Resources

• Doug Lea
  • Java Concurrency in Practice
• Maurice Herlihy & Nir Shavit
  • The Art of Multiprocessor Programming
Concurrency
Concurrency

• Why?
  • availability – minimize response lag, maximize throughput
  • modeling – simulating autonomous objects, animation
  • parallelism – exploit multiprocessors, overlapping I/O
  • protection – isolating activities in threads
Concurrency

• Why?
  • availability – minimize response lag, maximize throughput
  • modeling – simulating autonomous objects, animation
  • parallelism – exploit multiprocessors, overlapping I/O
  • protection – isolating activities in threads

• Why not?
  • complexity – dealing with safety, liveness, composition
  • overhead – higher resource usage
Java concurrency

- Active objects – threads
  - `java.lang.Thread`
  - `run()` method

- Passive objects
  - participate in multiple threads
  - can protect themselves from conflicting activities (synchronized)
  - can create and control new threads
Hardware mappings

• Shared memory multiprocessing
  • all objects visible in same machine
  • procedural message passing (method calls)
  • many more threads than CPUs

• Remote message passing
  • object access objects via remote references or copying
  • must marshal (serialize) messages
Design forces

- Forces that must be addressed at each level of design of a concurrent system
  - safety – “bad things don’t happen”
  - liveness – “good things eventually happen”
  - efficiency – performance
  - reusability – compositionality
Safety

• Want to prevent interference between concurrent activities
  • bad things do not happen

• problems:
  • storage conflicts, race conditions
  • transaction violations
  • inconsistent data, stale values, partially initialized objects
Liveness

- ensure activities make progress
  - good things eventually happen

- problems:
  - deadlock
  - livelock
  - starvation
  - lack of fairness
Java threads

- **Thread** class represents state of an independent activity
  - methods to start, sleep, etc.
  - weak scheduling guarantees
  - code that runs in a thread defined by Runnable interface
Thread creation

• Increment an array, in parallel:

```java
for (i = 0; i < A.length; i++) {
    th[i] = new Thread() {
        public void run() {
            A[i]++;
        }
    };
    th[i].start();
}
for (i = 0; i < A.length; i++) {
    th[i].join();
}
```
Problems

- Threads are very high overhead
- Creates a kernel data structure
- Allocates a call stack (100s of KB)

- Weak fairness guarantees
- Tedious to manage threads explicitly
Concurrency in new languages

- Newer languages support lighter-weight concurrency

- X10:

```java
foreach ((i) in A) {
    A(i)++;
}
```

- Loop executes in parallel.
• X10
  • new OO programming language from IBM
  • designed for high-performance computing, multicore

• Fine-grained concurrency
• Distribution
  • *Partitioned* global address space
• Atomicity operations
X10

• http://x10-lang.org

• Current version 2.0.3
• Language still under development

• Compiles to Java
  • slow!
• Compiles to C++ on top of MPI, LAPI, sockets
  • fast!
Basic X10 concurrency

- Three constructs:
  - `async S` – spawn a new activity
  - `finish S` – wait until all activities in $S$ terminate

- `async` and `finish` similar to `spawn` and `sync` in Cilk
Funicular [this name will change!]

- X10 concurrency features as a Scala library
- http://ranger.uta.edu/~nystrom/funicular
- Examples below use Scala syntax
- To use:
  - import funicular.Intrinsics._
Simple concurrent summation

```scala
val sums = Array.ofDim[Int](2);
finish {
  async { sums(0) = sum(1, 50, i => i*i) }
  async { sums(1) = sum(51, 100, i => i*i) }
}
val result = sums(0) + sums(1)
println(result)

// sum f(lo)..f(hi)
def sum(lo: Int, hi: Int, f: Int => Int) =
  (lo to hi).foldLeft(0)((x,y) => x + f(y))
```

Monday, May 3, 2010
Async

- `async S` – spawn a new activity
  - much lighter-weight than creating a thread
  - both syntactically and performance-wise
Finish

• `finish S` – wait until all activities spawned in S terminate

• Also collects any exceptions thrown by spawned activities
Deadlock

• Using `async` and `finish` guarantees no deadlock

• A parent activity can wait on spawned children
• But, children can’t wait on parent
• No wait-cycle => no deadlock
Implementation
Thread pool scheduler

• How to map activities to threads?

• Idea:
  • Maintain a fixed size thread pool

• Queue of activities to execute:
  • `async $ => add $ to queue`

• When thread is idle:
  • dequeue activity from queue and run it
Continuations

- Activities in the queue are continuations!
- saved control context to be invoked later
thread 1 finish { // task 1
    a();
async { // task 2
      b();
mental { // task 3
        c();
    }
    d();
}
async { // task 4
    e();
}
async { // task 5
    f();
}
g();
}
Thread pool scheduler

```java
finish {       // task 1
    a();
    async {    // task 2
        b();
        async {   // task 3
            c();
        }
        d();
    }
    async {    // task 4
        e();
    }
    async {    // task 5
        f();
    }
    g();
}
```
Thread pool scheduler

```javascript
finish { // task 1
  a();
  async { // task 2
    b();
    async { // task 3
      c();
    }
    d();
  }
  async { // task 4
    e();
  }
  async { // task 5
    f();
  }
  g();
}
```
Thread pool scheduler

```java
finish { // task 1
    a();
    async { // task 2
        b();
        async { // task 3
            c();
        }
        d();
    }
    async { // task 4
        e();
    }
    async { // task 5
        f();
    }
    g();
}
```

Monday, May 3, 2010
Thread pool scheduler

```javascript
finish {  // task 1
    a();
    async {  // task 2
        b();
        async {  // task 3
            c();
        }
        d();
    }
    async {  // task 4
        e();
    }
    async {  // task 5
        f();
    }
    g();
}
```

thread 1 ➔ thread 2

Monday, May 3, 2010
Thread pool scheduler

```javascript
finish {     // task 1
    a();
    async {    // task 2
        b();
        async {    // task 3
            c();
        }
        d();
    }
    async {    // task 4
        e();
    }
    async {    // task 5
        f();
    }
    g();
}
```

Monday, May 3, 2010
Thread pool scheduler

```javascript
finish {   // task 1
    a();
    async {   // task 2
        b();
        async {   // task 3
            c();
        }
        d();
    }
    async {   // task 4
        e();
    }
    async {   // task 5
        f();
    }
    g();
}
```
Thread pool scheduler

```javascript
finish { // task 1
    a();
    async { // task 2
        b();
        async { // task 3
            c();
        }
        d();
    }
    async { // task 4
        e();
    }
    async { // task 5
        f();
    }
    g();
}
```

thread 1 ➔ thread 2

Monday, May 3, 2010
Thread pool scheduler

```java
finish { // task 1
    a();
    async { // task 2
        b();
        async { // task 3
            c();
        }
        d();
    }
    async { // task 4
        e();
    }
    async { // task 5
        f();
    }
    g();
}
```

Monday, May 3, 2010
Thread pool scheduler

```java
finish { // task 1
    a();
    async { // task 2
        b();
        async { // task 3
            c();
        }
        d();
    }
    async { // task 4
        e();
    }
    async { // task 5
        f();
    }
    g();
}
```

thread 1

thread 2
Thread pool scheduler

```javascript
finish {     // task 1
  a();
  async {    // task 2
    b();
    async {   // task 3
      c();
    }
    d();
  }
  async {    // task 4
    e();
  }
  async {    // task 5
    f();
  }
  g();
}
```

Thread 1

Thread 2
Thread pool scheduler

```javascript
finish {     // task 1
    a();
    async {    // task 2
        b();
        async {  // task 3
            c();
        }
        d();
    }
    async {    // task 4
        e();
    }
    async {    // task 5
        f();
    }
    g();
}
```

thread 1

thread 2

Monday, May 3, 2010
Thread pool scheduler

```javascript
finish { // task 1
    a();
    async { // task 2
        b();
        async { // task 3
            c();
        }
        d();
    }
    async { // task 4
        e();
    }
    async { // task 5
        f();
    }
}
g();
```

Monday, May 3, 2010
Thread pool scheduler

```plaintext
finish {     // task 1
    a();
    async {    // task 2
        b();
        async {  // task 3
            c();
        }
        d();
    }
    async {    // task 4
        e();
    }
    async {    // task 5
        f();
    }
    g();
}
```

thread 1 → thread 2
Thread pool scheduler

```javascript
finish { // task 1
    a();
    async { // task 2
        b();
        async { // task 3
            c();
        }
        d();
    }
    async { // task 4
        e();
    }
    async { // task 5
        f();
    }
    g();
}
```

thread 2
Thread pool scheduler

```java
finish { // task 1
    a();
    async { // task 2
        b();
        async { // task 3
            c();
        }
        d();
    }
    async { // task 4
        e();
    }
    async { // task 5
        f();
    }
    g();
}
```
Thread pool scheduler

```
finish {  // task 1
    a();
    async {  // task 2
        b();
        async {  // task 3
            c();
        }
        d();
    }
    async {  // task 4
        e();
    }
    async {  // task 5
        f();
    }
    g();
}
```

thread 1
Thread pool scheduler

```c
finish { // task 1
    a();
    async { // task 2
        b();
        async { // task 3
            c();
        }
        d();
    }
    async { // task 4
        e();
    }
    async { // task 5
        f();
    }
    g();
}
```
Thread pool scheduler

• One queue for all threads
  • => lots of contention for the head of the queue

• Very bad performance
Work-stealing scheduler

- Pool of threads of fixed size
  - each thread maintains a deque (double-ended queue) of pending activities
  - \texttt{async \ S \Rightarrow add \ S \ to \ bottom \ of \ deque}
- When idle:
  - pop activity from bottom of deque
  - if deque empty, steal from top of deque of random thread
Work stealing scheduling

```cpp
thread 1 finish { // task 1
  a();
  async { // task 2
    b();
    async { // task 3
      c();
    }
    d();
  }
  async { // task 4
    e();
  }
  async { // task 5
    f();
  }
  g();
}
```

Monday, May 3, 2010
Work stealing scheduling

```javascript
thread 1

finish { // task 1
    a();
    async { // task 2
        b();
        async { // task 3
            c();
        }
        d();
    }
    async { // task 4
        e();
    }
    async { // task 5
        f();
    }
    g();
}
```
Work stealing scheduling

```javascript
finish {  // task 1
    a();
    async {    // task 2
        b();
        async {    // task 3
            c();
        }
        d();
    }
    async {    // task 4
        e();
    }
    async {    // task 5
        f();
    }
    g();
}
```
Work stealing scheduling

```java
finish {}  // task 1
    a();
    async {}  // task 2
        b();
        async {}  // task 3
            c();
        }
    d();
}
async {}  // task 4
    e();
}
async {}  // task 5
    f();
}
g();
// thread 1
// thread 2
```
Work stealing scheduling

```cpp
finish { // task 1
  a();
  async { // task 2
    b();
    async { // task 3
      c();
    }
    d();
  }
  async { // task 4
    e();
  }
  async { // task 5
    f();
  }
  g();
}
```

thread 1 ➠ thread 2
Work stealing scheduling

```javascript
finish { // task 1
    a();
    async { // task 2
        b();
        async { // task 3
            c();
        }
        d();
    }
    async { // task 4
        e();
    }
    async { // task 5
        f();
    }
}
g();
```

Monday, May 3, 2010
Work stealing scheduling

```javascript
finish { // task 1
    a();
    async { // task 2
        b();
        async { // task 3
            c();
        }
        d();
    }
    async { // task 4
        e();
    }
    async { // task 5
        f();
    }
}
g();
```

thread 1

thread 2
Work stealing scheduling

```cpp
finish { // task 1
    a();
    async { // task 2
        b();
        async { // task 3
            c();
        }
        d();
    }
    async { // task 4
        e();
    }
    async { // task 5
        f();
    }
    g();
}
```
Work stealing scheduling

```c
finish { // task 1
    a();
    async { // task 2
        b();
        async { // task 3
            c();
        }
        d();
    }
    async { // task 4
        e();
    }
    async { // task 5
        f();
    }
    g();
}
```

Monday, May 3, 2010
Work stealing scheduling

```
finish {     // task 1
  a();
  async {    // task 2
    b();
    async {    // task 3
      c();
    }
    d();
  }
  async {    // task 4
    e();
  }
  async {    // task 5
    f();
  }
  g();
}
```
Work stealing scheduling

```javascript
    finish {  // task 1
        a();
        async {  // task 2
            b();
            async {  // task 3
                c();
            }    // task 4
            d();
        }    // task 5
        async {  // task 4
            e();
        }
        async {  // task 4
            f();
        }
    }
    g();
```
Work stealing

• By giving each thread its own queue, avoid contention for the common queue

• Use double ended queue to avoid contention for the end of the queue

• Active thread uses one end of the queue

• Thief uses the other end of the queue

• Active thread is never blocked waiting on the queue
class Location {
    private double x, y;
    Location(double x, double y) { this.x = x; this.y = y; }
    synchronized double x() { return x; }
    double y() {
        synchronized (this) {
            return y;
        }
    }
    synchronized void move(double dx, double dy) {
        x += dx;
        y += dy;
    }
}
Java locks

• Every Java object possesses one lock
• manipulated only via `synchronized` keyword
• Class objects contain a lock used to protect statics
• Scalars like int are not Objects; must lock enclosing Object

• `synchronized` can be either a method or block qualifier
  
synchronized void f() { ... }
  ==

  void f() { synchronized (this) { ... } }
Java locks

- Java locks are reentrant
  - a thread hitting synchronized passes if the lock is free or if it already holds the lock, else blocks

- released after passing as many }s as {s for the lock—cannot forget to release the lock

- Synchronized also has side-effect of clearing locally cached values and forcing reloads from main memory
Storage conflicts

class Even {
    int n = 0;
    public int next() {
        // POST: n is always even
        ++n;
        ++n;
        return n;
    }
}

• Postcondition may fail due to storage conflicts
Locks and caching

• Locking generates traffic between threads and memory
  • Lock acquire forces reads from memory to cache
  • Lock release forces writes of cached updates to memory

• Without locking, there are no promises about if and when caches will be flushed or reloaded
  • without locking, a thread might never observe another thread writes
  • can lead to unsafe, nonsensical execution
Memory anomalies

• Should **acquire lock before use** of any field of any objects

• Should **release after update**

• If not, then might:
  • see **stale values** that do not reflect recent writes
  • see **inconsistent states** due to out-of-order writes
  • see **incompletely initialized** objects
class Cell {
    private long value;
    synchronized long get() { return value; }
    synchronized void set(long v) { value = v; }
    synchronized void swap(Cell o) {
        long t = get();
        long v = other.get();
        set(v);
        other.set(t);
    }
}

swap is a **transactional** method.
Can **deadlock**.
Determinate concurrency

- This is what to aim for
- Order of execution does not change result
- X10 does not guarantee determinancy
  - but can often get it
Implementing synchronized

`synchronized` implemented with reentrant locks

```java
synchronized (p) { ... }
==
p.lock.lock();
try { ... }
finally { p.lock.unlock(); }
```
A simple lock

• boolean state

• test-and-set (TAS) lock:
  • lock:
    while (state.getAndSet(true))   // atomically
  ;

• unlock:
  state = false;
Atomic get and set

• Implement atomic getAndSet with compare-and-swap instruction

• In Java: use AtomicBoolean.getAndSet
A better lock

• test-and-test-and-set (TTAS) lock:
  • lock:
    while (true) {
      while (state.get())
        ;
      if (! state.getAndSet(true)) // atomically
        return;
    }
  • unlock:
    state = false;
Refinements

• Exponential backoff if getAndSet fails

• Maintain a queue of threads waiting on the lock
  • enforces fairness
var v = 0
for (i <- 1 to 100 async) {
    v += i
}

Atomicity fail
Race condition: concurrent accesses to v.

```java
var v = 0
for (i <- 1 to 100 async) {
    v += i
}
```
Atomicity good

```java
var v = 0
for (i <- 1 to 100 async) {
    atomic {
        v += i
    }
}
```
Atomicity good

Use `atomic` to guarantee only one activity at a time will execute code.

```scala
var v = 0
for (i <- 1 to 100 async) {
    atomic {
        v += i
    }
}
```
Atomic

• Simpler concurrency control mechanism:
  • `atomic S` – run $S$ without interruption
Atomic vs. synchronized

• Compare:

```java
synchronized (this) {
    v += i
}
```

• vs.

```java
atomic {
    v += i
}
```

• No need to specify which lock to acquire.
Atomicity

• Most operations are not atomic
  • atomicity is expensive

• Use atomic sparingly
Atomic and deadlock

• Can deadlock if code in an atomic statement can block

• X10 prevents this from occurring
• Cannot perform blocking operations within an atomic statement
  • e.g., I/O

• Funicular: no checking that atomic statements do not cause deadlock
  • A library function, not a language feature
var v = new AtomicInteger
for (i <- 1 to 100 async) {
    v.getAndAdd(i)
}

- Use built-in atomic types for faster atomic operations
- Not as “clean” as `atomic`, though.
Atomic implementation

- In X10:
  - use one global lock
  - (actually one lock per place)
Transactional memory

- Treat sequence of loads and stores as a transaction

- Implemented using optimistic concurrency control

- Can be implemented as a library, but better to integrate into the language as atomic blocks
• Concurrent activities modify shared memory without regard to other activities

• At end of atomic block, verify that there was no interference

• If no interference, good (commit)
• If interference, roll back changes and redo transaction (abort)
TM benefits

• No thread waits for access to a resource
• No deadlock
• Composition
  • can nest another transaction in another
TM implementation

• Many alternatives, but two main approaches

  • do writes in-place
  • log operations
  • on commit, discard log
  • on abort, traverse log to undo effects of transaction

• do writes on a copy
• on abort, discard copy
• on commit, apply writes to memory
Concurrency Utilities

- Concurrency building blocks added in Java5
  - library for concurrency analogous to collections for data

- Goal: enhance scalability, performance, readability, thread safety of Java programs
- Goal: beat C performance for server apps
Concurrency utils: JSR 166

- Task scheduling framework
  - Executor interface, replaces direct use of Thread
- Callable and Future
- Synchronizers
  - Semaphore, CyclicBarrier, CountDownLatch
- Concurrent Collections
  - BlockingQueue
- Lock
- Atomic
Executor

• No direct Thread invocation

  • myExecutor.execute(aRunnable)

  • not new Thread(aRunnable).start()
Executor

```java
public interface Executor {
    void execute(Runnable cmd);
}

public interface ExecutorService extends Executor {
    ...
}

public class Executors {
    // factory methods
    static ExecutorService newFixedThreadPool(int n);
}

Executor pool = Executors.newFixedThreadPool(5);
pool.execute(runnable);
```
Thread pool example

class WebService {
    public static void main(String[] a) {
        Executor pool = Executors.newFixedThreadPool(5);
        ServerSocket socket = new ServerSocket(8080);
        while (true) {
            final Socket conn = socket.accept();
            final Runnable task = new Runnable() {
                public void run() {
                    new Handler().process(conn);
                }
            };
            pool.execute(task);
        }
    }
}
ScheduledExecutorService

- `schedule()`
  - run once after a fixed delay
- `scheduleAtFixedRate()`
  - repeat at fixed period
- `scheduleWithFixedDelay()`
  - repeat at fixed period with fixed delay between runs

- Operations return `ScheduleFuture`
  - supports `cancel()` operation
Synchronized

- Control access to a shared resource
- Ex: bank account

```java
synchronized double getBalance() {
    Account acct = verify(name, password);
    return acct.balance;
}

synchronized double getBalance() {
    synchronized (this) {
        Account acct = verify(name, password);
        return acct.balance;
    }
}

synchronized double getBalance() {
    Account acct = verify(name, password);
    synchronized (acct) {
        return acct.balance;
    }
}
```

Lock held for a long time.

Same as above.

Better. Only acct object is locked, and for a shorter time.
Locks

• Java provides basic locking via `synchronized`
• Good for many situations, but:
  • single monitor per object
  • not possible to interrupt thread waiting for lock
  • not possible to timeout when waiting for lock
  • block structure
    • difficult to acquire multiple locks

• Solution: Lock interface
interface Lock {
    void lock();
    void lockInterruptably();
    boolean tryLock();
    void unlock();
    Condition newCondition();
}
ReentrantLock

• Simplest concrete implementation of Lock
• Same semantics as synchronized, but with more features
• Generally better performance under contention than synchronized
• Multiple wait-sets supported using Condition interface

• But remember: Lock is not automatically released
Lock example

```java
Lock lock = new ReentrantLock();

public void accessResource() {
    lock.lock();
    try {
        ... do something ...
    } finally {
        lock.unlock();
    }
}
```
ReadWriteLock

- Has two locks controlling read and write access
- Multiple threads can acquire the read lock if no thread has the write lock
- Only one thread can acquire the write lock
- Better performance for read-mostly data access
ReadWriteLock

ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
Lock rl = rwl.readLock();
Lock wl = rwl.writeLock();
ArrayList<String> data = new ArrayList<String>();

public String getData(int pos) {
    rl.lock();
    try { return data.get(pos); } 
    finally { rl.unlock(); } 
}

public String addData(int pos, String value) {
    wl.lock();
    try { data.add(pos, value); } 
    finally { wl.unlock(); } 
}
Synchronizers

- Semaphore
  - manages fixed size pool of resources
- CountDownLatch
  - one or more threads wait for a set of threads to complete an action
- CyclicBarrier
  - set of threads wait until all reach a specified point
- Exchanger
  - two threads reach a fixed point and exchange data
BlockingQueue

• Provides thread-safe way for multiple threads to manipulate collection

interface BlockingQueue<E> {  
    void put(E e);
    boolean offer(E o);
    E take();
    E poll();
    int drainTo(Collection<? super E> c);
    ...
}
BlockingQueue implementations

• ArrayBlockingQueue
  • bounded queue, backed by array, FIFO
• LinkedBlockingQueue
  • optionally bounded queue, backed by linked nodes, FIFO
• PriorityBlockingQueue
  • unbounded queue
  • uses comparator or natural ordering (Comparable) to determine the order of the queue
private ArrayBlockingQueue mq;

public Logger(BlockingQueue<String> q) { mq = q; }

public void run() {
    try {
        while (true) {
            String msg = mq.take(); // blocks if queue is empty
            ... // do something
        }
    } catch (InterruptedException e) { } }
}
private ArrayBlockingQueue mq = new ArrayBlockingQueue<String>(10);
Logger logger = new Logger(mq);

public void run() {
    String msg;
    try {
        while (true) {
            ... // do something
            mq.put(msg); // blocks if queue is full
        }
    } catch (InterruptedException e) { }
}
Concurrent collections

- ConcurrentMap (interface)
  - extends Map interface with atomic operations
- ConcurrentHashMap
  - fully concurrent retrieval
  - tunable concurrency for updates
    - constructor takes #expected concurrent threads
- ConcurrentLinkedQueue
  - unbounded, thread-safe queue, FIFO
- CopyOnWriteArrayList
  - optimized for frequent iteration, infrequent modification
ConcurrentHashMap vs. synchronized

![Graph comparing scalability of Map implementations](image)

**Figure 11.3.** Comparing scalability of Map implementations.
Summary

• New concurrency features very powerful
  • not as simple as synchronized, but much better performance
• Take some time to learn them
• Use them