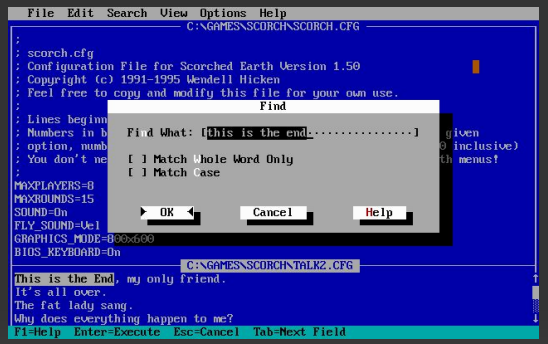
Java Concurrency

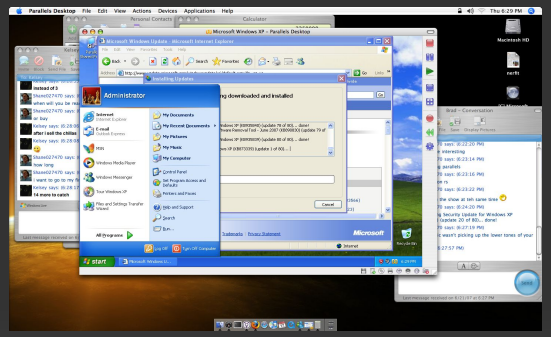
Silviu Marcu

Threads Intro

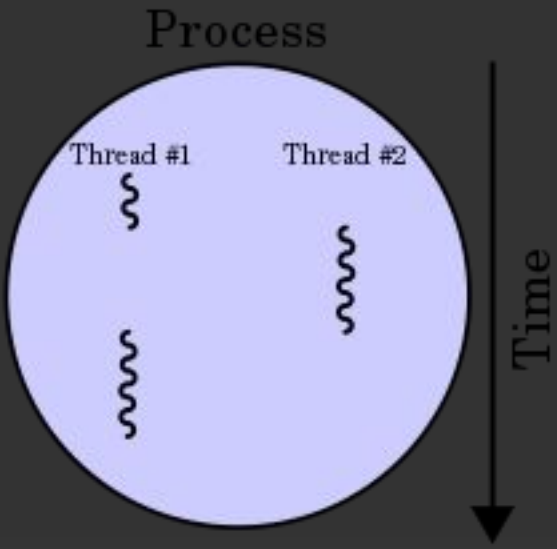
Single task



Multitasking



Multithreading



Threads benefits and risks

Benefits

- multi core
- simplicity of modeling
- handling async events
- responsiveness

Risks

- safety
- liveness
- performance

Thread Synchronization

Thread ERRORS

- Thread interference
- Memory Consistency

How to prevent these errors?
SYNCHRONIZATION

```
class Counter {
  private int c = 0;

  public void increment() {
     c++;
  }

  public void decrement() {
     c--;
  }

  public int value() {
     return c;
  }
}
```

Thread A: Retrieve c. (value 0)
Thread B: Retrieve c. (value 0)
Thread A: Increment retrieved value; result is 1.
Thread B: Decrement retrieved value; result is -1.
Thread A: Store result in c; c is now 1.
Thread B: Store result in c; c is now -1.

Thread Synchronization

Thread ERRORS

- Thread interference
- Memory Consistency

How to prevent these errors?
SYNCHRONIZATION

Memory consistency errors occur when different threads have inconsistent views of what should be the same data.

The key to avoiding memory consistency errors is understanding the *happens-before* relationship.

Actions to create happens-before relationships:
- synchronized
- volatile
- Thread.start, Thread.join

Java Concurrency concepts

Thread
Runnable

Thread.sleep(millisec)
threadObj.join()

InterruptedException
Thread.interrupted()
Interrupt status flag

synchronized - method, block, monitor lock, REENTRANT

Thread Safety

Writing thread-safe code is about MANAGING ACCESS TO STATE, in particular to SHARED MUTABLE STATE

Whenever more than one thread accesses a given state variable, and one of them might write to it, they all must coordinate their access to it using synchronization

Thread-safe classes encapsulate any needed synchronization so that clients need not provide their own.

Atomicity

- an atomic action happens all at once
- cannot stop in the middle (all or nothing)
- no side effects are visible until the action is complete

reads/writes are atomic for reference var and most primitive vars (except long and double)

reads/writes are atomic for all variables declared VOLATILE (including long and double)

To preserve state consistency, update related state variables in a single atomic operation.

Locking

Java provides a built-in locking mechanism for enforcing atomicity: the synchronized block

synchronized(lock) { ... access or modify state guarded by lock }

synchronized method - shorthand for a synchronized block that spans an entire method body and whose lock is the object on which the method is being invoked

The built-in locks are called INTRINSIC LOCKS or MONITOR LOCKS and they act as MUTEXES (mutual exclusion locks) The only way to acquire an intrinsic lock is to enter a synchronized block/method.

Locking

It is a common mistake to assume that synchronization needs to be used only when writing to shared variables.

For each mutable state variable that may be accessed by more than one thread, ALL ACCESSES to that variable must be performed with the same lock held (the variable is GUARDED by that lock)

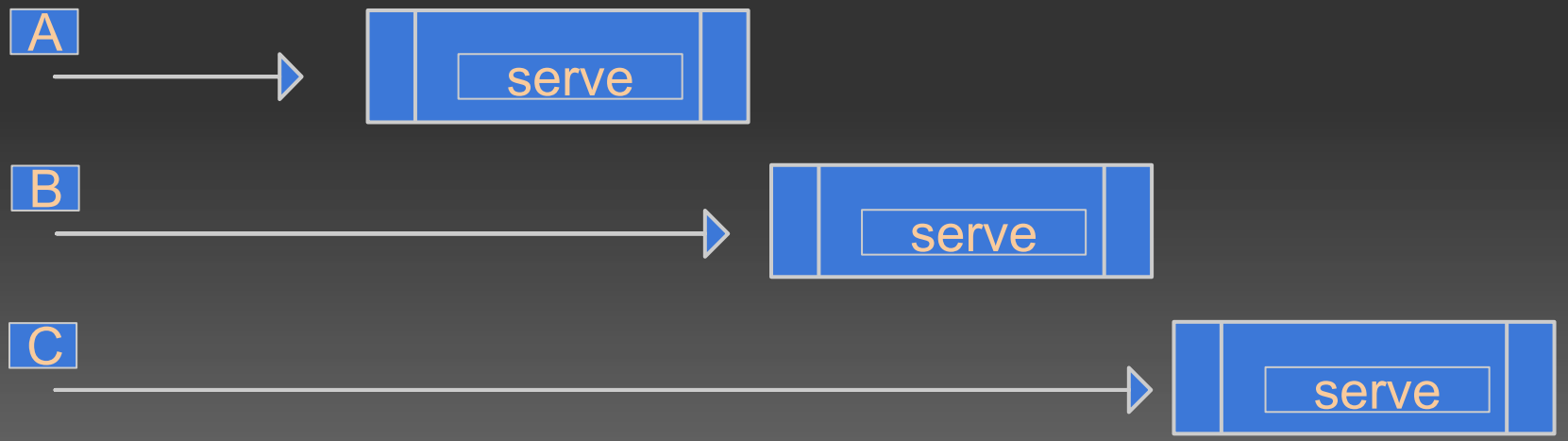
Intrinsic Locks are REENTRANT

```
public class X {
  public synchronized void syncMethod() { }
}
public class Y extends X {
  public synchronized void xyz() { syncMethod(); }
}
```

Liveness and performance

Tension between simplicity and performance

```
class SimpleMultiThreadedServer {
    public synchronized Response serve(Request) {
        io operations ...
        process state ...
        other operations
    }
}
```



Visibility

Locking is not just about mutual exclusion; it is also about memory visibility. To ensure that all threads SEE the most up-to-date values of shared mutable variables, the reading and writing threads must synchronize on a common lock;

VOLATILE

- nonatomic 64-bit operations (long, double) - use volatile or guarded by locks
- a volatile variable: compiler and runtime - do not reorder operations, not cached, a read of volatile var always returns the most recent write by any thread
- volatile performs no lock, lighter-weight mechanism
- code that uses volatile is harder to understand

- Publication and escape
- Thread confinement (not sharing state, ex swing dispatch thread)
- Thread Local (separate copy of the value for each thread)

Synchronized Collections

Legacy synchronized collection classes

- Vector, Hashtable, Collections.synchronizedXxx
- achieve thread safety by encapsulating their state and synchronizing every public method
- extra locking needed for compound actions (put if absent ...)

Locking a collection during iteration may be undesirable

- block other threads, performance issue, application scalability
- risk factor for deadlocks

Concurrent Collections - java 5.0

Designed for concurrent access from multiple threads

ConcurrentHashMap, CopyOnWriteArrayList
interface ConcurrentMap - compound actions: put-if-absent, replace, conditional remove.

Java 5.0 adds two new collection types: Queue and BlockingQueue

Replacing synchronized collections with concurrent collections can offer dramatic improvements with little risk.

ConcurrentHashMap

- uses a different locking strategy, offers better concurrency and scalability.

Finer grained locking mechanism, allowing many reading threads to access the map concurrently. Readers can access the map concurrently with writers, and a limited number of writers can modify the map concurrently.

ConcurrentMap interface

```
public interface ConcurrentMap<K,V> extends Map<K,V> {
    // insert only if not value is mapped from K
    V putIfAbsent (K key, V value);

    // remove only if k is mapped to v
    boolean remove (K key, V value);

    // replace value only if k is mapped to oldValue
    boolean replace (K key, V oldValue, V newValue);

    // replace value only if k is mapped to some value
    V replace (K key, V newValue);
}
```

CopyOnWriteArrayList / CopyOnWriteArraySet

- Concurrent replacement for synchronized List / Set
- Offer better concurrency in some common situations
- eliminates the need to lock or copy the collection during iteration

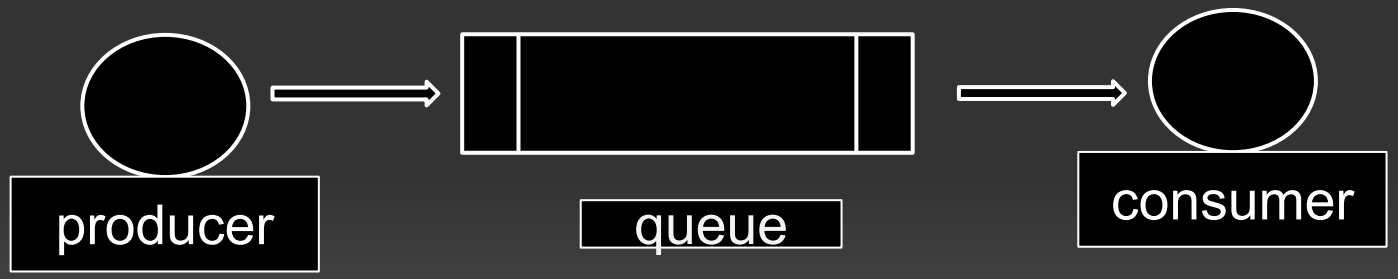
Thread safety: publishing an immutable object no further sync needed when accessing it.

They implement mutability by creating and republishing a new copy of the collection every time is modified.

Use when iteration is far more common than modification (example: event notification - iterating through event listener / register-unregister event listener)

Producer-consumer pattern

- Separates the identification of work to be done from the execution of that work by placing work items on a "to do" list for later processing
- Simplifies the development because it removes code dependencies between producer and consumer.



Blocking Queues

- blocking PUT and TAKE, also timed equivalents OFFER and POLL
- if queue is full PUT blocks until space available
- if queue is empty TAKE blocks until element available

Bounded / Unbounded Queues

Code Example [ ]

Blocking and interruptible methods

A thread may block/pause for several reasons:

- waiting I/O completion
- waiting to acquire a lock
- waiting to wake up from Thread.sleep
- waiting for the result of a computation in another thread

blocked thread states BLOCKED, WAITING, TIMED_WAITING

The distinction between a blocking operation and an ordinary operation that can take a long time to finish is that a blocked thread must wait for an event that is beyond its control before it can proceed (i/o completes, lock becomes available, external computation finishes)
state back to RUNNABLE and is eligible for scheduling

Thread interruptions

InterruptedException
Thread.sleep, BlockingQueue.put/take, Object.wait, etc...

When a method can throw InterruptedException, is is telling you that it is a blocking method, and if it is interrupted it will make an effort to stop blocking early.

Thread provides the interrupt() method for interrupting a thread and interrupted() / isInterrupted() to query if it is interrupted.

Thread interruptions

When your code calls a method that throws InterruptedException, then your method is a blocking method too, and MUST have a plan for responding to interruption.

Possible options:

1. Propagate the InterruptedException
2. Restore the interrupt (not always possible to propagate the exception)

DO NOT catch and ignore the InterruptedException, it deprives code higher up on the call stack of the opportunity to act on the interruption

Code Example [ ]

Thread interruptions

Dealing with non-interruptible blocking

Synchronous IO (java.io) InputStream.read/write
- closing underlying socket makes any read/write throw a SocketException

Intrinsic lock
- if a thread is blocked waiting for an intrinsic lock, there is nothing you can do .... Use explicit Lock classes (java 5) exposing lockInterruptibly

Synchronizers

A synchronizer is any object that coordinates the control flow of threads based on its state.

Latch

act as a gate, until the latch reaches the terminal state the gate is closed and no thread can pass. Once in terminal state, it cannot change state again

Semaphore

manages a set of virtual permits, activities can acquire and release permits, if no permit available acquire will block until one is

Barrier

similar to latches, all the threads must come together at a barrier point at the same time in order to proceed.

Code Example [ ]

Explicit LOCKS - java 5

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    void tryLock();
    void tryLock(long timeout, TimeUnit unit);
    void unlock();
    Condition newCondition();
}
```

ReentrantLock - same mutual exclusion and memory-visibility guarantees as synchronized.

- use try-finally to release a Lock

Explicit LOCKS - java 5

ReentrantReadWriteLock
Lock readLock() - returns the lock for reading
Lock writeLock() - returns the lock for writing

greater level of concurrency - single writer, multiple readers

Code Example [ ]

GUI applications (swing)

Most GUI frameworks are single threaded
GUI activity is confined to a single thread (event dispatch thread)
The application using a single threaded GUI framework can be multi threaded.

SWING single thread rule: Swing components and models should be created, modified and queried only from the event-dispatching thread.

GUI applications (swing)

SwingUtilities.invokeLater - schedules for execution on the event thread (callable from any thread)

SwingUtilities.invokeAndWait - schedules for execution on the event thread and blocks current thread until it completes (callable only from NON gui thread)

SwingUtilities.isEventDispatchThread - checks if the current thread is the event thread

SwingWorker - cancellation, completion notification, progress

Code Example [ ]

Task Execution

Task - logical unit of work
Executor - provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc.

An Executor is normally used instead of explicitly creating threads.

`Thread(new(RunnableTask())).start()` for each of a set of tasks,

```
Executor executor = anExecutor;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
```

Task Execution

ExecutorService

- *Future submit(Callable task)*
- *Future submit(Runnable task)*

Submits a value-returning task for execution and returns a Future representing the pending results of the task.

- List<Future> invokeAll(Collection<Callable> tasks)

- shutdown()
- shutdownNow()

Task Execution

Executors - factory and utility methods

Executors.newCachedThreadPool()

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

Executors.newSingleThreadExecutor()

Creates an Executor that uses a single worker thread operating off an unbounded queue.

Executors.newFixedThreadPool(int nthreads)

Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.

Atomic variables

AtomicBoolean, Integer, IntegerArray, ...
AtomicReference

boolean compareAndSet(expectedValue, updateValue)
boolean weakCompareAndSet(expectedValue, updateValue)

getAndIncrement()

Reference

Java Concurrency in Practice, Brian Goetz, Addison-Wesley

http://download.oracle.com/javase/tutorial/essential/concurrency/index.html

http://download.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html