HW3a Threads

This is part A of hw3. Part A uses classical synchronization and co-ordination to use multiple processors. Part B will introduce GUIs and networking. Both parts will be due midnight ending Mon March 3rd.

A. Thread Bank

For this assignment, you will build some classical non-GUI threaded code.

For this problem, you have an array of Account objects and a text file of transactions where each transaction moves an amount of money from one account to another. We start each account with a balance of 1000, then apply all the transactions to see what the final balance is of each account.

The trick here is to use threading to complete the operation more quickly...

- One thread reads the transactions from the file, one at a time, and adds them to the Buffer object.
- There are multiple worker threads, where each worker repeatedly gets a transaction from the buffer and performs that transaction on the accounts.

Here are the classes...

Account

Account is a simple, classical class that encapsulates an int balance and int number of transactions. The code for Account is provided for you. The ctor should take the initial balance and the transactions should start at 0. The Account has a synchronized apply(transaction) method that change the balance and increments the number of transactions. These methods are synchronized so that multiple threads can send them at the same time without corrupting the account.

Buffer

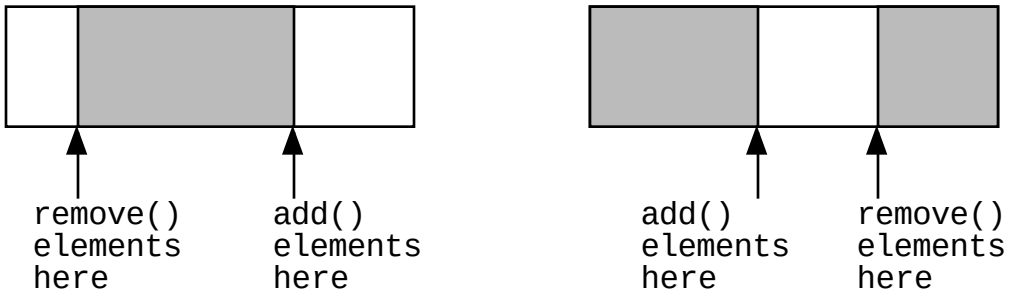
The Buffer object is a temporary storage area that holds the transactions before they are processed. The Buffer uses the Transaction class to store the ints from, to, and amount -- "from" and "to" are the account numbers and "amount" is the amount to transfer. The Transaction class is just a struct used for storage.

Buffer should respond to add(Transaction) which adds a transaction object to the buffer. The thread reading transactions from the file will call add(). Buffer should respond to a remove() operation that gets a Transaction object from the buffer. The worker threads that process the transactions will call remove(). Add() and remove() should be synchronized since multiple threads will be calling them at

the same time. Remove() should provide elements in the order in which they were added like a FIFO queue.

Buffer Implementation

Internally, the buffer should store the transactions in a fixed size Transaction[ ] array of size 64. The array should be used as a circular buffer to store the transactions. The buffer should track both a current add point and a current remove point. Both should wrap around from the right hand size back to 0. Keep a separate "size" int to keep track of how full the array is -- it's a headache trying to deduce the size from just the add and remove points.



Buffer Blocking

There are two cases where a thread using the buffer needs to block

1. an add() when the buffer is full (the add point has in effect caught up to the remove point).

2. a remove() when the buffer is empty (the remove point has caught up to the add point).

We will use the following simple strategy: We will use the buffer itself as the shared wait/notify object for both the adder and the removers. When an add() or remove() operation should wait on the buffer when the add() or remove() cannot proceed (the two conditions above). To balance the waits, when an add() increases the number of stored elements from 0 to 1, it should notifyAll() in case there are waiting removers. Likewise, when a remove() decreases the size from SIZE to SIZE-1, it should notifyAll() in case an adder is waiting.

A more complex solution could use separate objects for the add and remove wait queues and be more precise about where notifyAll() vs. notify() is required. However, the simple strategy above is pretty good. For this assignment, there will be no interruption, so you may ignore the InterruptedException from the wait().

Bank

The bank class should contain an array of 20 accounts and a buffer object. From the command line, the Bank should take the filename of transactions and the number of worker threads to use. In main(), the bank should create and start the

workers, read through the file and add transactions to the buffer. After all the transactions have been added, the Bank should add a null -- one for each worker. When each worker gets a null from remove(), it knows the data is finished and can exit cleanly. When all the workers are finished, the provided code in Bank and Account prints a summary for each account on one line, giving its account number, balance, and number of transactions (note the cute use of toString() in Account to accomplish this).

Worker

Worker should be an inner class of Bank that runs the following loop: try to get a transaction. If the transaction is null, exit the loop. For each transaction, withdraw the given amount from the "from" account and deposit it into the "to" account.

As usual, there are some starter materials in the course directory. Try running with 1 worker thread vs. 10 worker threads. The files 5k.txt and 100k.txt have transactions in them that balance out, so each account should end with the same balance it started with.

```
> java Bank test.txt 4
acct:0 bal:1490 trans:16
acct:1 bal:2140 trans:22
acct:2 bal:930 trans:8
...
```

Bad Accounting

When you have the basic account features working, add the following Bad Accounting feature. When the command line contains a third "limit" argument, the bank should keep track of "bad" transactions where a transaction has caused the account balance to be strictly less than the limit. The Bank object should respond to an addBad(transaction, balance) message and record those details about the transaction and account balance. The addBad() message should be sent to the Bank each time a transaction causes an account to go below the limit during processing. Modify Bank main() to print out a line for each bad transaction after the regular output...

```
> java Bank test.txt 4 0      ## note limit arg of "0"
...
acct:18 bal:2140 trans:22
acct:19 bal:930 trans:8
Bad transactions...
from:7 to:2 amt:100 bal:-140
from:11 to:19 amt:160 bal:-80
...
```

Add classes, ivars, methods, and parameters as needed to support the limit feature. In particular, the worker or account will need to know the limit and will need a pointer to the bank to send it messages. The bank will need a way to store bad transaction information -- consider using a simple nested class with a toString() method. If the limit command-line argument is not specified, then addBad() should not be called and the Bad Accounts printing should not happen. Note that which accounts go over the limit may change from one run to the next,

since it can depend on the exact order that the transactions are applied. The ending balance of each account should be the same every time however.