mode) {  
        this.mode = mode;  
        if (mode == 0) {  
            this.label = "A";  
            this.from = 12;  
            this.to = 2;  
            this.step = 2;  
        } else if (mode == 1) {  
            this.label = "B";  
            this.from = 11;  
            this.to = 1;  
            this.step = 2;  
        }  
    }  
  
    /**  
     * 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
        */
private void countDown(String label, int from, int to, int
step) {
    for (int i = from; i >= to; i-=step){
        myList.prepend(label+i);
    }
}

public void run() {
    if (mode == 0 || mode == 1){
        this.countDown(this.label, this.from, this.to,
this.step);
    }
    else if (mode == 2){
        myList.insert("A", 3);
    } else {
        myList.removeHead();
    }
}

private static DLList myList = new DLList();
private String label;
private int from, to;

```

```
    private int step;
    private int mode;
}

private static class DLLockTest implements Runnable {
    public DLLockTest(int mode) {
        this.mode = mode;
    }

    public void run() {
        if (mode == 0){
            myList.removeHead();
        }
        else if (mode == 1){
            myList.prepend("A");
        }
        KThread.yield();
    }

    private static DLList myList = new DLList();
    private int mode;
}

public static void yieldIfOughtTo() {
    numTimesBefore++;
    if (oughtToYield[(numTimesBefore-1) % oughtToYield.length]) {
        KThread.yield();
    }
}
```

```
/**  
 * Given this unique location, yield the  
 * current thread if it ought to. It knows  
 * to do this if yieldData[i][loc] 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 DLLList code should  
 * have a different loc number.  
 */  
  
public static void yieldIfShould(int loc) {  
    //System.out.println("yield if should");  
    yieldCount[loc]++;  
    boolean shouldYield = yieldData[loc][(yieldCount[loc]-1) %  
yieldData[loc].length];  
    // System.out.println(shouldYield);  
    if (shouldYield) {  
        //System.out.println(shouldYield);  
        KThread.yield();  
    }  
}  
  
/**  
 * 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();

// SelfTest_1();
SelfTest_2();
// SelfTest_3();

SelfTest_4();

SelfTestBad_1();
SelfTestBad_2();

}

/**
 * Results in: B1, B3, B5 ..., A8, A10, A12
 */
private static void SelfTest_1(){
    System.out.println("\n\nResults in: B1, B3, B5 ..., A8, A10,
A12");
    DLLListTest myList = new DLLList();
    yieldData = new boolean[][] {{false},{false}, {false},
{false}, {false}};
    KThread.yieldCount = new int[] {0, 0, 0, 0, 0};

    KThread forkedThread = new KThread(new DLLListTest("A", 12, 2,
2)).setName("forked thread");
    forkedThread.fork();
    KThread.yield();
    for (int i = 11; i >= 1; i-=2){
        myList.prepend("B"+i);
    }
}
```

```
        }

        forkedThread.join();
        System.out.println(DLLListTest myList);

    }

/***
 * Results in: B1, A2, B3, ..., A10, B11, A12
 */

private static void SelfTest_2(){
    boolean interruptStatus = Machine.interrupt().disable();

    System.out.println("\n\nResults in: B1, A2, B3, ..., A10,
B11, A12");
    DLLListTest myList = new DLLList();
    yieldData = new boolean[][] {{false},{false}, {true},
{false}, {false}};
    KThread.yieldCount = new int[] {0, 0, 0, 0, 0};

    KThread forkedThread = new KThread(new DLLListTest("A", 12, 2,
2)).setName("forked thread");
    forkedThread.fork();
    KThread.yield();
    for (int i = 11; i >= 1; i-=2){
        DLLListTest myList.prepend("B"+i);
        // yieldIfShould(2);
    }
    forkedThread.join();
    System.out.println(DLLListTest myList);
    Machine.interrupt().restore(interruptStatus);

}
```

```
/**  
 * Results in: B1, B3, A2, A4, ..., B11, A10, A12  
 */  
  
private static void SelfTest_3(){  
    System.out.println("\n\nResults in: B1, B3, A2, A4, ..., B11,  
A10, A12");  
    DLLListTest myList = new DLLList();  
    KThread.yieldData = new boolean[][] {{false}, {false}, {false},  
true}, {false}, {false}};  
    KThread.yieldCount = new int[] {0, 0, 0, 0, 0};  
  
    KThread forkedThread = new KThread(new DLLListTest("A", 12, 2,  
2)).setName("forked thread");  
    forkedThread.fork();  
    KThread.yield();  
    for (int i = 11; i >= 1; i-=2){  
        DLLListTest myList.prepend("B"+i);  
    }  
    forkedThread.join();  
    System.out.println(DLLListTest myList);  
}  
  
/**  
 * tests to see if removeAtHead waits for a node to be removed  
 * runs twice, one attempts to remove at head and then append,  
the other appends and then removes.  
 * both should output an empty list when done.  
 */  
  
private static void SelfTest_4(){  
    Lib.debug(dbgThread, "Enter KThread.SelfTest_4");  
    DLLListTest myList = new DLLList();
```

```

        yieldData = new boolean[][] {{false},{false}, {true},
{false}, {false}};

        System.out.println("\n\nCheck if removeAtHead sleeps until a
node has been added.");

        // removes then adds

        DLLockTest myList = new DLList();
        KThread forkedThread = new KThread(new
DLLockTest(1)).setName("forked thread");
        forkedThread.fork();

        DLLockTest myList.removeHead();
        KThread.yield(); // unnecessary, but kept just in case.
        forkedThread.join();
        System.out.println("Node removed first, then node added: \t"
+ myList.toString());

    }

/***
 * Broken Interleaving: Overwrites first node
 */
private static void SelfTestBad_1(){

    DLListBadTest myList = new DLList();
    System.out.println("\n\nBroken Interleaving: Overwrites first
node");
    System.out.println("\nResults in: B1, A2, B3, ..., A10, B11,
A12");
}

```

```
    yieldData = new boolean[][] {{true},{false}, {true}, {false}, {false}};
```

```
    KThread.yieldCount = new int[] {0, 0, 0, 0, 0};
```

```
    KThread forkedThread = new KThread(new DLLListBadTest(0)).setName("forked thread");
```

```
    forkedThread.fork();
```

```
    KThread.yield();
```

```
    for (int i = 11; i >= 1; i-=2){
```

```
        DLLListBadTest myList.prepend("B"+i);
```

```
        // yieldIfShould(2);
```

```
    }
```

```
    forkedThread.join();
```

```
    System.out.println(DLLListBadTest myList);
```

```
}
```

```
/**
```

```
 * Broken Interleaving: Results in Null Pointer
```

```
*/
```

```
private static void SelfTestBad_2(){
```

```
    yieldCount = new int[] {0, 0, 0, 0, 0};
```

```
    DLLListBadTest myList = new DLLList();
```

```
    System.out.println("\n\nBroken Interleaving: Results in Null
```

```
Pointer");
```

```
    yieldData = new boolean[][] {{false}, {false}, {true},
```

```
{true}, {false}};
```

```
    KThread.yieldCount = new int[] {0, 0, 0, 0, 0};
```

```
    KThread forkedThread = new KThread(new DLLListBadTest(2)).setName("forked thread");
```

```
    forkedThread.fork();
```

```
    KThread.yield();
```

```
DLLListBadTest myList.removeHead();
```

```
        forkedThread.join();
        System.out.println(DLLListBadTest myList);
    }

    /**
     * testsBounded Buffer
     */
    public static void BBSelfTest(){
        Lib.debug(dbgThread, "Enter KThread.BBSelfTest");
        System.out.println("\n\nBounded Buffer Tests:");
        // BBTest_1();
        // BBTest_2();
    }

    private static void BBTest_1(){
        System.out.println("\nCheck overflow, buffer size = 2.");
        BoundedBuffer buffer = new BoundedBuffer(2);

        KThread forkedThread = new KThread(new BBTest(buffer,
0)).setName("forked thread");
        forkedThread.fork();

        System.out.println("writing: +'a'");
        buffer.write('a');
        System.out.println("writing: +'b'");
        buffer.write('b');
        System.out.println("writing: +'c'");
        buffer.write('c');
        System.out.println("writing: +'d'");
        buffer.write('d');
        System.out.println("writing: +'e'");
        buffer.write('e');
    }
}
```

```
        forkedThread.join();
        System.out.println("Final buffer state: ");buffer.print();
    }

private static void BBTest_2(){
    System.out.println("\nCheck underflow, buffer size = 2.");
    BoundedBuffer buffer = new BoundedBuffer(2);

    KThread forkedThread = new KThread(new BBTest(buffer,
1)).setName("forked thread");
    forkedThread.fork();

    System.out.println("Reading char");
    System.out.println("read char: "+buffer.read());

    System.out.println("Reading char");
    System.out.println("read char: "+buffer.read());

    System.out.println("Reading char");
    System.out.println("read char: "+buffer.read());
    forkedThread.join();
    System.out.println("Final buffer state: ");buffer.print();

}

private static class BBTest implements Runnable{
    BoundedBuffer buffer;
    int mode;
```

```
char[] alphabet =
{'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q',
'r','s','t','u','v','w','x','y','z'};

public BBTest(BoundedBuffer buffer, int mode) {
    this.buffer = buffer;
    this.mode = mode;
}

public void run() {
    if (mode == 0){
        for (int i = 0; i < 3; i++) {
            System.out.println("Overflow - removing char:" +
buffer.read());
            KThread.yield();
        }
    } else if (mode == 1){

        for (int i = 0; i < 3; i++) {
            System.out.println("Underflow - writing char:
"+alphabet[i]);
            buffer.write(alphabet[i]);
            KThread.yield();
        }
    }
}

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 identifier 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 int numTimesBefore = 0;

// B1, B3, A2, A4 ...
private static boolean[] oughtToYield = {false, false, false,
true};

private static int[] yieldCount = {0, 0, 0, 0, 0};

private static boolean[][] yieldData;
}
```

SynchList:

```
Java
package nachos.threads;

import java.util.LinkedList;
import nachos.machine.*;
import nachos.threads.*;

/**
 * A synchronized queue.
 */
public class SynchList {
    /**
     * Allocate a new synchronized queue.
     */
    public SynchList() {
        list = new LinkedList<Object>();
        lock = new Lock();
        listEmpty = new Condition(lock);
    }

    public boolean isEmpty(){
        return list.isEmpty();
    }

    /**
     * Add the specified object to the end of the queue. If another
     * thread is
     *      * waiting in <tt>removeFirst()</tt>, it is woken up.
     *
     *      * @param o the object to add. Must not be <tt>null</tt>.
     */
}
```

```
public void add(Object o) {
    Lib.assertTrue(o != null);

    lock.acquire();
    list.add(o);
    listEmpty.wake();
    lock.release();
}

/**
 * Remove an object from the front of the queue, blocking until
the queue
 * is non-empty if necessary.
 *
 * @return the element removed from the front of the queue.
 */
public Object removeFirst() {
    Object o;

    lock.acquire();
    while (list.isEmpty())
        listEmpty.sleep();
    o = list.removeFirst();
    lock.release();

    return o;
}

private static class PingTest implements Runnable {
    PingTest(SynchList ping, SynchList pong) {
        this.ping = ping;
        this.pong = pong;
    }
}
```

```
}

public void run() {
    for (int i=0; i<10; i++)
        pong.add(ping.removeFirst());
}

private SynchList ping;
private SynchList pong;
}

/***
 * Test that this module is working.
 */
public static void selfTest() {
    SynchList ping = new SynchList();
    SynchList pong = new SynchList();

    new KThread(new PingTest(ping, pong)).setName("ping").fork();

    for (int i=0; i<10; i++) {
        Integer o = new Integer(i);
        ping.add(o);
        Lib.assertTrue(pong.removeFirst() == o);
    }
}

private LinkedList<Object> list;
private Lock lock;
private Condition listEmpty;
}
```

Bounded Buffer:

Java

```
package nachos.threads;

public class BoundedBuffer {
    private Lock bufferLock = new Lock();
    private int maxsize;
    private int size;
    private char[] buffer;
    private int nextIN;
    private int nextOUT;
    private Condition2 overflow = new Condition2(bufferLock);
    private Condition2 underflow = new Condition2(bufferLock);

    // non-default constructor with a fixed size
    public BoundedBuffer(int maxsize){
        this.maxsize = maxsize;
        this.buffer = new char[maxsize];
        this.size = 0;
        this.nextIN = 0;
        this.nextOUT = 0;
    }
    // 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(){
        bufferLock.acquire();
        if (size == 0) {
            underflow.sleep();
        }
    }
```

```
char toReturn = buffer[nextOUT];
nextOUT = (nextOUT + 1) % maxsize;
size--;

if (size == maxsize -1) { overflow.wake();}
bufferLock.release();
return toReturn;
}

// Write the given character c into the buffer, blocking until
// enough space is available to satisfy the request.
public void write(char c){
    bufferLock.acquire();
    while (size == maxsize){
        overflow.sleep();
    }
    buffer[nextIN] = c;
    nextIN=(nextIN + 1) % maxsize;
    size++;
    if (size == 1){underflow.wake();}
    bufferLock.release();
}

// Prints the contents of the buffer; for debugging only
public void print(){
    bufferLock.acquire();
    System.out.println(this.buffer);
    bufferLock.release();
}

public int size(){
    return size;
}
}
```

Condition2:

```
Java
package nachos.threads;

import nachos.machine.*;

/**
 * An implementation of condition variables that disables
interrupt()s for
* synchronization.
*
* <p>
* You must implement this.
*
* @see nachos.threads.Condition
*/
public class Condition2 {
    /**
     * Allocate a new condition variable.
     *
     * @param conditionLock the lock associated with this
condition
     *
     * variable. The current thread must hold this
     * lock whenever it uses <tt>sleep()</tt>,
     * <tt>wake()</tt>, or <tt>wakeAll()</tt>.
    */
    public Condition2(Lock conditionLock) {
        this.conditionLock = conditionLock;
    }

    /**

```

```
* Atomically release the associated lock and go to sleep on this
condition
* variable until another thread wakes it using <tt>wake()</tt>.

The
* current thread must hold the associated lock. The thread will
* automatically reacquire the lock before <tt>sleep()</tt>
returns.

*/
public void sleep() {
    /*
     */
    Lib.assertTrue(conditionLock.isHeldByCurrentThread());
    // disable interrupts
    boolean interruptStatus = Machine.interrupt().disable();
    // release the lock
    conditionLock.release();

    // add the current thread to a queue
    KThread thread = KThread.currentThread();
    waitQueue.add(thread);

    // block the current thread
    KThread.sleep();

    // enable interrupts (when the thread gets woken up)
    Machine.interrupt().restore(interruptStatus);

    // reacquire the lock
    conditionLock.acquire();
}
```

```
/**  
 * Wake up at most one thread sleeping on this condition  
variable. The  
 * current thread must hold the associated lock.  
 */  
  
public void wake() {  
    Lib.assertTrue(conditionLock.isHeldByCurrentThread());  
    // disable interrupts  
    boolean interruptStatus = Machine.interrupt().disable();  
    // release the lock  
    conditionLock.release();  
  
    // Only wakes a thread if there is one on the queue  
    if (!waitForQueue.isEmpty()) {  
        // remove the next thread from the queue  
        KThread nextThread = (KThread) waitForQueue.removeFirst();  
        // wake the new thread  
        nextThread.ready();  
    }  
  
    // enable interrupts (when the thread gets woken up)  
    Machine.interrupt().restore(interruptStatus);  
    // reacquire the lock  
    conditionLock.acquire();  
}  
  
/**  
 * Wake up all threads sleeping on this condition variable. The  
current  
 * thread must hold the associated lock.  
 */
```

```
public void wakeAll() {
    Lib.assertTrue(conditionLock.isHeldByCurrentThread());

    // disable interrupts
    boolean interruptStatus = Machine.interrupt().disable();
    // release the lock
    conditionLock.release();

    // Only wakes a thread if there is one on the queue
    while (!waitForQueue.isEmpty()){
        // remove the next thread from the queue
        KThread nextThread = (KThread) waitForQueue.removeFirst();
        // wake the new thread
        nextThread.ready();
    }

    // enable interrupts (when the thread gets woken up)
    Machine.interrupt().restore(interruptStatus);
    // reacquire the lock
    conditionLock.acquire();
}

private Lock conditionLock;
private SynchList waitForQueue = new SynchList();
}
```

ThreadedKernel

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.BBSelfTest();
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;  
}
```