The Design and Engineering of Concurrency Libraries

Doug Lea
SUNY Oswego

Outline

- Overview of Java concurrency support
  - java.util.concurrent
- Some APIs, usages, and underlying algorithms for:
  - Task-based parallelism
    - Executors, Futures, ForkJoinTasks
    - Implementation using weak memory idioms
  - Synchronization
  - Queues
  - Other Collections, Sets, and Maps
- With occasional code walk-throughs
  - See http://gee.cs.oswego.edu/dl/concurrency-interest/index.html

Developing Libraries

- Potentially rapid and wide adoption
  - Trying new library easier than new language
- Support best ideas for structuring programs
  - Improve developer productivity, application quality
  - Drive new ideas
- Continuous evaluation
  - Developer feedback on functionality, usability, bugs
  - Ongoing software engineering, quality assurance
- Explore edges among compilers, runtimes, applications
  - Can be messy, hacky

Diversity

Parallel and concurrent programming have many roots

- **Functional, Object-oriented, and ADT-based procedural patterns are all well-represented; including:**
  - **Parallel (function) evaluation**
  - **Bulk operations on aggregates (map, reduce etc)**
  - **Shared resources (shared registries, transactions)**
  - **Sending messages and events among objects**
- **But none map uniformly to platforms**
  - **Beliefs that any are most fundamental are delusional**
  - **Arguments that any are “best” are silly**
  - **Libraries should support multiple styles**
    - **Avoiding policy issues when possible**

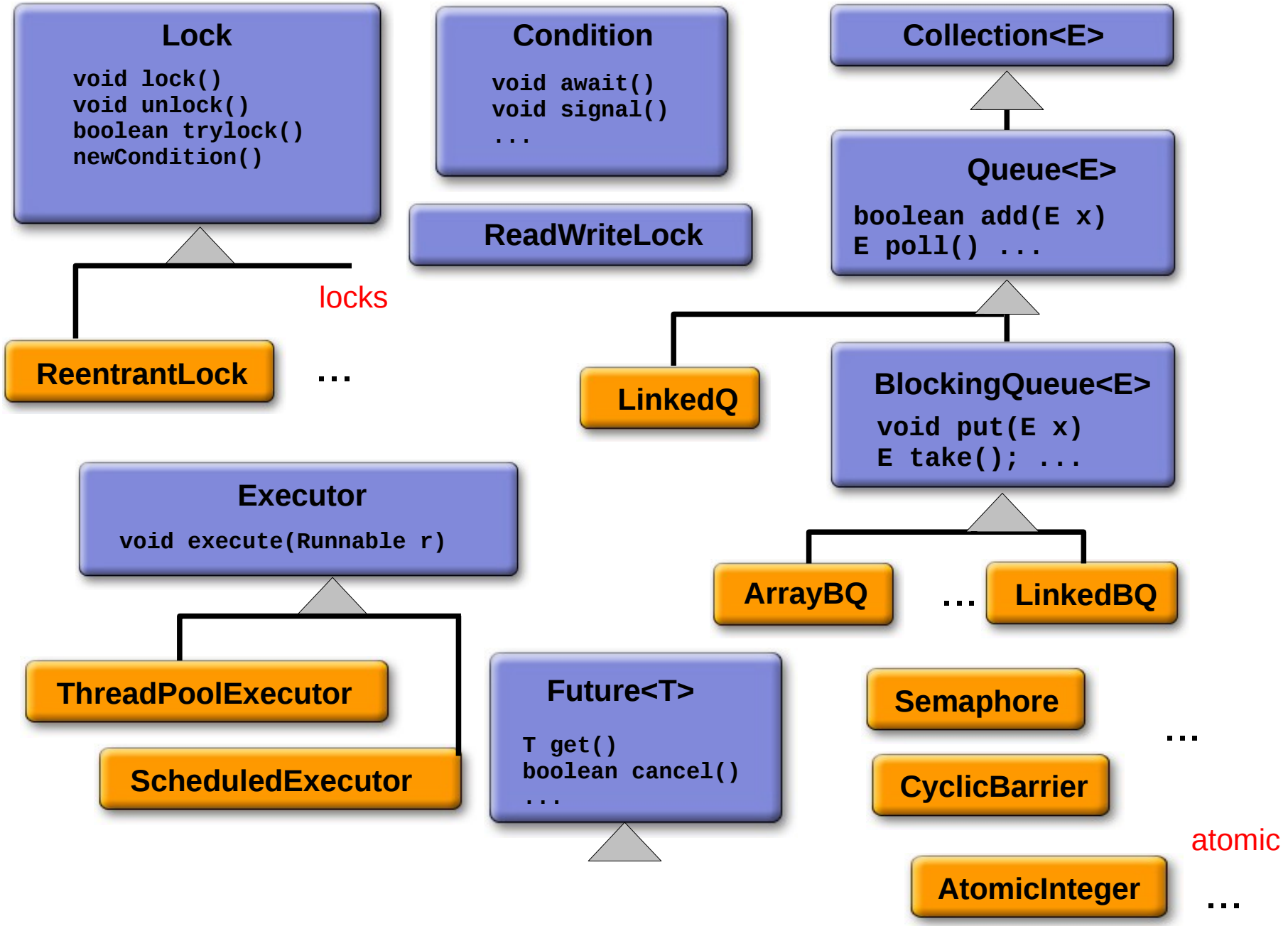
Core Java 1.x Concurrency Support

- **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 V5

- Executor framework
  - ThreadPools, Futures, CompletionService
- Atomic vars (`java.util.concurrent.atomic`)
  - JVM support for compareAndSet (CAS) operations
- Lock framework (`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

Main j.u.c components



java.util.concurrent V6-V8

- More Executors
  - ForkJoinPool; support for parallel java.util.Streams
- More Queues
  - LinkedTransferQueue, ConcurrentLinkedDeque
- More Collections
  - ConcurrentSkipList{Map, Set}, ConcurrentSets
- More Atomics
  - Weak access methods, LongAdder
- More Synchronizers
  - Phasers, StampedLocks
- More Futures
  - ForkJoinTask, CompletableFuture

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 ...

Exposing Parallelism

Old Elitism: Hide from most programmers

- "Programmers think sequentially"
- "Only an expert should try to write a <X>"
- "<Y> is a kewl hack but too weird to export"

End of an Era

- Few remaining hide-able speedups (Amdahls law)
- Hiding is impossible with multicores, GPUs, FPGAs

New Populism: Embrace and rationalize

- Must integrate with defensible programming models, language support, and APIs
- Some residual quirkiness is inevitable

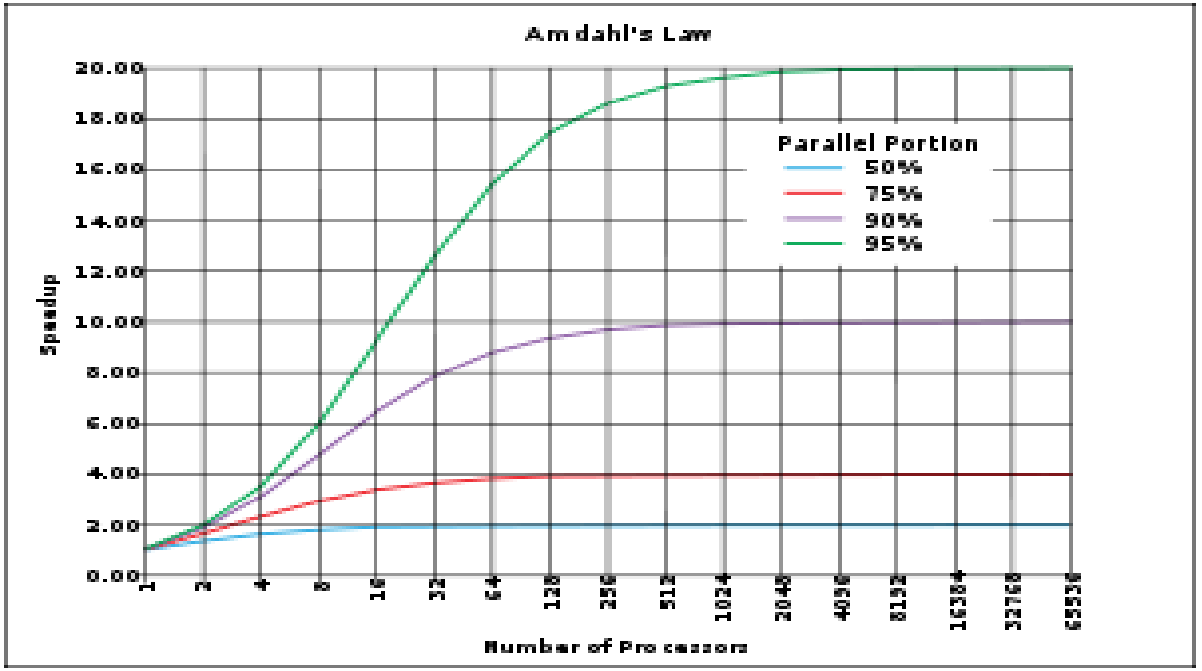
Parallelizing Arbitrary Expressions

- Instruction-level parallelism doesn't scale well
  - But can use similar ideas on multicores
  - With similar benefits and issues
- Example: `val e = f(a, b) op g(c, d)  // scala`
- Easiest if rely on shallow dependency analysis
  - Methods f and g are pure, independent functions
  - Can exploit commutativity and/or associativity
- Other cases require harder work
  - To find smaller-granularity independence properties
    - For example, parallel sorting, graph algorithms
  - Harder work → more bugs; sometimes more payoff

Limits of Parallel Evaluation

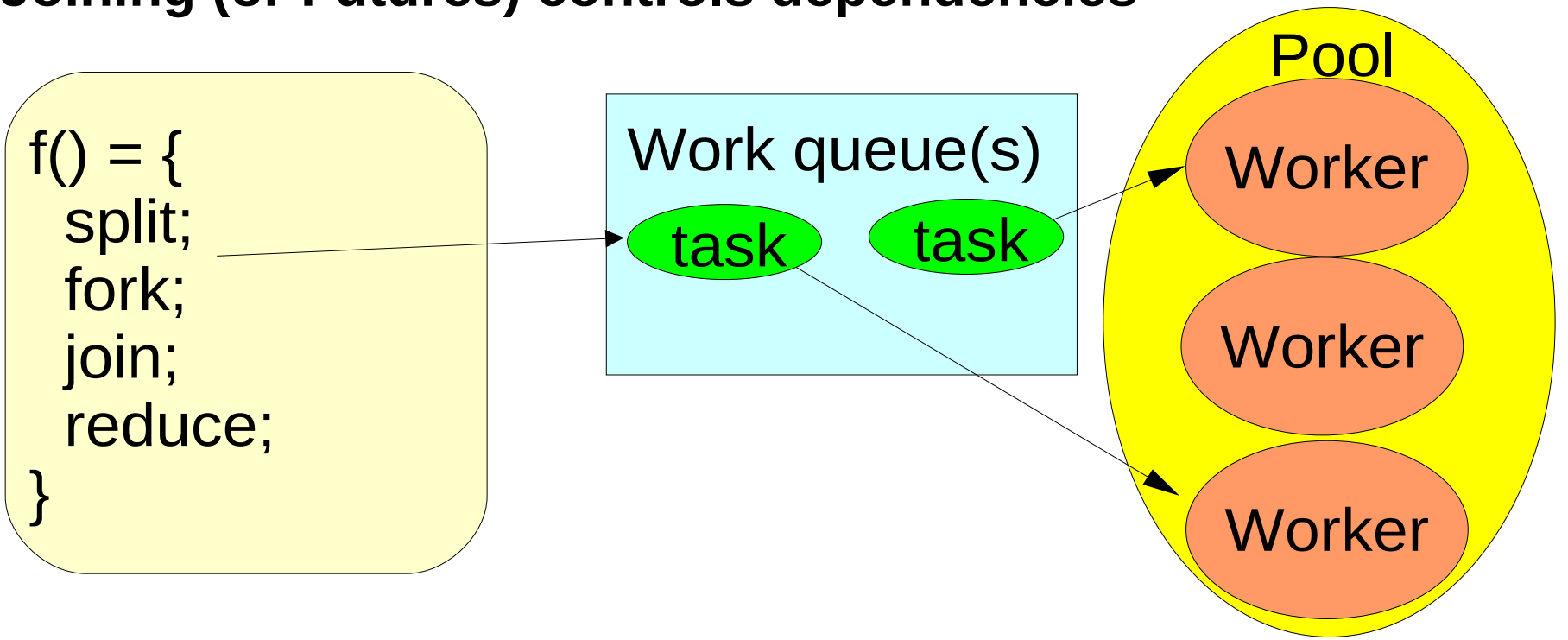
- Why can't we always parallelize to turn any O(N) problem into O(N / #processors)?
  - Sequential dependencies and resource bottlenecks
- For program with serial time S, and parallelizable fraction f, max speedup regardless of #proc is 1 / ((1 – f) + f / S)
- Can also express in terms of critical paths or tree depths

Wikipedia



Task-Based Parallel Evaluation

- **Programs can be broken into tasks**
  - **Under some appropriate level of granularity**
- **Workers/Cores continually run tasks**
  - **Sub-computations are forked as subtask objects**
- **Sometimes need to wait for subtasks**
  - **Joining (or Futures) controls dependencies**



Executors

A GOF-ish pattern with a single method interface

```
interface Executor { void execute(Runnable w); }
```

- Separate work from workers (what vs how)
  - `ex.execute(work)`, not `new Thread(..).start()`
  - The “work” is a passive closure-like action object
- Executors implement execution policies
  - Might not apply well if execution policy depends on action
    - Can lose context, locality, dependency information
- Reduces active objects to very simple forms
  - Base interface allows trivial implementations like `work.run()` or `new Thread(work).start()`
  - Normally use group of threads: ExecutorService

Executor Framework

- Standardizes asynchronous task invocation
  - Use `anExecutor.execute(aRunnable)`
  - Not: `new Thread(aRunnable).start()`
- Two styles – non-result-bearing and result-bearing:
  - Runnables/Callables; FJ Actions vs Tasks
- A small framework, including:
  - `Executor` – something that can `execute tasks`
  - `ExecutorService` extension – shutdown support etc
  - `Executors` utility class – configuration, conversion
  - `ThreadPoolExecutor, ForkJoinPool` – implementation
  - `ScheduledExecutor` for time-delayed tasks
  - `ExecutorCompletionService` – hold completed tasks

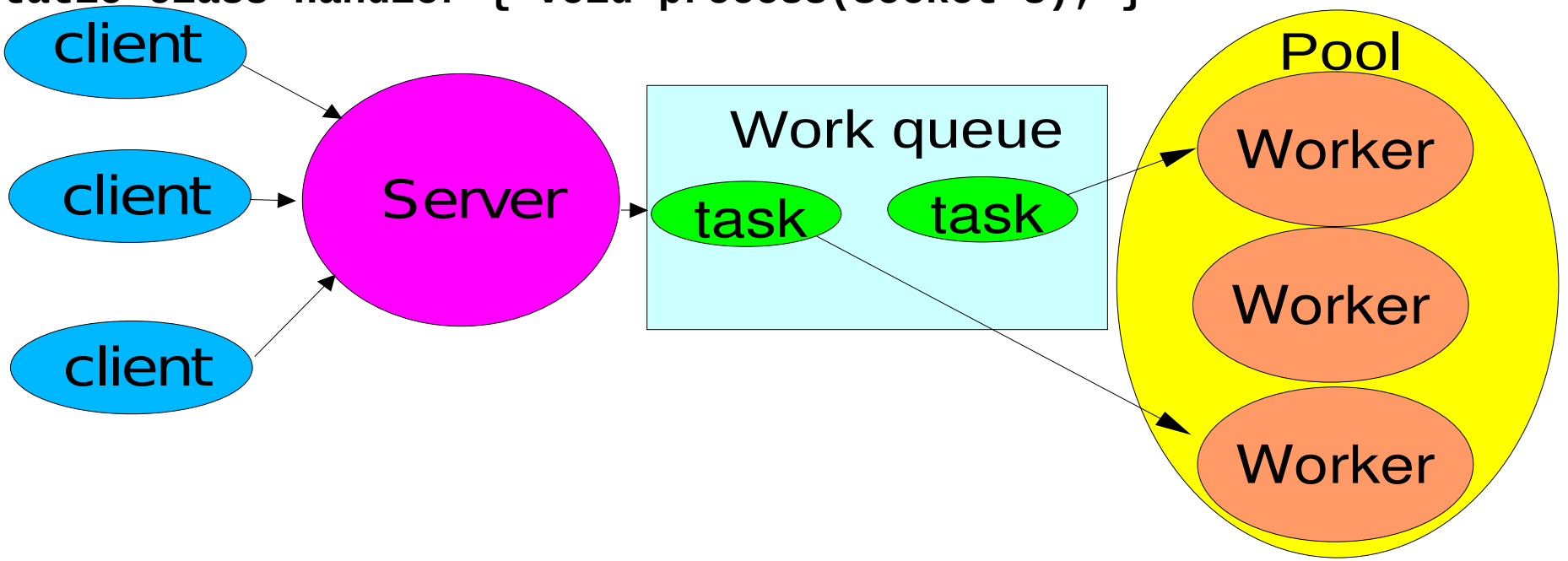
ExecutorServices

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

- Two main implementations
  - ThreadPoolExecutor (also via Executors factory methods)
    - Single (use-supplied) BlockingQueue for tasks
    - Tunable target and max threads, saturation policy, etc
    - Interception points before/after running tasks
  - ForkJoinPool
    - Distributed work-stealing queues
    - Internally tracks joins to control scheduling
    - Assumes tasks do not block on IO or other sync

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
  - 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 implementations
  - `FutureTask<V>, ForkJoinTask<V>`

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();
     }
   }
}
```

ForkJoinTasks extend Futures

- `V join()`
  - Same semantics as get, but no checked exceptions
  - Usually appropriate when computationally based
    - If not, users can rethrow as RuntimeException
- `void fork()`
  - Submits task to the same executor as caller is running under
- `void invoke()`
  - Same semantics as { t.fork(); t.join; }
  - Similarly for invokeAll
- Plus many small utilities

Parallel Recursive Decomposition

Typical algorithm

```
Result solve(Param problem) {
  if (problem.size <= THRESHOLD)
    return directlySolve(problem);
  else {
    in-parallel {
      Result l = solve(leftHalf(problem));
      Result r = solve(rightHalf(problem));
    }
    return combine(l, r);
  }
}
```

- To use FJ, must convert method to task object
  - "in-parallel" can translate to invokeAll(leftTask, rightTask)
- The algorithm itself drives the scheduling
- Many variants and extensions

Implementing ForkJoin Tasks

Queuing: Work-stealing

- Each worker forks to own deque; but steals from others or accepts new submission when no work

Scheduling: Locally LIFO, random-steals FIFO

- Cilk-style: Optimal for divide-and-conquer
- Ignores locality: Cannot tell if better to use another core on same processor, or a different processor

Joining: Helping and/or pseudo-continuations

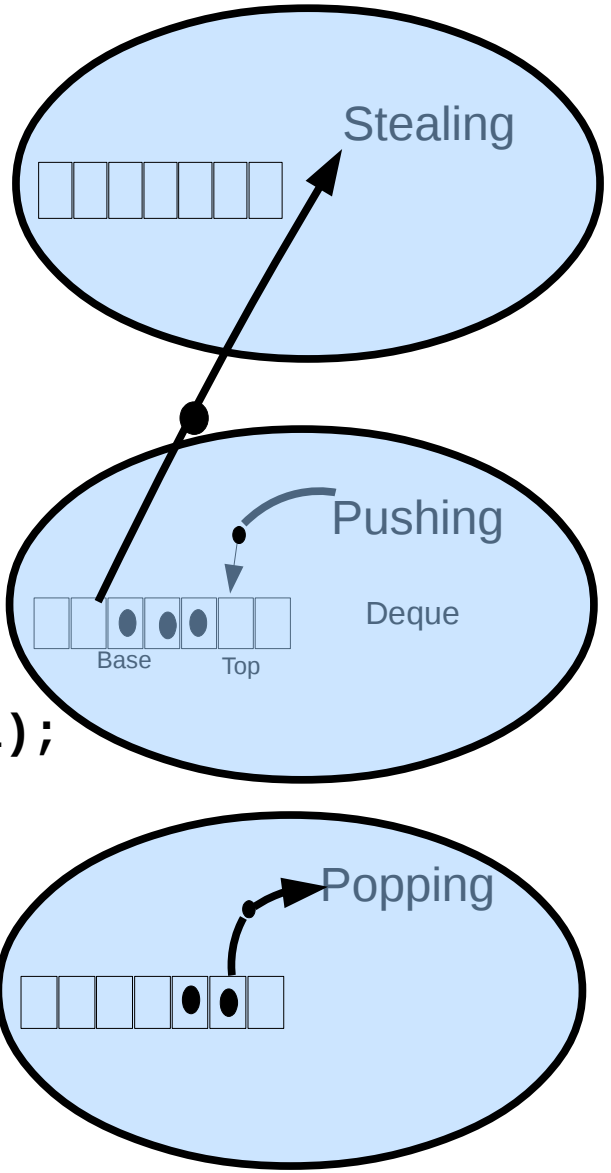
- Try to steal a child of stolen task; if none, block but (re)start a spare thread to maintain parallelism

Overhead: Task object with one 32-bit int status

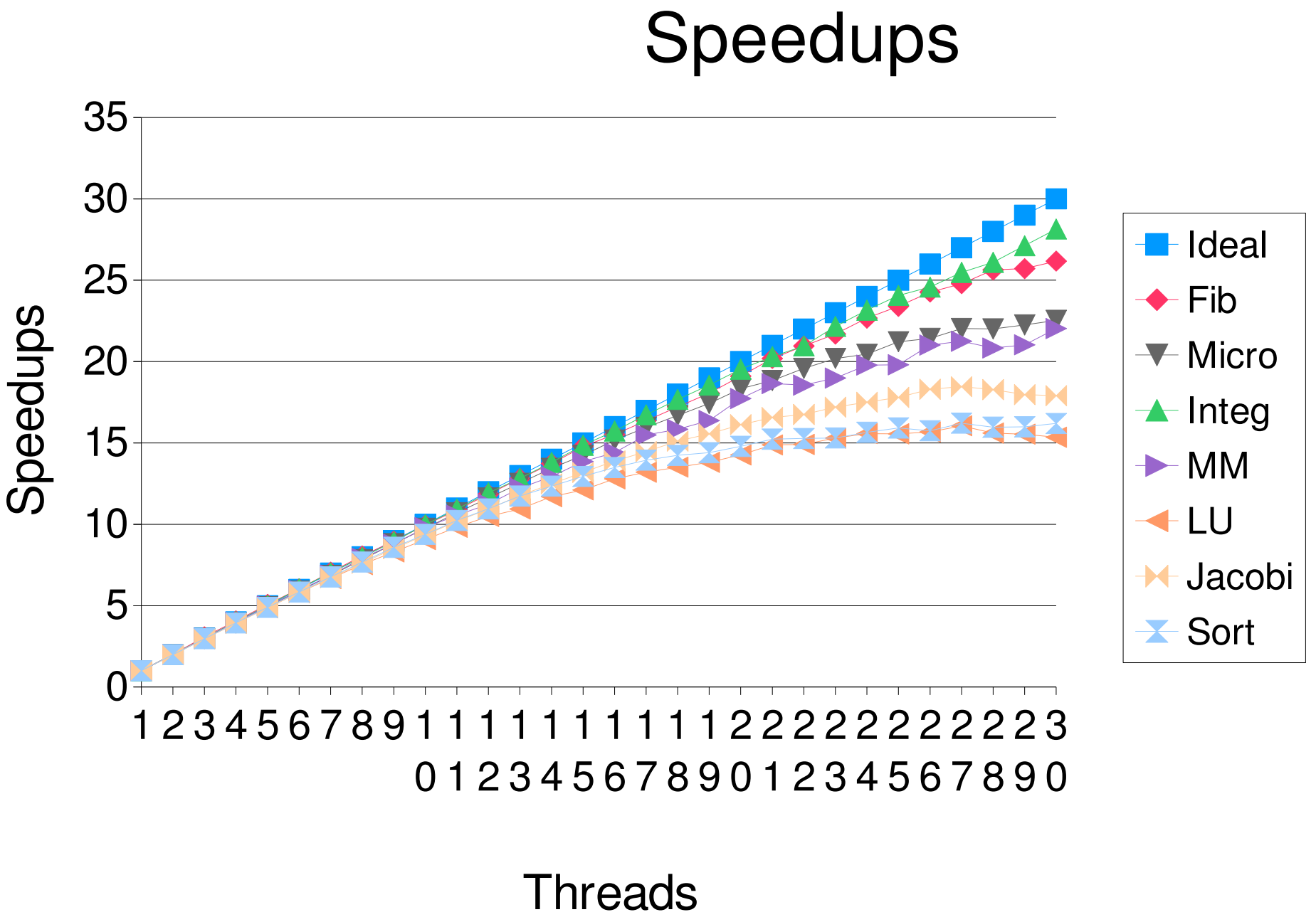
- Payoff after ~100-1000 instructions per task body

ForkJoin Sort (Java)

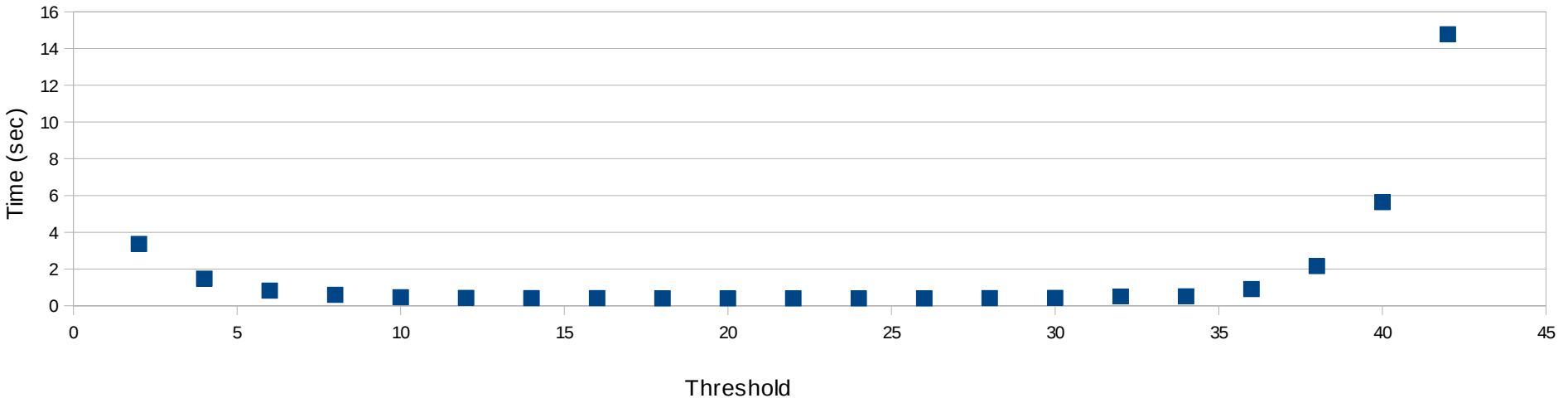

```
class SortTask extends RecursiveAction {
  final long[] array;
  final int lo; final int hi;

  SortTask(long[] array, int lo, int hi) {
    this.array = array;
    this.lo = lo; this.hi = hi;
  }

  protected void compute() {
    if (hi - lo < THRESHOLD)
      sequentiallySort(array, lo, hi);
    else {
      int m = (lo + hi) >>> 1;
      SortTask r = new SortTask(array, m, hi);
      r.fork();
      new SortTask(array, lo, m).compute();
      r.join();
      merge(array, lo, mid, hi);
    }
  }
    // …
 }
```



Speedups on 32way Sparc



Granularity Effects



Recursive Fibonacci(42) running on Niagara2

```
compute() {
if (n <= Threshold) seqFib(n);
else invoke(new Fib(n-1), new Fib(n-2)); ...}
```

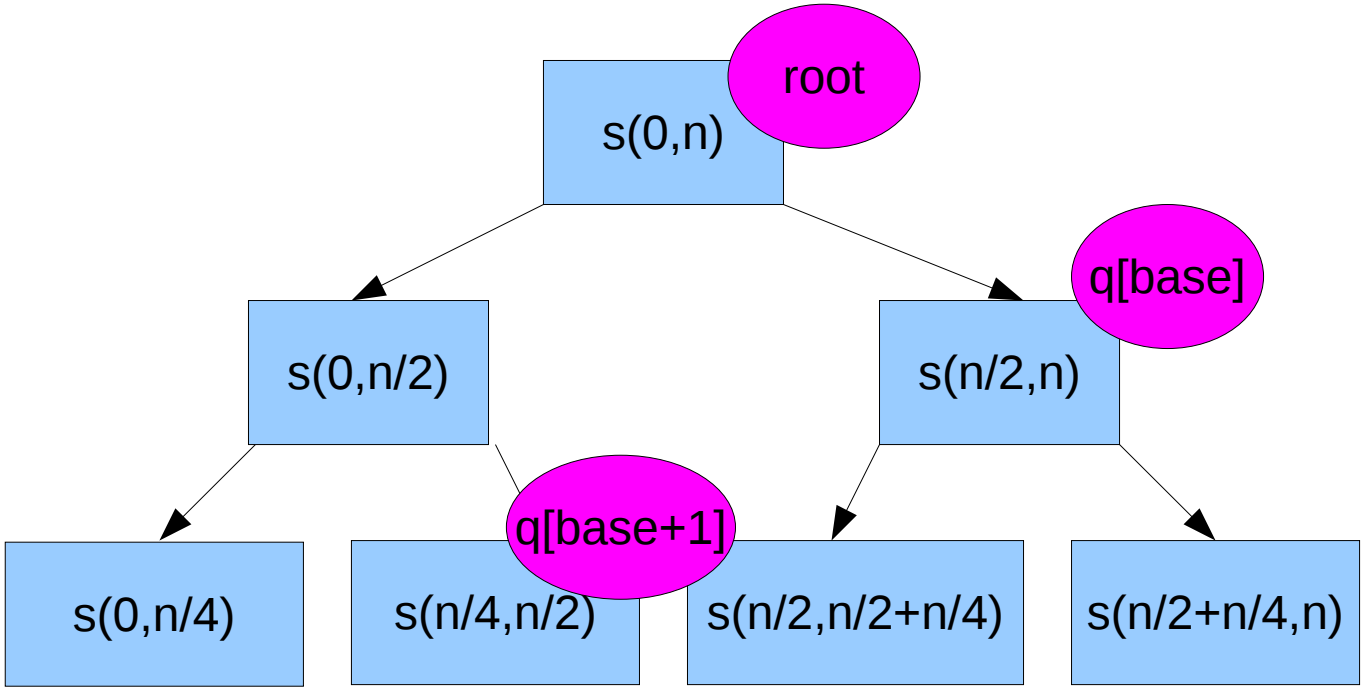
- When do you bottom out parallel decomposition?
  - A common initial complaint but usually easy decision
    - Very shallow sensitivity curves near optimal choices
  - And usually easy to automate – except for problems so small that they shouldn't divide at all

Why Work-Stealing

- Portable scalability
  - Programs work well with any number of processors/cores
  - 15+ years of experience (most notably in Cilk)
- Load-balancing
  - Keeps processors busy, improves throughput
- Robustness
  - Can afford to use small tasks (as few as 100 instructions)
- But not a silver bullet – need to overcome / avoid ...
  - Basic versions ignore processor memory affinities
  - Task propagation delays can hurt for loop constructions
  - Overly fine granularities can hit big overhead wall
  - Restricted sync restricts range of applicability
  - Sizing/Scaling issues past a few hundred processors

Computation Trees and Deques

- For recursive decomposition, deques arrange tasks with the most work to be stolen first. (See Blelloch et al for alternatives)
- Example: method s operating on array elems 0 ... n:



Blocking

- The cause of many high-variance slowdowns
  - More cores → more slowdowns and more variance
    - Blocking Garbage Collection accentuates impact
- Reducing blocking
  - Help perform prerequisite action rather than waiting for it
  - Use finer-grained sync to decrease likelihood of blocking
  - Use finer-grained actions, transforming ...
    From: Block existing actions until they can continue
    To: Trigger new actions when they are enabled
- Seen at instruction, data structure, task, IO levels
  - Lead to new JVM, language, library challenges
    - Memory models, non-blocking algorithms, IO APIs

IO

- Long-standing design and API tradeoff:
  - Blocking: suspend current thread awaiting IO (or sync)
  - Completions: Arrange IO and a completion (callback) action
- Neither always best in practice
  - Blocking often preferable on uniprocessors if OS/VM must reschedule anyway
  - Completions can be dynamically composed and executed
    - But require overhead to represent actions (not just stack-frame)
    - And internal policies and management to run async completions on threads. (How many OS threads? Etc)
  - Some components only work in one mode
- Ideally support both when applicable
  - Completion-based support problematic in pre-JDK8 Java
    - Unstructured APIs lead to “callback hell”

Blocking vs Completions in Futures

- Java.util.concurrent Futures hit similar tradeoffs
  - Completion support hindered by expressibility
    - Initially skirted “callback hell” by not supporting any callbacks. But led to incompatible 3rd party frameworks
  - JDK8 lambdas and functional interfaces enabled introduction of CompletableFutures (CF)
- CF supports fluent dynamic composition

```
CompletableFuture.supplyAsync(()->generateStuff()).
  thenApply(stuff->reduce(stuff)).thenApplyAsync(x->f(x)).
  thenAccept(result->print(result)); // add .join() to wait
```

  - Plus methods for ANDed, ORed, and flattened combinations
    - In principle, CF alone suffices to write any concurrent program
  - Not fully integrated with JDK IO and synchronization APIs
    - Adaptors usually easy to write but hard to standardize
    - Tools/languages could translate into CFs (as in C# async/await)

Using Weak Idioms

- Want good performance for core libraries and runtime systems
  - Internally use some common non-SC-looking idioms
  - Most can be seen as manual “optimizations” that have no impact on user-level consistency
  - But leaks can show up as API usage rules
    - Example: cannot fork a task more than once
- Used extensively in implementing FJ

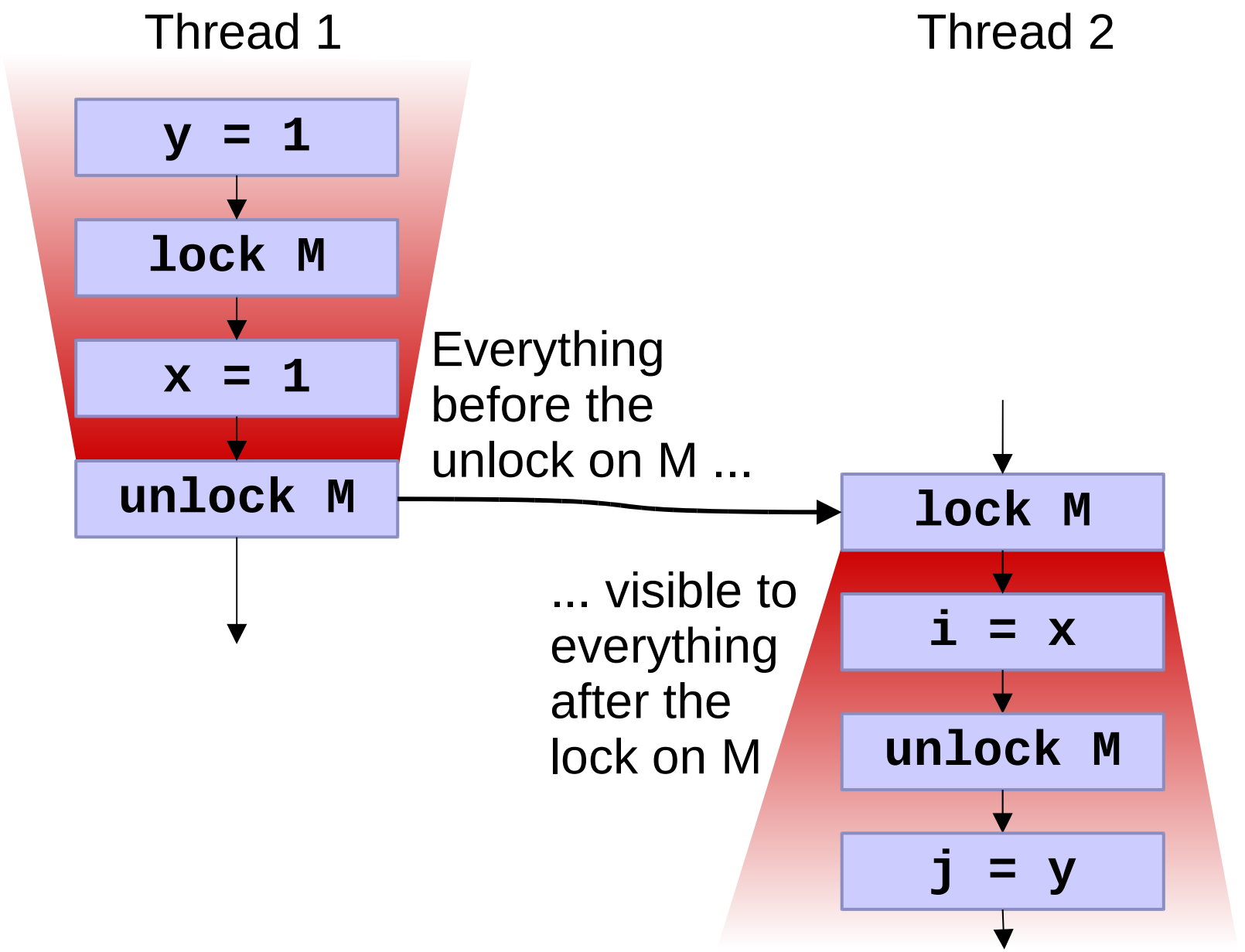
Consistency

- Processors do not intrinsically guarantee much about memory access orderings
  - Neither do most compiler optimizations
    - Except for classic data and control dependencies
  - Not a bug
    - Globally ordering all program accesses can eliminate parallelism and optimization → unhappy programmers
- Need memory model to specify guarantees and how to get them when you need them
  - Initial Java Memory Model broken
  - JSR133 overhauled specs but still needs some work

Memory Models

- Distinguish sync *accesses* (locks, volatiles, atomics) from normal accesses (reads, writes)
- Require strong ordering properties among sync
  - Usually “strong” means Sequentially Consistent
- Allow as-if-sequential reorderings among normal
  - Usually means: obey seq data/control dependencies
- Restrict reorderings between sync vs normal
  - Rules usually not obvious or intuitive
  - Special rules for cases like final fields
- There's probably a better way to go about all this

JSR-133 Main Rule



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

Additional JSR-133 Rules

- Variants of lock rule apply to volatile fields and thread control
  - Writing a volatile has same memory effects as unlock
  - Reading a volatile has same memory effects as lock
  - Similarly for thread start and termination
- Final fields
  - All threads read final value so long as 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

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

Enhanced Volatiles (and Atomics)

- Support extended atomic access primitives
  - CompareAndSet (CAS), getAndSet, getAndAdd, ...
- Provide intermediate ordering control
  - May significantly improve performance
    - Reducing fences also narrows CAS windows, reducing retries
  - Useful in some common constructions
    - Publish (release) → acquire
      - No need for StoreLoad fence if only owner may modify
    - Create (once) → use
      - No need for LoadLoad fence on use because of intrinsic dependency when dereferencing a fresh pointer
  - Interactions with plain access can be surprising
    - Most usage is idiomatic, limited to known patterns
    - Resulting program need not be sequentially consistent

Expressing Atomics

- C++/C11: standardized access methods and modes
- Java: JVM “internal” intrinsics and wrappers
  - Not specified in JSR-133 memory model, even though some were introduced internally in same release (JDK5)
  - Ideally, a bytecode for each mode of (load, store, CAS)
    - Would fit with No L-values (addresses) Java rules
  - Instead, intrinsics take object + field offset arguments
    - Establish on class initialization, then use in `Unsafe` API calls
    - Non-public; truly “unsafe” since offset args can't be checked
      - Can be used outside of JDK using odd hacks if no security mgr
      - j.u.c supplies public wrappers that interpose (slow) checks
- JEP 188 and 193 (targeting JDK9) will provide first-class specs, and improved APIs

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 semantics for volatile
  - `get` acts as volatile-read
  - `set` acts as volatile-write
  - `compareAndSet` acts as volatile-read and volatile-write
  - `weakCompareAndSet` ordered wrt accesses to same var

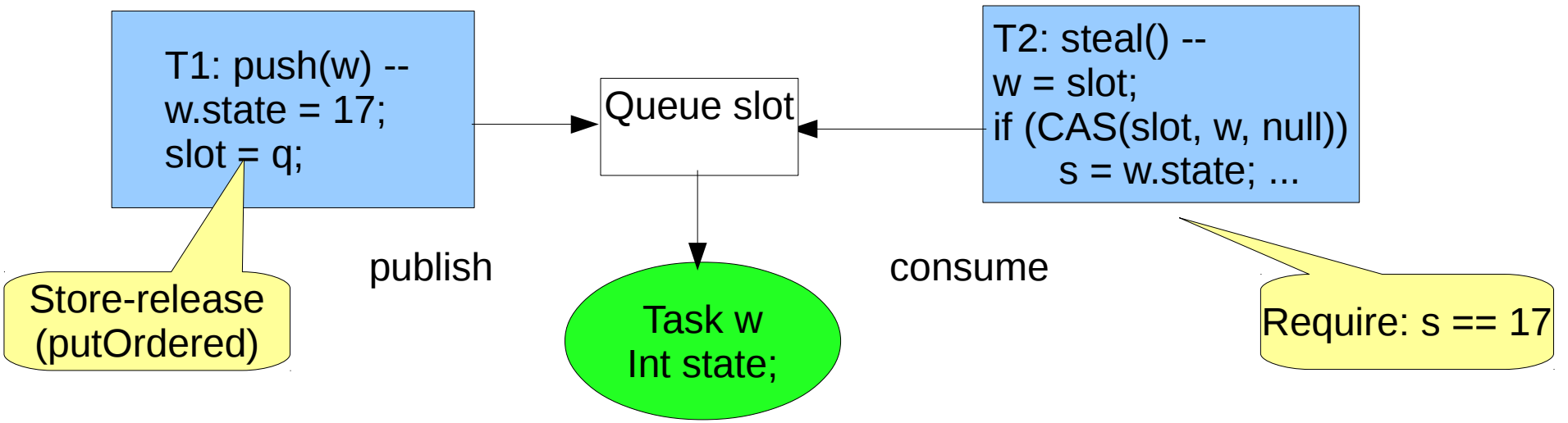
Publication and Transfers

```
Class X { int field; X(int f) { field = f; } }
```

- For shared var v (other vars thread-local):
  - P: `p.field = e; v = p;` || C: `c = v; f = c.field;`
- Weaker protocols avoid more invalidation
- Use weakest that ensures that C:f is usable, considering:
  - "Usable" can be algorithm- and API-dependent
  - Is write to v final? including:
    - Write Once (null → x), Consume Once (x → null)
  - Is write to x.field final?
    - Is there a unique uninitialized value for field
  - Are reads validated?
  - Consistency with reads/writes of other shared vars

Example: Transferring Tasks

- **Work-stealing Queues perform ownership transfer**
  - **Push: make task available for stealing or popping**
    - **Needs release fence (weaker, thus faster than full volatile)**
  - **Pop, steal: make task unavailable to others, then run**
    - **Needs CAS with at least acquire-mode**

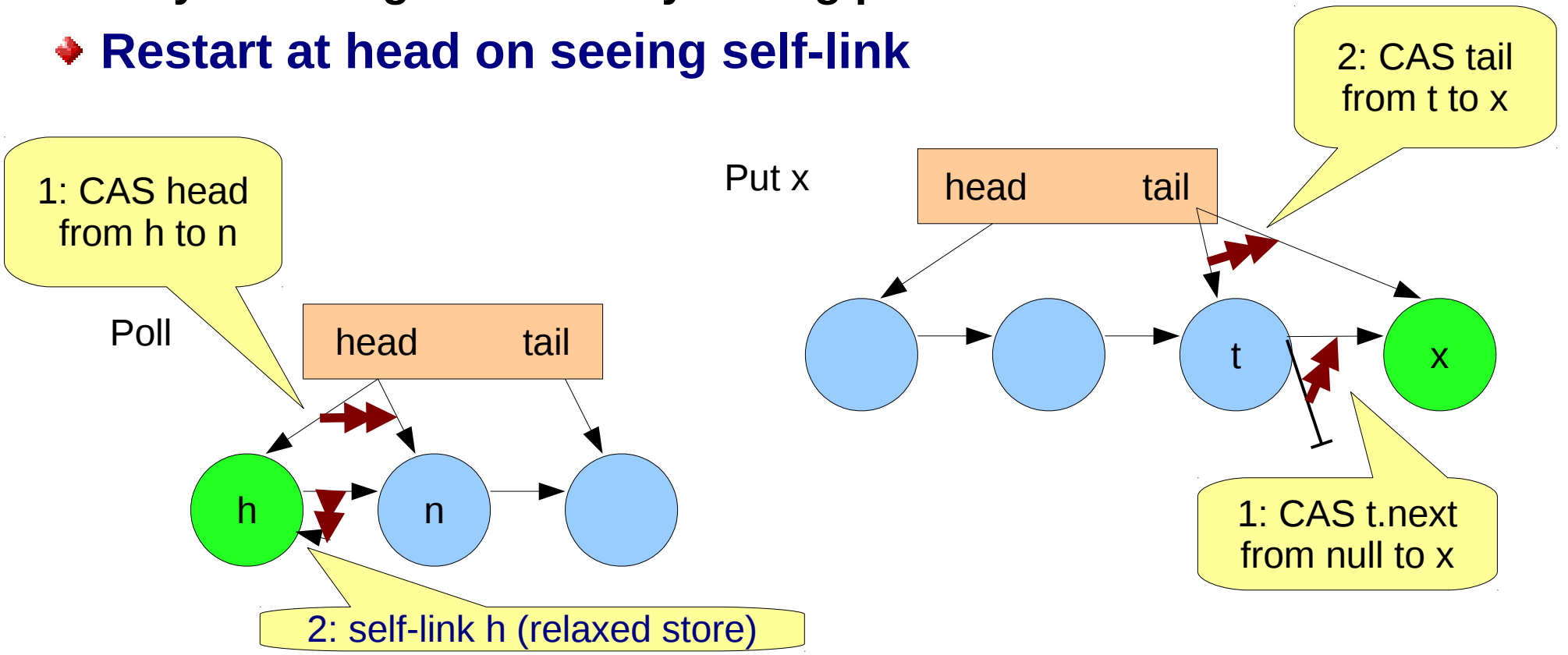


Task Deque Algorithms

- Deque ops (esp push, pop) must be very fast/simple
- One atomic op per push+{pop/steal}
  - This is minimal unless allow duplicate execs or arbitrary postponement (See Maged Michael et al PPoPP 09)
  - Competitive with procedure call stack push/pop
    - Less than 5X cost for empty fork+join vs empty method
- Uses (circular) `array` with `base` and `top` indices
  - Push(t): `storeFence; array[top++] = t;`
  - Pop(t): `if (CAS(array[top-1], t, null)) --top;`
  - Steal(t): `if (CAS(array[base], t, null)) ++base;`
- NOT strictly non-blocking but probabilistically so
  - A stalled ++base precludes other steals
  - But if so, stealers try elsewhere (randomized selection)

Example: ConcurrentLinkedQueue

- **Extend Michael & Scott Queue (PODC 1996)**
  - **CASes on different vars (head, tail) for put vs poll**
  - **If CAS of tail from t to x on put fails, others try to help**
    - **By checking consistency during put or take**
  - **Restart at head on seeing self-link**



Synchronizers

- Shared-memory sync support
  - Queues, Futures, Locks, Barriers, etc
  - Shared is faster than unshared messaging
    - But can be less scalable for point-to-point
  - Provides stronger guarantees: Cache coherence
  - Can be more error-prone: Aliasing, races, visibility
  - Exposing benefits vs complexity is policy issue
- Support Actors, Messages, Events
  - Supply mechanism, not policy

Builtin Synchronization

- Every Java object has 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

Synchronizer Framework

- Any of: Locks, RW locks, semaphores, futures, handoffs, etc., could be to build 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
  - 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.
  - Doesn't try to work for all possible synchronizers, but enough to be both efficient and widely useful
    - Phasers, Exchangers don't use AQS

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
  - Main implementation is AQS-based `ReentrantLock`
- `lock()` and `unlock()` can occur in different 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

AQS Acquire/Release Support

- **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:**

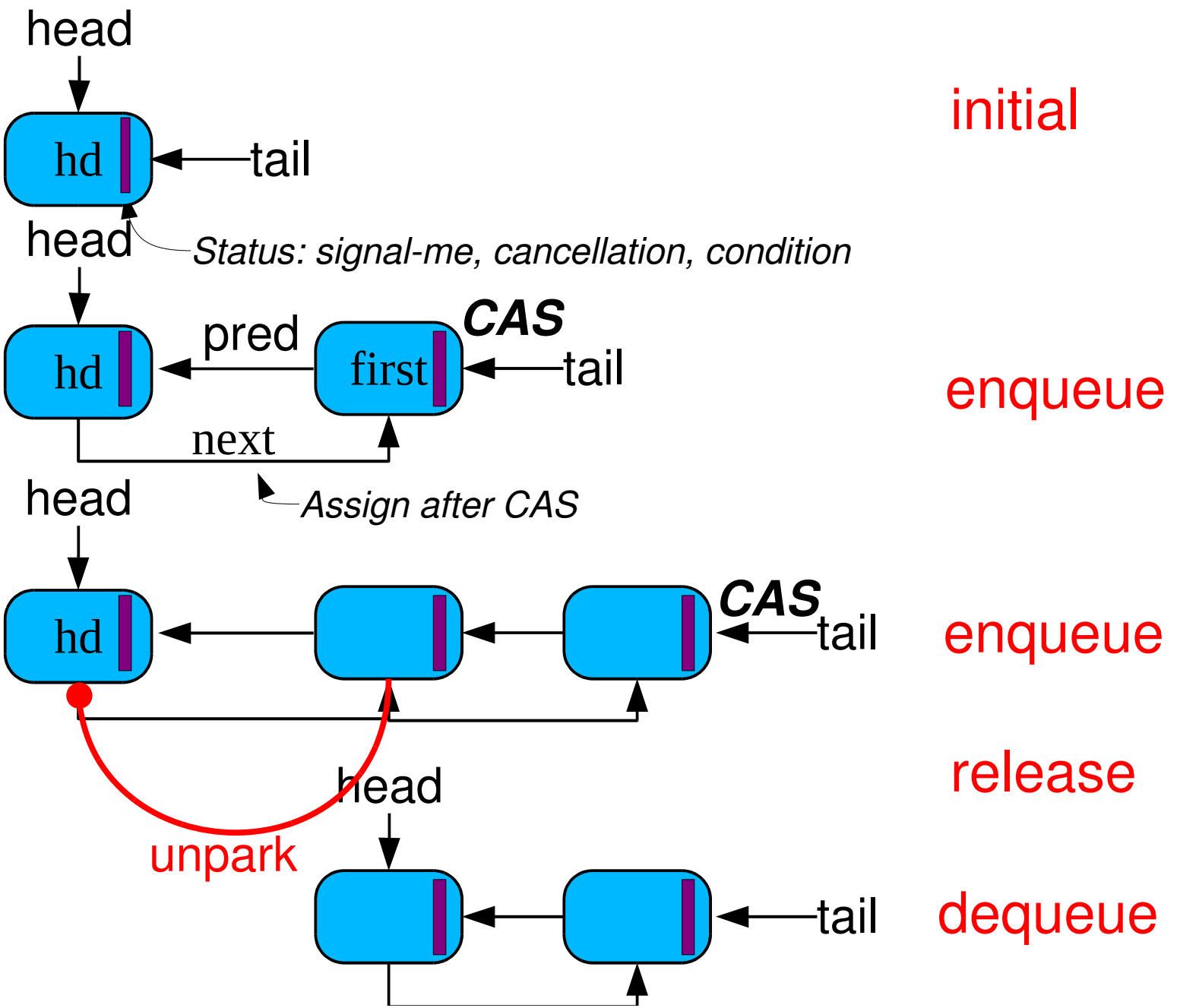
  update synchronization state;
  if (state may permit a blocked thread to acquire)
  * unblock one or more queued threads;*

- **AQS atomically maintains synchronization state**
  - An int representing e.g., whether lock is in locked state
  - **Blocks and unblocks threads**
    - Using LockSupport.park/unpark
  - **Maintains queues**

AQS Queuing

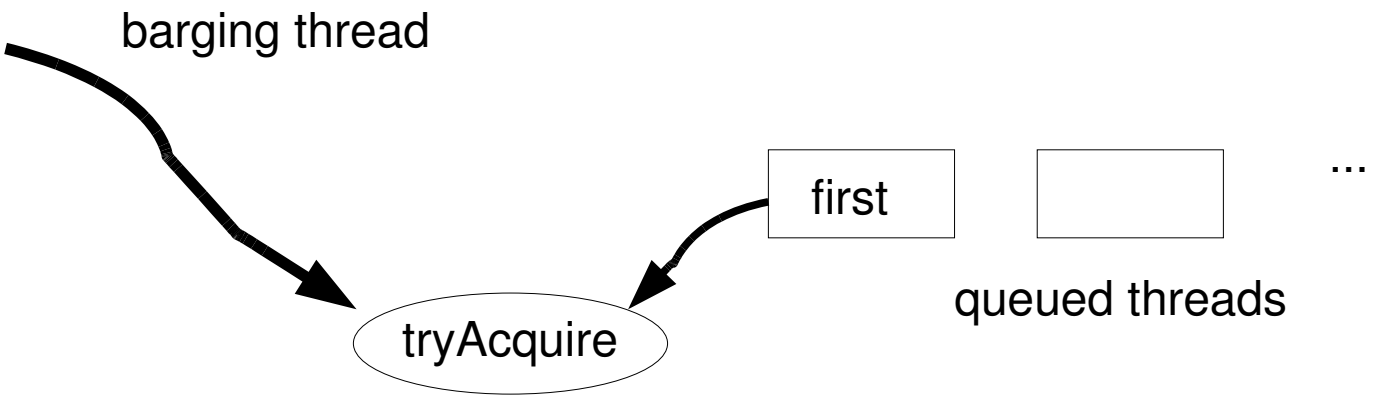
- An extension CLH locks
  - Single-CAS insertion using explicit pred pointers
- Modified as blocking lock, not spin lock
  - Acquirability based on sync state, not node state
    - Signal status information for a node held in its predecessor
  - Add timeout, interrupt, fairness, exclusive vs shared modes
- Also next-pointers to enable signalling (unpark)
  - Wake up successor (if needed) upon release
  - Not atomically assigned; Use pred ptrs as backup
- Lock Conditions use same rep, 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



Performance

- Uncontended overhead (ns/lock)

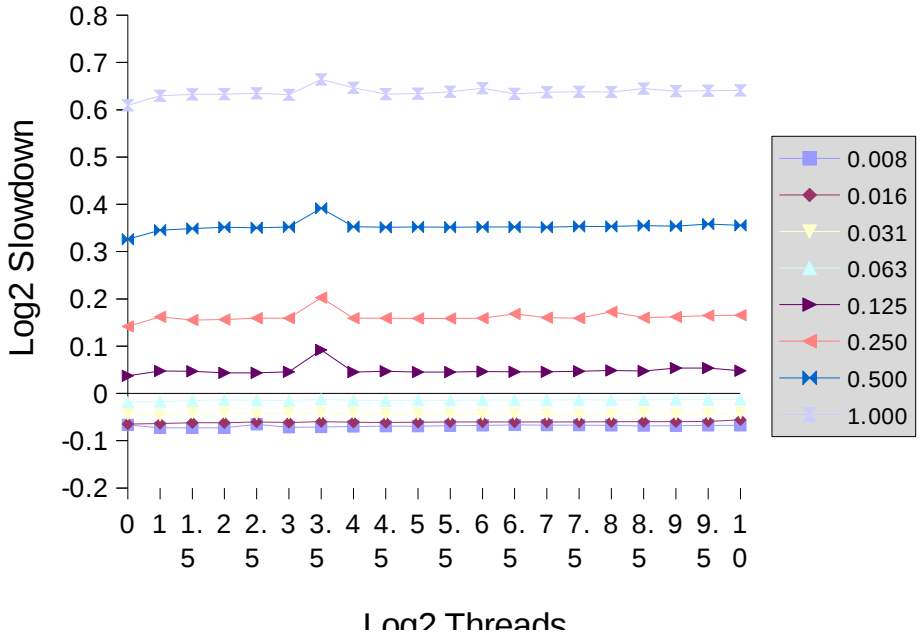
Machine	*Builtin*	*Mutex*	*Reentrant*	*Fair*
1P	18	9	31	37
2P	58	71	77	81
2A	13	21	31	30
4P	116	95	109	117
1U	90	40	58	67
4U	122	82	100	115
8U	160	83	103	123
24U	161	84	108	119

- On saturation FIFO-with-Barging keeps locks busy

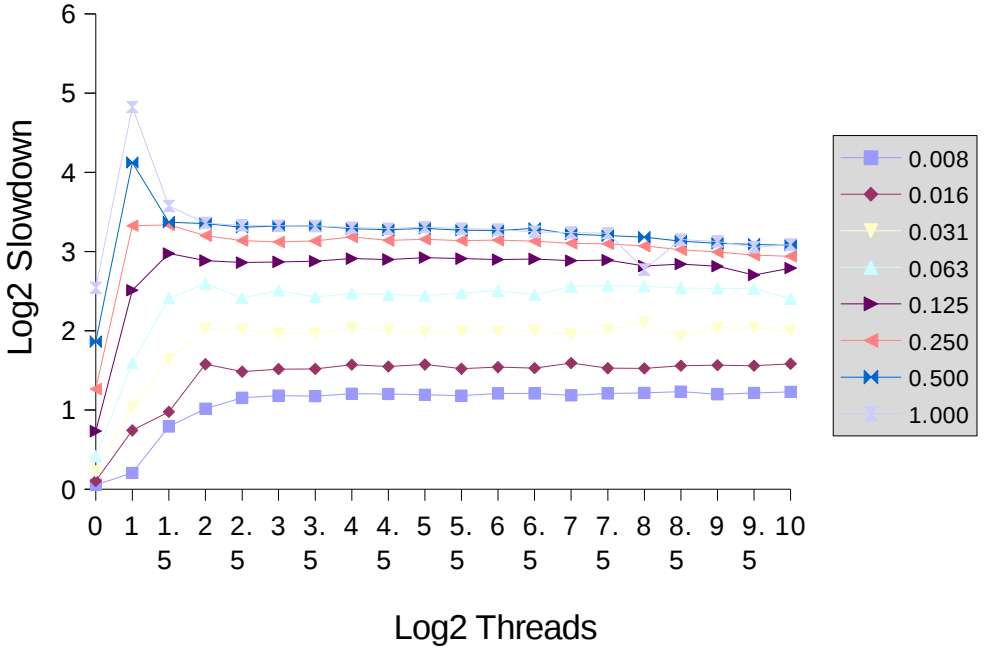
Machine	*Builtin*	*Mutex*	*Reentrant*	*Fair*
1P	521	46	67	8327
2P	930	108	132	14967
2A	748	79	84	33910
4P	1146	188	247	15328
1U	879	153	177	41394
4U	2590	347	368	30004
8U	1274	157	174	31084
24U	1983	160	182	32291

Throughput under Contention

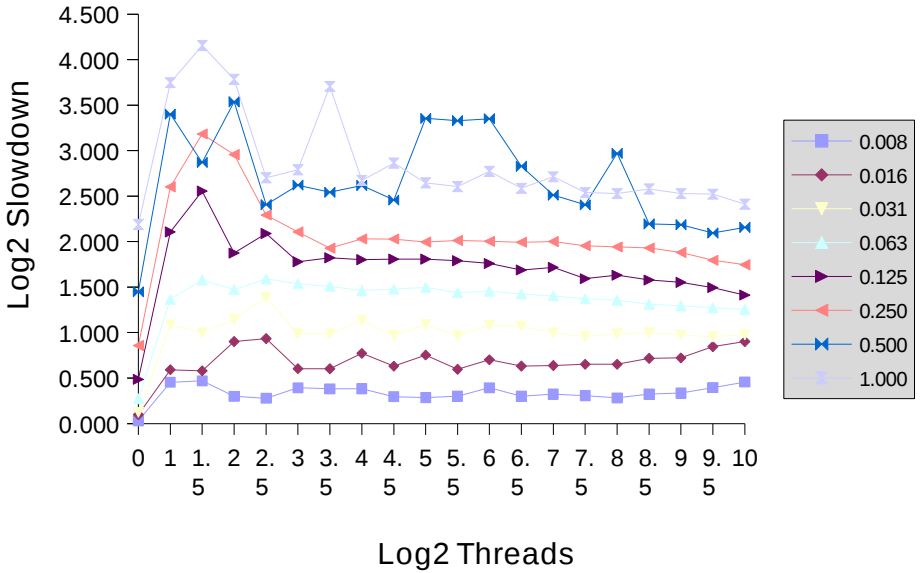
Sparc Uniprocessor



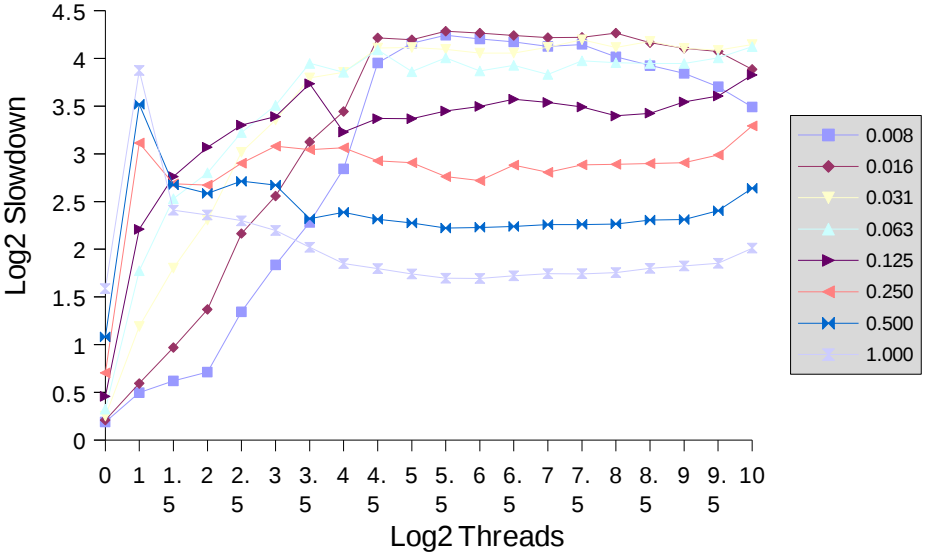
Dual hyperthread Xeon / linux



Dual P3/linux



24-way Ultrasparc 3



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
- Library code never hides 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**
  - Exit without producing or signalling errors
  - Callers can poll cancellation status if necessary
    - May require rollback or recovery
- **Continuation (ignoring cancellation status)**
  - When partial actions cannot be backed out
  - When it doesn't matter
- **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

Queues

- Can act as synchronizers, collections, or both
- As channels, may support:
  - Always available to insert without blocking: add(x)
  - Fallible add: boolean offer(x)
  - Non-blocking attempt to remove: poll()
  - Block on empty: take()
  - Block on full: put()
  - Block until received: transfer();
  - Versions with timeouts

Queue APIs

```
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);
}
interface TransferQueue<E> extends BlockingQueue<E> {
  void transfer(E x) throws InterruptedException; // ...
}
```

- `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(msqQ.take());
      }
    }
    catch(InterruptedException ie) { ... }
  }
}
```

No-API Queues

- Nearly any array or linked list can be used as queue
  - Often the case when array or links needed anyway
  - Common inside other j.u.c. code (like ForkJoin)
  - Avoids a layer of wrapping
  - Avoids overhead of supporting unneeded methods
- Example: Treiber Stacks
  - Simplest CAS-based Linked “queue”
    - LIFO ordering
- Work-Stealing deques are array-based example

Treiber Stack

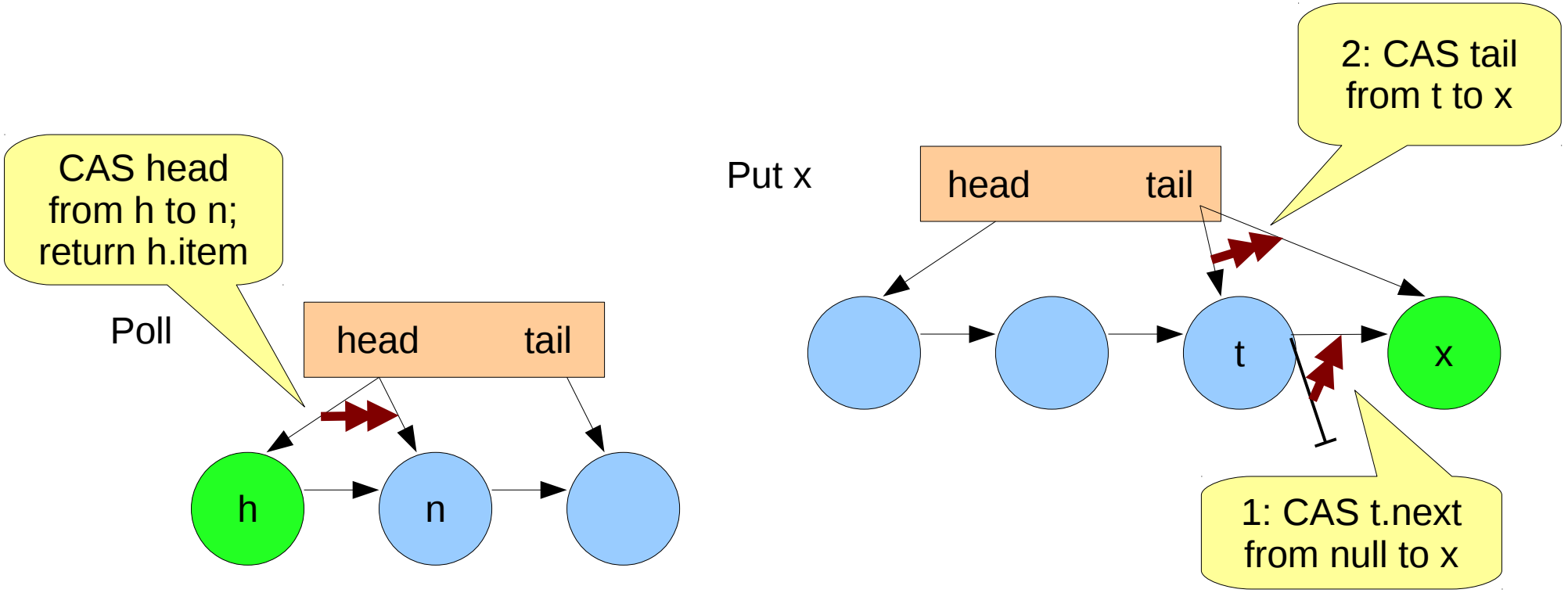
```
interface LIFO<E> { void push(E x); E pop(); }

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

  final 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;
  }

}
```

ConcurrentLinkedQueue

- **Michael & Scott Queue (PODC 1996)**
  - **Use retriable CAS (not lock)**
  - **CASes on different vars (head, tail) for put vs poll**
  - **If CAS of tail from t to x on put fails, others try to help**
    - **By checking consistency during put or take**



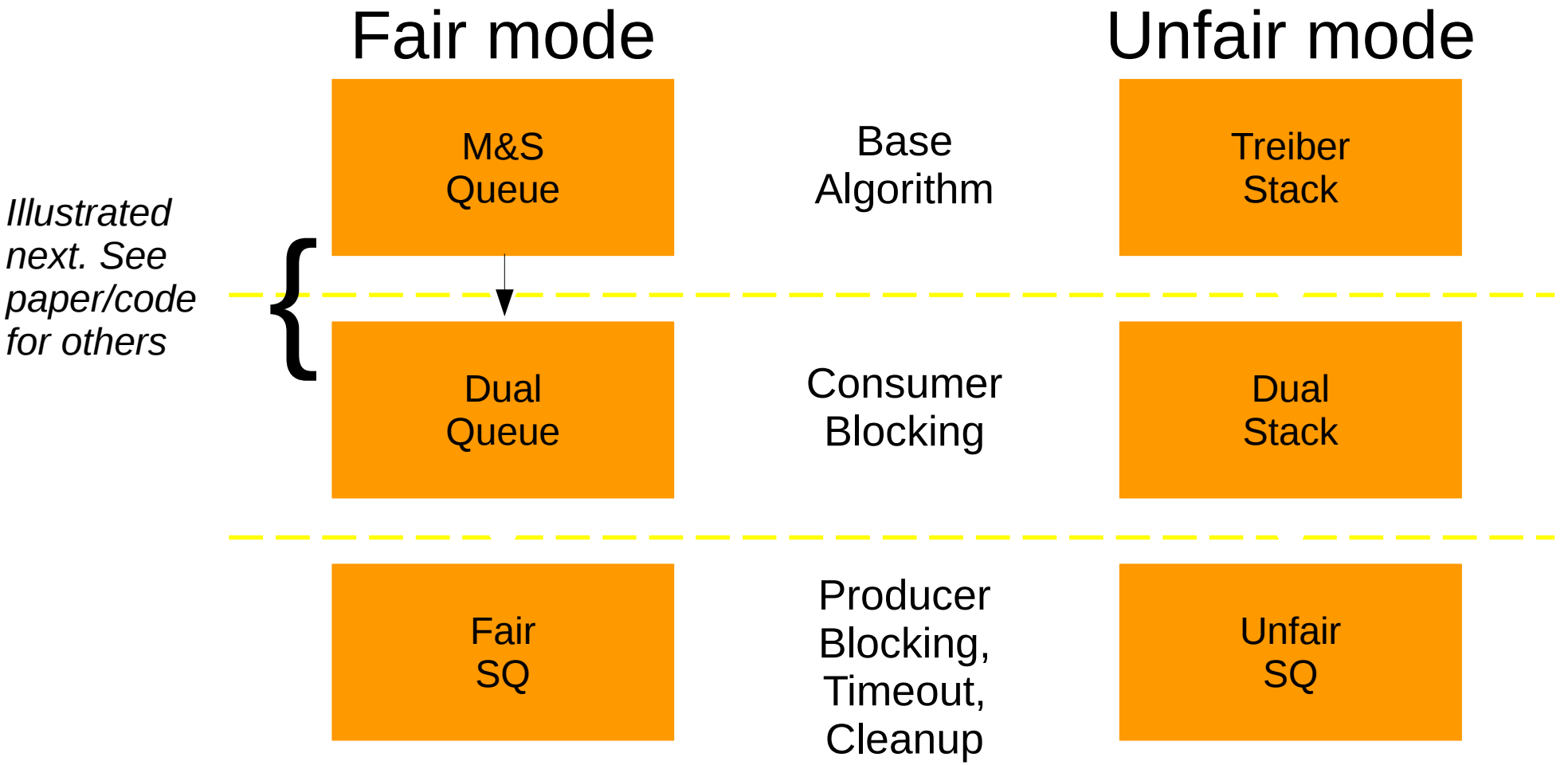
Classic Monitor-Based Queues

```
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
```

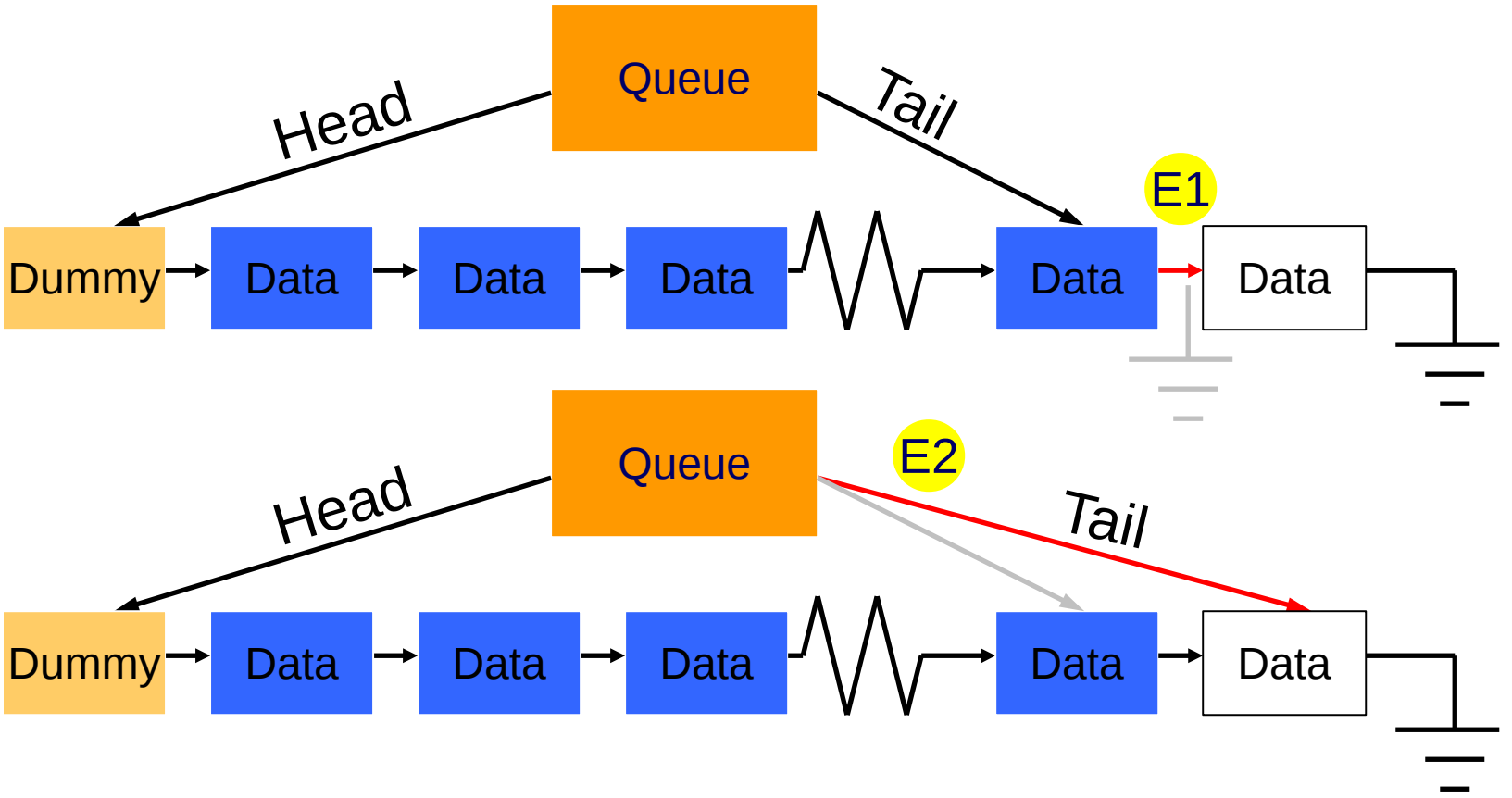
SynchronousQueues

- 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

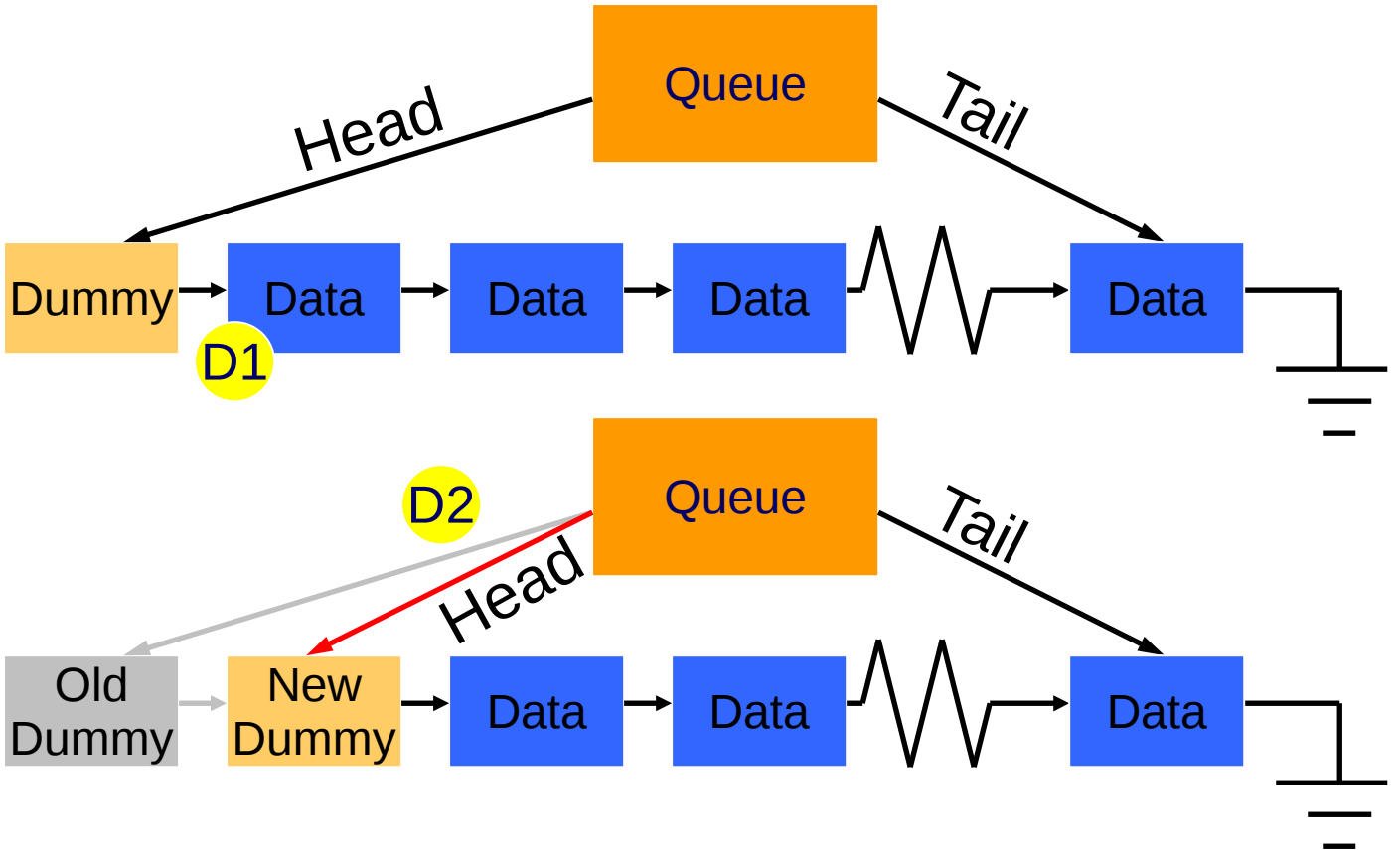
Dual SynchronousQueue Derivation



M&S Queue: Enqueue



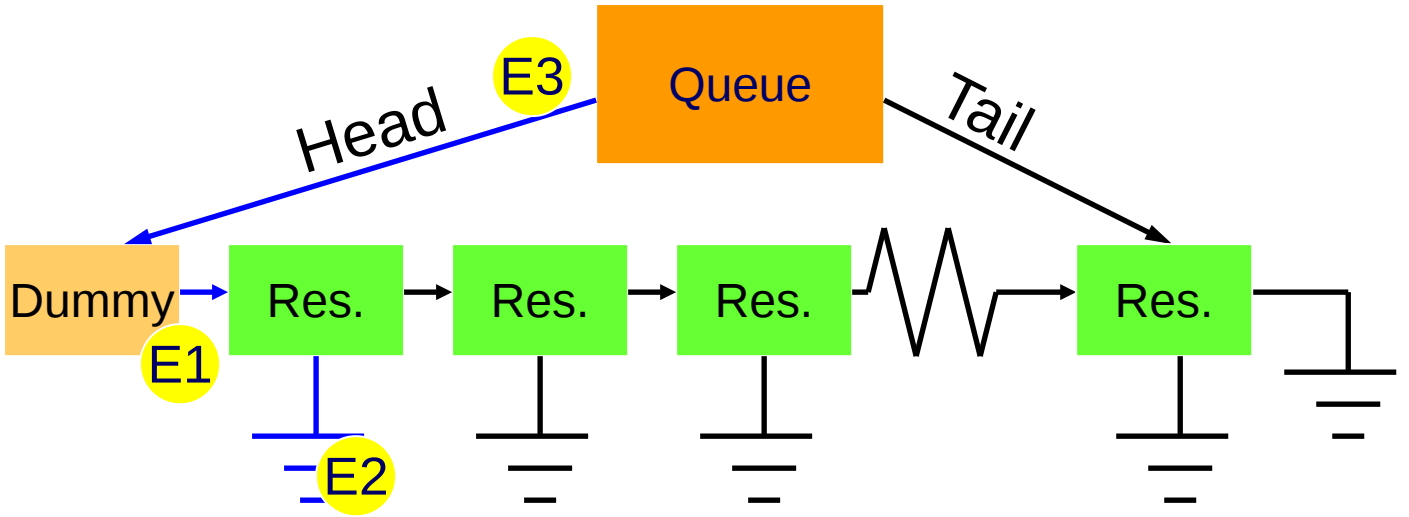
M&S Queue: Dequeue



Dual M&S Queues

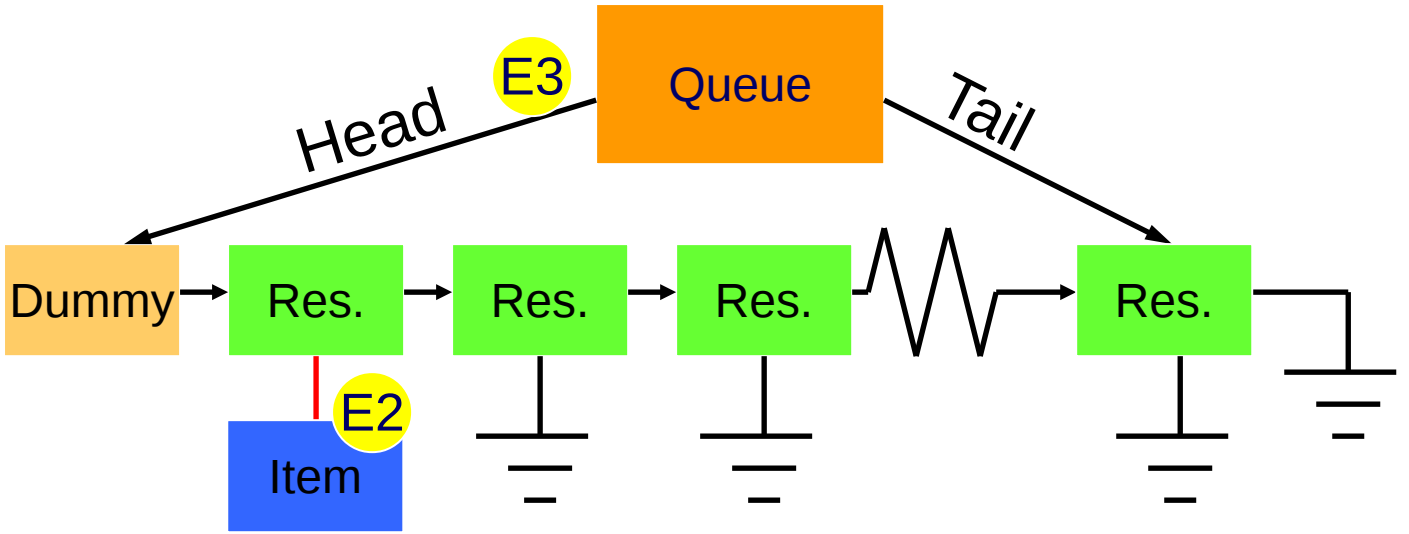
- Separate data, request nodes (flag bit)
  - Queue always all-data or all-requests
- Same behavior as M&S queue for data
- Reservations are antisymmetric to data
  - dequeue enqueues a reservation node
  - enqueue satisfies oldest reservation
- Tricky consistency checks needed
  - Dummy node can be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

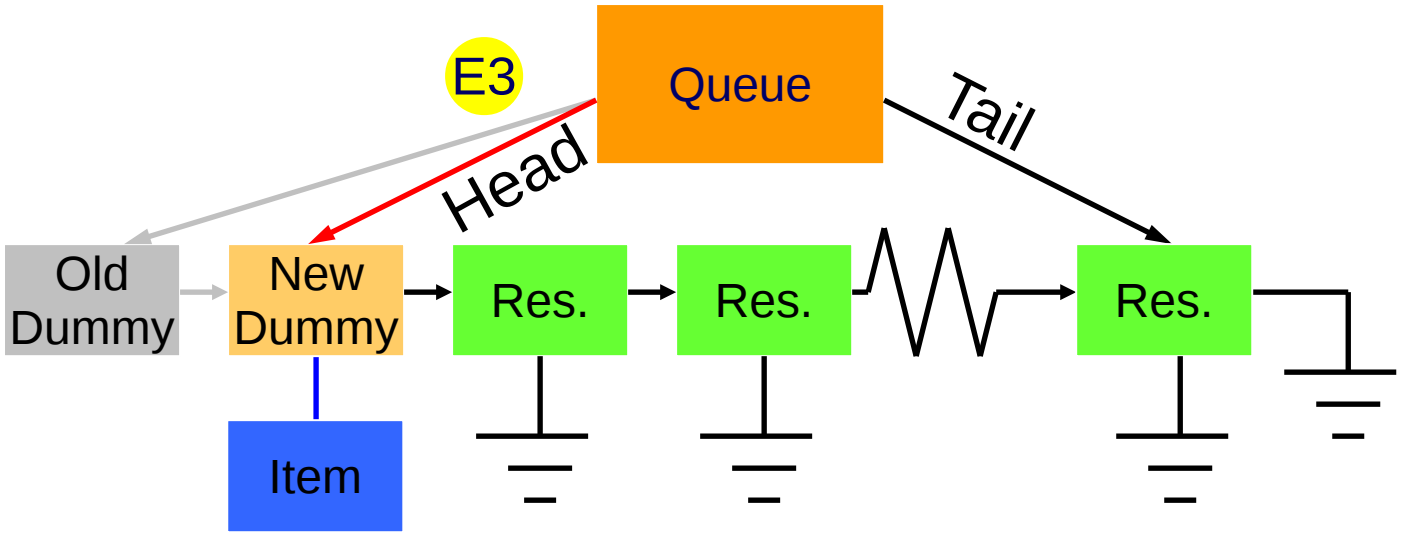


E1 Read dummy's next ptr

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

Queues, Events and Consistency

- Consistency issues are intrinsic to event systems
  - Example: vars x,y initially 0 → events x, y unseen
    - Node A: send x = 1; // (multicast send)
    - Node B: send y = 1;
    - Node C: receive x; receive y; // see x=1, y=0
    - Node D: receive y; receive x; // see y=1, x=0
- On shared memory, can guarantee agreement
  - JMM: declare x, y as volatile
- Remote consistency is expensive
  - Atomic multicast, distributed transactions; failure models
- Usually, weaker consistency is good enough
  - Example: Per-producer FIFO

Collections

- Multiple roles
  - Representing ADTs
  - Shared communication media
- An increasing common focus
  - Transactionality
  - Isolation
  - Bulk parallel operations

Semi-Transactional ADTs

- Explicitly concurrent objects used as resources
  - Support conventional APIs (Collections, Maps)
    - Examples: Registries, directories, message queues
  - Programmed in low-level JVMese – compareAndSet (CAS)
    - Often vastly more efficient than alternatives
- Roots in ADTs and Transactions
  - ADT: Opaque, self-contained, limited extensibility
  - Transactional: All-or-nothing methods
- Atomicity limitations; no transactional removeAll etc
  - But usually can support *non*-transactional bulk *parallel* ops
    - (Need for transactional parallel bulk ops is unclear)
  - Possibly only transiently concurrent
    - Example: Shared outputs for bulk parallel operations

Concurrent Collections

- Non-blocking data structures rely on simplest form of hardware transactions
  - CAS (or LL/SC) tries to commit a single variable
  - Frameworks layered on CAS-based data structures can be used to support larger-grained transactions
  - HTM (or multiple-variable CAS) would be nicer
    - But not a magic bullet
- Evade most hard issues in general transactions
  - Contention, overhead, space bloat, side-effect rollback, etc
  - But special cases of these issues still present
    - Complicates implementation: Hard to see Michael & Scott algorithm hiding in LinkedTransferQueue

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

Contention in Shared Data Structures

Mostly-Write

- Most producer-consumer exchanges
- Apply combinations of a small set of ideas
  - Use non-blocking sync via compareAndSet (CAS)
  - Reduce point-wise contention
  - Arrange that threads help each other make progress

Mostly-Read

- Most Maps & Sets
- Structure to maximize concurrent readability
  - Without locking, readers see legal (ideally, linearizable) values
  - Often, using immutable copy-on-write internals
  - Apply write-contention techniques from there

Collections Design Options

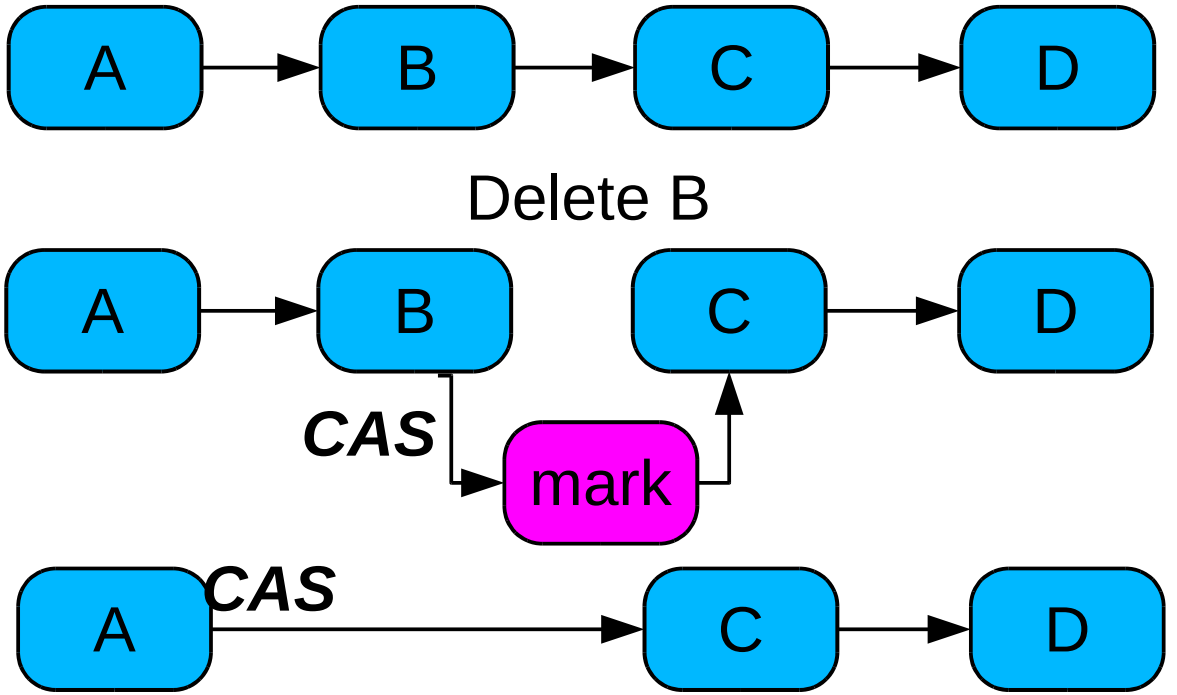
- **Large design space, including**
  - **Locks: Coarse-grained, fine-grained, ReadWrite locks**
  - **Concurrently readable – reads never block, updates use locks**
  - **Optimistic – never block but may spin**
  - **Lock-free – concurrently readable and updatable**

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**

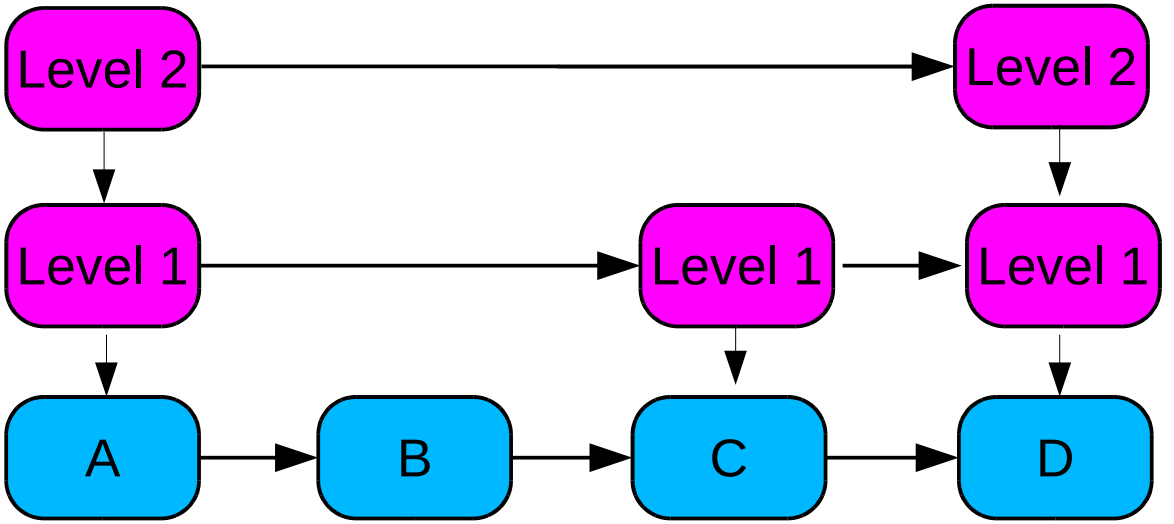
Linear Sorted Lists

- **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**



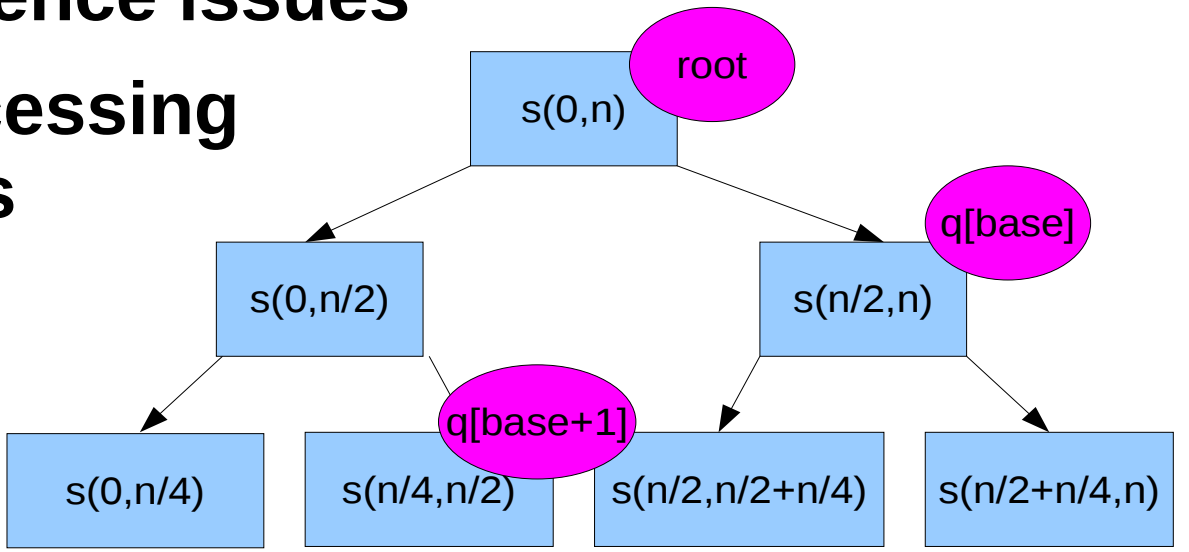
ConcurrentSkipListMap

- 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



Bulk Operations

- SIMD: Apply operation to all elements of a collection
  - Procedures: Color all my squares red
  - Mappings: Map these student IDs to their grades
  - Reductions: Calculate the sum of these numbers
- A special case of basic parallel evaluation
  - Any number of components; same operation on each
  - Same independence issues
- Can arrange processing in task trees/dags
  - Array Sum:



QoS and Memory Management

- GC can be ill-suited for stream-like processing:
  - Repeat: Allocate → read → process → forget
- RTSJ Scoped memory
  - Overhead, run-time exceptions (vs static assurance)
- Off-heap memory
  - Direct-allocated ByteBuffers hold data
    - Emulation of data structures inside byte buffers
  - Manual storage management (pooling etc)
  - Manual synchronization control
  - Manual marshalling/unmarshalling/layout
    - Project Panama will enable declarative layout control
- Alternatives?

Memory Placement

- Memory contention, false-sharing, NUMA, etc can have huge impact
  - Reduce parallel progress to memory system rates
    - JDK8 @sun.misc.Contended allows pointwise manual tweaks
  - Some GC mechanics worsen impact; esp card marks
    - When writing a reference, JVM also writes a bit/byte in a table indicating that one or more objects in its address range (often 512bytes wide) may need GC scanning
    - The card table can become highly contended
      - Yang et al (ISMM 2012) report 378X slowdown
- JVMs cannot allow precise object placement control
  - But can support custom layouts of plain bits (struct-like)
    - JEP for Value-types (Valhalla) + Panama address most cases?
  - JVMs oblivious to higher-level locality constraints
    - Including “ThreadLocal”!

Randomization

- Common components inject algorithmic randomness
  - Hashing, skip lists, crypto, numerics, etc
    - Fun fact: The Mark I (1949) had hw random number generator
  - Visible effects; e.g., on collection traversal order
    - API specs do not promise deterministic traversal order
      - Bugs when users don't accommodate
- Can be even more useful in concurrency
  - Fight async and system non-determinism with algorithmic non-determinism
    - Hashed striping, backoffs, work-stealing, etc
  - Implicit hope that central limit theorem applies
    - Combining many allegedly random effects → lower variance
    - Often appears to work, but almost never provably
      - Formal intractability is an impediment for some real-time use

Postscript: Developing Libraries

- Design is a social process
  - Single visions are good, those that pass review better
- Specification and documentation require broad review
  - Even so, by far most submitted j.u.c bugs are spec bugs
- Release engineering requires perfectionism
  - Lots of QA: tests, reviews. Still not enough
- Widespread adoption requires standardization
  - JCP both a technical and political body
- Need tutorials, examples, etc written at many different levels
  - Users won't read academic papers to figure out how/why to use
- Creating new components leads to new developer problems
  - Example: New bug patterns for findBugs