Java Concurrency -- java.util.concurrent

Doug Lea
SUNY Oswego
dl@cs.oswego.edu

Credits: Some slides co-authored by David Holmes, Bill Scherer, Brian Goetz

Outline

- Overview of Java concurrency support
  - Core support
  - java.util.concurrent
- Selections of APIs, usages, and underlying algorithms for:
  - Executing tasks
    - Executors, Threads
  - Atomicity and Synchronization
    - Locks, Atomics, Synchronizers
  - Collections
    - Queues
    - Maps and Sets

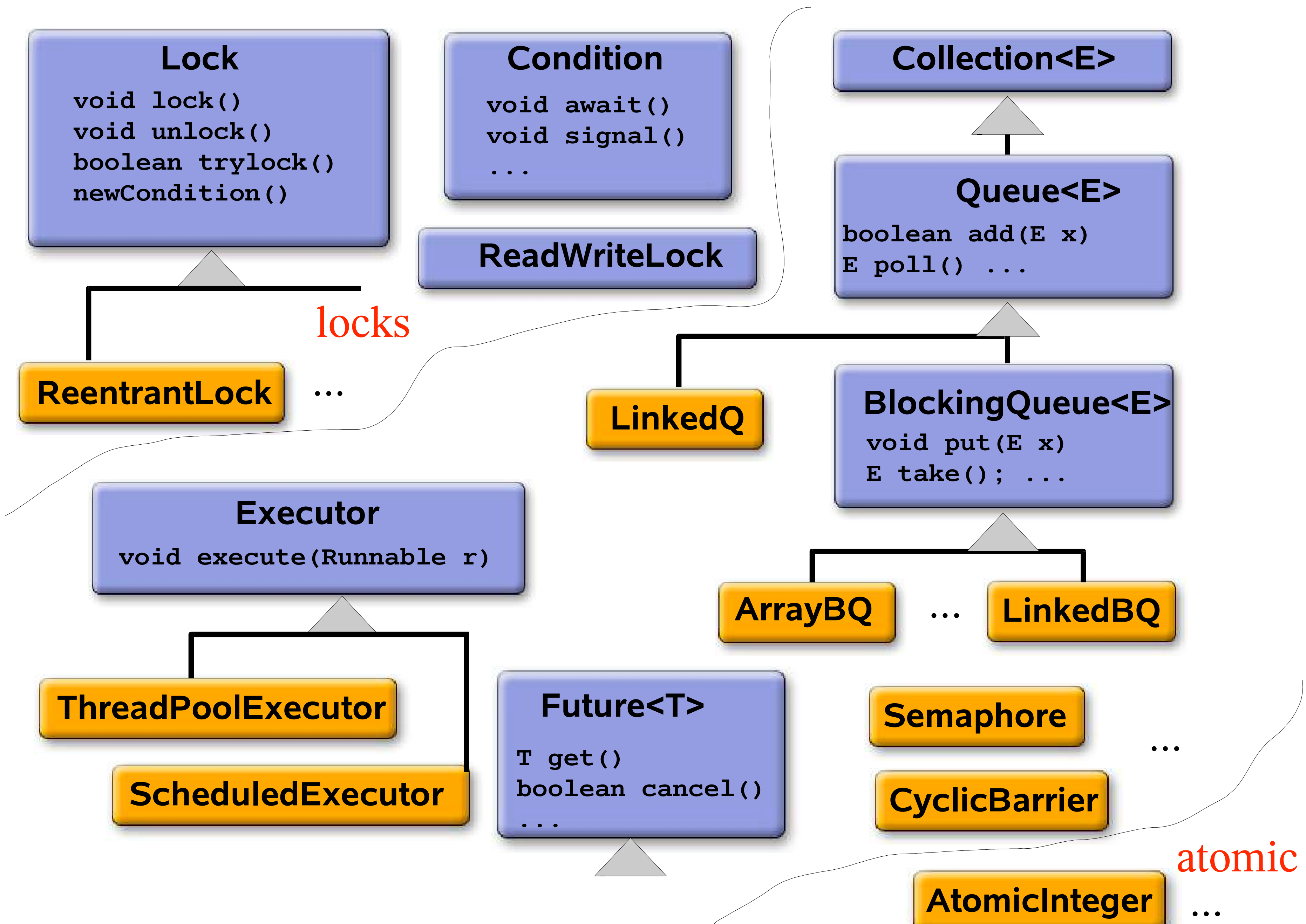
Core Java Concurrency Support

- **Built-in language features:**
  - `synchronized` **keyword**
    - **"monitors" part of the object model**
  - `volatile` **modifier**
    - **Roughly, reads and writes act as if in synchronized blocks**
- **Core library support:**
  - `Thread` **class methods**
    - `start, sleep, yield, isAlive, getID, interrupt, isInterrupted, interrupted, ...`
  - `Object` **methods:**
    - `wait, notify, notifyAll`

java.util.concurrent

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

Main j.u.c components

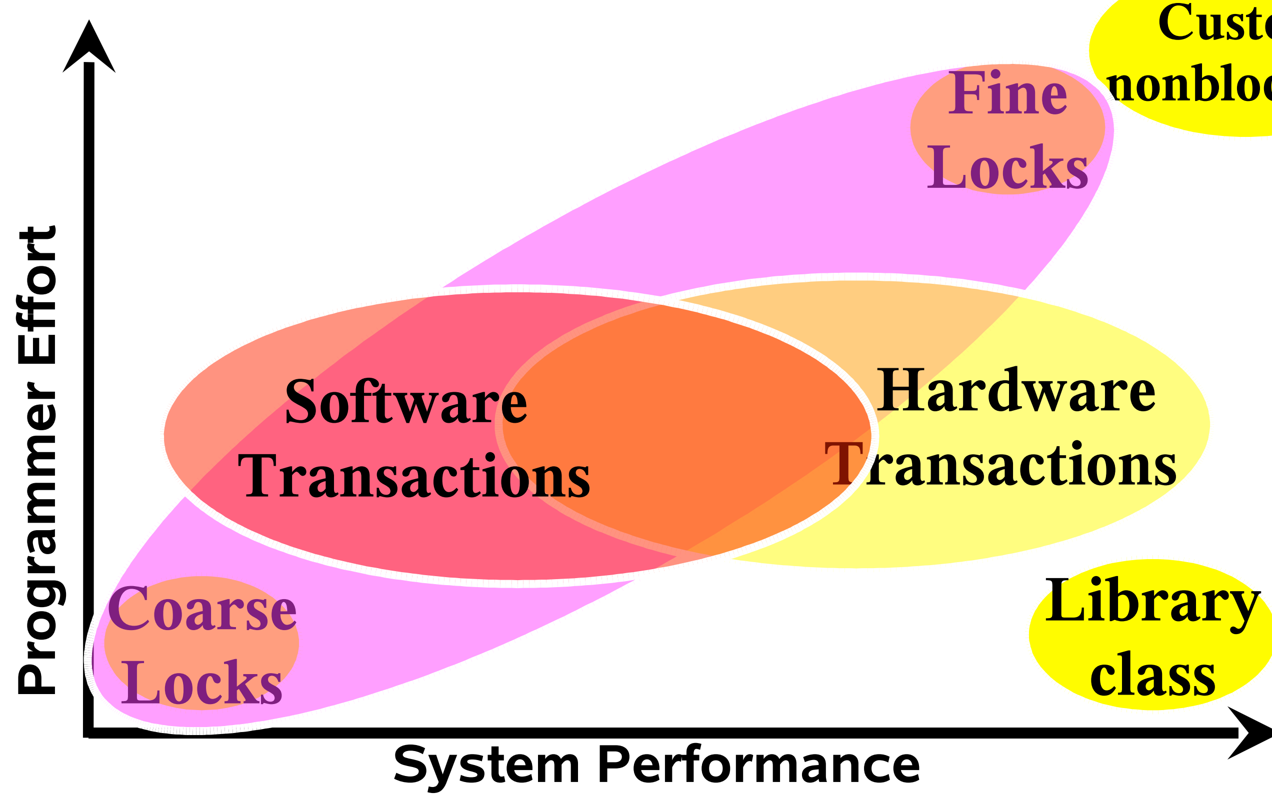


Engineering j.u.c

- Main goals
  - Scalability – work well on big SMPs
  - Overhead – work well with few threads or processors
  - Generality – no niche algorithms with odd limitations
  - Flexibility – clients choose policies whenever possible
  - Manage Risk – gather experience before incorporating
- Adapting best known algorithms; continually improving them
  - LinkedQueue based on M. Michael and M. Scott lock-free queue
  - LinkedBlockingQueue is (was!) an extension of two-lock queue
  - ArrayBlockingQueue adapts classic monitor-based algorithm
- Leveraging Java features to invent new ones
  - GC, OOP, dynamic compilation etc
  - Focus on nonblocking techniques
    - SynchronousQueue, Exchanger, AQS, SkipLists ...

Why Do Researchers Write Library Code?

- Make difficult constructions usable by componentizing them
  - Improve usability of existing languages and platforms
  - Compromise as little as possible between very fast and very easy to use. Mix of API design, algorithm design, SE.
- Coexists with goal of making constructions easier
  - New languages, platforms, computing models



Executors

- Standardizes asynchronous task invocation
  - Use `anExecutor.execute(aRunnable)`
  - Not old style: `new Thread(aRunnable).start()`
- Two styles supported, non-result-bearing and result-bearing:

```
interface Runnable    { void run(); }
interface Callable<T>{ T call() throws Exception;}
```

- 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

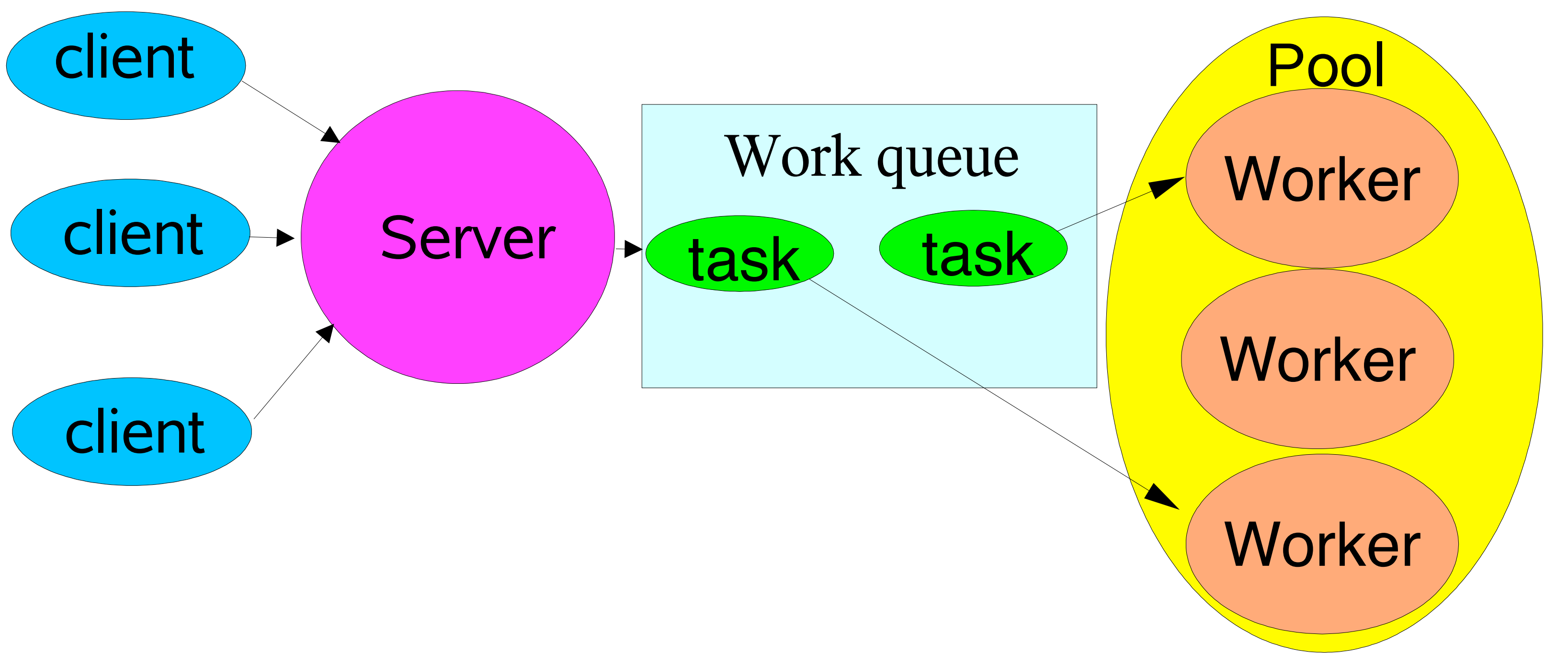
ExecutorServices

- Usually create with factory methods in `Executors` class
  - `newFixedThreadPool(int N)`
    - A fixed pool of N, working from an unbounded queue
  - `newCachedThreadPool`
    - A variable size pool that grows as needed and shrinks when idle
  - Or use highly tunable underlying `ThreadPoolExecutor`
- Lifecycle Control

```
interface ExecutorService extends Executor { //...
   void shutdown();
   List<Runnable> shutdownNow();
   boolean isShutdown();
   boolean isTerminated();
   boolean awaitTermination(long to, TimeUnit unit);
 }
```

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); }
}
```



Futures

- Encapsulates waiting for the result of an asynchronous computation launched in another thread
  - The callback is encapsulated by the Future object
- Usage pattern
  - Client initiates asynchronous computation
  - Client receives a “handle” to the result: a Future
  - Client performs additional tasks prior to using result
  - Client requests result from Future, blocking if necessary until result is available
  - Client uses result
- Main implementation is class `FutureTask<V>`
  - Wraps either a `Callable` or `Runnable`
    - `FutureTask(Callable<V> callable)`
    - `FutureTask(Runnable r, V result)`

Methods on Futures

- `V get()`
  - Retrieves the result held in this `Future` object, blocking if necessary until the result is available
  - Timed version throws `TimeoutException`
  - If cancelled then `CancelledException` thrown
  - If computation fails throws `ExecutionException`
- `boolean cancel(boolean mayInterrupt)`
  - Attempts to cancel computation of the result
  - Returns true if successful
  - Returns false if already complete, already cancelled or couldn't cancel for some other reason
  - Parameter determines whether cancel should interrupt the thread doing the computation
    - Only the implementation of `Future` can access the thread

Futures and Executors

```
<T> Future<T> submit(Callable<T> task)
```

- Submit the task for execution and return a `Future` representing the pending result

```
Future<?> submit(Runnable task)
```

- Use `isDone()` to query completion

```
<T> Future<T> submit(Runnable task, T result)
```

- Submit the task and return a `Future` that wraps the given `result` object

```
<T> List<Future<T>>
      invokeAll(Collection<Callable<T>> tasks)
```

- Executes the given tasks and returns a list of `Futures` containing the results
- Timed version too

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();
     }
   }
}
```

Builtin Synchronization

- Every Java object has an associated lock acquired via:
  - synchronized statements

```
synchronized( foo ){
    // execute code while holding foo's lock
}
```

  - synchronized methods

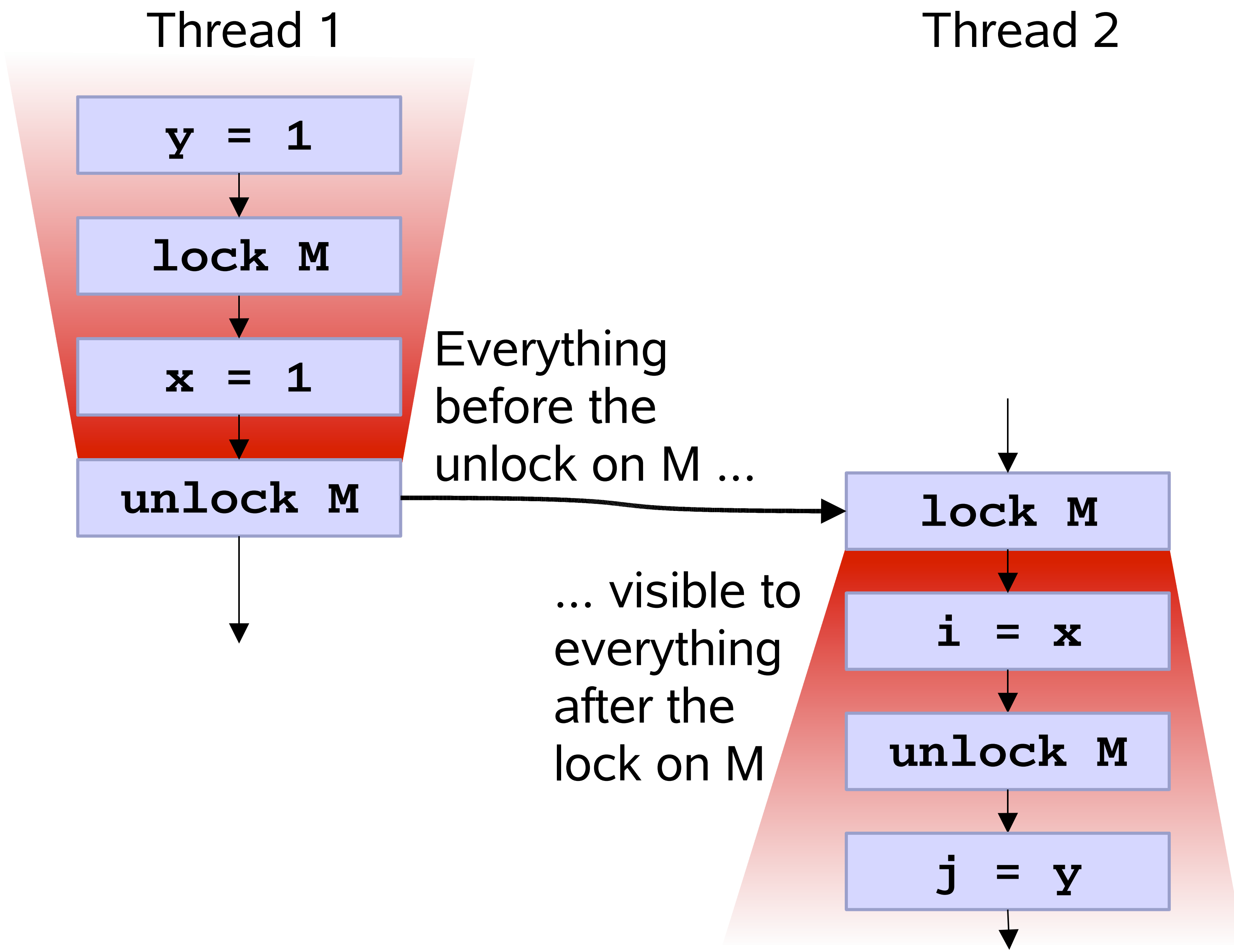
```
public synchronized void op1(){
    // execute op1 while holding 'this' lock
}
```

- Only one thread can hold a lock at a time
  - If the lock is unavailable the thread is blocked
- Locks are granted per- thread
  - So called reentrant or recursive locks
- Locking and unlocking are automatic
  - Can't forget to release a lock
  - Locks are released when a block goes out of scope

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

Happens-Before

- Underlying relationship between reads and writes of variables
  - Specifies the possible values of a read of a variable
- For a given variable:
  - If a write of the value v1 happens-before the write of a value v2, and the write of v2 happens-before a read, then that read may not return v1
  - Properly ordered reads and writes ensure a read can only return the most recently written value
- If an action A synchronizes-with an action B then A happens-before B
  - So correct use of synchronization ensures a read can only return the most recently written value

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

Example: AtomicInteger

```
class AtomicInteger {
  AtomicInteger(int initialValue);
  int get();
  void set(int newValue);
  int getAndSet(int newValue);
  boolean compareAndSet(int expected, int newVal);
  boolean weakCompareAndSet(int expected, int newVal);
  //      prefetch                     postfetch
  int getAndIncrement();   int incrementAndGet();
  int getAndDecrement();   int decrementAndGet();
  int getAndAadd(int x);   int addAndGet(int x);
}
```

- Integrated with JSR133 memory model semantics for volatile
  - `get` acts as volatile-read
  - `set` acts as volatile-write
  - `compareAndSet` acts as volatile-read and volatile-write
  - `weakCompareAndSet` ordered wrt other accesses to same var

Treiber Stack

```
interface LIFO<E> { void push(E x); E pop(); }

class TreiberStack<E> implements LIFO<E> {
  static class Node<E> {
    volatile Node<E> next;
    final E item;
    Node(E x) { item = x; }
  }

  AtomicReference<Node<E>> head =
        new AtomicReference<Node<E>>();

  public void push(E item) {
    Node<E> newHead = new Node<E>(item);
    Node<E> oldHead;
    do {
      oldHead = head.get();
      newHead.next = oldHead;
    } while (!head.compareAndSet(oldHead, newHead));
  }
```

TreiberStack(2)

```
public E pop() {
   Node<E> oldHead;
   Node<E> newHead;
   do {
     oldHead = head.get();
     if (oldHead == null) return null;
     newHead = oldHead.next;
   } while (!head.compareAndSet(oldHead,newHead));

   return oldHead.item;
 }

}
```

Exercise

- 1. Write a similar LockBasedStack<E> implements LIFO<E>
- 2. Write a program that creates N tasks, each repeatedly pushing and popping 100K elements
  - Use a cachedThreadPool
  - Use System.nanoTime() for timing
  - Lots of little details are up to you
- 3. Make two versions of this, for two kinds of stack
- 4. Compare times using the two versions for different values of N on machines with different numbers of processors

Synchronizers

- Different APIs for different styles of (blocking) synchronization
  - Locks, RW locks, barriers, semaphores, futures, handoffs...
  - Any *could* be used a basis for building others, but *shouldn't*.
    - Overhead, complexity, ugliness
- Class AbstractQueuedSynchronizer (AQS) provides common underlying functionality
  - Expressed in terms of acquire/release operations
    - Implements a concrete synch scheme
    - Doesn't try to work for all possible synchronizers, but enough to be both efficient and widely useful
  - Structured using a variant of GoF template-method pattern
    - Synchronizer classes define only the code expressing rules for when it is permitted to acquire and release.
  - Also encapsulates twisty control paths etc

Synchronizer Class Example

```
class Mutex {

  private class Sync
    extends AbstractQueuedSynchronizer {

    public boolean tryAcquire(int ignore) {
      return compareAndSetState(0, 1);
    }
    public boolean tryRelease(int ignore) {
       setState(0); return true;
    }
  }

  private final Sync sync = new Sync();

  public void lock()   { sync.acquire(0); }
  public void unlock() { sync.release(0); }
}
```

Lock APIs

- `java.util.concurrent.locks.Lock`
  - Allows user-defined classes to implement locking abstractions with different properties to those of built-in object locks
  - Main implementation is AQS-based `ReentrantLock`
- `lock()` and `unlock()` can occur in different lexical scopes
  - Unlocking is no longer automatic
  - Use `try`/`finally`
- Lock acquisition can be interrupted or allowed to time-out
  - `lockInterruptibly()`, `boolean tryLock()`, `boolean tryLock(long time, TimeUnit unit)`
- Supports multiple `Condition` objects

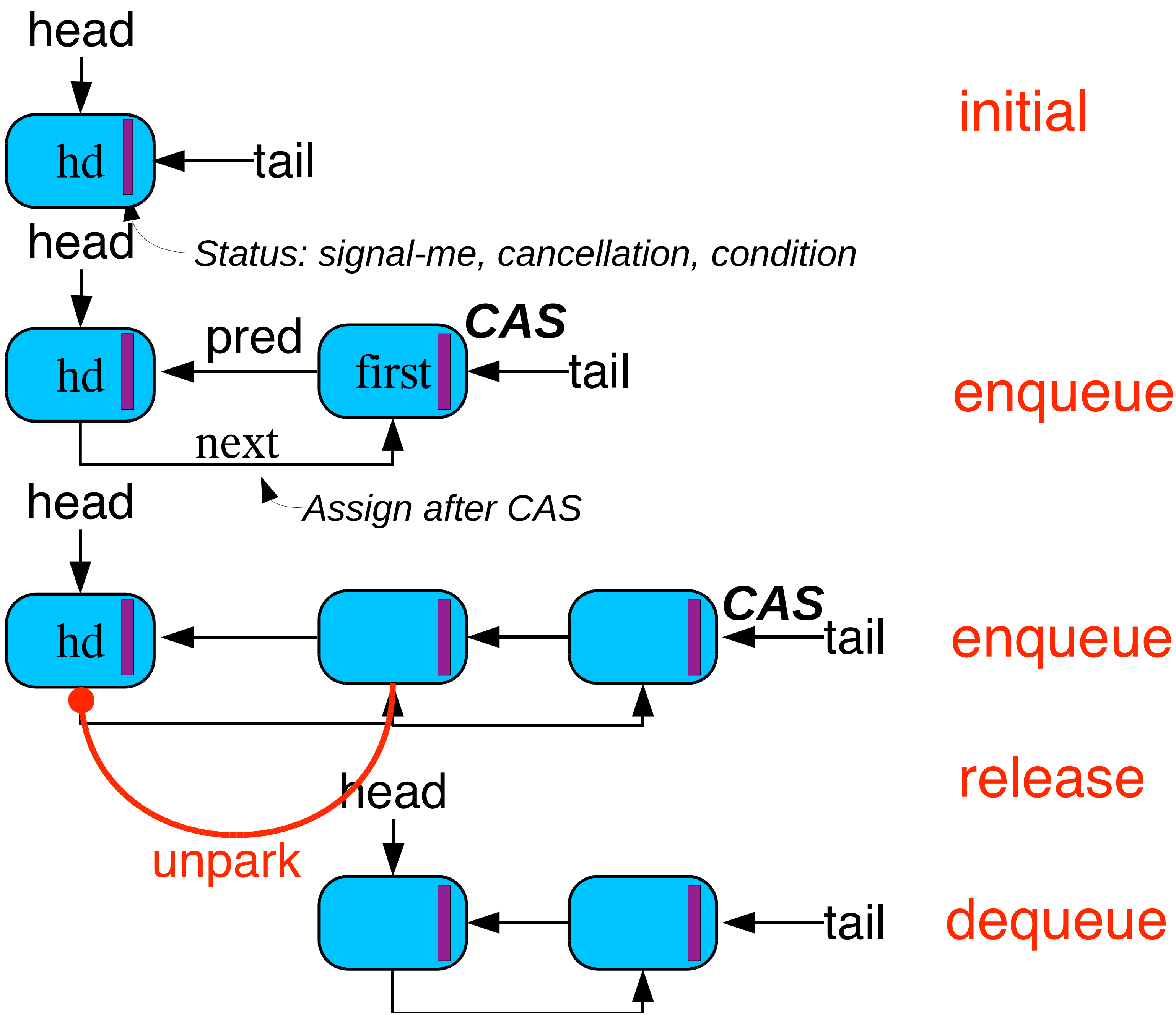
Acquire/Release

- Acquire:
  while (synchronization state does not allow acquire) {
  enqueue current thread if not already queued;
  possibly block current thread;
  }
  dequeue current thread if it was queued;
- Release:
  update synchronization state;
  if (state may permit a blocked thread to acquire)
  unblock one or more queued threads;
- Three integrated aspects of support
  - Atomically maintain synchronization state
    - An int representing e.g., whether lock is in locked state
  - Blocking and unblocking threads
    - Using LockSupport.park/unpark
  - Queuing

AQS Queuing

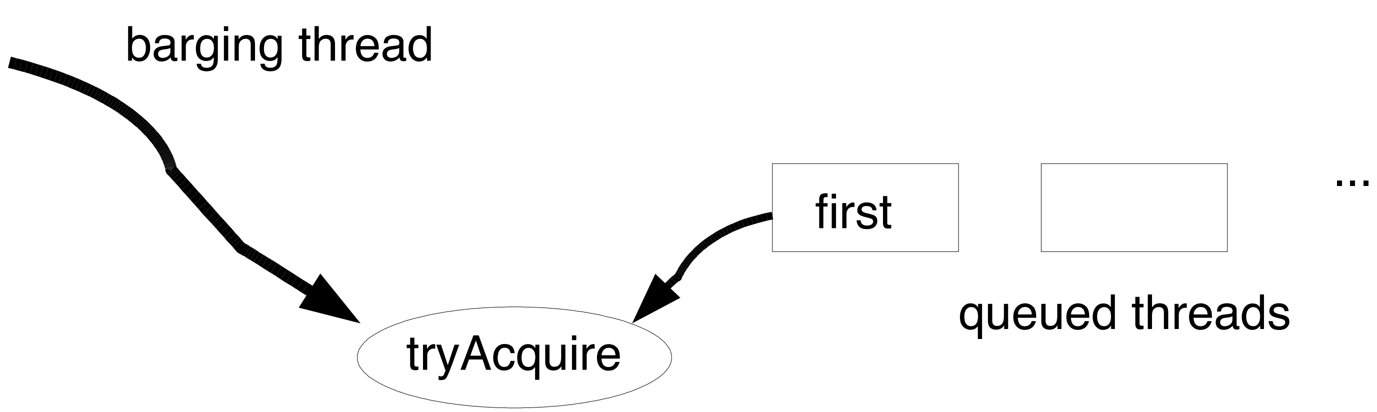
- An extension of an extension of CLH locks
  - CLH handles cancellation better and usually faster than MCS (See Scott & Scherer)
- Modified as blocking lock, not spin lock
  - Acquirability based on sync state, not just node state
  - Wake up successor (if needed) upon release
- Supports timeout, interrupt, fairness, exclusive vs shared modes
- Fast single-CAS queue insertion using explicit pred pointers
  - Signal status information for a node held in its predecessor
- Also next-pointers to enable signalling (unpark)
  - Not atomically assigned; Use pred ptrs as backup
- Lock Conditions use same representations, different queues
  - Condition signalling via queue transfer

Queuing Mechanics



FIFO with Barging

- **Incoming threads and unparked first thread may race to acquire**
  - **Reduces the expected time that a lock (etc) is needed, available, but not yet acquired.**
  - **FIFOness avoids most unproductive contention**
- **Disable barging by coding tryAcquire to fail if current thread is not first queued thread**
  - **Worthwhile for preventing starvation only when hold times long *and* contention high**
  - **Possible but much too costly to automatically adapt, so rely on user to set mode**



Performance

- **Uncontended overhead either somewhat better or worse than builtin depending on JVM, processor and usage context (ns/lock)**

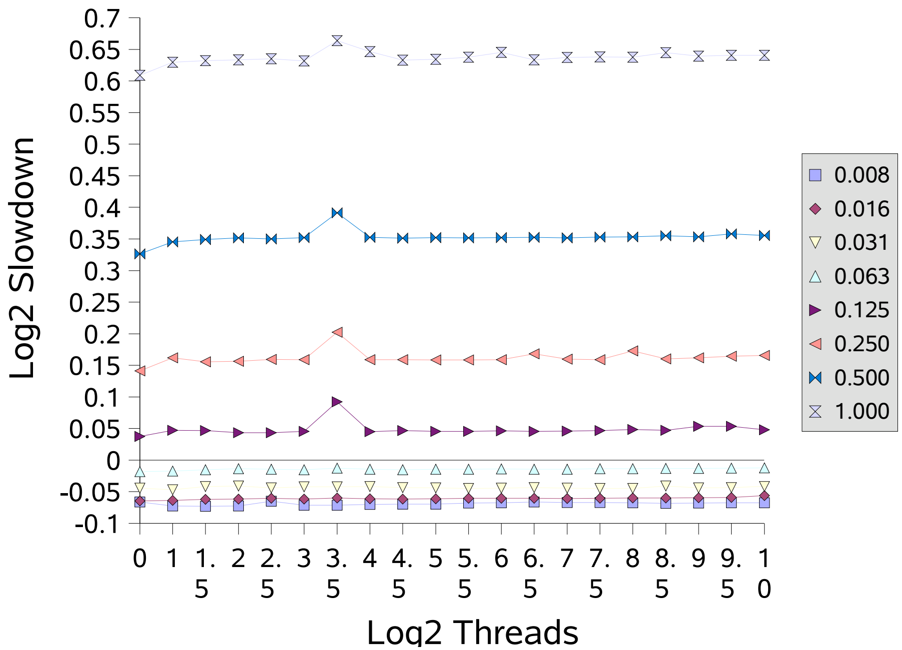
Machine	*Builtin*	*Mutex*	*Reentrant*	*Fair*
1P	18	9	31	37
2P	58	71	77	81
2A	13	21	31	30
4P	116	95	109	117
1U	90	40	58	67
4U	122	82	100	115
8U	160	83	103	123
24U	161	84	108	119

- **On saturation (256 threads) FIFO-with-Barging keeps locks busy**

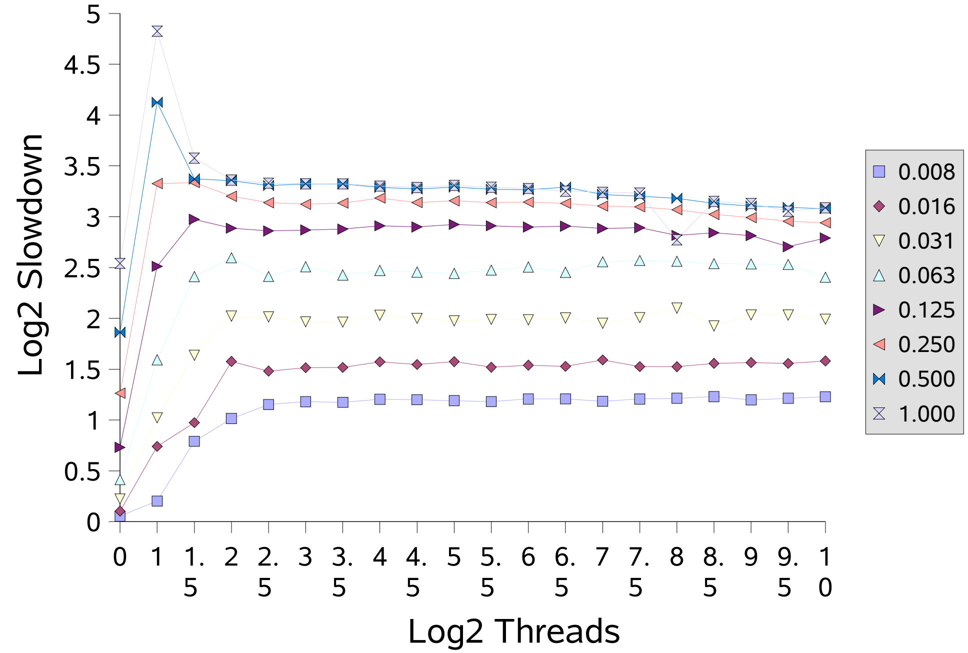
Machine	*Builtin*	*Mutex*	*Reentrant*	*Fair*
1P	521	46	67	8327
2P	930	108	132	14967
2A	748	79	84	33910
4P	1146	188	247	15328
1U	879	153	177	41394
4U	2590	347	368	30004
8U	1274	157	174	31084
24U	1983	160	182	32291

Throughput under Contention

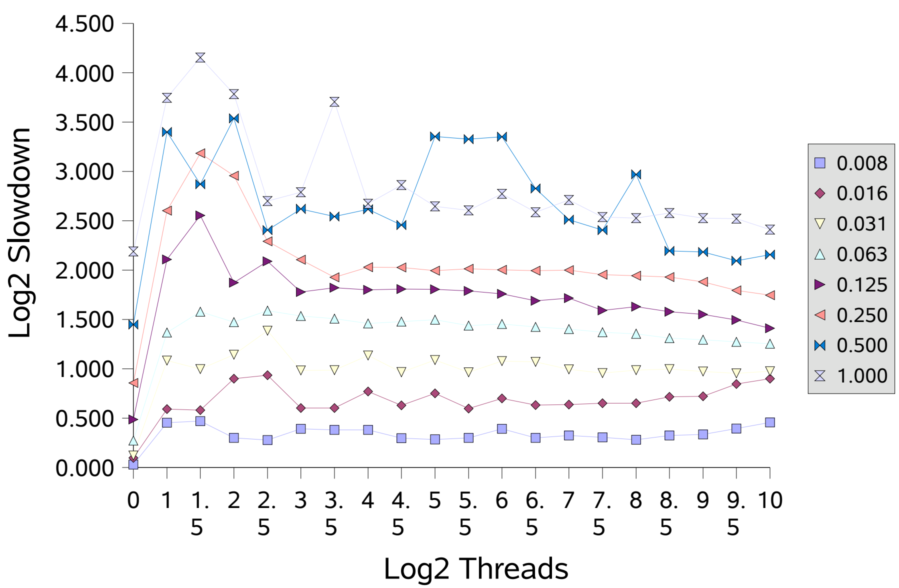
Sparc Uniprocessor



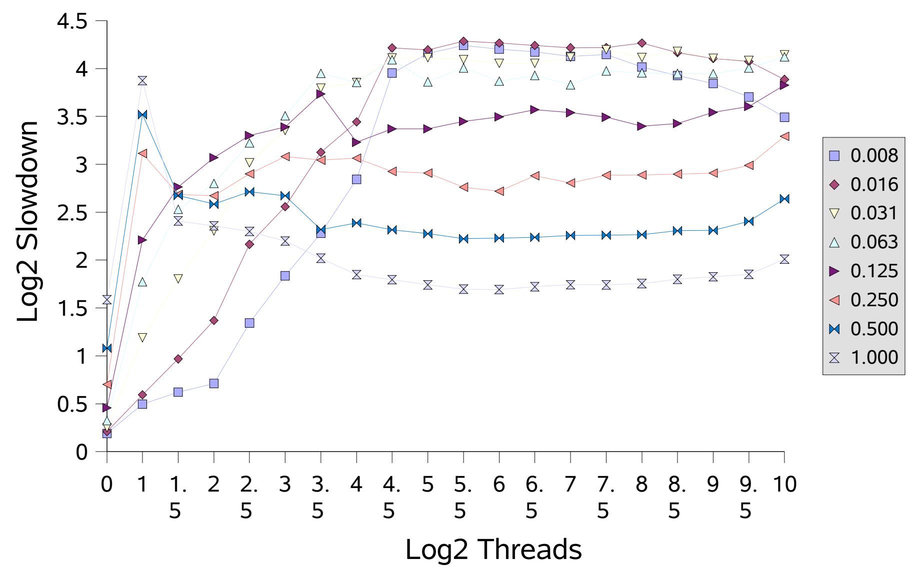
Dual hyperthread Xeon / linux



Dual P3/linux



24-way Ultrasparc 3



Queues

```
interface Queue<E> extends Collection<E>    { // ...
  boolean offer(E x);
  E poll();
  E peek();
}

interface BlockingQueue<E> extends Queue<E> { // ...
  void put(E x) throws InterruptedException;
  E take() throws InterruptedException;
  boolean offer(E x, long timeout, TimeUnit unit);
  E poll(long timeout, TimeUnit unit);
}
```

- **Note: `Collection` already supports lots of methods – iterators, remove(x), etc. These can be more challenging to implement than the queue methods. People rarely use them, but sometimes desperately need them.**

Using BlockingQueues

```
class LogWriter {
  private BlockingQueue<String> msgQ =
    new LinkedBlockingQueue<String>();

  public void writeMessage(String msg) throws IE {
    msgQ.put(msg);
  }

  // run in background thread
  public void logServer() {
    try {
      for(;;) {
        System.out.println(msgQ.take());
      }
    }
    catch(InterruptedException ie) { ... }
  }
}
```

Classic Monitor-Based Queues

```
class BoundedBuffer<E> implements Queue<E> { // ...
    Lock lock = new ReentrantLock();
    Condition notFull  = lock.newCondition();
    Condition notEmpty = lock.newCondition();
    Object[] items = new Object[100];
    int putptr, takeptr, count;
    public void put(E x)throws IE {
      lock.lock(); try {
        while (count == items.length)notFull.await();
        items[putptr] = x;
        if (++putptr == items.length) putptr = 0;
        ++count;
        notEmpty.signal();
      } finally { lock.unlock(); }
    }
    public E take() throws IE {
      lock.lock(); try {
        while (count == 0) notEmpty.await();
        Object x = items[takeptr];
        if (++takeptr == items.length) takeptr = 0;
        --count;
        notFull.signal();
        return (E)x;
      } finally { lock.unlock(); }
  } } // j.u.c.ArrayBlockingQueue class is along these lines
```

SynchronousQueues

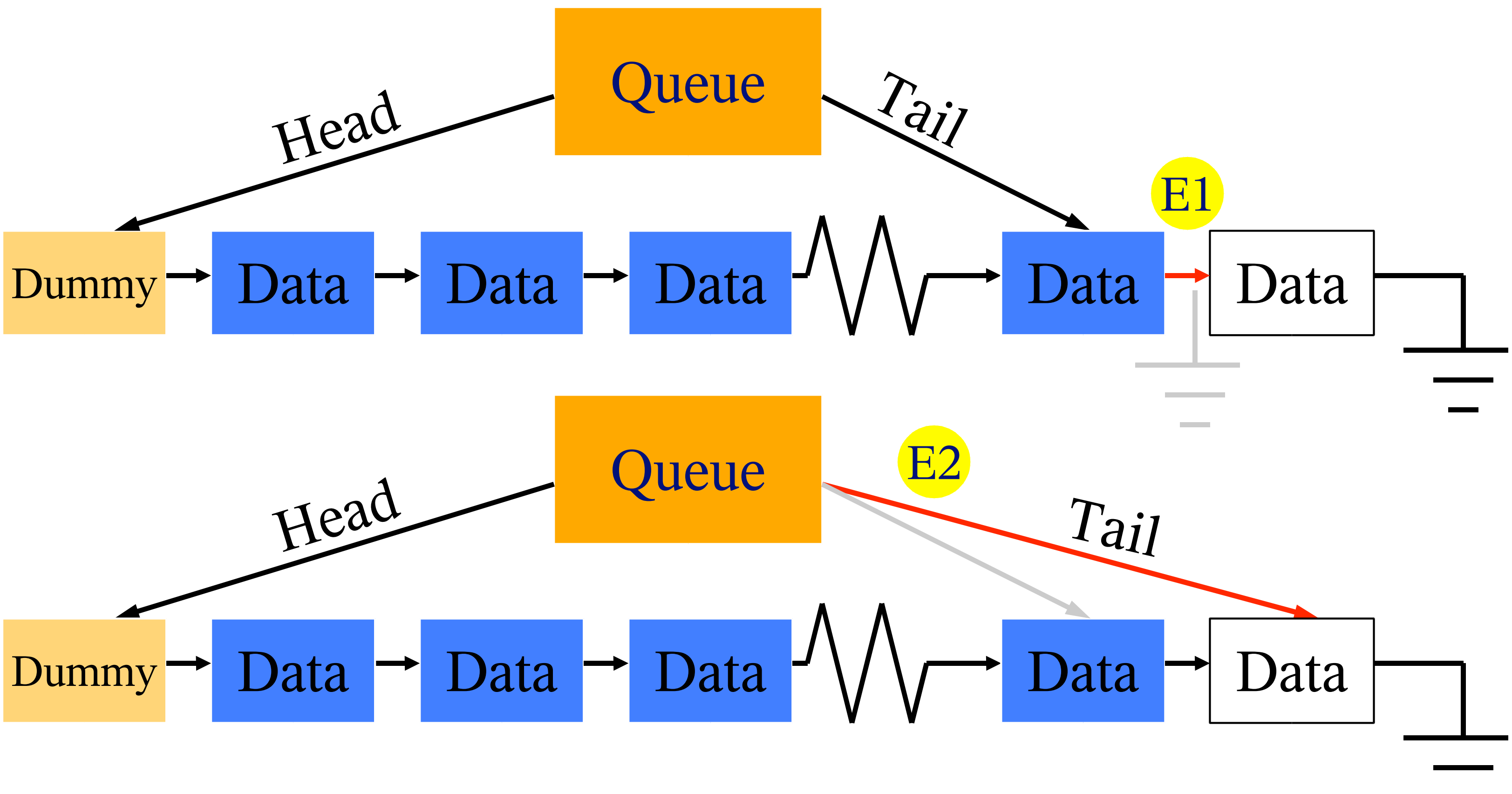
- Tightly coupled communication channels
  - Producer awaits consumer and vice versa
- Seen throughout theory and practice of concurrency
  - Implementation of language primitives
    - CSP handoff, Ada rendezvous
  - Message passing software
  - Handoffs
    - Java.util.concurrent.ThreadPoolExecutor
- Historically, expensive to implement
  - But lockless mostly nonblocking approach very effective

Dual SynchronousQueue Derivation

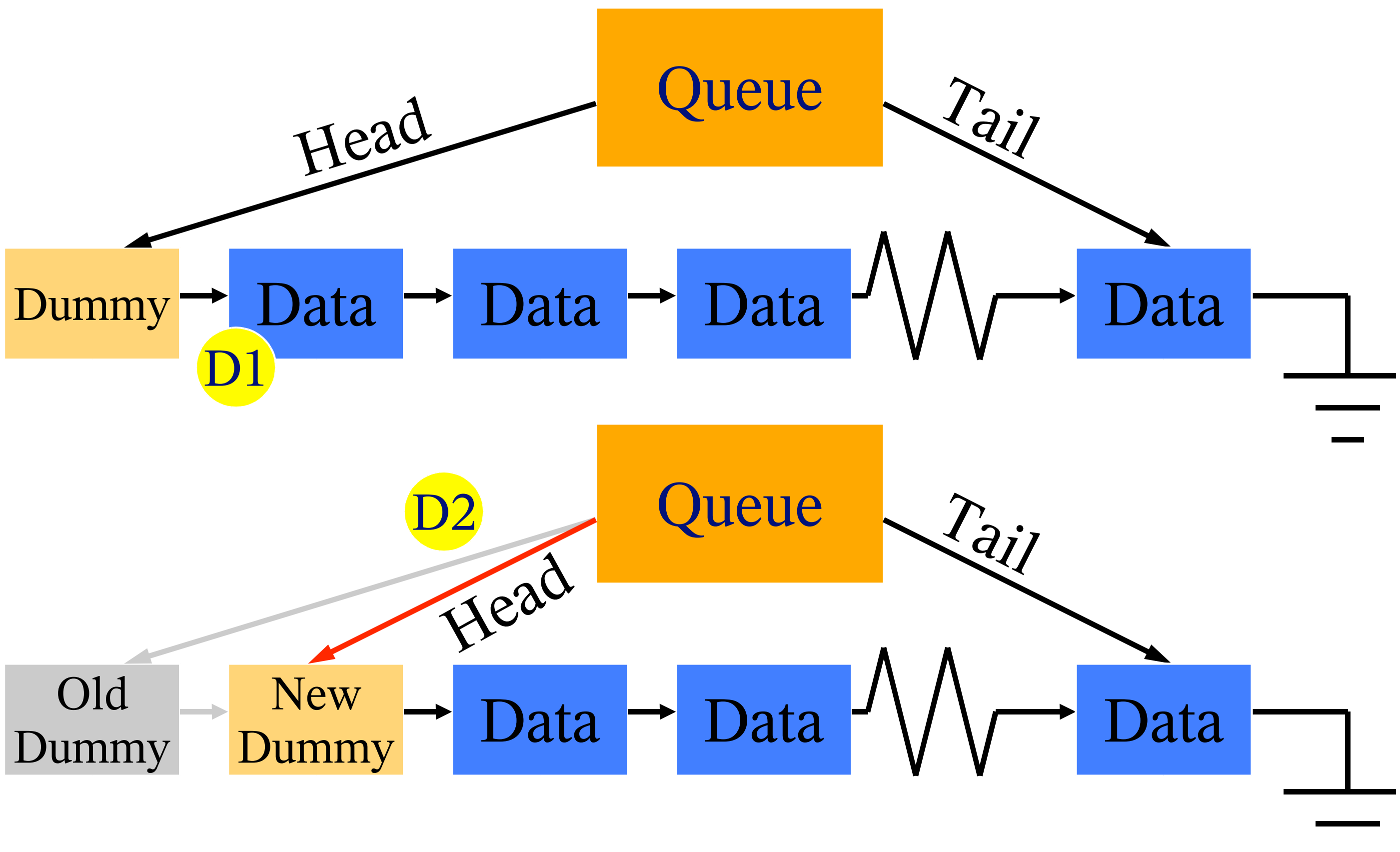
Illustrated next. See paper/code for others

Fair mode		Unfair mode
M&S Queue	Base Algorithm	Treiber Stack
Dual Queue	Consumer Blocking	Dual Stack
Fair SQ	Producer Blocking, Timeout, Cleanup	Unfair SQ

M&S Queue: Enqueue



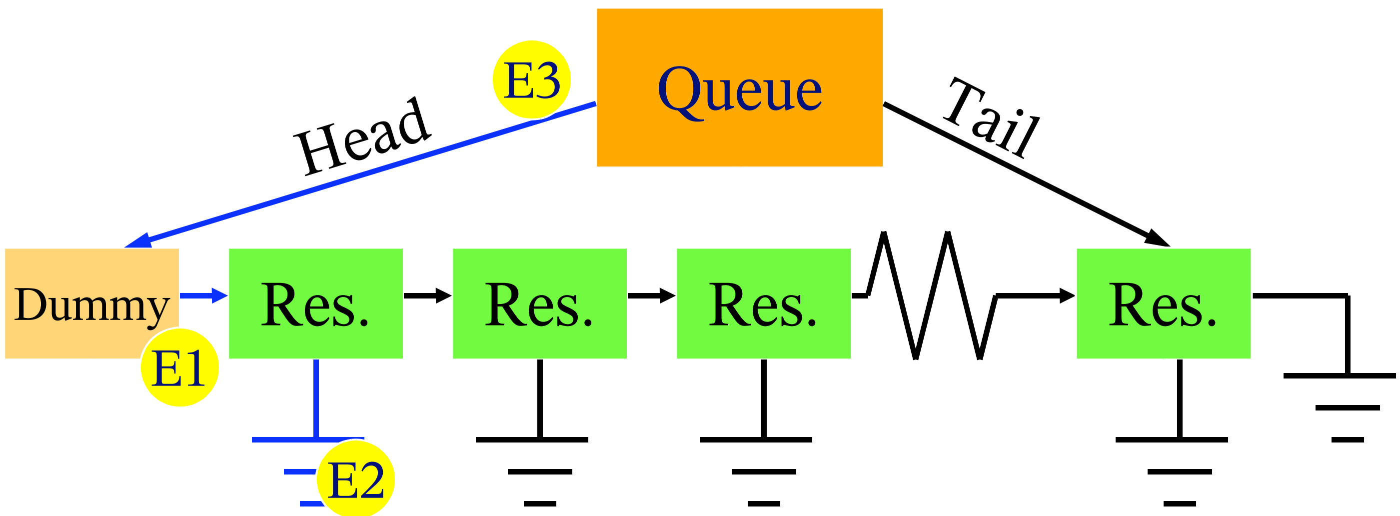
M&S Queue: Dequeue



Dual M&S Queues

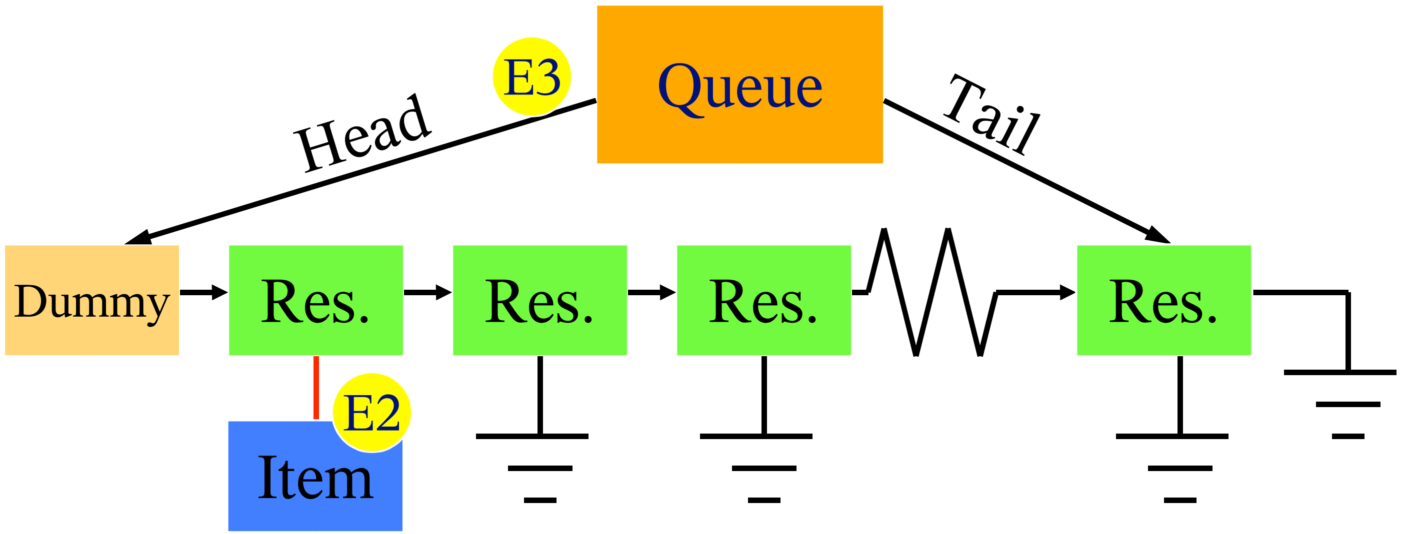
- Separate data, request nodes (flag bit)
  - Queue always all-data or all-requests
- Same behavior as M&S queue for data
- Reservations are antisymmetric to data
  - dequeue enqueues a reservation node
  - enqueue satisfies oldest reservation
- Tricky consistency checks needed
  - Dummy node can be datum or reservation
  - Extra state to watch out for (more corner cases)

DQ: Enqueue item when requests exist



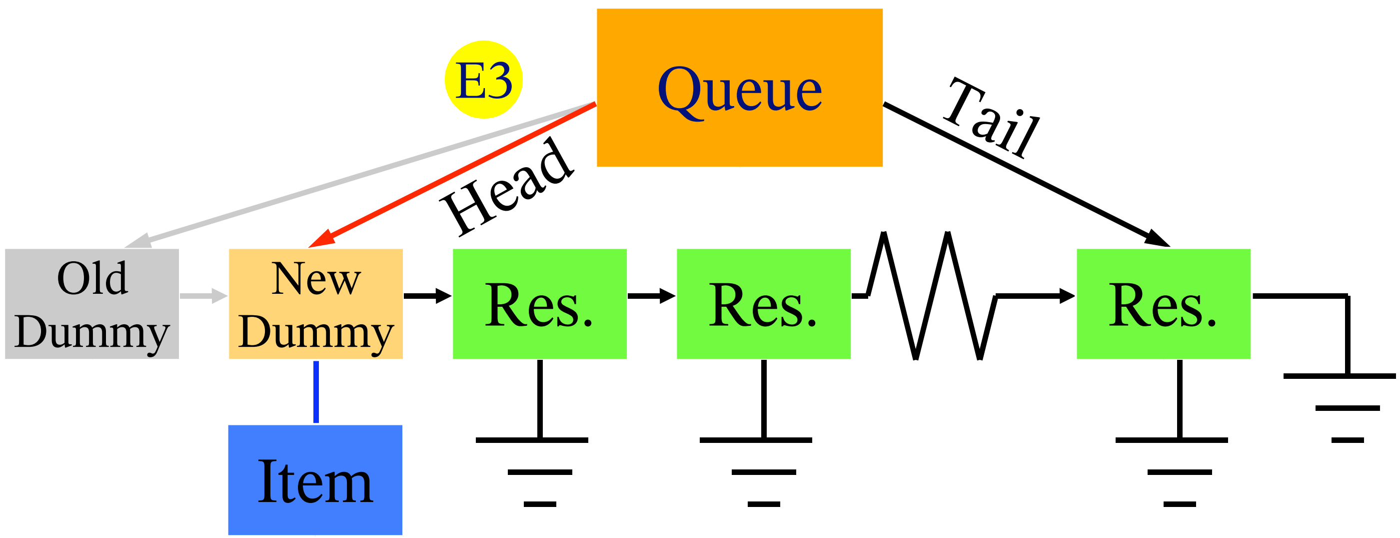
- E1 Read dummy's next ptr
- E2 CAS reservation's data ptr from null to item
- E3 Update head ptr

DQ: Enqueue (2)



- E1 Read dummy's next ptr
- E2 CAS reservation's data ptr from null to item
- E3 Update head ptr

DQ: Enqueue (3)

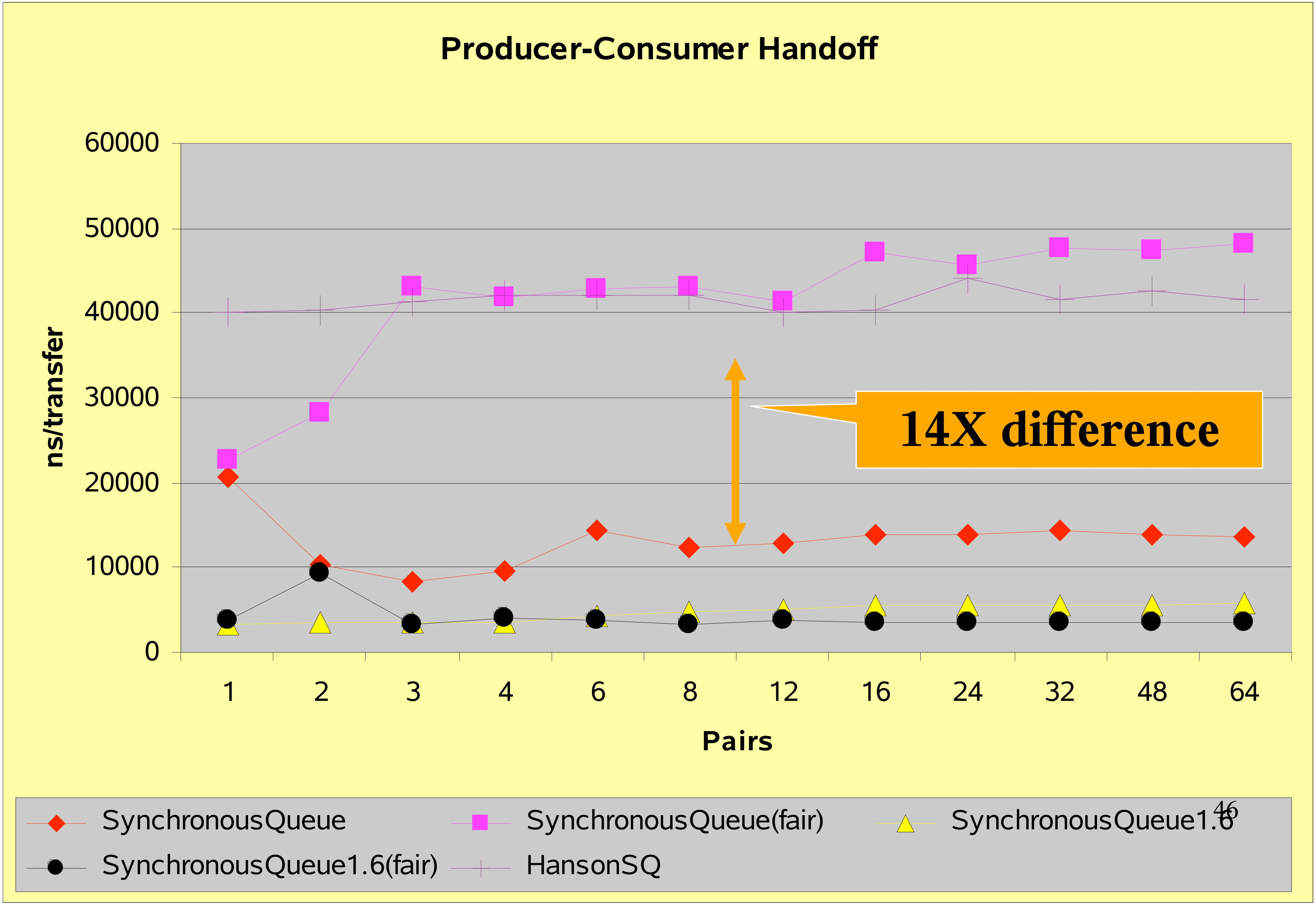


- E1 Read dummy's next ptr
- E2 CAS reservation's data ptr from null to item
- E3 Update head ptr

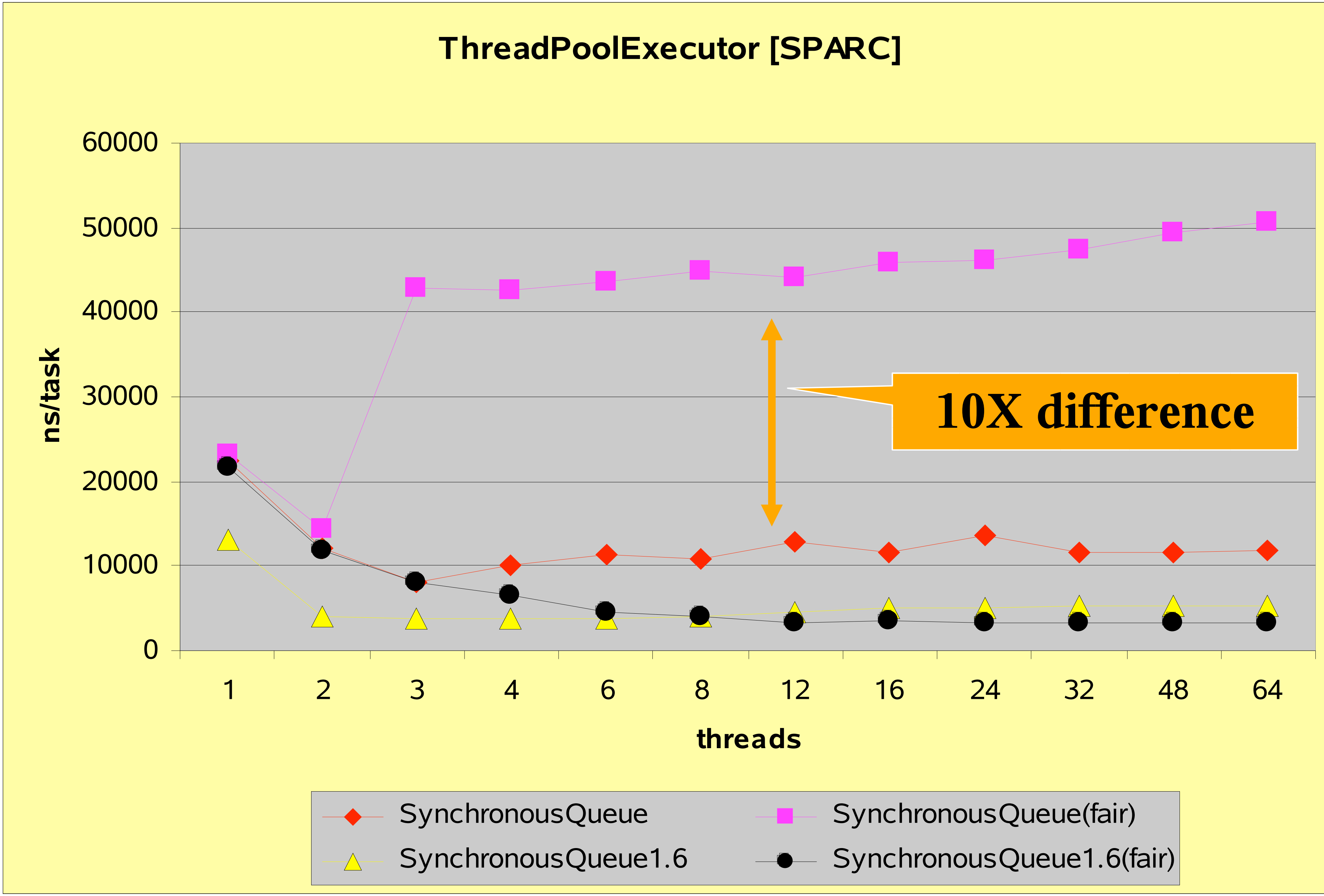
Synchronous Dual Queue

- Implementation extends dual queue
- Consumers already block for producers
  - Add blocking for the “other direction”
- Add item ptr to data nodes
  - Consumers CAS from null to “satisfying request”
  - Once non-null, any thread can update head ptr
  - Timeout support
    - Producer CAS from null back to self to indicate unusable
    - Node reclaimed when it reaches head of queue: seen as fulfilled node
- See the paper and code for details

SQ Performance (16way sparc)



ThreadPoolExecutor Impact (16way sparc)



Collection Usage

- 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 lock-based 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

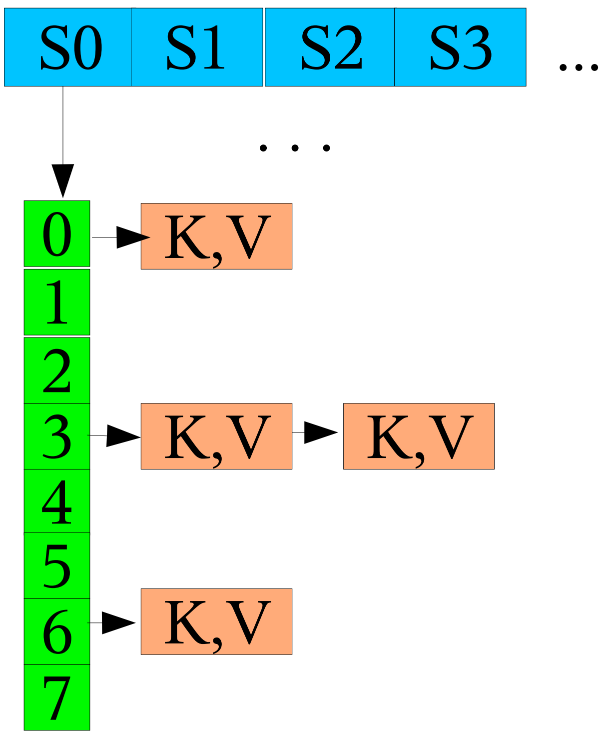
- Large design space, including
  - Locks: Coarse-grained, fine-grained, ReadWrite locks
  - Concurrently readable – reads never block, updates use locks
  - Optimistic – never block but may spin
  - Lock-free – concurrently readable and updatable
- Most initial JSR-166 collections concurrently readable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

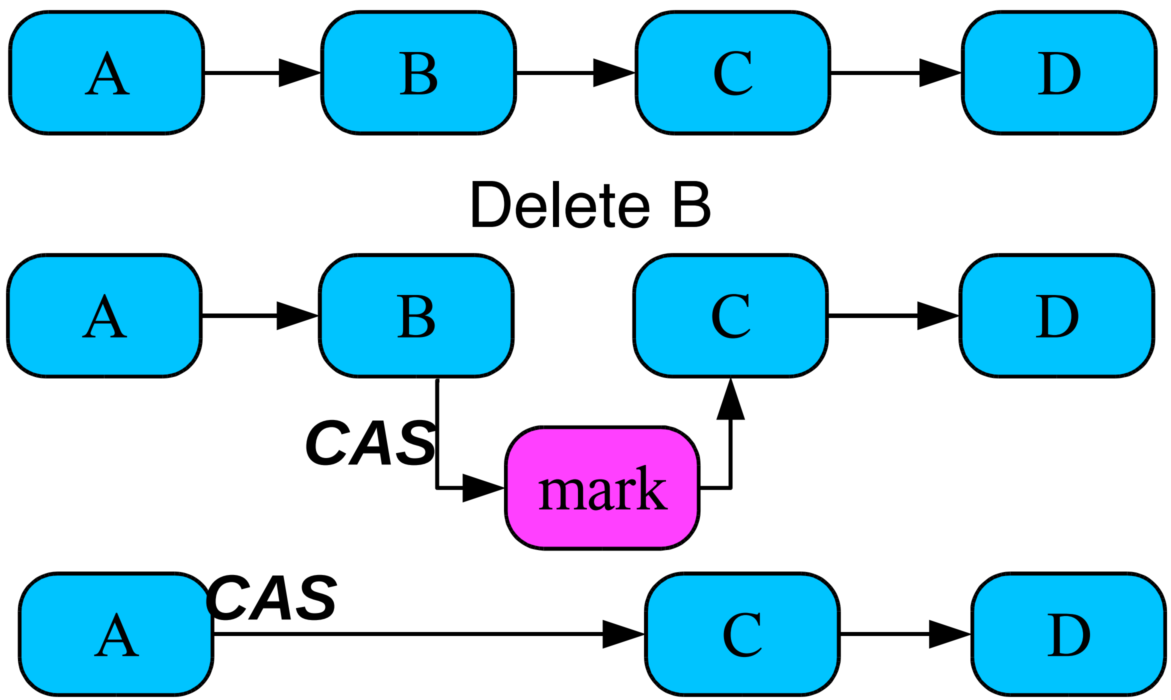
Concurrent Hash Maps

- A segment is a mini hash map
  - A resizable array of lists
  - Array and list both concurrently readable during updates
  - Need functional-list style copying on remove etc
- Use multiple segments to alleviate update contention
  - Index on low bits of hash
  - Per-segment lock used in put, remove, resize
  - Lock-free possible but hard to avoid overhead



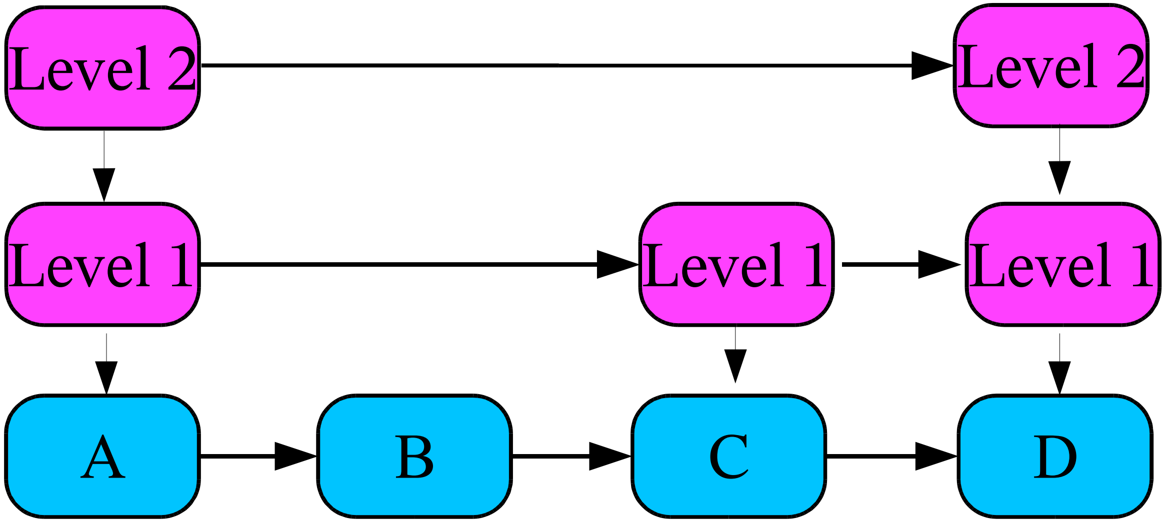
Linear Sorted Lists

- 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*
  - In turn apply to ConcurrentSkipListMaps



ConcurrentSkipListMap

- Each node has random number of index levels
  - Each index a separate node, not array element
  - Each level on average twice as sparse
- Base list uses sorted list insertion and removal algorithm
- Index nodes use cheaper variant because OK if (rarely) lost



Parallel Operations on Collections

- Analogs of HPC-style parallel array constructions
- But more challenging
  - Irregular data structures
    - Usually cannot statically partition
  - Cope with exceptions
    - Usually desire exception in one task to cancel others
  - Cope with cancellation
    - Interrupts
    - Timeouts
- Custom constructions can use ExecutorCompletionService
  - But not simple to use and doesn't always apply
  - Needs better support to be more widely useful
- One (very) special form exists in ExecutorService.invokeAll

Exercise

- Implement

```
class Applyer {
  final ExecutorService exec;
  Applyer(ExecutorService ex) { exec = ex; }
  void applyToAll(Collection<T> c,
                  Procedure<T> action);
  // anything else
}
```

- Where

```
interface Procedure<T> { void call(T arg); }
```

- Try it on a list of Integers, where the action is just to print them

Background: Interrupts

- `void Thread.interrupt()`
  - NOT asynchronous!
  - Sets the interrupt state of the thread to true
  - Flag can be tested and an `InterruptedException` thrown
  - Used to tell a thread that it should *cancel* what it is doing:
    - May or may not lead to thread termination
- What could test for interruption?
  - Methods that throw `InterruptedException`
    - sleep, join, wait, various library methods
  - I/O operations that throw `IOException`
    - But this is broken

By convention, *most* methods that throw an interrupt related exception clear the interrupt state first.

Checking for Interrupts

- `static boolean Thread.interrupted()`
  - Returns true if the current thread has been interrupted
  - Clears the interrupt state
- `boolean Thread.isInterrupted()`
  - Returns true if the specified thread has been interrupted
  - Does not clear the interrupt state
- Golden rule for library code:

  Never hide the fact an interrupt occurred

  - Either re-throw the interrupt related exception, or
  - Re-assert the interrupt state:

    `Thread.currentThread().interrupt();`

Responding to Interruptions

- Early return
  - Clean up and exit without producing or signalling errors
  - May require rollback or recovery
  - Callers can poll cancellation status if necessary to find out why action was not carried out
- Continuation (ignoring cancellation status)
  - When it is too dangerous to stop
  - When partial actions cannot be backed out
  - When it doesn’t matter (but consider lowering priority)
- Re-throwing `InterruptedException`
  - When callers must be alerted on method return
- Throwing a general failure Exception
  - When interruption is one of many reasons method can fail