CS193j, Stanford
Winter, 2001-02

Handout #24
Nick Parlante

HW3a Threads

This is part (a) of hw3. This is a small threading exercise. Part (b) will combine a GUI with networking. Both parts will be due midnight ending Tue March 5th.

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 want to 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 ctor should take the initial balance and the transactions should start at 0. The Account should have deposit(int amt) and withdraw(int amt) methods that change the balance and increment the number of transactions. These methods should be 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 should define a nested Transaction class that stores 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. To other classes, the name of the nested Transaction class is Buffer.Transaction.

Buffer should respond to an add(Transaction) message that adds a transaction object to the buffer. The add() operation should never fail. Buffer should respond to a remove() message that gets a Transaction object from the buffer. This returns null if there are no transactions in the buffer. As with the account, add() and remove() should be synchronized since multiple threads will be trying to run them at the same time.

Buffer Implementation

Internally, the buffer should store the transactions in an array initially of size 10. Use a single "size" int to keep track of how full the array is. It's simplest if the remove() pulls transactions from the buffer in LIFO order -- like a stack. When an add() happens and the array is full, allocate a new array that is twice the size of the old array and copy the contents over. This complication is internal to the buffer class. (The little double-size-on-overfill strategy is a standard bit of programming everyone should know.)

Bank

The bank class should contain an array of 20 accounts, a buffer object, and a "done" boolean that is initially false. 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, set done to true, and wait for all the workers to finish. When all the workers are finished, the bank should print a summary for each account on one line, giving its account number, balance, and number of transactions.

Worker

Worker should be an inner class of Bank that runs the following loop: try to get a transaction. If the transaction is null check to see if "done" is true, and if it is exit the loop. If done is false, sleep for 200 milliseconds to give the buffer a chance to fill up (you may do nothing on the exception). 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 cancel out, so the end balances should be 1000. Temporarily remove the "synchronized" keywords -- notice that the program now gets the wrong answer! Use the "time" command running on a saga.stanford.edu or tree.stanford.edu (multiple processors) and you may see greater than 100% cpu utilization. Processing is truly happening in parallel-- neato! (The measurement is not entirely accurate since our worker-sleep loop uses CPU without doing useful work.)

```
> time java Bank 100k.txt 4
```

Improvements

A better implementation of the Buffer would not use a sleep-loop in the worker. Intead the buffer and the workers could use a wait/notify convention so that the workers could block efficiently when there are no transactions. However, our version is not too bad. Also, the threading would buy more if the worker computation were more expensive than a little arithmetic. However, the Bank demonstrates the basic pattern of forking off multiple workers to do work in parallel.