javaf / savings-account

A savings account is used to store salary/savings in a bank. This is a concurrent object (RAM) based account. For educational purposes only.

repl.it/@wolfram77/savings-account#readme.md

MIT License

0 stars 0 forks

Star Unwatch

Code Issues Pull requests Actions Projects Wiki Security

master

wolfram77 output in README ... 11 hours ago 7

View code

README.md

A savings account is used to store salary/savings in a bank. This is a concurrent object (RAM) based account. For educational purposes only.

`withdraw()` from an account blocks until the account has the necessary balance.
`transfer()` from an account also blocks until the source account has the necessary balance. Preferred withdrawls are performed, if waiting.

> Note: Using 2 separate conditions avoids unnecessary waking up of non-preferred withdrawls when preferred withdrals are present.

```
deposit(k):
1. Acquire common lock.
2. Deposit to account.
3. Signal that account has currency.
4. Release common lock.



withdraw(k, pref):
1. Acquire common lock.
```

```
2. Update pending preferred withdrawls.
3. Wait until sufficient balance available.
4. Withdraw from account.
5. If non-zero balance, then signal it.
6. Update pending preferred withdrawls.
7. Release common lock.



transfer(k, from, pref):
1. Withdraw from source account.
2. Deposit to this account.



## OUTPUT
Setting up accounts ...

Starting transfers ...
6: [21] 100 from 3
4: [55] 100 from 5
7: [145] 100 from 1
0: [195] 100 from 9
5: [199] 100 from 6
2: [68] 100 from 8
1: [36] 100 from 6
3: [39] 100 from 0
9: [22] 100 from 3
8: [73] 100 from 2
3: [139] done
0: [195] done
4: [155] done
9: [122] done

Boss donates 1000 to all
5: [1199] done
2: [1068] done
1: [1036] done
8: [1173] done
7: [1245] done
6: [1021] done
```

See SavingsAccount.java for code, Main.java for test, and repl.it for output.

references

- The Art of Multiprocessor Programming :: Maurice Herlihy, Nir Shavit

Languages

92 lines (77 sloc) 2.39 KB

```
1  import java.util.concurrent.locks.*;
2
3  // A savings account is used to store salary/savings in
4  // a bank. This is a concurrent object (RAM) based
5  // account. For educational purposes only.
6  //
7  // withdraw() from an account blocks until the account
8  // has the necessary balance.
9  // transfer() from an account also blocks until the
10 // source account has the necessary balance.
11 // Preferred withdrawls are performed, if waiting.
12 //
13 // Note: Using 2 separate conditions avoids unnecessary
14 // waking up of non-preferred withdrawls when preferred
15 // withdrals are present.
16
17 class SavingsAccount {
18   Lock lock;
19   Condition hasNormal;
20   Condition hasPreferred;
21   int balance;
22   int preferred;
23   // lock: common lock
24   // hasNormal: account has balance (no preferred deposits)
25   // hasPreferred: account has balance (some preferred)
26   // balance: amount of currency in account
27   // preferred: pending preferred withdrawls
```

```
28
29
30    public SavingsAccount() {
31      lock = new ReentrantLock();
32      hasNormal = lock.newCondition();
33      hasPreferred = lock.newCondition();
34      balance = 0;
35      preferred = 0;
36    }
37
38
39    // 1. Acquire common lock.
40    // 2. Deposit to account.
41    // 3. Signal that account has currency.
42    // 4. Release common lock.
43    public void deposit(int k) {
44      lock.lock();   // 1
45      balance += k;  // 2
46      signal();      // 3
47      lock.unlock(); // 4
48    }
49
50
51    // 1. Acquire common lock.
52    // 2. Update pending preferred withdrawls.
53    // 3. Wait until sufficient balance available.
54    // 4. Withdraw from account.
55    // 5. If non-zero balance, then signal it.
56    // 6. Update pending preferred withdrawls.
57    // 7. Release common lock.
58    public void withdraw(int k, boolean pref)
59    throws InterruptedException {
60      lock.lock();           // 1
61      if (pref) preferred++; // 2
62
63      while (balance<k) await(pref); // 3
64      balance -= k;                  // 4
65
66      if (balance>0) signal(); // 5
67      if (pref) preferred--;   // 6
68      lock.unlock();           // 7
69    }
70
71
72    // 1. Withdraw from source account.
73    // 2. Deposit to this account.
74    public void transfer(int k,
75    SavingsAccount from, boolean pref)
```

```
76    throws InterruptedException {
77      from.withdraw(k, pref); // 1
78      deposit(k);             // 2
79    }
80
81
82    private void signal() {
83      if (preferred>0) hasPreferred.signal();
84      else hasNormal.signal();
85    }
86
87    private void await(boolean pref)
88    throws InterruptedException {
89      if (pref) hasPreferred.await();
90      else hasNormal.await();
91    }
92  }
```

javaf / savings-account

Code Issues Pull requests Actions Projects Wiki Security

master

savings-account / Main.java / Jump to

wolfram77 withdraw() needs to signal too! History

1 contributor

Raw Blame

77 lines (65 sloc) 1.76 KB

```
1  class Main {
2    static SavingsAccount[] accounts;
3    static int N = 10;
4    // accounts: savings accounts
5    // N: number of accounts
6
7    static Thread person(int id) {
8      return new Thread(() -> {
9        try {
10       int f = (int) (N * Math.random());
11       SavingsAccount mine = accounts[id];
12       SavingsAccount from = accounts[f];
13       log(id+": ["+mine.balance+"] 100 from "+f);
14       mine.transfer(100, from, false);
15       log(id+": ["+mine.balance+"] done");
16       } catch (InterruptedException e) {}
17     });
18   }
19
20   // Setup N accounts with random balance.
21   static void setupAccounts(int kmax) {
22     accounts = new SavingsAccount[N];
23     for (int i=0; i<N; i++) {
24       int k = (int) (kmax * Math.random());
25       accounts[i] = new SavingsAccount();
26       accounts[i].deposit(k);
27     }
```

```
28    }
29
30    // Start all transfers (by person).
31    static Thread[] startTransfers() {
32      Thread[] t = new Thread[N];
33      for (int i=0; i<N; i++)
34        t[i] = person(i);
35      for (int i=0; i<N; i++)
36        t[i].start();
37      return t;
38    }
39
40    // Deposit amount to all accounts.
41    static void depositAll(int k) {
42      for (int i=0; i<N; i++)
43        accounts[i].deposit(k);
44    }
45
46    // Wait for all transfers to complete.
47    static void awaitTransfers(Thread[] t) {
48      try {
49      for (int i=0; i<t.length; i++)
50        t[i].join();
51      } catch (InterruptedException e) {}
52    }
53
54
55    public static void main(String[] args) {
56      log("Setting up accounts ...");
57      setupAccounts(200);
58
59      log("\nStarting transfers ...");
60      Thread[] t = startTransfers();
61      sleep(1000);
62
63      log("\nBoss donates 1000 to all");
64      depositAll(1000);
65      awaitTransfers(t);
66    }
67
68
69    static void sleep(long ms) {
70      try { Thread.sleep(ms); }
71      catch (InterruptedException e) {}
72    }
73
74    static void log(String x) {
75      System.out.println(x);
```

```
76      }
77  }
```