Concurrency: Where to draw the lines

http://gee.cs.oswego.edu

Concurrency:
Where to draw the lines

Doug Lea
SUNY Oswego
dl@cs.oswego.edu

Outline

- Layers of concurrency support
  - Some design options
- Selected background
  - Memory models
  - Concurrency libraries
  - Isolates

Supporting Concurrency

- Concurrency is unavoidable, and unavoidably diverse
  - No use taking religious stance about which style is best
- Common approaches
  - Threads-and-Monitors (classic Java, pthreads)
  - Asynchronous task frameworks, Futures, Events
  - Optimistic and lock-free synchronization
  - Message passing – synch or asynch, thread or process-based
  - Resource control – semaphores, monitoring, etc
  - Parallel decomposition – barriers, etc
  - Transactional – lightweight or databases
- Languages/platforms must support these
  - Otherwise programmers will build from what they are given.

Layers

- Targets
  - Processors
  - (Hardware) Systems
  - Operating Systems
  - Virtual Machines
  - Libraries and Middleware
  - Components
  - Applications
- Functions and properties
  - Ordering and Consistency
  - Atomicity
  - Waiting
  - Task-switching
  - Notifications
  - Monitoring
- Typical tradeoffs
  - Overhead, throughput, latency, scalability

Sample Design Issues

- Doing something is better than doing nothing
  - Stalling hurts throughput, and doesn't help anything else
  - Speculation, chip-level multithreading etc
- Unless that something hurts others
  - Spinning causes memory contention
- Or there is nothing to do
  - Power management
- But switching tasks can be expensive
  - Minimizing overhead: Pools, work-stealing
- And reliance on future actions of other tasks is risky
  - Minimizing before/after-style control (e.g., lock/unlock)
- And abruptly killing other tasks is even more risky
  - Minimizing reliance on whether cleanup occurs

Core VM support

- Adherence to a memory model
  - Including support for atomic variables
- Threads
  - Possibly multiple granularities (tasks, active-events, sessions)
  - Scheduling: Task-stealing, blocking, unblocking, cancelling
- Processes or Isolates
  - Resource control
  - Interprocess messaging
- Binding control
  - Threads, sessions, objects etc as containers
  - Versioning and rollback
- Integration with IO
  - Channels, buffers

Library-Centric Concurrency

- Rely on library/middleware for most user-visible concurrency
  - Avoid global reliance on, say, Monitor-style concurrency
- Efficiency
  - Many algorithms and data structures are both simpler and faster if they can rely on GC and dynamic optimization
  - Can make more informed engineering tradeoffs about Scalability vs overhead, general vs special-case etc
- Planning for change
  - Concurrency is again a hot area in research and engineering
  - Expect even better approaches to emerge for lightweight transactions, task coordination, collections, etc
- Downstream consequences
  - On debugging, monitoring, profiling, static analysis, error detectors, design tools

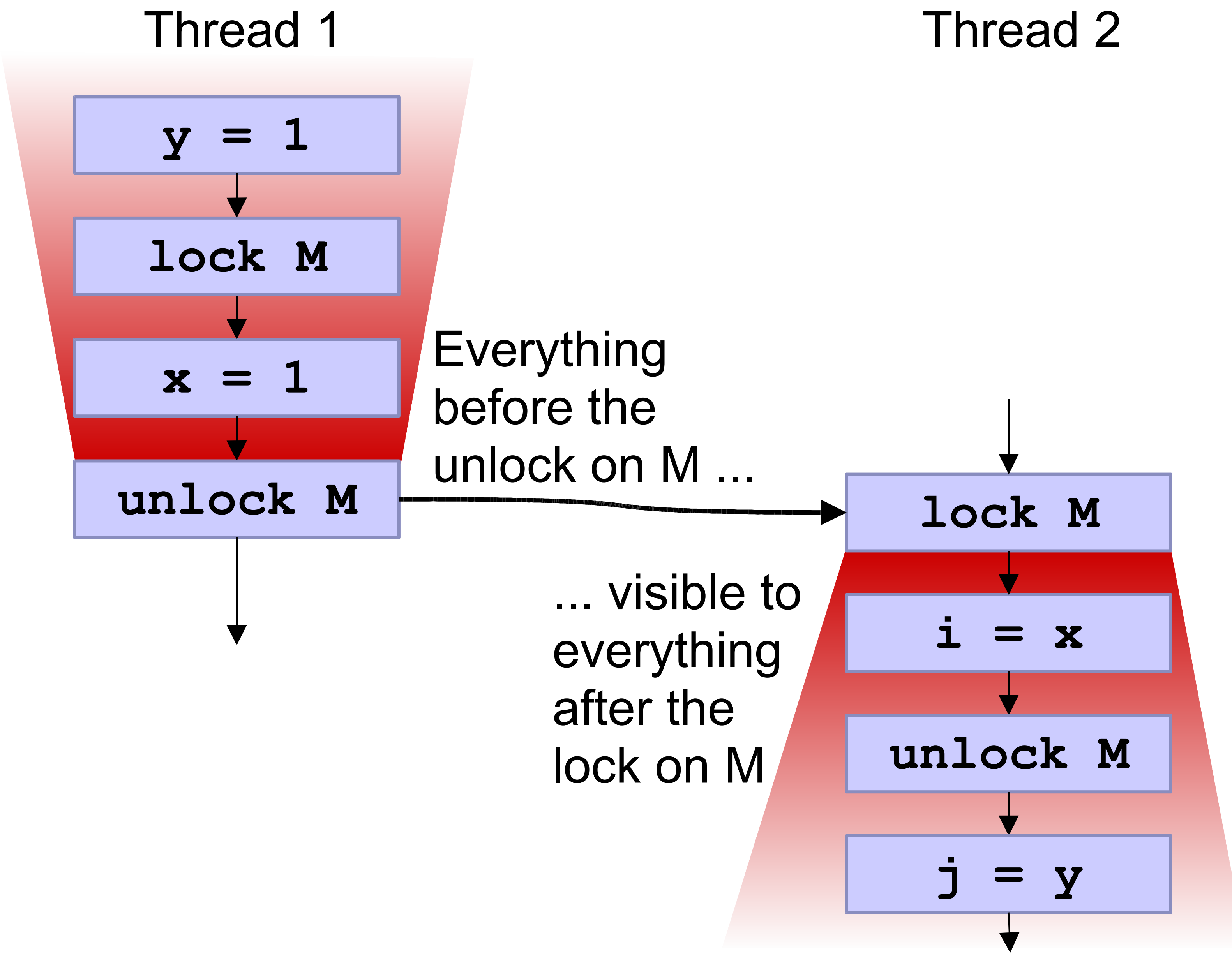
Some Challenges

- Where does VM end and middleware begin?
  - May require trust framework so VM will believe library author
  - May require APIs accessible only by trusted middleware
- Teaching VM about optimizations
  - Example: Minimizing memory barriers
  - Requires new forms of metadata
  - Similar to current work in C++ library optimization
- Accommodating Processor, System, OS differences
  - Example: LL/SC vs CAS vs new chip-level primitives
- Avoiding constructs that reward complexity and sleaze
  - Example: Lock bits in object headers
- Syntactic integration with language
  - Example: Expressing lightweight transactions

JSR-133 Memory Model

- A memory model specifies how threads and objects interact
  - Atomicity
    - Ensuring mutual exclusion for field updates
  - Visibility
    - Ensuring changes made in one thread are seen in other threads
  - Ordering
    - Ensuring that you aren't surprised by the order in which statements are executed
- Original JLS spec was broken and impossible to understand
  - Unwanted constraints, omissions, inconsistencies
- The basic JSR-133 rules are easy. The formal spec is not.
  - Spec complexity mainly in clarifying optimization issues

JSR-133 Main Rule



Additional JSR-133 Rules

- Variants of lock rule apply to volatile fields and thread control
  - Writing a volatile has same basic memory effects as unlock
  - Reading a volatile has same basic memory effects as lock
  - Similarly for thread start and termination
  - Details differ from locks in minor ways
- Final fields
  - All threads read final value so long as it is always assigned before the object is visible to other threads. So DON'T write:

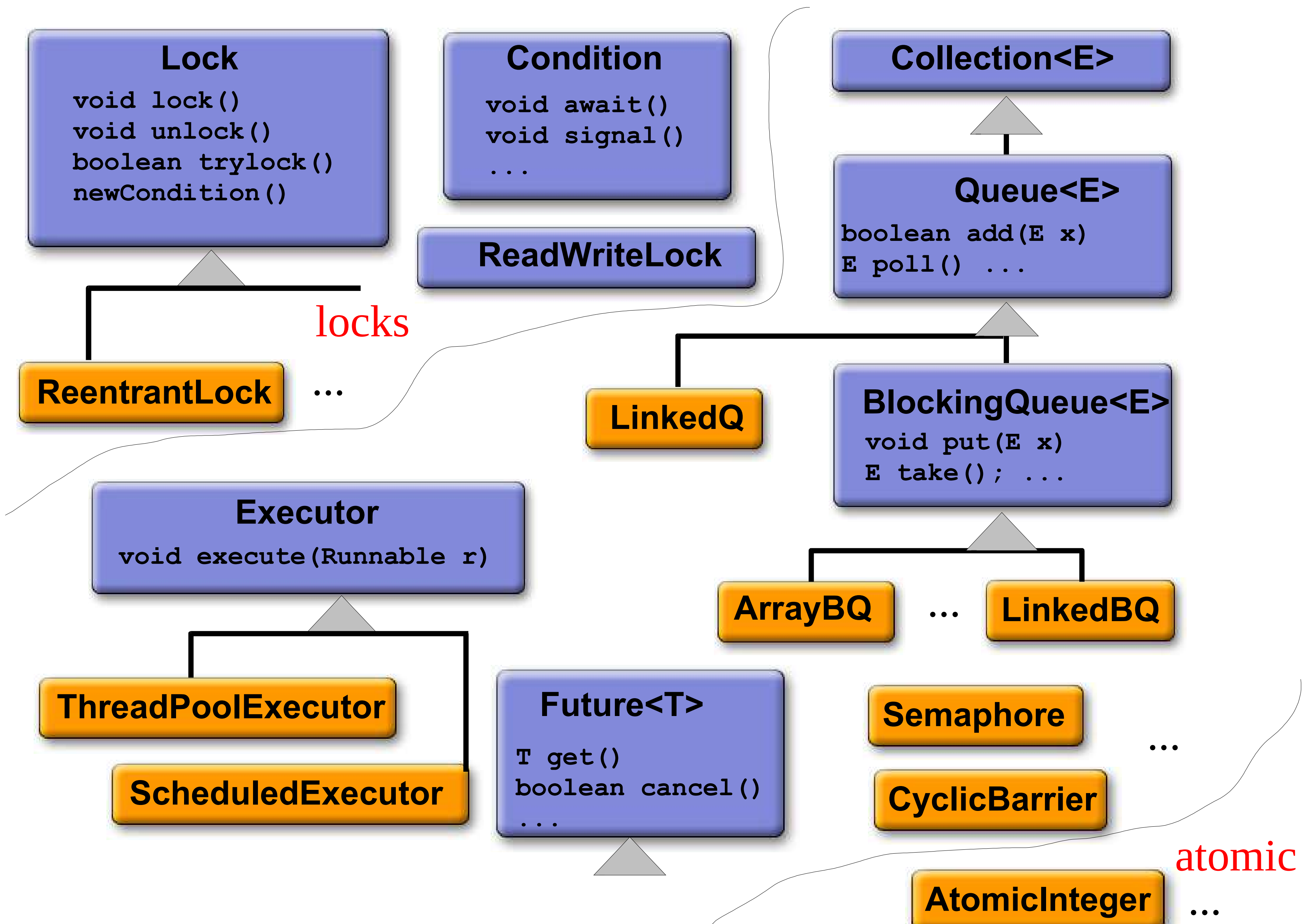
```
class Stupid implements Runnable {
  final int id;
  Stupid(int i) { new Thread(this).start(); id = i; }
  public void run() { System.out.println(id); }
}
```

- Extremely weak rules for unsynchronized, non-volatile, non-final reads and writes
  - type-safe, not-out-of-thin-air, but can be reordered, invisible

java.util.concurrent

- Queue framework
  - Queues & blocking queues
- Concurrent collections
  - Lists, Sets, Maps geared for concurrent use
- Executor framework
  - ThreadPools, Futures, CompletionService
- Synchronizers
  - Semaphores, Barriers, Exchangers, CountDownLatches
- Lock framework (subpackage `java.util.concurrent.locks`)
  - Including Conditions & ReadWriteLocks
- Atomic variables (subpackage `java.util.concurrent.atomic`)
  - JVM support for compareAndSet operations
- Other miscellany

Main JSR-166 components

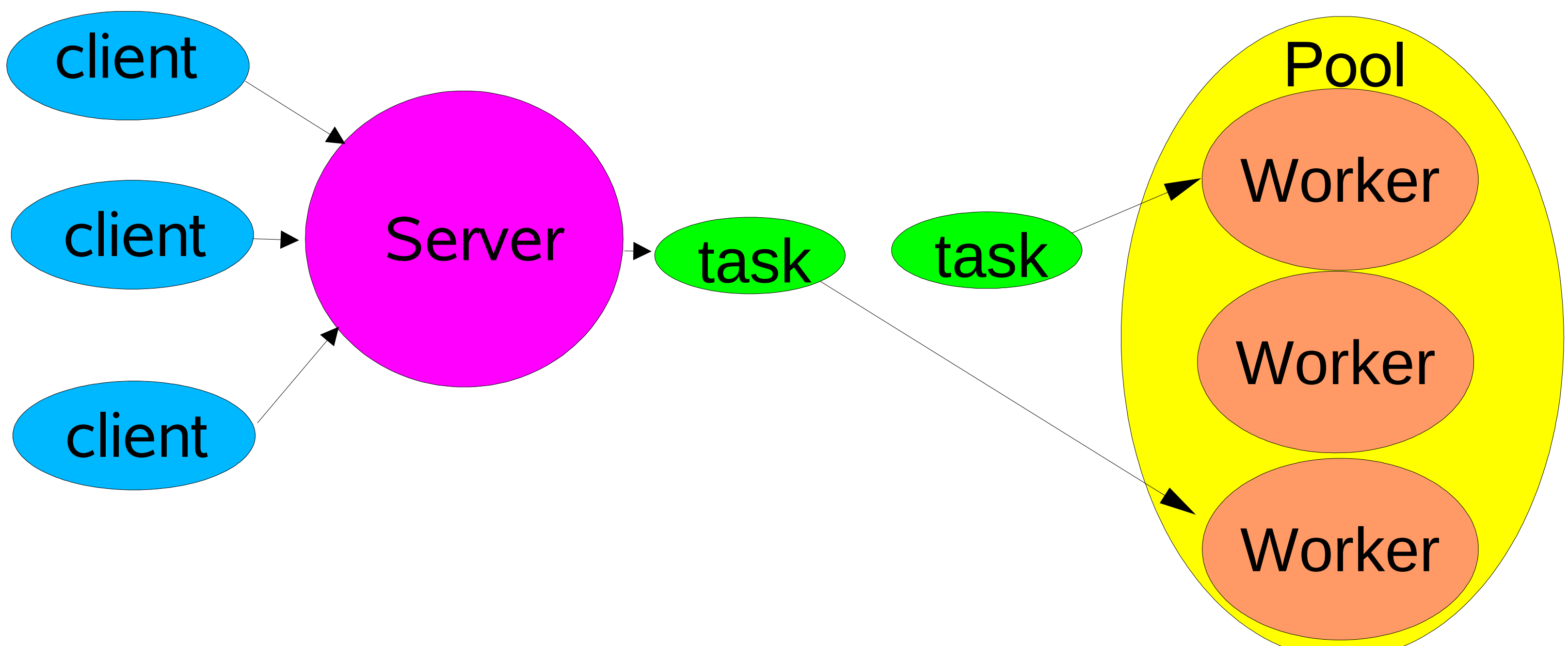


Example framework: Executors

- Standardize asynchronous task invocation
  - Use `anExecutor.execute(aRunnable)`
  - Not `new Thread(aRunnable).start()`
- Two styles supported:
  - Actions: `Runnables`
  - Functions (indirectly): `Callables`
- A small framework, including:
  - `Executor` – something that can `execute Runnables`
  - `ExecutorService` extension -- shutdown support etc
  - `Executors` utility class – configuration, conversion
  - `ThreadPoolExecutor` – tunable implementation
  - `ScheduledExecutor` for time-delayed tasks
  - `ExecutorCompletionService` – maintain completed tasks

Executor Example

```
class Server {
  public static void main(String[] args) throws Exception {
    Executor pool = Executors.newFixedThreadPool(3);
    ServerSocket socket = new ServerSocket(9999);
    for (;;) {
      final Socket connection = socket.accept();
      pool.execute(new Runnable() {
        public void run() {
          new Handler().process(connection);
        }});
    }
  }
  static class Handler { void process(Socket s); }
}
```



Future Example

```
class ImageRenderer { Image render(byte[] raw); }

class App { // ...
   ExecutorService exec = ...;          // any executor
   ImageRenderer renderer = new ImageRenderer();

   public void display(final byte[] rawimage) {
     try {
       Future<Image> image = exec.submit(new Callable(){
         public Object call() {
           return renderer.render(rawImage);
         }});

       drawBorders(); // do other things while executing
       drawCaption();

       drawImage(image.get()); // use future
     }
     catch (Exception ex) {
       cleanup();
     }
   }
}
```

Atomic Variables

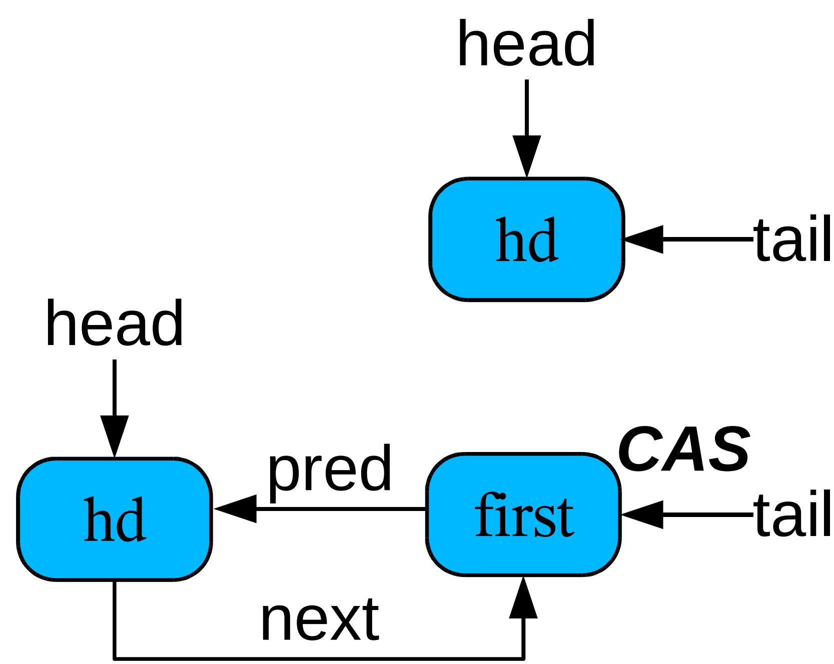
- Classes representing scalars supporting

  `boolean compareAndSet(expectedValue, newValue)`

  - Atomically set to `newValue` if currently hold `expectedValue`
  - Also support variant: `weakCompareAndSet`
    - May be faster, but may spuriously fail (as in LL/SC)
- Classes: { *int, long, reference* } X { *value, field, array* } plus boolean value
  - Plus AtomicMarkableReference, AtomicStampedReference
    - (emulated by boxing in J2SE1.5)
- JVMs can use best construct available on a given platform
  - Compare-and-swap, Load-linked/Store-conditional, Locks

Synchronizers

- Locks, semaphores, latches, futures etc all rely on class AbstractQueuedSynchronizer for queuing and blocking
- Based on a blocking extension of CLH locks
  - Block using LockSupport.park when not head of queue or cannot acquire state – an atomic int controlled by client class
- Fast single-CAS queue insertion using explicit pred pointers
- Also next-pointers to enable signalling (unpark)
  - Not atomically assigned
  - Use pred ptrs as backup
- Many options: timeout, cancellation, fairness, exclusive vs shared, associated Conditions
- See CSJP paper for details



Collections (Lists, Sets, Maps)

- Large APIs, but what do people do with them?
  - Informal workload survey using pre-1.5 collections
- Operations:
  - About 83% read, 16% insert/modify, <1% delete
- Sizes:
  - Medians less than 10, very long tails
  - Concurrently accessed collections usually larger than others
- Concurrency:
  - Vast majority only ever accessed by one thread
    - But many apps use thread-safe collections anyway
  - Others contended enough to be serious bottlenecks
  - Not very many in-between
- Lock-based collections don't usually fit well with usage patterns

Collections Design Options

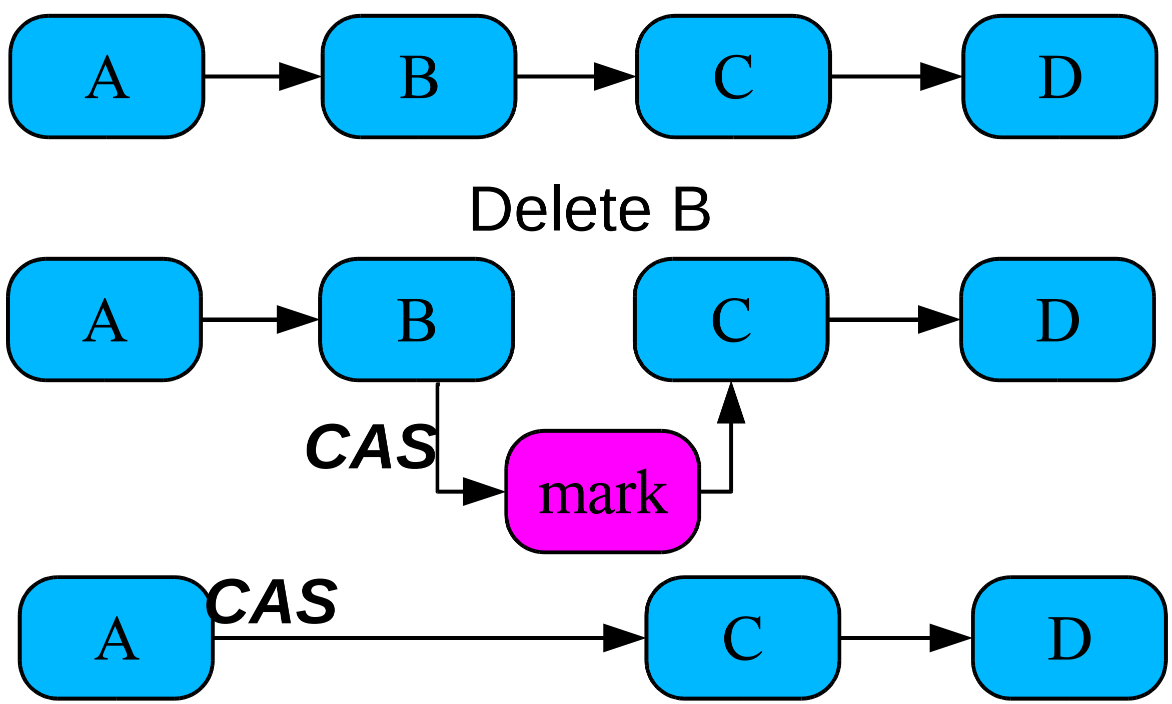
- Large design space, including
  - Locks: Coarse-grained, fine-grained, ReadWrite locks
  - Concurrently readable – reads never block, updates use locks
  - Optimistic – never block but may spin
  - Lock-free – concurrently readable and updatable
- Most initial JSR-166 collections concurrently readable
  - Several lock-free additions are being done as RFEs

Rough guide to tradeoffs for typical implementations

	Read overhead	Read scaling	Write overhead	Write scaling
Coarse-grained locks	**Medium**	**Worst**	**Medium**	**Worst**
Fine-grained locks	**Worst**	**Medium**	**Worst**	**OK**
ReadWrite locks	**Medium**	**So-so**	**Medium**	**Bad**
Concurrently readable	**Best**	**Very good**	**Medium**	**Not-so-bad**
Optimistic	**Good**	**Good**	**Best**	**Risky**
Lock-free	**Good**	**Best**	**OK**	**Best**

Example lock-free collection idiom

- **Linking a new object** ***can*** **be cheaper/better than marking a pointer**
  - **Less traversal overhead but need to traverse at least 1 more node during search; also can add GC overhead if overused**
- **Can apply to M. Michael's sorted lists, using deletion marker nodes**
  - **Maintains property that ptr from deleted node is** ***changed***
  - **Can in turn apply to Skip Lists (now in package `jsr166x`)**



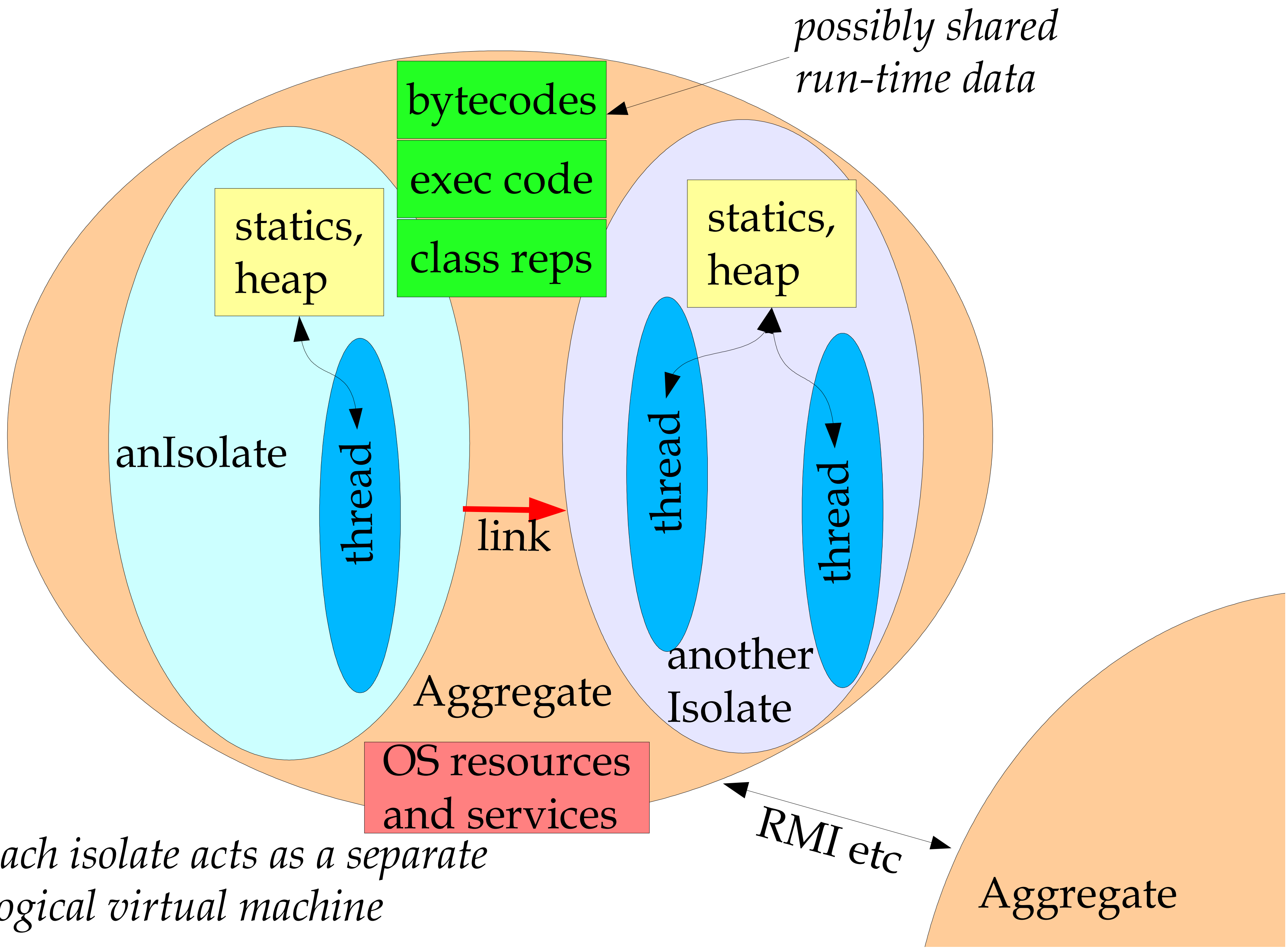
Overview of Isolates

Isolate *noun*. pronounciation: *isolet*.
1. A thing that has been isolated, as by geographic, ecologic or social barriers -
American Heritage Dictionary

Status

- **At public review draft in JSR-121.**
  - **Originally targetted for J2SE1.5, but triaged out**
- **Tentatively scheduled for next major J2SE release.**
  - **Will be partially overhauled**
- **J2ME versions will probably appear sooner.**

Aggregates vs Isolates vs Threads



each isolate acts as a separate
logical virtual machine

Three Implementation Styles

- One Isolate per OS process
  - Internal sharing via OS-level shared memory, comms via IPC
    - class representations, bytecodes, compiled code, immutable statics, other internal data structures

Likely for J2SE

- All Isolates in one OS address space / process managed by aggregate
  - Isolates still get own versions of all statics/globals
    - including AWT thread, shutdown hooks, ...

Likely for J2ME

- LAN Cluster JVMs
  - Isolates on different machines under a common administrative domain. *NOT* a substitute for RMI
    - Little or no internal sharing

Still research

Main Classes

- `public final class Isolate`
  - Create with name of class with a "main", arguments to main, plus optional standard IO bindings, classpath, security, system property and other context settings.
  - Methods to start, stop, and terminate created isolate
  - Event-based monitoring of life cycle events
- `public abstract class Link`
  - A pipe-like data channel to another isolate, that can pass:
    - byte arrays, ByteBuffers, Strings and serializable types
    - SocketChannels, FileChannels and other IO types
    - Isolates, Links
  - (Will be reworked in upcoming revision.)

Target Usage Patterns

- Minimizing startup time and footprint
  - User-level "java" program, web-start, etc can start JVM if not already present then fork Isolate
  - OS can start JVM at boot time to run daemons
- Partitioning applications
  - Contained applications (*lets)
    - Applets, Servlets, Xlets, etc can run as Isolates
    - Container utility services can run as Isolates
  - Service Handler Forks
    - ServerSocket.accept can launch handler for new client as Isolate
    - Pools of "warm" Isolates

More Usage Patterns

- Parallel execution on cluster JVMs
  - Java analogs of Beowulf clusters
    - Maybe using MPI-like protocol over Links
  - Need partitioning and load-balancing frameworks
- Fault-tolerance
  - Fault detection and re-activation frameworks
  - Redundancy via multiple Isolates
- CSP style programming
  - Always use Isolates instead of Threads
  - Practically suitable only for coarse-grained designs