Monitors and Threads in Java

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 15, 7 March 2023

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
- 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
- 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
- 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
- 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
- "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`
- 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
- 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();
    }
  }
}
```

Locks and conditions

- Can associate multiple condition variables with a lock
  - Bounded buffer implemented as circular queue
  - `put()` blocks if buffer is full, `take()` blocks if buffer is empty

```
class BoundedBuffer {
  ...

  final Object[] items = new Object[100];
  int putptr, takeptr, count;

  public void put(Object x) {
    ...
  }

  public Object take() {
    ...
  }
}
```

Locks and conditions

- Can associate multiple condition variables with a lock
  - Bounded buffer implemented as circular queue
  - `put()` blocks if buffer is full, `take()` blocks if buffer is empty
- Java interface `Condition`
  - Methods `await()` and `signal()`
  - Separate conditions to indicate buffer empty and buffer full

```
class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  final Object[] items = new Object[100];
  int putptr, takeptr, count;

  public void put(Object x) {
    ...
  }

  public Object take() {
    ...
  }
}
```

Locks and conditions

- Can associate multiple condition variables with a lock
  - Bounded buffer implemented as circular queue
  - `put()` blocks if buffer is full, `take()` blocks if buffer is empty
- Java interface `Condition`
  - Methods `await()` and `signal()`
  - Separate conditions to indicate buffer empty and buffer full
- `put()` awaits `notFull`, signals `notEmpty`

```
class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final Object[] items = new Object[100];
  int putptr, takeptr, count;

  public void put(Object x) throws InterruptedException {
    lock.lock();
    try {
      while (count == items.length)
        notFull.await();
      ### Add an item to the buffer
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }

  public Object take() {
    ...
  }
}
```

Locks and conditions

- Can associate multiple condition variables with a lock
  - Bounded buffer implemented as circular queue
  - `put()` blocks if buffer is full, `take()` blocks if buffer is empty
- Java interface `Condition`
  - Methods `await()` and `signal()`
  - Separate conditions to indicate buffer empty and buffer full
- `put()` awaits `notFull`, signals `notEmpty`
- `take()` awaits `notEmpty`, signals `notFull`

```
class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final Object[] items = new Object[100];
  int putptr, takeptr, count;

  public void put(Object x) throws InterruptedException {
    ...
  }

  public Object take() throws InterruptedException {
    lock.lock();
    try {
      while (count == 0)
        notEmpty.await();
      ### Remove an item x from the buffer
      notFull.signal();
      return x;
    } finally {
      lock.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
  - Can attach multiple condition variables to a lock, `await()` and `signal()`

Creating threads in Java

- Have a class extend `Thread`
- Define a function `run()` where execution can begin in parallel
- Invoking `p[i].start()` initiates `p[i].run()` in a separate thread
  - Directly calling `p[i].run()` does not execute in separate thread!
- `sleep(t)` suspends thread for `t` milliseconds
  - Static function — use `Thread.sleep()` if current class does not extend `Thread`
  - Throws `InterruptedException` — later

```
public class Parallel extends Thread{
  private int id;
  public Parallel(int i){ id = i; }
  public void run(){
    for (int j = 0; j < 100; j++){
      System.out.println("My id is "+id);
      try{
        sleep(1000);        // Sleep for 1000 ms
      }
      catch(InterruptedException e){}
    }
  }
}

public class TestParallel {
  public static void main(String[] args){
    Parallel p[] = new Parallel[5];
    for (int i = 0; i < 5; i++){
       p[i] = new Parallel(i);
       p[i].start();  // Start p[i].run()
    }                 // in concurrent thread
  }
}
```

Java threads . . .

- Cannot always extend `Thread`
  - Single inheritance
- Instead, implement `Runnable`
- To use `Runnable` class, explicitly create a `Thread` and `start()` it

```
public class Parallel implements Runnable{
  // only the line above has changed
  private int id;
  public Parallel(int i){ ... } // Constructor
  public void run(){ ... }

}

public class TestParallel {
  public static void main(String[] args){
    Parallel p[] = new Parallel[5];
    Thread t[]   = new Thread[5];

    for (int i = 0; i < 5; i++){
       p[i] = new Parallel(i);
       t[i] = new Thread(p[i]);
             // Make a thread t[i] from p[i]
       t[i].start();  // Start off p[i].run()
                      // Note: t[i].start(),
    }                 //   not p[i].start()
  }
}
```

Life cycle of a Java thread

A thread can be in six states — thread status via `t.getState()`

- New: Created but not `start()`ed.
- Runnable: `start()`ed and ready to be scheduled.
  - Need not be actually "running"
  - No guarantee made about how scheduling is done
  - Most Java implementations use time-slicing
- Not available to run
  - Blocked — waiting for a lock, unblocked when lock is granted
  - Waiting — suspended by `wait()`, unblocked by `notify()` or `notfifyAll()`
  - Timed wait — within `sleep(..)`, released when sleep timer expires
- Dead: thread terminates.

Interrupts

- One thread can interrupt another using `interrupt()`
  - `p[i].interrupt();` interrupts thread `p[i]`
- Raises `InterruptedException` within `wait()`, `sleep()`
- No exception raised if thread is running!
  - `interrupt()` sets a status flag
  - `interrupted()` checks interrupt status and clears the flag
- Detecting an interrupt while running or waiting

```
public void run(){
  try{
    j = 0;
    while(!interrupted() && j < 100){
      System.out.println("My id is "+id);
      sleep(1000);   // Sleep for 1000 ms
      j++;
    }
  }
  catch(InterruptedException e){}
}
```

More about threads . . .

- Check a thread's interrupt status
  - Use `t.isInterrupted()` to check status of `t`'s interrupt flag
  - Does not clear flag
- Can give up running status
  - `yield()` gives up active state to another thread
  - Static method in `Thread`
  - Normally, scheduling of threads is handled by OS — preemptive
  - Some mobile platforms use cooperative scheduling — thread loses control only if it yields
- Waiting for other threads
  - `t.join()` waits for `t` to terminate

Summary

- To run in parallel, need to extend `Thread` or implement `Runnable`
  - When implmenting `Runnable`, first create a `Thread` from `Runnable` object
- `t.start()` invokes method `run()` in parallel
- Threads can become inactive for different reasons
  - Block waiting for a lock
  - Wait in internal queue for a condition to be notified
  - Wait for a sleep timer to elapse
- Threads can be interrupted
  - Be careful to check both `interrupted` status and handle `InterruptException`
- Can yield control, or wait for another thread to terminate