CSC 335: Project 1 – Break that Code
Due Monday, April 22, 2024 at start of class

Objectives:

- To practice running multiple threads
- To understand and articulate how shared data structures between concurrent threads can lead to incorrect results
- To begin using Nachos system calls like **fork** and **yield**
- To break completely working code…

We have studied how *interleavings* between threads can cause incorrect results in shared variables and data structures. You're going to demonstrate that in this project by implementing interleavings of your doubly-linked list that you built in the previous project. Those interleavings will cause your doubly-linked list to crash or to be left in an invalid state, even though it completely works in a single-threaded environment.

BEFORE YOU BEGIN: make a backup of your previous project since you'll be adding onto it. In case of catastrophe, you want a known state to get back to.

Part 1: Make your own multi-threaded program

First, we'll practice making a multi-threaded program in Nachos that does not cause errors yet. We are still working inside the Nachos kernel, doing various kinds of self-tests just like in the previous project. So context switching will only occur when we deliberately tell a thread to yield. But that leads to an implementation problem. Revisit slide 38, shown below, which did an interleaving between two threads that were both running `insertAtTail`:

T_0: insert 0	T_1: insert 1
`Node newNode = new Node(0);`	
`if (last == null) {` *[true]*	
`    first = newNode;`	
`    last = newNode;`	
	`Node newNode = new Node(1);`
	`if (last == null){` *[false]*
`    first.next = null;`	
`}`	

Once T_1 starts, it is supposed to yield right after the first if statement. But if we add a line to do that:

```
public void insertAtTail(Integer key)
{
    Node newNode = new Node(key);
    if (last == null) {  // empty list
        currentThread.yield();
        first = newNode;
        last = newNode;
        first.next = null;
    }
```

then <u>every</u> thread will yield at that point, even though T_0 is **not** supposed to. You need a way to specify a point in your code for a yield to happen *conditionally*, i.e. it should happen for one thread but not another. To handle this, write a (static!) method called `yieldIfOughtTo()` that will keep track of the number of times it has been executed by any thread. And based on that count, it will either yield or not. Do this using two shared (i.e. instance) variables in the KThread class: an array of booleans named `oughtToYield` and a shared int named `numTimesBefore`. The latter keeps track of the number of times `yieldIfOughtTo` has been executed before (so it increments each time `yieldIfOughtTo` runs). If `oughtToYield[i]` is true, that means we should yield if we've already executed the `yieldIfOughtTo` method `i` times before.

For example, to create the context switch from T_1 to T_0 from slide 38 above, initialize `oughtToYield[0]` to false and `oughtToYield[1]` to true. That makes the following happen:

```
public void insertAtTail(Integer key)
{
    Node newNode = new Node(key);
    if (last == null) {  // empty list
        yieldIfOughtTo();
```

T_0 reaches this line first, when it has been executed 0 times before. Look up `oughtToYield[0]`. Since it's false, T_0 does not yield.

T_1 reaches this line next, when it has been executed 1 time before. Look up `oughtToYield[1]`. Since it's true, T_1 yields like it should.

Now, `yieldIfOughtTo` will only work if there's a single point in your code where context switches happen, but that's all we need to demonstrate multithreading.

Make it Happen

Read the class description at the top of KThread which gives a nice summary of how new threads are created. Then make a new private inner class called **DLListTest** that is modeled after the PingTest class that you analyzed in the last project. DLListTest should contain this method:

```
/**
 * Prepends multiple nodes to a shared doubly-linked list. For each
 * integer in the range from...to (inclusive), make a string
 * concatenating label with the integer, and prepend a new node
 * containing that data (that's data, not key).  For example,
 * countDown("A",8,6,1) means prepend three nodes with the data
 * "A8", "A7", and "A6" respectively.  countDown("X",10,2,3) will
 * also prepend three nodes with "X10", "X7", and "X4".
 *
 * This method should conditionally yield after each node is inserted.
 * Print the list at the very end.
 *
 * Preconditions: from>=to and step>0
 *
 * @param label string that node data should start with
 * @param from integer to start at
 * @param to integer to end at
 * @param step subtract this from the current integer to get to the next integer
 */
public void countDown(String label, int from, int to, int step) {
```

DLListTest also needs a single shared doubly-linked list as an instance variable.   Make it static to guarantee that there is only one such object.

Finally, you need to write code so that you'll have two threads total, each running `countDown`. (**PITFALL ALERT**: how many new threads will you need to create?) This will be similar to what `KThread.selfTest` did in Project 0. Write the following method in KThread:

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

Run Nachos using the same proj1 working directory as the last project. Debug your code until it works.   You will show me that it works by taking a screenshot of your output for each of the following test cases:

1. Create an interleaving so all the "A" nodes are prepended before all the "B" nodes.
2. Create an interleaving where the "B" nodes alternate with the "A" nodes, so that the integer part of the data is in sorted order: B1, A2, B3, A4, …, B11, A12.
3. Create an interleaving where two "A" nodes are prepended, then two "B" nodes, then two "A" nodes, etc.

You should only be altering the `oughtToYield` array to make the different interleavings happen.

Part 2: Break it down
Time to break your doubly-linked list.

1. Analyze your code and create at least two interleavings that will cause concurrency errors to occur. One error must cause your code to crash (a fatal error) while the other must be a non-fatal error that leaves your doubly-linked list in an invalid state.

2. For each error, cause it to occur by adding another method similar to `DLL_selfTest` above. That method will use one or more private inner classes (like DLListTest) to contain the behavior of each thread you fork. Remember that these threads need to use a shared static DLList object between them.

   In all likelihood, you will need *multiple* locations in your DLList code that conditionally yield. Do the same trick as in part 1, but just up the dimensions by one. `oughtToYield` becomes a 2D array of booleans called `yieldData`. And the int `numTimesBefore` becomes an int array named `yieldCount`. Each column of `yieldData` is like one instance of `oughtToYield`. It represents one line in your code (one location) where yielding might occur. And `yieldCount[i]` represents the number of times you've previously executed the conditional yield at location `i`. All you need now is a conditional yield method whose job is to use `yieldData` and `yieldCount` to figure out if it's time to yield. Let's call that `yieldIfShould`:

```
/**
 * 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 DLList code should
 *             have a different loc number.
 */
public static void yieldIfShould(int loc) {
```

   Now you can call `yieldIfShould` at different points in your code where different threads may or may not yield when they reach those points. Just make sure to use a unique location number for the argument. This will open up many more possibilities of interleaving errors for you.

3. Finally, document your interleavings in a writeup that you will turn in with this project. Each interleaving should contain

   - a drawing of the interleaving similar to slide 38 on the first page of this document.
   - a text description of the error. Be sure to include the initial state of the list and the arguments of whatever methods you call. For the non-fatal error, be sure to include what the sequential orderings would have been so I can see that your result is different.
   - a screenshot of the error when run. Embed this screenshot in your pdf file.

Note: When you move on to a new interleaving, do not remove calls to `yieldIfShould` in your code that dealt with previous interleavings. You can "disable" those locations simply by putting falses in the right places in the `yieldData` array. When I grade this, I should be able to simply comment out all other tests I don't want to run and see the result described in your writeup.

Turning it in

The ONE pdf file you will turn in must have

1. The three screenshots from Part 1.
2. The writeup from Part 2 described on the previous page.
3. The source code of **just** the Java classes you added code to.

Place this pdf into your "nachos" project folder and zip it up. Upload the zip file to Nexus.

Grading

This project is worth 50 points.

- 15 points for the working multi-threaded program in Part 1 (5 pts per interleaving)
- 30 points for the interleavings in Part 2 (15 pts each)
- 5 points for the writeup. The harder it is for me to understand what your interleavings are doing, the fewer points you will receive.

Gentle Reminder

Programming projects are *individual* projects. I encourage you to talk to others about the general nature of the project and ideas about how to pursue it. However, the technical work, the writing, and the inspiration behind these must be substantially your own. You must cite anyone else who contributes in any way to the project by adding appropriate comments to the code. Similarly, if you include information that you have gleaned from other sources, you must cite them as references. Looking at, and/or copying, other people's code is inappropriate, and will be considered an honor code violation.