HW2 Threads

The three parts (a, b, c) of HW2 are due midnight ending Fri May 11th. Part(c) will be on a separate handout.

Part (a) of the homework is a warm-up exercise to get you accustomed to threads. The code is a bit contrived (!), but it demonstrates the core issues of threaded programming.

Part (a) -- Magic Thread

Consider the following code which is available for you in the file MagicThread.java in the class directory. This code has several threading problems which it will be your pleasure to fix. Here's the code...

```
// MagicThread.java
/*
 The MagicStrings class stores some magic strings
 which the client can access one by one. The magic
 strings are a big secret, so the client can only
 get them one at a time.

 Each instance of MagicThread retrieves all the strings
 in order and concats them together along with the name of a
 staffer to make a little spell like this:
 "fibbity fabbity foo : Jason".

 The magic words have to all be there and be in the right
 order, or the spell doesn't work.

 Each magic thread concats its spell onto the "answer"
 static when it is done.
*/


/*
 Store magic strings -- give them
 out to the client one at a time.
*/
class MagicStrings {
   private String[] strings;
   private int index;

   public MagicStrings() {
      strings = new String[] {"fibbity", "fabbity", "foo"};
      index = 0;
   }

   /*
    Return the next magic string, or null
    if there are no more.
   */
   public String nextString() {
```

```
      if (index<strings.length) {
         index++;
         return(strings[index-1]);
      }
      else {
         return(null);
      }
   }

   /*
    Reset to the start so that the
    next call to nextString() will return
    the first magic string.
   */
   public void reset() {
      index = 0;
   }
}


/*
 Takes a pointer to a magic object.
 Gets the strings out of it and concats
 them together.
 When done, puts the whole thing into the
 "answer" static.
*/
class MagicThread extends Thread {
   private MagicStrings magic;
   private String name;
   private static StringBuffer answer;


   public MagicThread(MagicStrings magic, String name) {
      this.magic = magic;
      this.name = name;
   }


   public void run() {
      String result = new String();
      String next;

      // see the magic strings in order
      magic.reset();
      while ((next = magic.nextString()) != null) {
         // concat them together
         result = result + next + " ";
      }
      answer.append(result + ": " + name + "\n");
   }


   /*
    Create one magic object and three
    threads to create a spell for each CS person.
    The output should look like...
    fibbity fabbity foo : Eric
```

```
     fibbity fabbity foo : Julie
     fibbity fabbity foo : Nick

     although the order of the rows may be different.
    */
    public static void main(String[] args) {
       MagicStrings magic = new MagicStrings();


       Thread t1 = new MagicThread(magic, "Jason");
       t1.start();
       Thread t2 = new MagicThread(magic, "Saurabh");
       t2.start();
       Thread t3 = new MagicThread(magic, "Nick");
       t3.start();

       answer = new StringBuffer();

       // Wait for the three to finish
       try {
          t1.join();
          t2.join();
          t3.join();
       }
       catch (InterruptedException ignored) {}

       System.out.println(answer);
    }
```

The challenge is to fix the various problems with the code. Your changes will be to the MagicThread class. Your solution should continue to use a single MagicStrings object.

Part i — Null Pointer

There's a threading problem in the code that leads to a NullPointerException (NPE) in rare cases. This bug can be fixed by moving one line from one place in the code to another. Please move the line and append a "// part i" comment to it. Afterwards, the code should still have mutex problems, but at least it won't generate the NPE.

Part ii — Mutex Violation

There's a mutex violation that causes the spells to be wrong sometimes. Sometimes the problem will exhibit itself and sometimes it won't. Add a single Thread.yield(); call so that the mutex error almost always exhibits itself, and append a "// part ii" comment to the line.

Part iii — Mutex Repair

Now add synchronization inside MagicThread.run() so that the code runs correctly, even with the part (ii) yield() present.

Part iv — Performance

Finally, change the code inside MagicThread.run() to make it run faster with the following strategy: **minimize the amount of work that is done while holding a**

lock. Avoid computation, as much as possible, while holding the lock. Computation while not holding the lock is relatively cheap. The part ii and iii code should still be present. To change code, comment out the old lines and put in the new. Avoid String where StringBuffer is better. You may assume there are at most 100 strings returned by MagicStrings.

HW2b — Thread Bank

Homework 2b is a classical threading application. It takes the threading concepts you would have seen in any Operating Systems or concurrency course taught since 1975, and applies them in Java. The most important concepts are critical sections (synchronize) and coordination (wait/notify).

Thread Bank

The apparent operation of Thread Bank is quite simple...

Have a ThreadBank object which contains 20 Accounts numbered 0..19.

Each Account object starts with an initial balance of 1000.

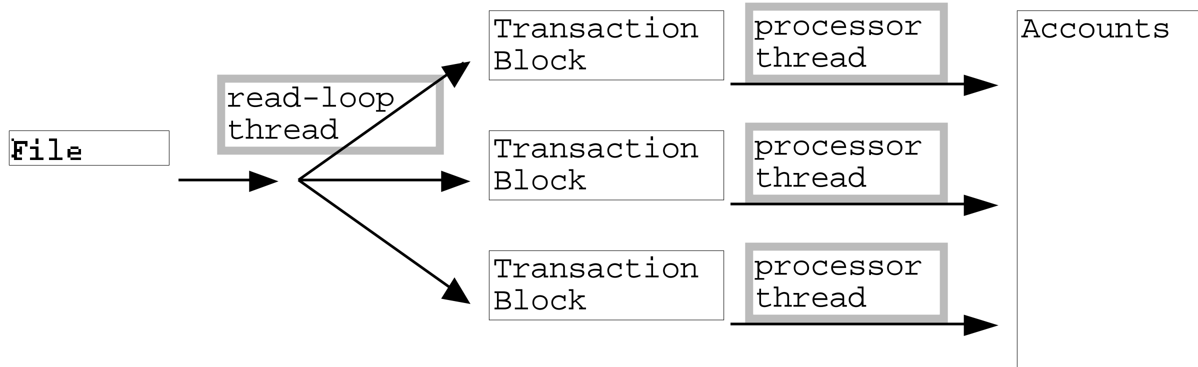
There is a text file of transactions where each transaction is represented by three ints separated by spaces on a line... *from_account_num to_account_num amount*

Read through and perform all of the transactions in the file.

Print out the ending balances of the accounts.

Threads Threads Everywhere But Not A Thought To Think

Fortunately, the above dull computation can be spruced up with extensive use of threads. The concurrent version has a main thread that reads transactions out of the file and passes them out round-robin among three processing threads (0, 1, 2, 0, 1, 2, ...). The processing threads each get transactions from the main thread and apply them one at a time to the relevant accounts. Each processing thread has its own "TransactionBlock" that buffers up the transactions to be processed.



Realism
Though not a 100% accurate simulation of the banking system, the code structure presented here is realistic in many respects. It will take advantage of multiple hardware processors— it is possible to run this program where it does 4 CPU seconds of computation but completes in 3 seconds — yay! Its structure is especially well suited to problems where the "transaction" operation is expensive (our little banking addition/subtraction transaction is not especially expensive, but the structure is correct).

Classes
Here are the classes you should build for ThreadBank...

Account
The Account object should just store a balance. It should support getBalance() and changeBalance(delta) messages. The Account messages should not be synchronized — that needs to be handled one level higher.

Transaction
The Transaction object just holds the three ints on their trip from the file to the processing threads. It's ok to just code the Transaction like a struct instead of an object...

```
class Transaction {
   public int from;
   public int to;
   public int amount;
}
```

TransactionBlock
Each processing thread has one TransactionBlock (TB) that holds its incoming transactions. The main thread puts transactions in the TransactionBlock while the processor removes them and processes them. The TB is a traditional looking reader/writer buffer object.

For efficiency, the TB should be implemented as a fixed array of 64 pre-allocated transaction objects. The TB should support...

- void add(int from, int to, int amount) — add a transaction to the TB. Do not call new(); use one of the pre-allocated transactions. Internally, this should wait if the TB is completely full.

- Transaction acquire() — get a transaction from the TB. Internally, this should wait if the TB is completely empty. For efficiency, this should just return a sharing pointer to the transaction object owned by the TB. Acquire() should retrieve the transactions in FIFO order.

- void release(Transaction trans) -- the client is done with the acquired transaction and so returns it to the TB for re-use. The client must balance each call to acquire() with a call to release() before the next acquire(). The transaction is passed back to the TB just for assert error checking -- the TB knows which transaction is being returned.

The TB may assume that there will be one thread calling add() and one thread calling acquire()/release(). The TB interface is structured for efficiency::

- It should use its fixed pool of 64 transactions without calling new(). If you remember nothing else about Java performance, remember that new() is slow.

- The transactions do not need to move around inside the TB. Instead, the TB should use a few ints to keep track of its state (detailed below).

The TB should keep three ints to keep track of the current addition index, the current acquire index, and the current total number of elements. (Write a comment for each int variable describing its exact semantics so your code will be consistent with itself.) The three methods should be synchronized to keep the updates to the TB consistent as the multiple threads execute against it — a classic mutex application. Add() should wait() when there is no room, and release() should notify() when it makes room (in case add() was blocked). Likewise, acquire() should wait() when there are no transactions, and add() should notify(). The wait() and notify() can be on the TransactionBlock itself. Because we only have a single writer and a single reader, the notifications are only necessary at the very end cases: going from 0 to 1 available transactions or going from 0 to 1 available spaces.

ThreadBank

The ThreadBank object should contain an array of 20 Account objects. The most interesting method is..

```
public void doTransaction(int from, int to, int amount)
```

doTransaction() should perform one transaction in a thread safe way — multiple threads will be calling doTransaction() at the same time. For this assignment, it's required that both account locks be held before either balance is touched. The

doTransaction() method should not be synchronized — that synchronization must occur at a lower level. Internally, doTransaction() has three phases...

1. Check to see that the two accounts mentioned in the transaction exist. If an account does not exist, it should be created with an initial balance of 1000. The (rare) operation of creating an account should be done under a special createAccount mutex to prevent multiple threads from trying to create an account at the same time. For efficiency, the existence of an account can be checked outside of the creation mutex, but it must be re-checked and changed inside the mutex. Convince yourself that doing an `==null` test on the pointer outside of the mutex will work (pointer assignment is guaranteed to be atomic in Java). (You could just create all the accounts at the beginning, but it would be too easy. Also, this model makes more sense if there are 10000 account slots in the Bank, but only some of them are going to be mentioned in the transactions.)

2. Obtain locks for the two accounts to be changed. Use a synchronize(Account) {..} block for each account. It is required that you hold both locks simultaneously before touching either balance. Notice that by obtaining locks as needed account by account, we allow multiple threads to do transactions on the Bank at the same time while they are on different accounts. This is a more realistic approach than simply synchronizing all of doTransaction(). Important: you must acquire the lower-numbered lock first, then the higher numbered lock, otherwise there is a chance of deadlock (try it and see!). This is the classic "obtain locks in a consistent order everywhere" trick.

3. Having verified that both accounts exist, and having obtained locks for both of them, perform the transaction and get out.

Processor

Internally, the ThreadBank will create three Processor threads to do the processing. The Processor class should be an internal class to ThreadBank (so it can see the accounts[] array) and a subclass of Thread. In its constructor, each Processor should take a pointer to a TransactionBlock that it will use to get transactions from the Bank. In its run() loop, the Processor should get transactions from its TransactionBlock and apply them to the ThreadBank. A transaction with a "from" account of -1 signals that there are no more transactions and the Processor should exit normally.

ThreadBank.processFile(File file)

The whole process initiates with a single call to processFile()....

1) Create three TransactionBlocks and Processors and start them.

2) Create a streamTokenizer on the File, loop on it reading out the three ints for the transactions, and add() the transactions round-

robin to the TransactionBlocks that the Processors are waiting on.
To save you from the tedious I/O code, here it is....

```
FileInputStream input = new FileInputStream(file);
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
StreamTokenizer tokenizer = new StreamTokenizer(reader);

...

int from, to,  amount;
while (true) {
   try {
      read = tokenizer.nextToken();
      if (read == StreamTokenizer.TT_EOF) break;
      from = (int)tokenizer.nval;

      tokenizer.nextToken();
      to = (int)tokenizer.nval;

      tokenizer.nextToken();
      amount = (int)tokenizer.nval;
   }
   catch(NumberFormatException e) {
      System.out.println("Error:could not parse int '" +
                         tokenizer.sval + "'");
      System.exit(-1);
   }

   // send the transaction off for processing
}
```

3. When the file is exhausted, add() the -1 transaction to each TransactionBlock and do a join() on each Processor to wait for it to exit.

4. Print out a summary of the state of the Accounts which now exist in the accounts array, one per line...
   account_num balance

Main

The main() function in ThreadBank should create a Bank object and send it a processFile(File) message to process the one file specified as a command line argument.

Error Handling

You may ignore all exceptions, but you may wish to print out some sort of error message to help your debugging.

Testing

First test with no extra threads — just call doTransaction() straight from the file reading loop. When that's working, try adding a single Processor thread. Finally, try multiple Processor threads. If when you add multiple threads the answer comes out wrong every now and then — you have a concurrency bug. Be sure to

do some testing on the sagas (multiple processors) to try to expose your concurrency bugs.

The file "small.tr" contains some trivial transactions to get started, but it does not push the concurrency. The file "5k.tr" contains 5000 transactions that happen to all cancel out, so at the end, all the accounts should have a balance of 1000. The files 25k.tr and 100k.tr are the same but with more transactions.

You can use the unix "time" command to see if the program is using more than one hardware processor on the sagas. If the load is low enough on the machine, you have a chance of exceeding 100% utilization. Use the "top" and "uptime" command to see how heavily loaded your machine is. You'd prefer a load under 1.0.

```
saga2:~/java/BankThread> time java ThreadBank 25k.tr > /dev/null
2.66u 2.31s 0:03.77 131.8%
```

Here are the relevant parts of the "time" section in the csh man page

```
     time              Control automatic timing of commands.  Can
                       be  supplied  with one or two values.  The
                       first is the reporting  threshold  in  CPU
                       seconds.  The  second  is a string of tags
                       and text  indicating  which  resources  to
                       report  on.   A  tag is a percent sign (%)
                       followed by  a  single  upper-case  letter
                       (unrecognized tags print as text):

                              %U   Number of seconds of CPU  time
                                   devoted to the user's process.
                              %S   Number of seconds of CPU  time
                                   consumed   by  the  kernel  on
                                   behalf of the user's process.
                              %E   Elapsed (wallclock)  time  for
                                   the command.
                              %P   Total CPU time - U (user) plus
                                   S  (system)  - as a percentage
                                   of E (elapsed) time.
```

I've found it to be quite rare to get more than 100% CPU use, but it does happen. It's rare because so many other people are doing stuff, that the odds of getting both CPUs is poor, plus if your job gets swapped off one or both CPUs during the normal course of Unix scheduling, wall clock time, which is what the "time" command measures, keeps running. Also, the single threaded java class loading and startup times operations dilute our concurrency times, but you can compensate for that somewhat by using 25k.tr, and running jobs in quick succession so the disk hits are all cached. You could try using tree which has a bunch of processors (16?), but the 50 other people on tree at the same time as you conspire to never give you more than one CPU. On the single processor elaines, you will never get higher than 100% CPU use.

Note: sometime in your lifetime, you will have a personal 16 processor machine, and you will be one of the few people who has written a program that can actually use the multiple processors! At present, multiprocessor machines are rare, and the concurrent software to use mutiple processors is even more rare.