Monitors in Java

Type	📒 Lecture
Date	@March 6, 2022
Lecture #	1
Lecture URL	https://youtu.be/Srz0mewFaEg
Notion URL	https://21f1003586.notion.site/Monitors-in-Java-5f68cc1c54cf4033ade01d2bc87bb9c2
Week #	11

Monitors

- Monitor is like a class in an OO languages
  - 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 assocaited with target
    return true;
  }

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

- 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

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
    }
}
```

- 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 the lock in external queue
- `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

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) { ... }
    }
}
```

- 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 "o"
        ...
    }
}
```

- Can convert methods from “externally” synchronized to “internally” synchronized

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

- “Anonymous” `wait()`, `notify()`, `notifyAll()` abbreviate `this.wait()`, `this.notify()`, `this.notifyAll()`

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

```
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();
        }
    }
}
```

- 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

Summary

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

Thread in Java

Type	📒 Lecture
Date	@March 6, 2022
Lecture #	2
Lecture URL	https://youtu.be/htwvFDP5t5I
Notion URL	https://21f1003586.notion.site/Thread-in-Java-ae49acbc0e1342c6923cb09d9dc00572
Week #	11

Creating threads in Java

- Have a class extend `Thread`
- Define a function `run()` where exeuction 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`

```
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 1000ms
            } 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
        }
    }
}
```

Typical output

```
My id is 0
My id is 3
My id is 2
My id is 1
My id is 4
My id is 0
My id is 2
My id is 3
My id is 4
My id is 1
My id is 0
My id is 3
My id is 1
My id is 2
My id is 4
My id is 0
...
```

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 {
    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 1000ms
            } catch(InterruptedException e) {}
        }
    }
}

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]
            // Start off p[i].run()
            t[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 `notifyAll()`
  - 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 the 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);
            j++;
        }
    } catch(InterruptedException e) {}
}
```

More about threads

- Check a thead'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 implementing `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

Concurrency Programming

Type	📒 Lecture
Date	@March 6, 2022
Lecture #	3
Lecture URL	https://youtu.be/yI_WSIaFbFQ
Notion URL	https://21f1003586.notion.site/Concurrency-Programming-7a9088ef19fd46c187bc5f562405aa56
Week #	11

An exercise in concurrent programming

- A narrow North-South bridge can accomodate traffic only in one direction at a time
- When a car arrives at the bridge
  - Cars on the bridge going in the same direction ⇒ can cross
  - No other car on the bridge ⇒ can cross (implicitly sets direction)
  - Cars on the bridge going in opposite direction ⇒ wait for the bridge to be empty
- Cars waiting to cross from one side may enter bridge in any order after direction switches in their favour
- When bridge becomes empty and cars are waiting, yet another car can enter in the opposite direction and makes them all wait some more

An example

- Design a class `Bridge` to implement consistent one-way access for cars on the highway
  - Should permit multiple cars to be on the same bridge at one time (all going in the same direction)
- `Bridge` has a public method `public void cross (int id, boolean d, int s)`
  - `id` is the indentity of the car
  - `d` indicates direction
    - `true` is North
    - `false` is South
  - `s` indicates time taken to cross (milliseconds)

`public void cross(int id, boolean d, int s)`

- Method `cross` prints out diagnostics
  - A car is stuck waiting for the direction to change

    `Car 10 going South stuck at Fri Feb 25 12:42:13 IST 2022`
  - The direction changes

    `Car 10 switches bridge direction to South at Fri Feb 25 12:42:13 IST 2022`
  - A car enters the bridge

    `Car 10 going South enters bridge at Fri Feb 25 12:42:13 IST 2022`
  - A car leaves the bridge

    `Car 10 leaves at Fri Feb 25 12:42:14 IST 2022`

Analysis

- The “data” that is shared is the `Bridge`
- State of the bridge is represented by two quantities
  - Number of cars on the bridge — `int bcount`
  - Current direction of bridge — `boolean direction`
- The method `public void cross(int id, boolean d, int s)` changes the state of the bridge
  - Concurrent execution of `cross` can cause problems
- but making `cross` a synchronized method is too restrictive
  - Only one car on the bridge at a time
  - Problem description explicitly disallows such a solution
- Break up `cross` into a sequence of actions

    - `enter` — get on the bridge
    - `travel` — drive across the bridge
    - `leave` — get off the bridge
    - `enter` and `leave` can print out the diagnostics required
- Which of these affect the state of the bridge?
    - `enter` → increment the number of cars, perhaps change the direction
    - `leave` → decrement the number of cars
- Make `enter` and `leave` synchronized
- `travel` is just a means to let time elapse — use `sleep`

Code for `cross`

```
public void cross(int id, boolean d, int s) {
    // Get onto the bridge (if you can)
    enter(id, d);

    // Takes time to cross the bridge
    try {
        Thread.sleep(s);
    } catch(InterruptedException e) {}

    // Get off the bridge
    leave(id);
}
```

Entering the bridge

- If the direction of this car matches the direction of the bridge, it can enter
- If the direction does not match but the number of cars is > 0, wait
- Otherwise, `wait()` for the state of the bridge to change

Code for `enter`

```
private synchronized void enter(int id, boolean d) {
    Date date;

    // While there are cars going in the wrong direction
    while(d != direction && bcount > 0) {
        date = new Date();
        System.out.println("Card " + id + " going " + direction_name(d) + " stuck at " + date);

        // Wait for our turn
        try {
            wait();
        } catch(InterruptedException e) {}
    }
```

```
    ...
}
```

```
private synchronized void enter(int id, boolean d) {
    ...
    while(d != direction && bcount > 0) { ... wait() ...}
    ...
    if(d != direction) {    // Switch direction, if needed
        direction = d;
        date = new Date();
        System.out.println("Card " + id + " switches bridge direction to "
            + direction_name(direction) + " at " + date);
    }

    bcount++;   // Register our presence on the bridge
    date = new Date();
    System.out.println("Car " + id + " going " + direction_name(d) + " enter bridge at" + date);
}
```

Code for `leave`

Leaving the bridge is much simpler

- Decrease the car count
- `notify()` waiting cars .. provided car count is zero

```
private synchronized void leave(int id) {
    Date date = new Date();
    System.out.println("Car " + id + " left at " + date);

    // Check out
    bcount--;

    // If everyone on the bridge has checked out
    // Notify the cars waiting on the opposite side
    if(bcount == 0) {
        notifyAll();
    }
}
```

Summary

- Concurrent programming can be tricky
- Need to synchronize access to shared resources
- while allowing concurrency

Thread Safe Collection

Type	📒 Lecture
Date	@March 6, 2022
Lecture #	4
Lecture URL	https://youtu.be/yDmnozJBKic
Notion URL	https://21f1003586.notion.site/Thread-Safe-Collection-56a0ffd9d4cb4f31a0d70b2472f85078
Week #	11

Concurrency and collections

```
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 the balance across all accounts
        double balance = 0.00;

        for(int i = 0; i < 100; ++i) {
            balance += accounts[i];
        }

        return balance;
    }
}
```

- Synchronize access to bank account array to ensure consistent updates
- Non-interfering updates can safely happen in parallel
  - Updates to different accounts, `accounts[i]` and `accounts[j]`
- Insistence on sequential access affects performance

Thread safety and correctness

- Thread safety guarantees consistency of individual updates
- If two threads increment `accounts[i]` neither update is lost
- Individual updates are implemented in an atomic manner
- Does not say anything about sequences of updates
- Formally, linearizability
- Contrast with serielizability in databases, where transactions (sequences of updates) appear atomic

Threads safe collections

- To implement threads safe collections, use locks to make local updates atomic
- Granularity of locking depends on data structure
  - In an array, sufficient to protect `a[i]`
  - In a linked list, restrict access to nodes on either side of insert/delete
- Java provides built-in collection types that are thread safe
  - `ConcurrentMap` interface, implemented as `ConcurrentHashMap`

    - `BlockingQueue`, `ConcurrentSkipList`
    - Appropriate low level locking is done automatically to ensure consistent local updates
- Remember that these only guarantee atomicity of individual updates
- Sequences of updates (transfer from one account to another) still need to be manually synchronized to work properly

Using thread safe queues for synchronization

- Use a thread safe queue for simpler synchronization of shared objects
- Producer-Consumer system
    - Producer threads insert items into the queue
    - Consumer threads retrieve them
- Bank account example
    - Transfer threads insert transfer instructions into shared queue
    - Update thread processes instructions from the queue, modifies the bank account
    - Only the update thread modifies the data structure
    - No synchronization necessary

Blocking queues

- Blocking queues block when ...
    - we try to add an element when the queue is full
    - we try to remove an element when the queue is empty
- Update thread tries to remove an item to process, waits if nothing is available
- In general, use blocking queues to coordinate multiple producer and consumer threads
    - Producers write intermediate results into the queue
    - Consumers retrieve these results and make further updates
- Blocking automatically balances the workload
    - Producers wait if consumers are slow and the queue fills up

    - Consumers wait if producers are slow to provide items to process

Summary

- When updating collections, locking the entire data structure for individual updates is wasteful
- Sufficient to protect access within a local portion of the structure
    - Ensure that two updates do not overlap
    - Region to protect depends on the type of collection
    - Implement using lower level locks of suitable granularity
- Java provides built-in thread safe collections
- One of these is a blocking queue
    - Use a blocking queue to coordinate producers and consumers
    - Ensure safe access to a shared data structure without explicit synchronization