Monitors in Java

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming Concepts using Java

Week 11

Monitors

- Monitor is like a class in an OO language
  - Data definition — to which access is restricted across threads
  - Collections of functions operating on this data — all are implicitly mutually exclusive
- Monitor guarantees mutual exclusion — if one function is active, any other function will have to wait for it to finish
- Implicit queue associated with each monitor
  - Contains all processes waiting for access

```
monitor bank_account{
  double accounts[100];

  boolean transfer (double amount,
                        int source,
                        int target){
    if (accounts[source] < amount){
      return false;
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    return true;
  }

  double audit(){
    // compute balance across all accounts
    double balance = 0.00;
    for (int i = 0; i < 100; i++){
      balance += accounts[i];
    }
    return balance;
  }
}
```

Condition variables

- Thread suspends itself and waits for a state change — `q[source].wait()`
- Separate internal queue, vs external queue for initially blocked threads

```
monitor bank_account{
  double accounts[100];
  queue q[100];  // one internal queue
                 // for each account
  boolean transfer (double amount,
                    int source,
                    int target){
    while (accounts[source] < amount){
      q[source].wait();  // wait in the queue
                         // associated with source
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    q[target].notify();  // notify the queue
                         // associated with target
    return true;
  }

  // compute the balance across all accounts
  double audit(){ ...}
}
```

Condition variables

- Thread suspends itself and waits for a state change — `q[source].wait()`
- Separate internal queue, vs external queue for initially blocked threads
- Notify change — `q[target].notify()`

```
monitor bank_account{
  double accounts[100];
  queue q[100];  // one internal queue
                 // for each account
  boolean transfer (double amount,
                    int source,
                    int target){
    while (accounts[source] < amount){
      q[source].wait();  // wait in the queue
                         // associated with source
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    q[target].notify();  // notify the queue
                         // associated with target
    return true;
  }

  // compute the balance across all accounts
  double audit(){ ...}
}
```

Condition variables

- Thread suspends itself and waits for a state change — `q[source].wait()`
- Separate internal queue, vs external queue for initially blocked threads
- Notify change — `q[target].notify()`
- Signal and exit — notifying process immediately exits the monitor
- Signal and wait — notifying process swaps roles with notified process
- Signal and continue — notifying process keeps control till it completes and then one of the notified processes steps in

```
monitor bank_account{
  double accounts[100];
  queue q[100];  // one internal queue
                 // for each account
  boolean transfer (double amount,
                    int source,
                    int target){
    while (accounts[source] < amount){
      q[source].wait();  // wait in the queue
                         // associated with source
    }
    accounts[source] -= amount;
    accounts[target] += amount;
    q[target].notify();  // notify the queue
                         // associated with target
    return true;
  }

  // compute the balance across all accounts
  double audit(){ ...}
}
```

Monitors in Java

- Monitors incorporated within existing class definitions

```
public class bank_account{
 double accounts[100];

 public synchronized boolean
   transfer(double amount, int source, int target){
  while (accounts[source] < amount){ wait(); }
  accounts[source] -= amount;
  accounts[target] += amount;
  notifyAll();
  return true;
 }

 public synchronized double audit(){
  double balance = 0.0;
  for (int i = 0; i < 100; i++)
    balance += accounts[i];
  return balance;
 }

 public double current_balance(int i){
  return accounts[i];   // not synchronized!
 }
}
```

Monitors in Java

- Monitors incorporated within existing class definitions
- Function declared `synchronized` is to be executed atomically

```
public class bank_account{
 double accounts[100];

 public synchronized boolean
   transfer(double amount, int source, int target){
  while (accounts[source] < amount){ wait(); }
  accounts[source] -= amount;
  accounts[target] += amount;
  notifyAll();
  return true;
 }

 public synchronized double audit(){
  double balance = 0.0;
  for (int i = 0; i < 100; i++)
    balance += accounts[i];
  return balance;
 }

 public double current_balance(int i){
  return accounts[i];   // not synchronized!
 }
}
```

Monitors in Java

- Monitors incorporated within existing class definitions
- Function declared synchronized is to be executed atomically
- Each object has a lock
  - To execute a synchronized method, thread must acquire lock
  - Thread gives up lock when the method exits
  - Only one thread can have the lock at any time

```
public class bank_account{
 double accounts[100];

 public synchronized boolean
   transfer(double amount, int source, int target){
  while (accounts[source] < amount){ wait(); }
  accounts[source] -= amount;
  accounts[target] += amount;
  notifyAll();
  return true;
 }

 public synchronized double audit(){
  double balance = 0.0;
  for (int i = 0; i < 100; i++)
    balance += accounts[i];
  return balance;
 }

 public double current_balance(int i){
  return accounts[i];   // not synchronized!
 }
}
```

Monitors in Java

- Monitors incorporated within existing class definitions
- Function declared synchronized is to be executed atomically
- Each object has a lock
  - To execute a synchronized method, thread must acquire lock
  - Thread gives up lock when the method exits
  - Only one thread can have the lock at any time
- Wait for lock in external queue

```
public class bank_account{
 double accounts[100];

 public synchronized boolean
   transfer(double amount, int source, int target){
  while (accounts[source] < amount){ wait(); }
  accounts[source] -= amount;
  accounts[target] += amount;
  notifyAll();
  return true;
 }

 public synchronized double audit(){
  double balance = 0.0;
  for (int i = 0; i < 100; i++)
    balance += accounts[i];
  return balance;
 }

 public double current_balance(int i){
  return accounts[i];   // not synchronized!
 }
}
```

Monitors in Java

- wait() and notify() to suspend and resume

```
public class bank_account{
 double accounts[100];

 public synchronized boolean
   transfer(double amount, int source, int target){
  while (accounts[source] < amount){ wait(); }
  accounts[source] -= amount;
  accounts[target] += amount;
  notifyAll();
  return true;
 }

 public synchronized double audit(){
  double balance = 0.0;
  for (int i = 0; i < 100; i++)
    balance += accounts[i];
  return balance;
 }

 public double current_balance(int i){
  return accounts[i];   // not synchronized!
 }
}
```

Monitors in Java

- `wait()` and `notify()` to suspend and resume
- Wait — single internal queue

```
public class bank_account{
 double accounts[100];

 public synchronized boolean
   transfer(double amount, int source, int target){
  while (accounts[source] < amount){ wait(); }
  accounts[source] -= amount;
  accounts[target] += amount;
  notifyAll();
  return true;
 }

 public synchronized double audit(){
  double balance = 0.0;
  for (int i = 0; i < 100; i++)
    balance += accounts[i];
  return balance;
 }

 public double current_balance(int i){
  return accounts[i];   // not synchronized!
 }
}
```

Monitors in Java

- `wait()` and `notify()` to suspend and resume
- Wait — single internal queue
- Notify
  - `notify()` signals one (arbitrary) waiting process
  - `notifyAll()` signals all waiting processes
  - Java uses signal and continue

```
public class bank_account{
 double accounts[100];

 public synchronized boolean
   transfer(double amount, int source, int target){
  while (accounts[source] < amount){ wait(); }
  accounts[source] -= amount;
  accounts[target] += amount;
  notifyAll();
  return true;
 }

 public synchronized double audit(){
  double balance = 0.0;
  for (int i = 0; i < 100; i++)
    balance += accounts[i];
  return balance;
 }

 public double current_balance(int i){
  return accounts[i];   // not synchronized!
 }
}
```

Object locks . . .

- Use object locks to synchronize arbitrary blocks of code

```
public class XYZ{
  Object o = new Object();

  public int f(){
    ..
    synchronized(o){ ... }
  }

  public double g(){
    ..
    synchronized(o){ ... }
    }
  }
}
```

Object locks . . .

- Use object locks to synchronize arbitrary blocks of code
- f() and g() can start in parallel
- Only one of the threads can grab the lock for o

```
public class XYZ{
  Object o = new Object();

  public int f(){
    ..
    synchronized(o){ ... }
  }

  public double g(){
    ..
    synchronized(o){ ... }
    }
  }
}
```

Object locks . . .

- Use object locks to synchronize arbitrary blocks of code
- f() and g() can start in parallel
- Only one of the threads can grab the lock for o
- Each object has its own internal queue

```
Object o = new Object();

public int f(){
  ..
  synchronized(o){
     ...
     o.wait();   // Wait in queue attached to "o"
     ...
  }
}

public double g(){
  ..
  synchronized(o){
     ...
     o.notifyAll();   // Wake up queue attached to
     ...
  }
}
```

Object locks . . .

- Use object locks to synchronize arbitrary blocks of code
- `f()` and `g()` can start in parallel
- Only one of the threads can grab the lock for `o`
- Each object has its own internal queue
- Can convert methods from "externally" synchronized to "internally" synchronized

```
public double h(){
  synchronized(this){
    ...
  }
}
```

Object locks . . .

- Use object locks to synchronize arbitrary blocks of code
- `f()` and `g()` can start in parallel
- Only one of the threads can grab the lock for `o`
- Each object has its own internal queue
- Can convert methods from "externally" synchronized to "internally" synchronized
- "Anonymous" `wait()`, `notify()`, `notifyAll()` abbreviate `this.wait()`, `this.notify()`, `this.notifyAll()`

```
public double h(){
  synchronized(this){
    ...
  }
}
```

Object locks . . .

- Actually, `wait()` can be "interrupted" by an `InterruptedException`

Object locks . . .

- Actually, `wait()` can be “interrupted” by an `InterruptedException`
- Should write

```
try{
  wait();
}
catch (InterruptedException e) {
  ...
};
```

Object locks ...

- Actually, `wait()` can be “interrupted” by an `InterruptedException`
- Should write

```
try{
  wait();
}
catch (InterruptedException e) {
  ...
};
```

- Error to use `wait()`, `notify()`, `notifyAll()` outside synchronized method
  - `IllegalMonitorStateException`

Object locks . . .

- Actually, `wait()` can be "interrupted" by an `InterruptedException`
- Should write

```
try{
  wait();
}
catch (InterruptedException e) {
  ...
};
```

- Error to use `wait()`, `notify()`, `notifyAll()` outside synchronized method
  - `IllegalMonitorStateException`
- Likewise, use `o.wait()`, `o.notify()`, `o.notifyAll()` only in block synchronized on `o`

Reentrant locks

- Separate `ReentrantLock` class

```
public class Bank
{
  private Lock bankLock = new ReentrantLock();
  ...
  public void
     transfer(int from, int to, int amount) {
    bankLock.lock();
    try {
       accounts[from] -= amount;
       accounts[to] += amount;
    }
    finally {
       bankLock.unlock();
    }
  }
}
```

Reentrant locks

- Separate `ReentrantLock` class
- Similar to a semaphore
  - `lock()` is like `P(S)`
  - `unlock()` is like `V(S)`

```
public class Bank
{
  private Lock bankLock = new ReentrantLock();
  ...
  public void
     transfer(int from, int to, int amount) {
    bankLock.lock();
    try {
       accounts[from] -= amount;
       accounts[to] += amount;
    }
    finally {
       bankLock.unlock();
    }
  }
}
```

Reentrant locks

- Separate `ReentrantLock` class
- Similar to a semaphore
  - `lock()` is like `P(S)`
  - `unlock()` is like `V(S)`
- Always `unlock()` in `finally` — avoid abort while holding lock

```
public class Bank
{
  private Lock bankLock = new ReentrantLock();
  ...
  public void
     transfer(int from, int to, int amount) {
    bankLock.lock();
    try {
       accounts[from] -= amount;
       accounts[to] += amount;
    }
    finally {
       bankLock.unlock();
    }
  }
}
```

Reentrant locks

- Separate `ReentrantLock` class
- Similar to a semaphore
  - `lock()` is like `P(S)`
  - `unlock()` is like `V(S)`
- Always `unlock()` in `finally` — avoid abort while holding lock
- Why reentrant?
  - Thread holding lock can reacquire it
  - `transfer()` may call `getBalance()` that also locks `bankLock`
  - Hold count increases with `lock()`, decreases with `unlock()`
  - Lock is available if hold count is 0

```
public class Bank
{
  private Lock bankLock = new ReentrantLock();
  ...
  public void
     transfer(int from, int to, int amount) {
    bankLock.lock();
    try {
       accounts[from] -= amount;
       accounts[to] += amount;
    }
    finally {
       bankLock.unlock();
    }
  }
}
```

Summary

- Every object in Java implicitly has a lock
- Methods tagged `synchronized` are executed atomically
  - Implicitly acquire and release the object's lock
- Associated condition variable, single internal queue
  - `wait()`, `notify()`, `notifyAll()`
- Can synchronize an arbitrary block of code using an object
  - `sycnchronized(o) { ... }`
  - `o.wait()`, `o.notify()`, `o.notifyAll()`
- Reentrant locks work like semaphores