İTÜ

Spin Locks and Contention

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

İSTANBUL TEKNİK ÜNİVERSİTESİ

Focus so far: Correctness and Progress

İTÜ

- Models
  - Accurate (we never lied to you)
  - But idealized (so we forgot to mention a few things)
- Protocols
  - Elegant
  - Important
  - But naïve

İSTANBUL TEKNİK ÜNİVERSİTESİ

New Focus: Performance

İTÜ

- Models
  - More complicated (not the same as complex!)
  - Still focus on principles (not soon obsolete)
- Protocols
  - Elegant (in their fashion)
  - Important (why else would we pay attention)
  - And realistic (your mileage may vary)

İSTANBUL TEKNİK ÜNİVERSİTESİ

Kinds of Architectures

İTÜ

- SISD (Uniprocessor)
  - Single instruction stream
  - Single data stream
- SIMD (Vector)
  - Single instruction
  - Multiple data
- MIMD (Multiprocessors)
  - Multiple instruction
  - Multiple data.

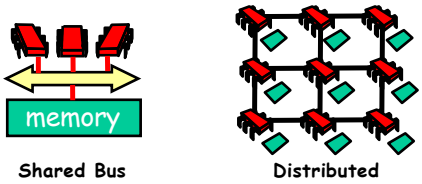
Our space

(1)

İSTANBUL TEKNİK ÜNİVERSİTESİ

MIMD Architectures

İTÜ



- Memory Contention
- Communication Contention
- Communication Latency

İSTANBUL TEKNİK ÜNİVERSİTESİ

Today: Revisit Mutual Exclusion

İTÜ

- Think of performance, not just correctness and progress
- Begin to understand how performance depends on our software properly utilizing the multiprocessor machine's hardware
- And get to know a collection of locking algorithms…

İSTANBUL TEKNİK ÜNİVERSİTESİ

What Should you do if you can't get a lock?

- Keep trying
  - "spin" or "busy-wait"
  - Good if delays are short
- Give up the processor
  - Good if delays are long
  - Always good on uniprocessor

our focus

Review: Test-and-Set

- Boolean value
- Test-and-set (TAS)
  - Swap true with current value
  - Return value tells if prior value was true or false
- Can reset just by writing false
- TAS aka "getAndSet"

Review: Test-and-Set

```
public class AtomicBoolean {

 boolean value;

 public synchronized boolean getAndSet(boolean newValue) {
   boolean prior = value;
   value = newValue;
   return prior;
 }

}
```

Review: Test-and-Set

```
AtomicBoolean lock = new AtomicBoolean(false)

…

boolean prior = lock.getAndSet(true)
```

Test-and-Set Locks

- Locking
  - Lock is free: value is false
  - Lock is taken: value is true
- Acquire lock by calling TAS
  - If result is false, you win
  - If result is true, you lose
- Release lock by writing false

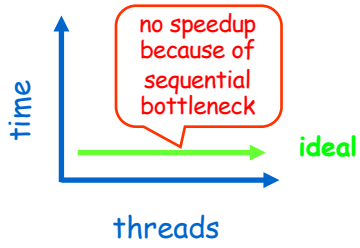
Test-and-set Lock

```
class TASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (state.getAndSet(true)) {}
 }

 void unlock() {
  state.set(false);
 }}
```

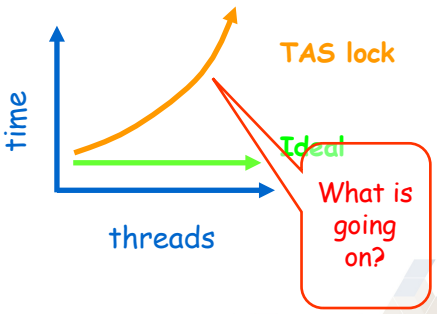
Performance

- Experiment
  - n threads
  - Increment shared counter 1 million times
- How long should it take?
- How long does it take?

Graph



Mystery #1



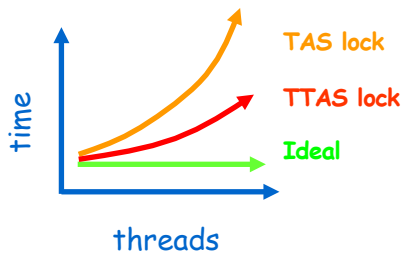
Test-and-Test-and-Set Locks

- Lurking stage
  - Wait until lock "looks" free
  - Spin while read returns true (lock taken)
- Pouncing state
  - As soon as lock "looks" available
  - Read returns false (lock free)
  - Call TAS to acquire lock
  - If TAS loses, back to lurking

Test-and-test-and-set Lock

```
class TTASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (true) {
   while (state.get()) {}
   if (!state.getAndSet(true))
    return;
  }
}
```

Mystery #2



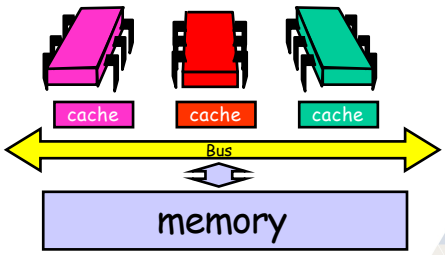
Mystery

- Both
  - TAS and TTAS
  - Do the same thing (in our model)
- Except that
  - TTAS performs much better than TAS
  - Neither approaches ideal

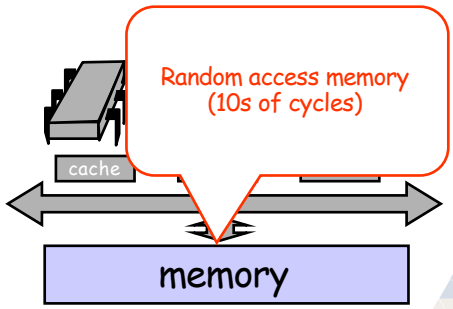
Opinion

- Our memory abstraction is broken
- TAS & TTAS methods
  - Are provably the same (in our model)
  - Except they aren't (in field tests)
- Need a more detailed model …

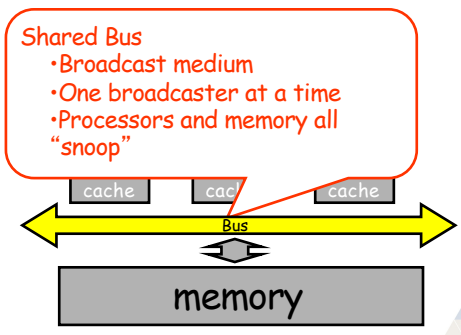
Bus-Based Architectures



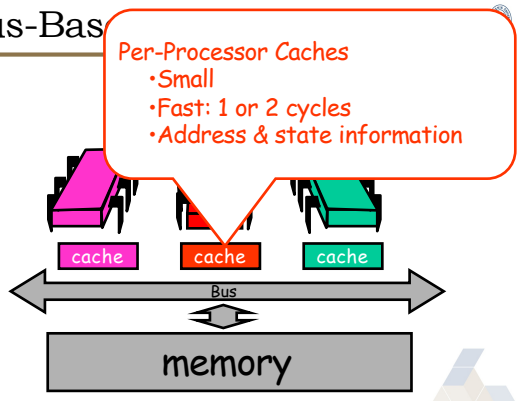
Bus-Based Architectures



Bus-Based Architectures



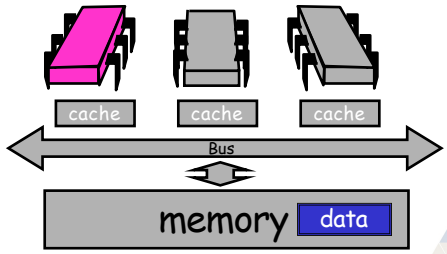
Bus-Bas



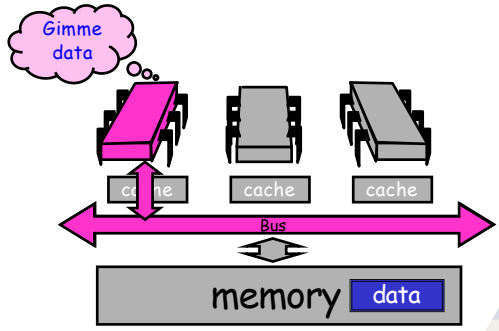
Jargon Watch

- Cache hit
  - "I found what I wanted in my cache"
  - Good Thing™
- Cache miss
  - "I had to shlep all the way to memory for that data"
  - Bad Thing™

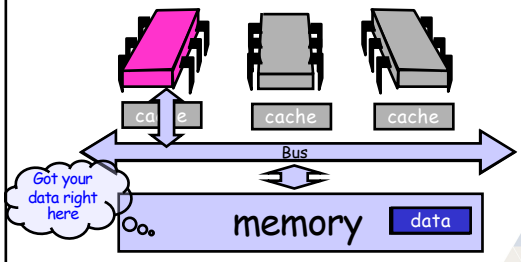
Processor Issues Load Request



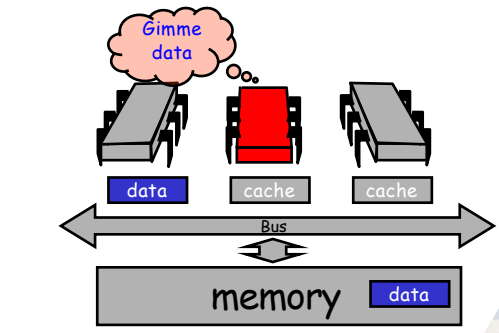
Processor Issues Load Request



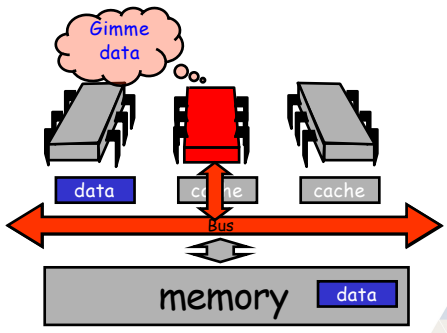
Memory Responds



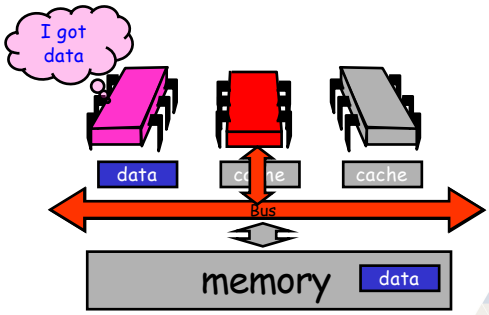
Processor Issues Load Request



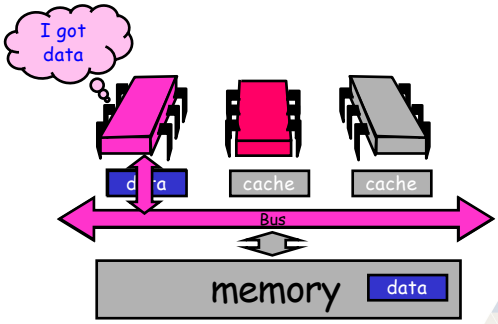
Processor Issues Load Request



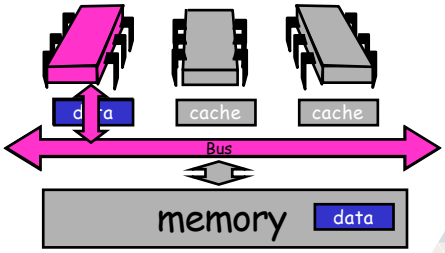
Processor Issues Load Request



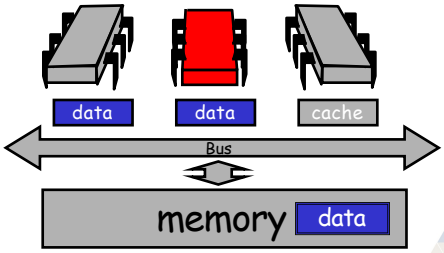
Other Processor Responds



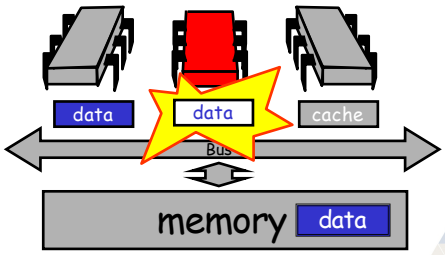
Other Processor Responds



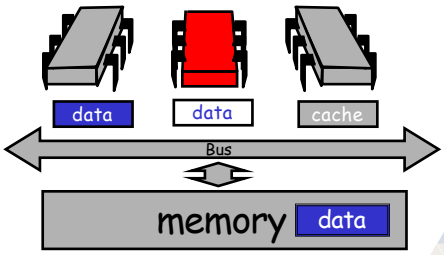
Modify Cached Data



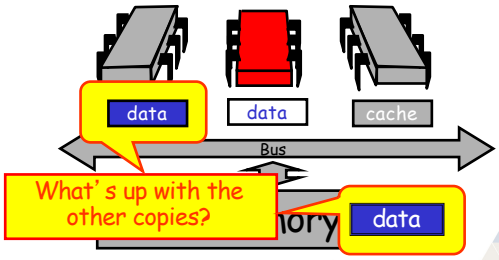
Modify Cached Data



Modify Cached Data



Modify Cached Data



Cache Coherence

- We have lots of copies of data
  - Original copy in memory
  - Cached copies at processors
- Some processor modifies its own copy
  - What do we do with the others?
  - How to avoid confusion?

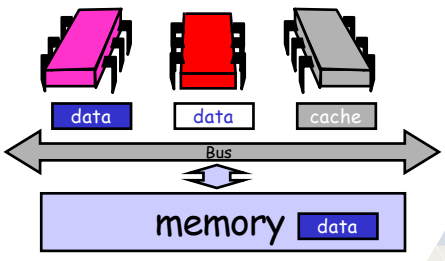
Write-Back Caches

- Accumulate changes in cache
- Write back when needed
  - Need the cache for something else
  - Another processor wants it
- On first modification
  - Invalidate other entries
  - Requires non-trivial protocol …

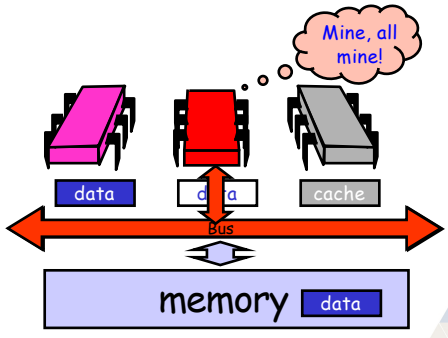
Write-Back Caches

- Cache entry has three states
  - Invalid: contains raw seething bits
  - Valid: I can read but I can't write
  - Dirty: Data has been modified
    - Intercept other load requests
    - Write back to memory before using cache

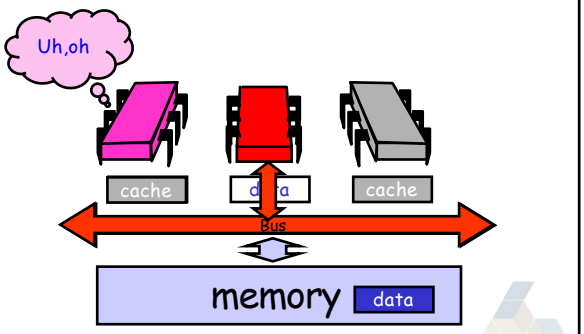
Invalidate



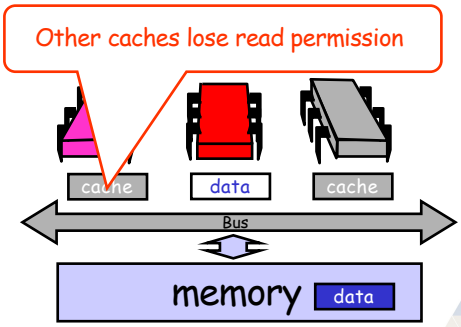
Invalidate



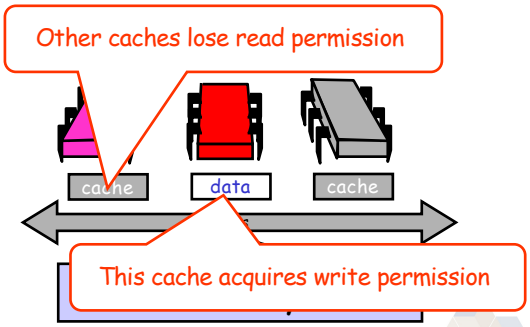
Invalidate



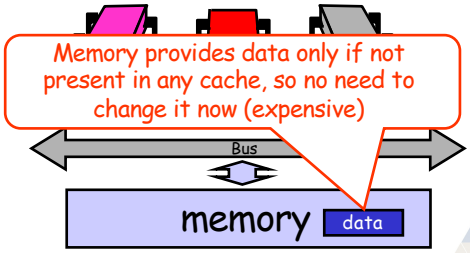
Invalidate



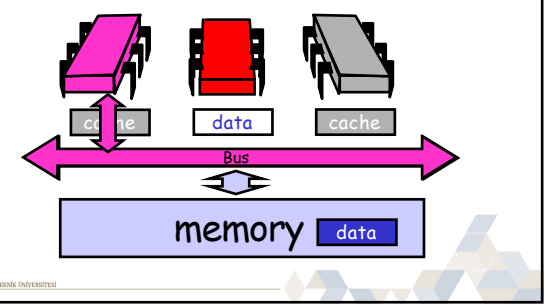
Invalidate



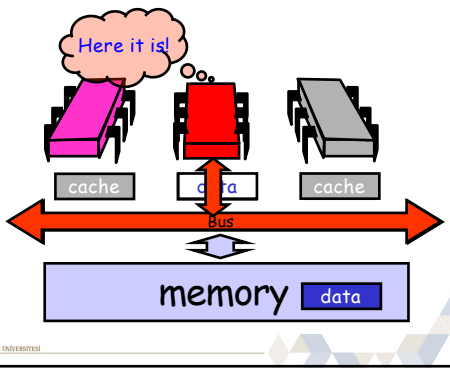
Invalidate



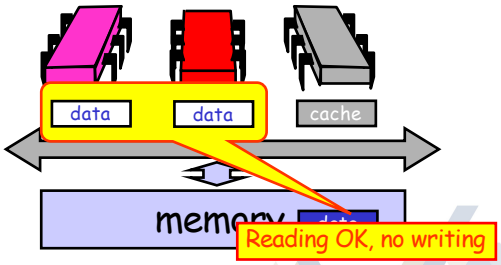
Another Processor Asks for Data



Owner Responds



End of the Day …



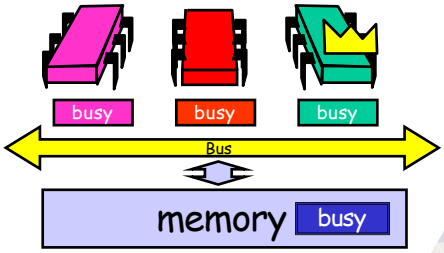
Simple TASLock

- TAS invalidates cache lines
- Spinners
  - Miss in cache
  - Go to bus
- Thread wants to release lock
  - delayed behind spinners

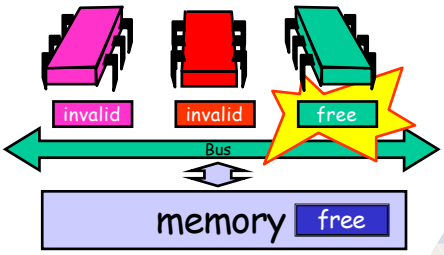
Test-and-test-and-set

- Wait until lock "looks" free
  - Spin on local cache
  - No bus use while lock busy
- Problem: when lock is released
  - Invalidation storm …

Local Spinning while Lock is Busy

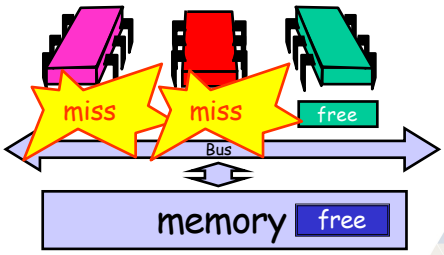


On Release



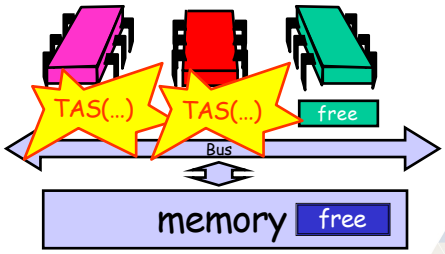
On Release

Everyone misses, rereads



On Release

Everyone tries TAS

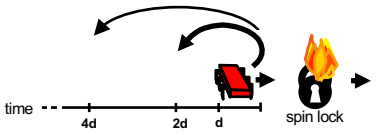


Problems

- Everyone misses
  - Reads satisfied sequentially
- Everyone does TAS
  - Invalidates others' caches
- Eventually quiesces after lock acquired
  - How long does this take?

Dynamic Example: Exponential Backoff

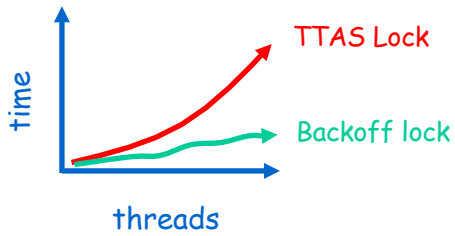
- If I fail to get lock
  - wait random duration before retry
  - Each subsequent failure doubles expected wait



Exponential Backoff Lock

```
public class Backoff implements lock {


 public void lock() {
  int delay = MIN_DELAY;
  while (true) {
   while (state.get()) {}
   if (!lock.getAndSet(true))
    return;
   sleep(random() % delay);
   if (delay < MAX_DELAY)
    delay = 2 * delay;
 }
}}
```

Spin-Waiting Overhead



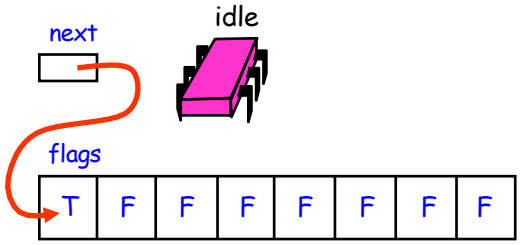
Backoff: Other Issues

- Good
  - Easy to implement
  - Beats TTAS lock
- Bad
  - Must choose parameters carefully
  - Not portable across platforms

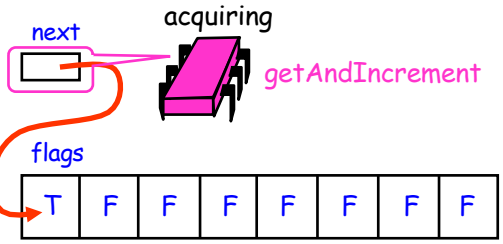
Idea

- Avoid useless invalidations
  - By keeping a queue of threads
- Each thread
  - Notifies next in line
  - Without bothering the others

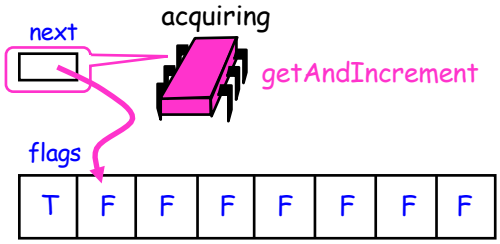
Anderson Queue Lock



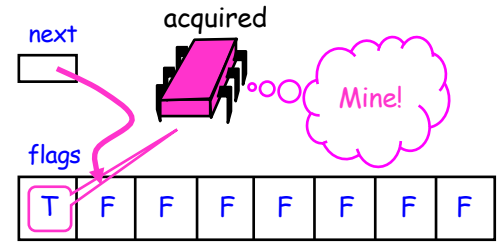
Anderson Queue Lock



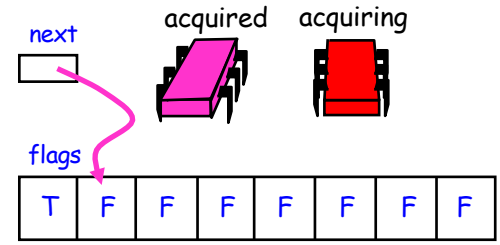
Anderson Queue Lock



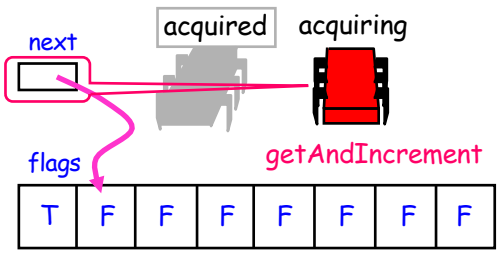
Anderson Queue Lock



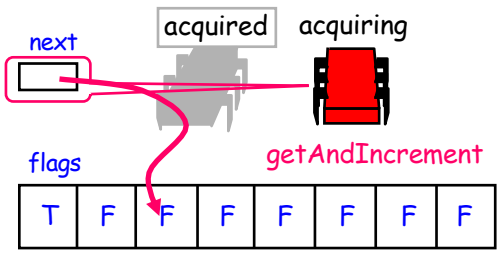
Anderson Queue Lock



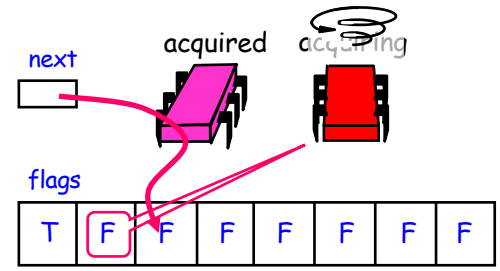
Anderson Queue Lock



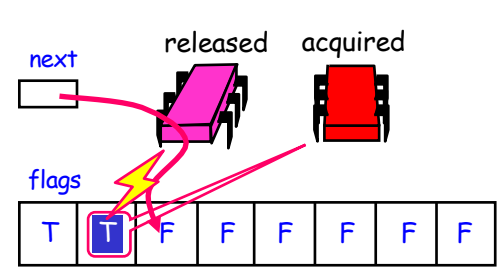
Anderson Queue Lock



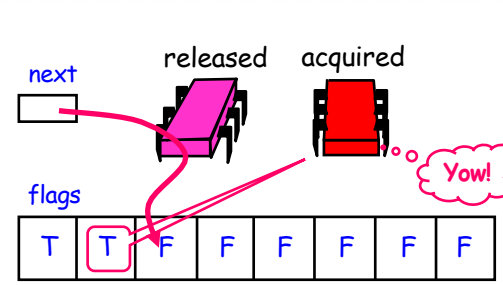
Anderson Queue Lock



Anderson Queue Lock



Anderson Queue Lock



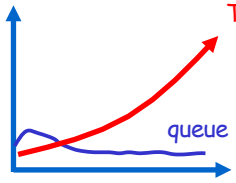
Anderson Queue Lock

```
class ALock implements Lock {
 boolean[] flags={true,false,…,false};
 AtomicInteger next= new AtomicInteger(0);
 int[] slot = new int[n];
```

Anderson Queue Lock

```
public lock() {
 mySlot = next.getAndIncrement();
 while (!flags[mySlot % n]) {};
 flags[mySlot % n] = false;
}

public unlock() {
 flags[(mySlot+1) % n] = true;
}
```

Performance



- Shorter handover than backoff
- Curve is practically flat
- Scalable performance
- FIFO fairness

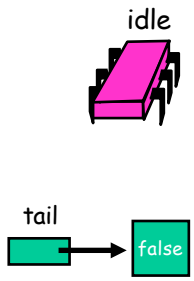
Anderson Queue Lock

- Good
  - First truly scalable lock
  - Simple, easy to implement
- Bad
  - Space hog
  - One bit per thread
    - Unknown number of threads?
    - Small number of actual contenders?

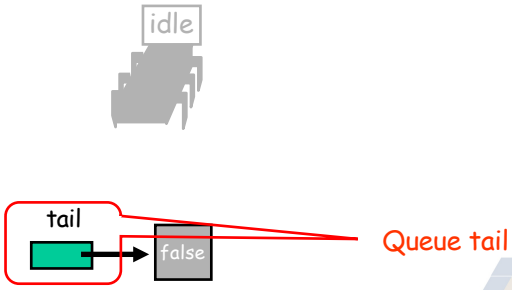
CLH Lock

- FIFO order
- Small, constant-size overhead per thread

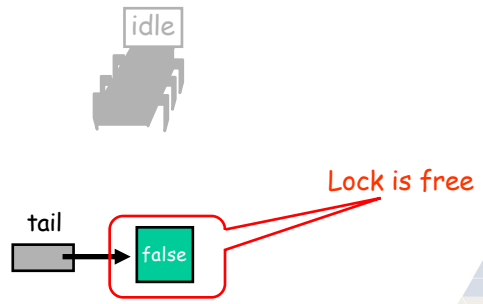
Initially



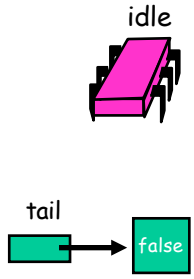
Initially



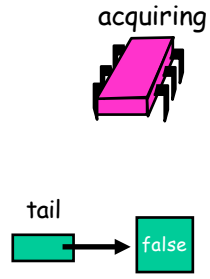
Initially



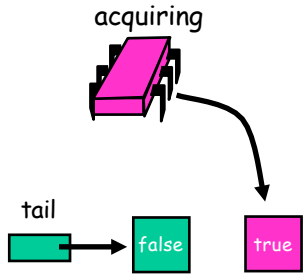
Initially



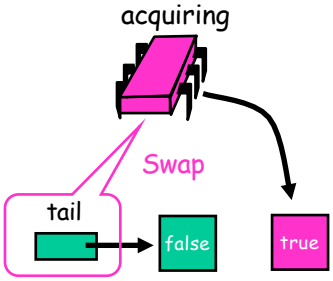
Purple Wants the Lock



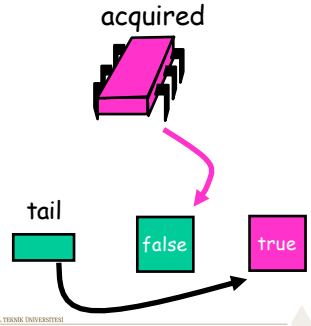
Purple Wants the Lock



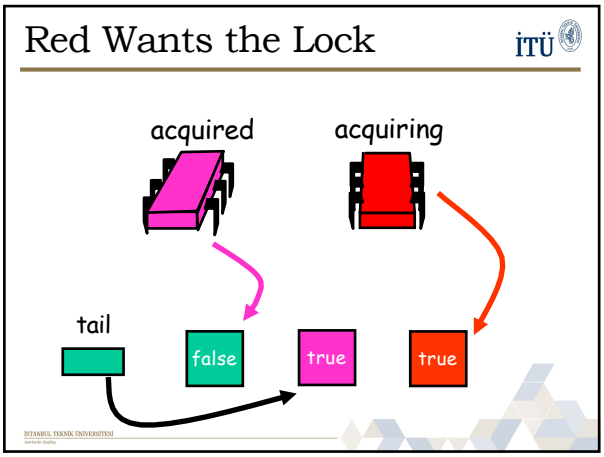
Purple Wants the Lock



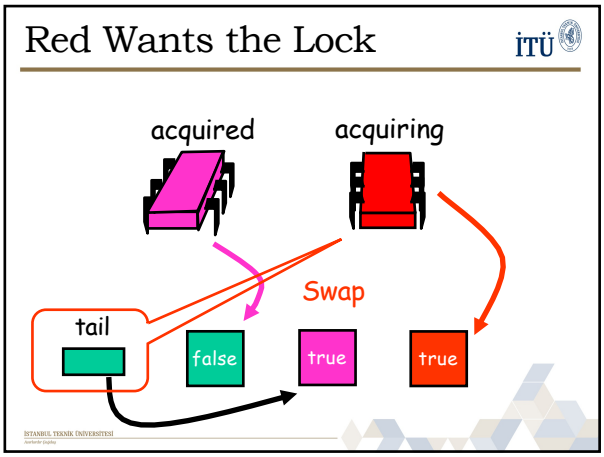
Purple Has the Lock



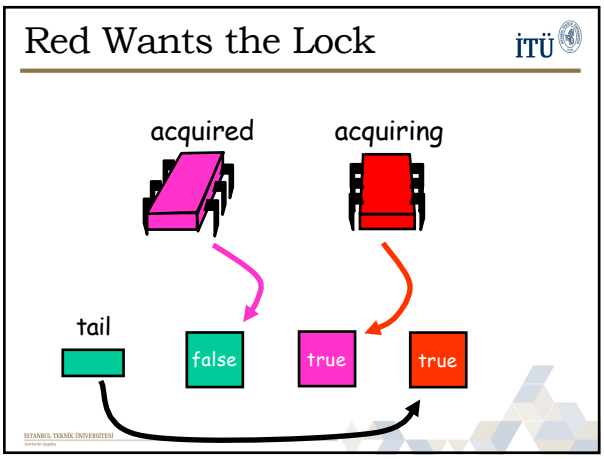
Red Wants the Lock



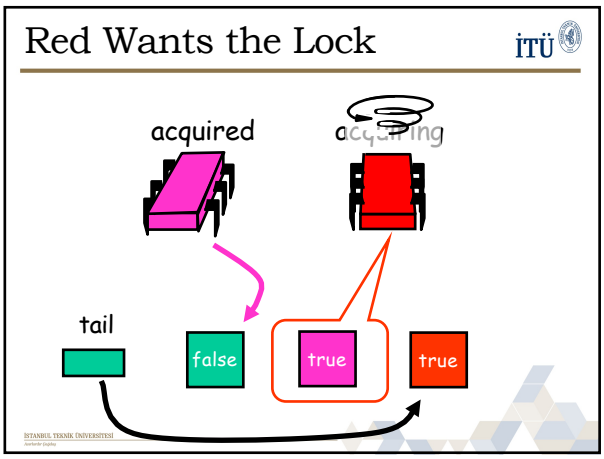
Red Wants the Lock



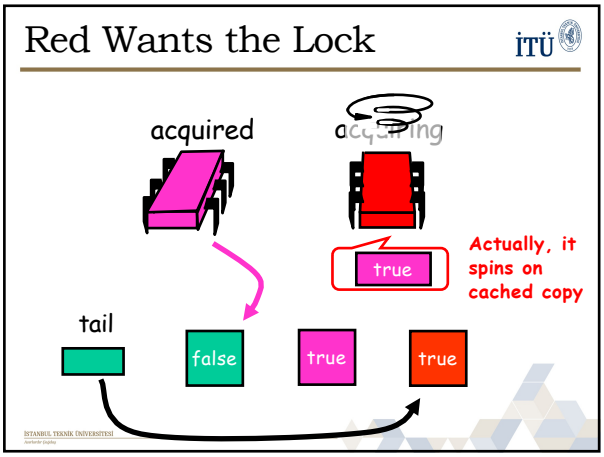
Red Wants the Lock



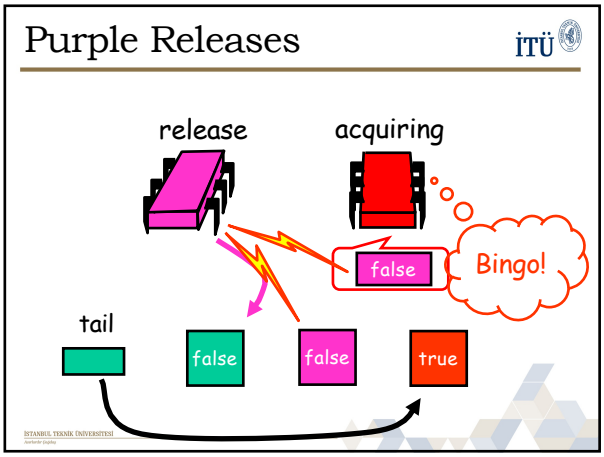
Red Wants the Lock



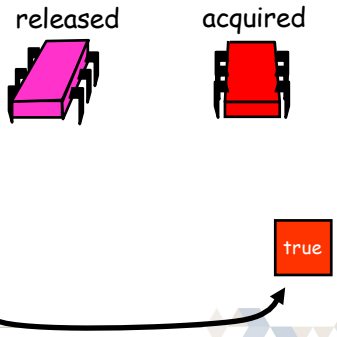
Red Wants the Lock



Purple Releases



Purple Releases



CLH Queue Lock

```
class Qnode {
 AtomicBoolean locked =
   new AtomicBoolean(true);
}
```

CLH Queue Lock

```
class CLHLock implements Lock {
 AtomicReference<Qnode> tail;
 ThreadLocal<Qnode> myNode = new Qnode();
 ThreadLocal<Qnode> pred = new Qnode();
 public void lock() {
  pred = tail.getAndSet(myNode);
  while (pred.locked) {}
 }
}
```

CLH Queue Lock

```
Class CLHLock implements Lock {
 …
 public void unlock() {
  myNode.locked.set(false);
  myNode = pred;
 }
}
```

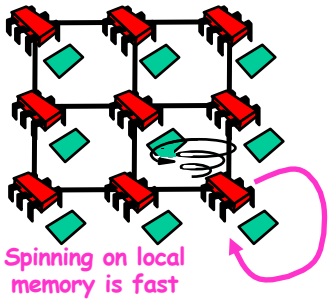
CLH Lock

- Good
  - Lock release affects predecessor only
  - Small, constant-sized space
- Bad
  - Doesn't work for uncached NUMA architectures

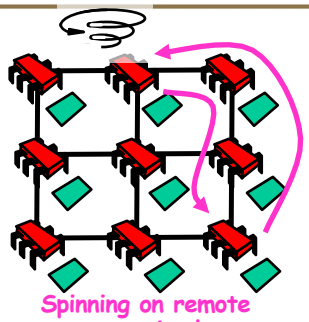
NUMA Architecturs

- Acronym:
  - Non-Uniform Memory Architecture
- Illusion:
  - Flat shared memory
- Truth:
  - No caches (sometimes)
  - Some memory regions faster than others

NUMA Machines



NUMA Machines



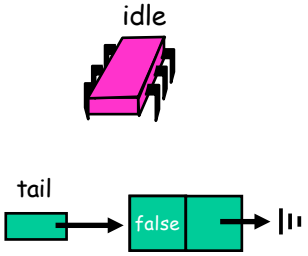
CLH Lock

- Each thread spin's on predecessor's memory
- Could be far away …

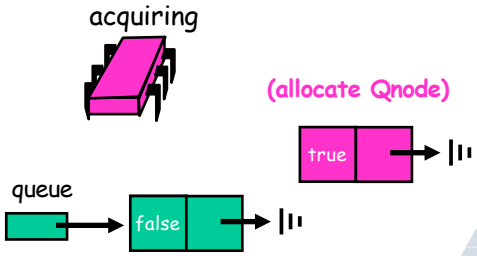
MCS Lock

- FIFO order
- Spin on local memory only
- Small, Constant-size overhead

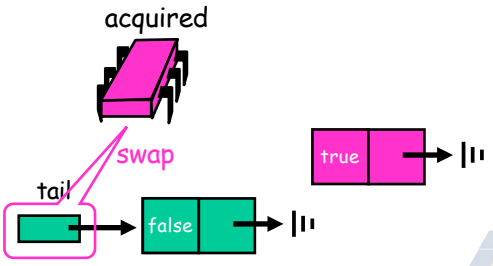
Initially



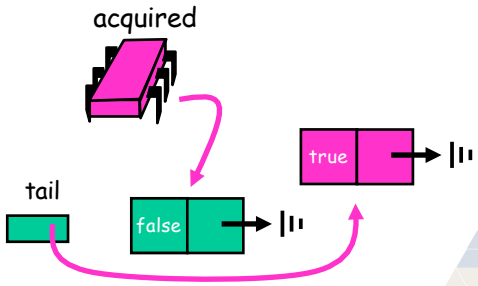
Acquiring



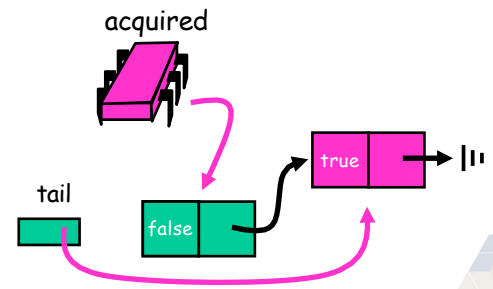
Acquiring



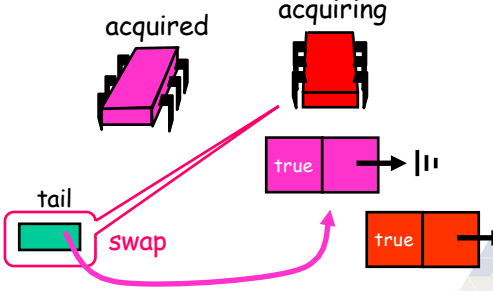
Acquiring



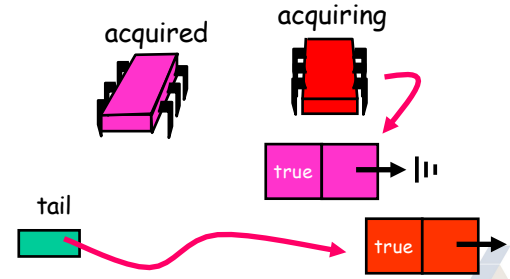
Acquired



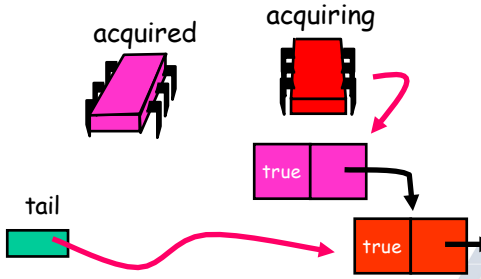
Acquiring



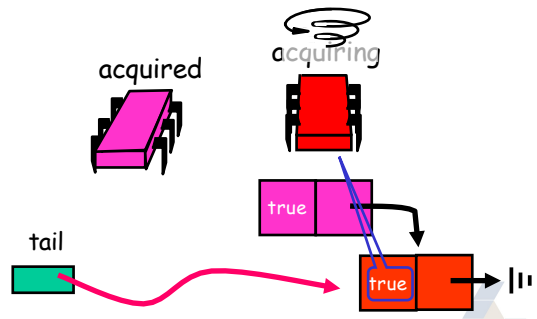
Acquiring



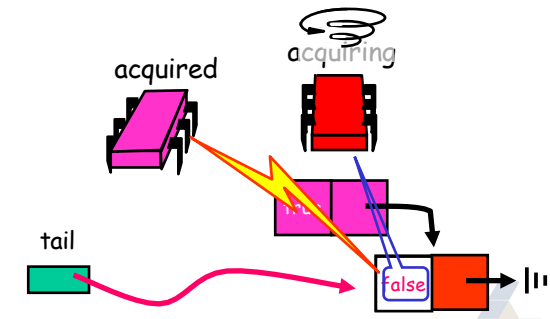
Acquiring



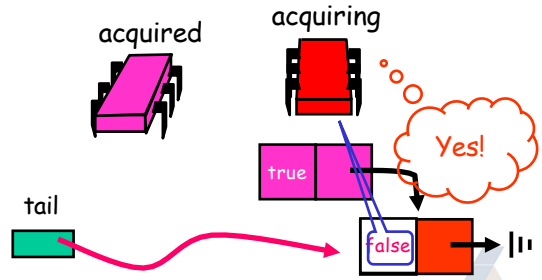
Acquiring



Acquiring



Acquiring



MCS Queue Lock

```
class Qnode {
 boolean locked = false;
 qnode   next   = null;
}
```

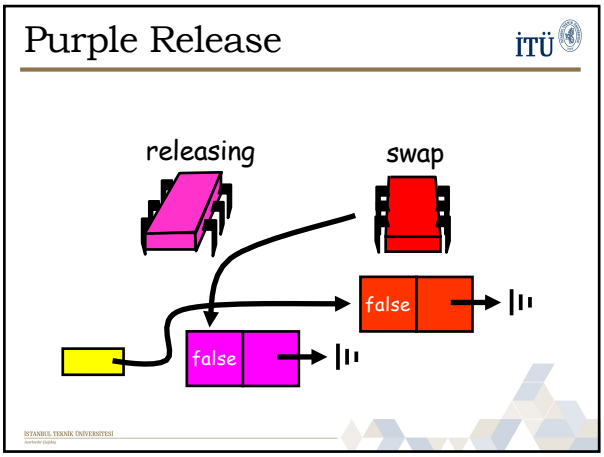
MCS Queue Lock

```
class MCSLock implements Lock {
 AtomicReference tail;
 public void lock() {
  Qnode qnode = new Qnode();
  Qnode pred = tail.getAndSet(qnode);
  if (pred != null) {
   qnode.locked = true;
   pred.next = qnode;
   while (qnode.locked) {}
  }}}
```

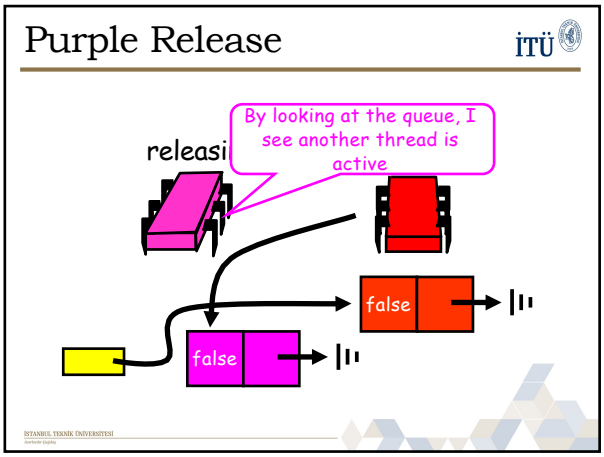
MCS Queue Unlock

```
class MCSLock implements Lock {
 AtomicReference tail;
 public void unlock() {
  if (qnode.next == null) {
   if (tail.CAS(qnode, null)
    return;
   while (qnode.next == null) {}
  }
 qnode.next.locked = false;
}}
```

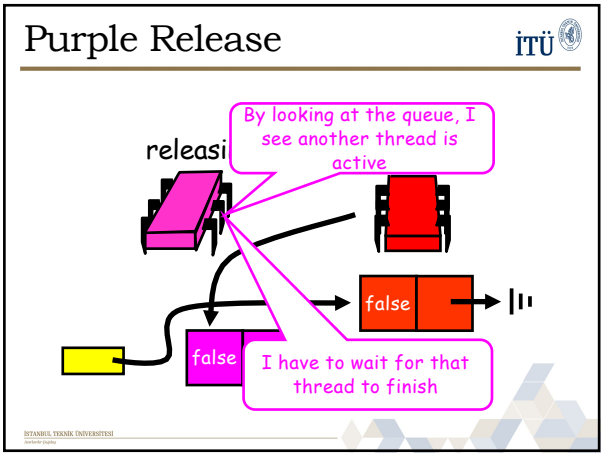
Purple Release



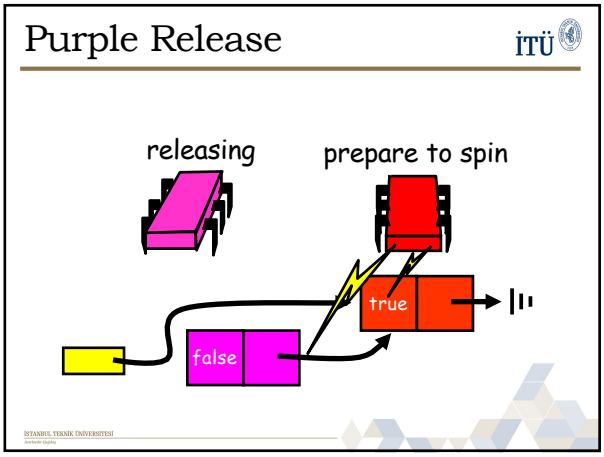
Purple Release



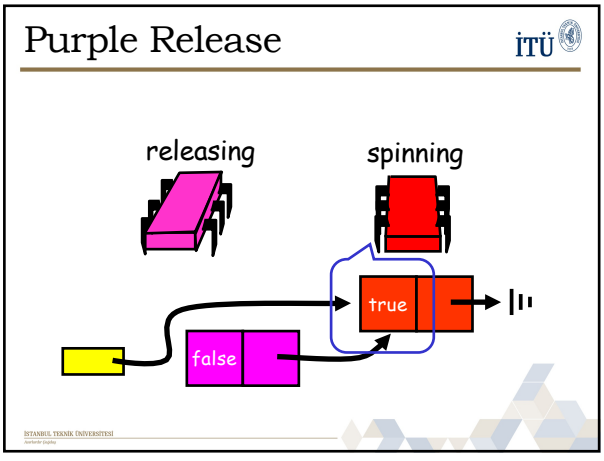
Purple Release



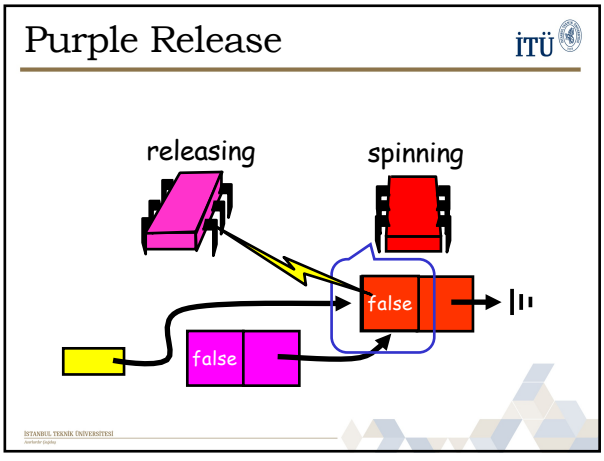
Purple Release



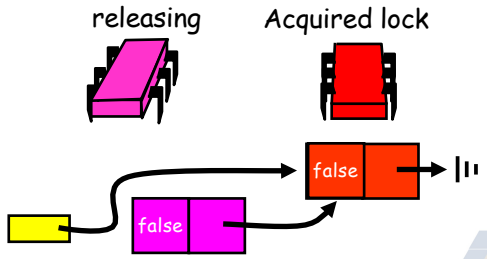
Purple Release



Purple Release



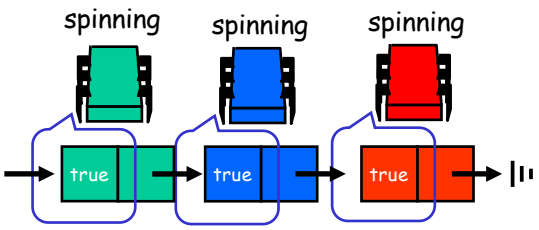
Purple Release



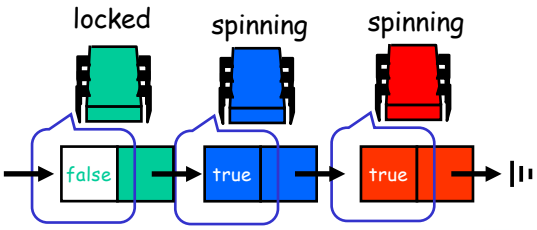
Abortable Locks

- What if you want to give up waiting for a lock?
- For example
  - Timeout
  - Database transaction aborted by user

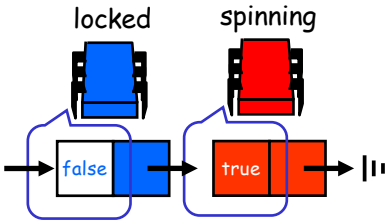
Queue Locks



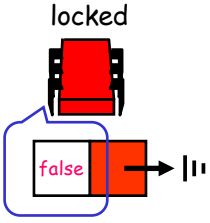
Queue Locks



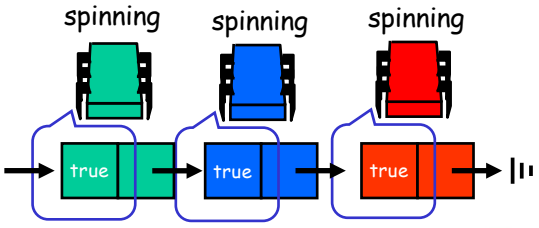
Queue Locks



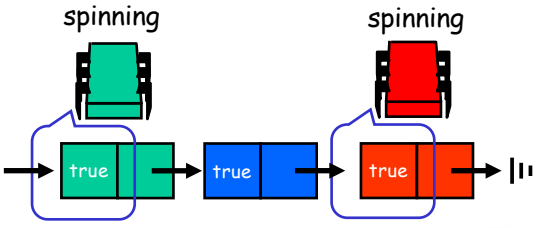
Queue Locks



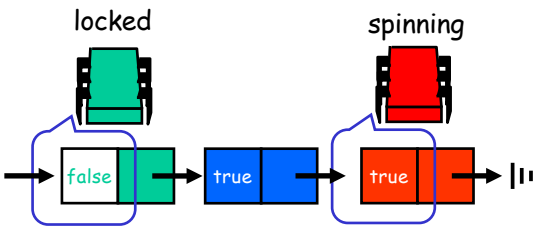
Queue Locks



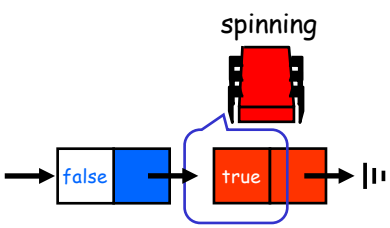
Queue Locks



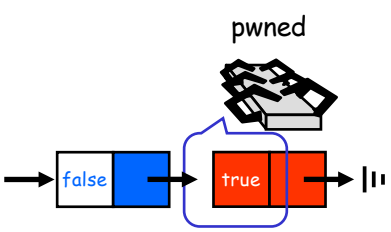
Queue Locks



Queue Locks



Queue Locks



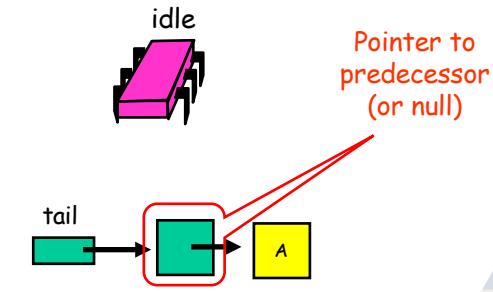
Queue Locks

- Can't just quit
  - Thread in line behind will starve
- Need a graceful way out

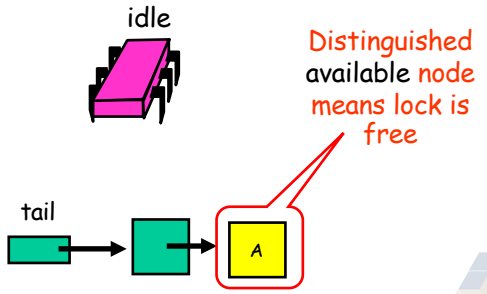
Abortable CLH Lock

- When a thread gives up
  - Removing node in a wait-free way is hard
- Idea:
  - let successor deal with it.

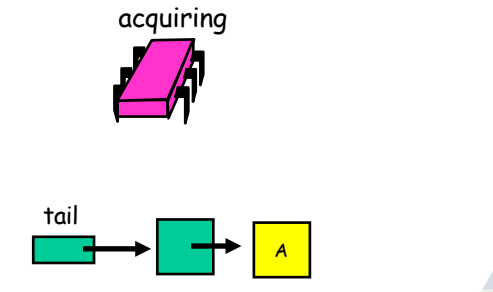
Initially



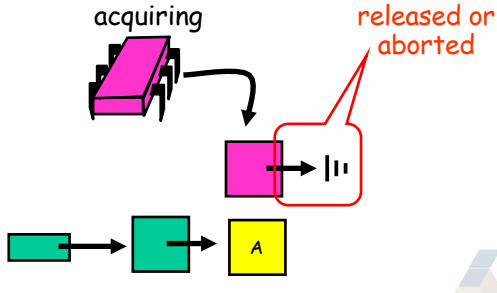
Initially



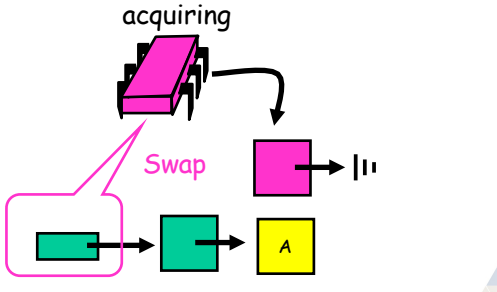
Acquiring



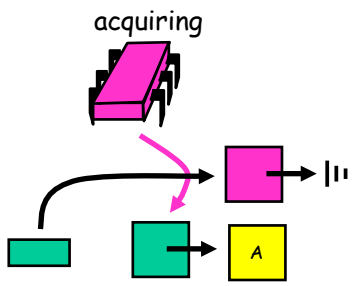
Acquiring



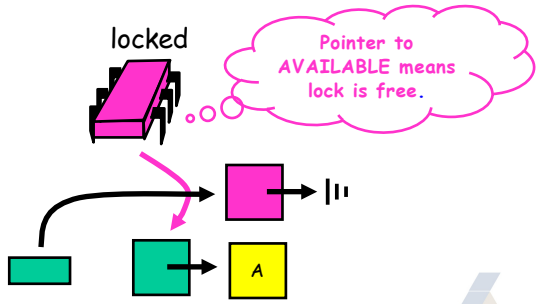
Acquiring



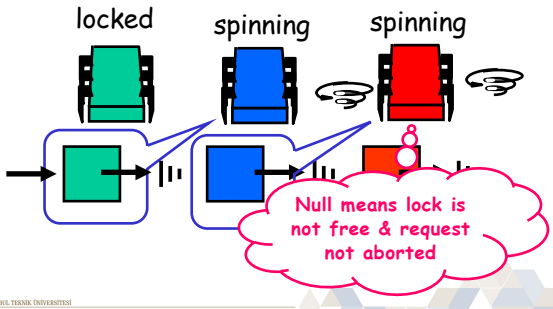
Acquiring



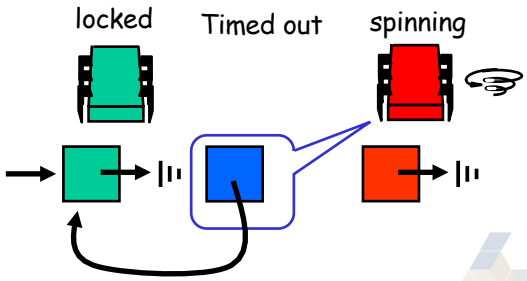
Acquired



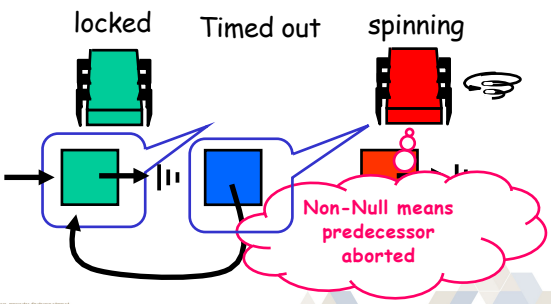
Normal Case



One Thread Aborts

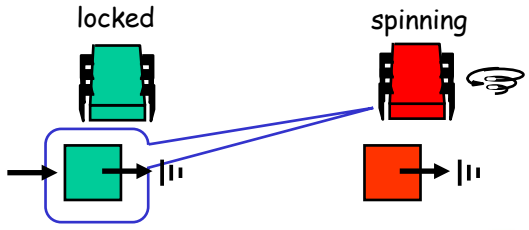


Successor Notices

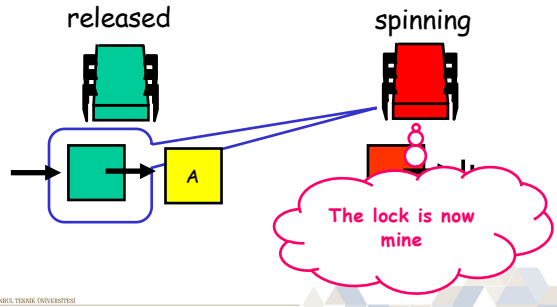


Recycle Predecessor's Node

Spin on Earlier Node

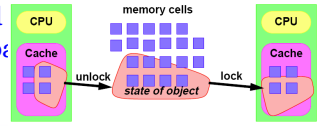


Spin on Earlier Node



What's in place in Java™

- Objects represent a state and procedures over it
- Every Thread represents the state of a single activity touching different objects
- Synchronized block ensures thread-local cache flush/reload
- Volatile keyword cache flush/relo



Volatile variables

Declaring a volatile Java variable means:

- The value of this variable will never be cached thread-locally: all reads and writes will go straight to "main memory";
- Access to the variable acts as though it is enclosed in a synchronized block, synchronized on itself.

As of Java 5, accessing a volatile variable creates a memory barrier: it effectively synchronizes all cached copies of variables with main memory, just as entering or exiting a synchronized block that synchronizes on a given object.

Locks

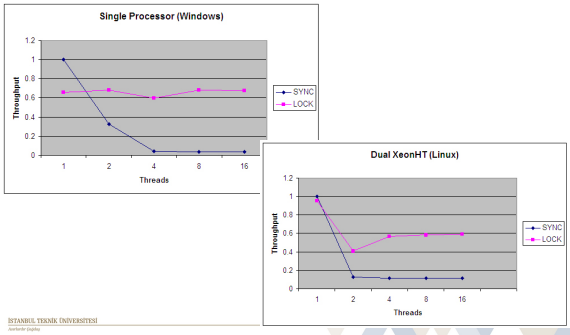
- A generalization of built-in locking behavior, but with several different types of locks with superior functionality
- Reentrant lock: This has the same overall semantics as the "synchronized" keyword, with the exception that locks must be explicitly obtained and released
  - Provides better throughput than synchronized when multiple threads are racing for the same lock
- ReadWrite lock: A pair of locks, one for reading and one for writing. The read lock can be simultaneously held by multiple threads. The write lock is exclusive
- AbstractQueuedSynchronizer: provides a framework for implementing locks and related synchronizers

Locks

- Flexibility at expense of verbosity
- Overcomes limitations of synchronized
  - Doesn't force block structured locking/unlocking
  - Allow interruptible lock acquisition and "try lock"
  - Can define customized implementations
  - Fair/Unfair

```
lock.lock();
try {
  action();
}
finally {
  lock.unlock();
}
```

How much faster?



Lock API

- Concrete ReentrantLock implementation
  - Fast, scalable with synchronized block semantics, and additional query methods
  - Also FairReentrantLock subclass with

```
interface Lock {
  void lock();
  void lockInterruptibly() throws IE;
  boolean trylock();
  boolean trylock(long timeout,TimeUnit unit)throws IE;
  void unlock();
  Condition newCondition();
}
```

Lock Example

```
class ParticleUsingLock {
  private int x, y;
  private final Random rng = new Random();
  private final Lock lock = new ReentrantLock();
  public void move() throws InterruptedException {
    lock.lockInterruptibly(); // allow cancellation
    try {
      x += rng.nextInt(3) - 1;
      y += rng.nextInt(3) - 1;
    }
    finally { lock.unlock(); }
  }
 . . .
```

Lock Example

```
 . . .
  public void draw(Graphics g) {
    int lx, ly;
    lock.lock(); // no interrupts - AWT Event Thread
    try {
      lx = x; ly = y;
    }
    finally { lock.unlock(); }
    g.drawRect(lx, ly, 10, 10);
  }
}
```

Read-Write Locks

- A pair of locks for enforcing multiple-reader, single-writer access
  - Each used in the same way as ordinary locks
- Concrete **`ReentrantReadWriteLock`**
  - Almost always the best choice for apps
  - Each lock acts like a reentrant lock

```
interface ReadWriteLock {
  Lock readock();
  Lock writeLock();
}
```

Semaphores

- Semaphores can be seen as permit holders
  - Create with initial number of permits
  - **`acquire`** takes a permit, waiting if necessary
  - **`release`** adds a permit
  - But no actual permits change hands.
  - Semaphore just maintains the current count.
- Can use for both “locking” and “synchronizing”
  - With initial permits=1, can serve as a lock
  - Useful in buffers, resource controllers
  - Use in designs prone to missed signals
  - Semaphores “remember” past signals

İTÜ

Barrier Synchronization

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

İSTANBUL TEKNİK ÜNİVERSİTESİ

Simple Video Game

İTÜ

- Prepare frame for display
  - By graphics coprocessor
- "soft real-time" application
  - Need at least 35 frames/second
  - OK to mess up rarely

Simple Video Game

İTÜ

```
while (true) {
  frame.prepare();
  frame.display();
  }
```


Simple Video Game

İTÜ

```
while (true) {
  frame.prepare();
  frame.display();
  }
```

- What about overlapping work?
  - 1st thread displays frame
  - 2nd prepares next frame

Two-Phase Rendering

İTÜ

```
while (true) {
  if (phase) {
    frame[0].display();
  } else {
    frame[1].display();
  }
  phase = !phase;
}
```

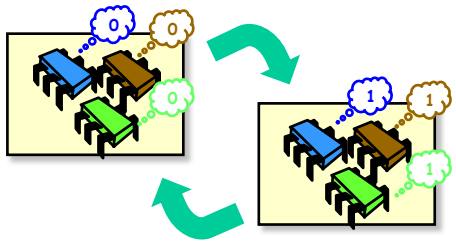
```
while (true) {
  if (phase) {
    frame[1].prepare();
  } else {
    frame[0].prepare();
  }
  phase = !phase;
}
```


Synchronization Problems

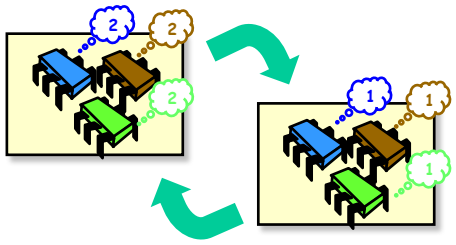
İTÜ

- How do threads stay in phase?
- Too early?
  - "we render no frame before its time"
- Too late?
  - Recycle memory before frame is displayed

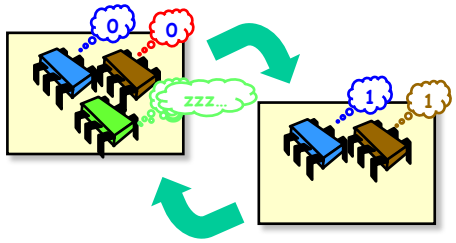
Ideal Parallel Computation



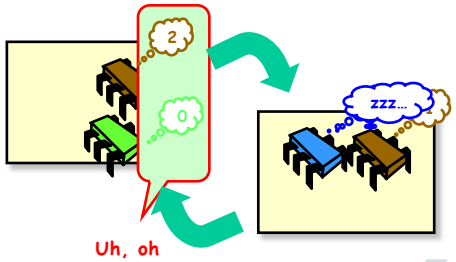
Ideal Parallel Computation



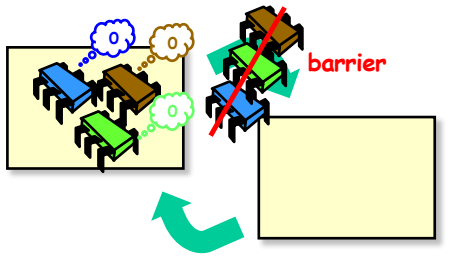
Real-Life Parallel Computation



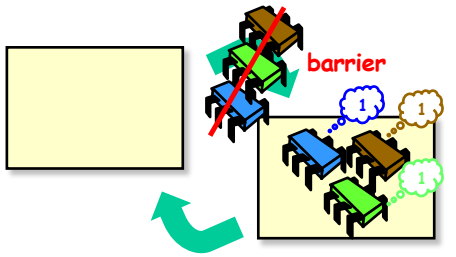
Real-Life Parallel Computation



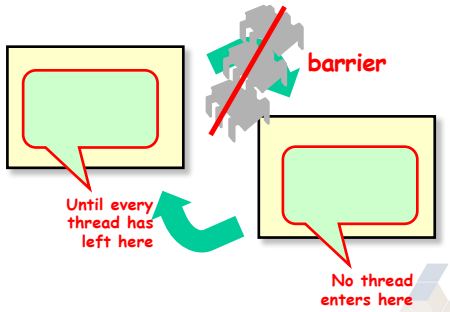
Barrier Synchronization



Barrier Synchronization



Barrier Synchronization



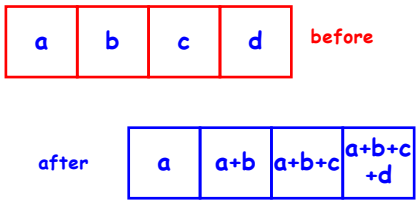
Why Do We Care?

- Mostly of interest to
  - Scientific & numeric computation
- Elsewhere
  - Garbage collection
  - Less common in systems programming
  - Still important topic

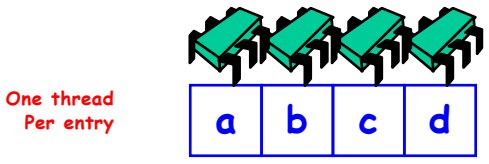
Duality

- Dual to mutual exclusion
  - Include others, not exclude them
- Same implementation issues
  - Interaction with caches …
    - Invalidation?
    - Local spinning?

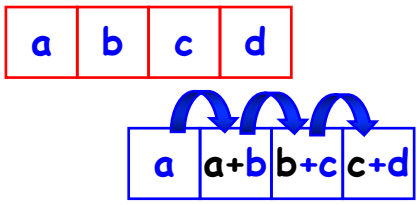
Example: Parallel Prefix



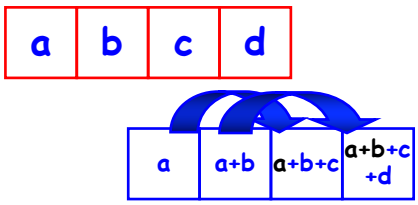
Parallel Prefix



Parallel Prefix: Phase 1



Parallel Prefix: Phase 2



Parallel Prefix

- N threads can compute
  - Parallel prefix
  - Of N entries
  - In $\log_2$ N rounds
- What if system is asynchronous?
  - Why we need barriers

Prefix

```
class Prefix extends Thread {
 private int[] a;
 private int i;
 private Barrier b;
 public Prefix(int[] a, Barrier b, int i) {
  a = a;
  b = b;
  i = i;
 }
```


Where Do the Barriers Go?

```
public void run() {
 int d = 1, sum = 0;
 while (d < N) {
  if (i >= d){
    sum = a[i-d];
    a[i] += sum;
  }
  d = d * 2;
 }
}
```


Where Do the Barriers Go?

```
public void run() {
 int d = 1, sum = 0;
 while (d < N) {
  if (i >= d){
    sum = a[i-d];
    b.await();
    a[i] += sum;
  }
  d = d * 2;
 }
}
```


Where Do the Barriers Go?

```
public void run() {
 int d = 1, sum = 0;
 while (d < N) {
  if (i >= d){
    sum = a[i-d];
    b.await();
    a[i] += sum;
    b.await();
  }
  d = d * 2;
 }
}
```

Make sure everyone reads before anyone writes

Where Do the Barriers Go?

```
public void run() {
 int d = 1, sum = 0;
 while (d < N) {
  if (i >= d){
    sum = a[i-d];
    b.await();
    a[i] += sum;
    b.await();
  }
  d = d * 2;
 }
}
```

Make sure everyone reads before anyone writes

Make sure everyone writes before anyone reads

Barrier Implementations

- Cache coherence
  - Spin on locally-cached locations?
  - Spin on statically-defined locations?
- Latency
  - How many steps?
- Symmetry
  - Do all threads do the same thing?

Barriers

```
public class Barrier {
 AtomicInteger count;
 int size;
 public Barrier(int n){
  count = AtomicInteger(n);
  size = n;
 }
 public void await() {
  if (count.getAndDecrement()==1) {
   count.set(size);
  } else {
   while (count.get() != 0);
 }}}}
```


Barriers

```
public class Barrier {
 AtomicInteger count;
 int size;
 public Barrier(int n){
  count = AtomicInteger(n);
  size = n;
 }
 publi
  if (count.getAndDecrement()==1) {
   count.set(size);
  } else {
   while (count.get() != 0);
 }}}}
```

What's wrong with this protocol?

Reuse

```
Barrier b = new Barrier(n);
while ( mumble() ) {
  work();
  b.await()
}
```

Do work

synchronize

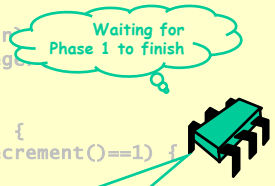
repeat

Barriers

```
public class Barrier {
 AtomicInteger count;
 int size;
 public Barrier(int n){
  count = AtomicInteger(n);
  size = n;
 }
 public void await() {
  if (count.getAndDecrement()==1) {
   count.set(size);
  } else {
   while (count.get() != 0);
 }}}}
```

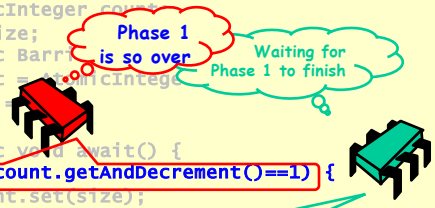
Barriers

```
public class Barrier {
 AtomicInteger count;
 int size;
 public Barrier(int n) {
  count = AtomicInteger(n);
  size = n;
 }
 public void await() {
  if (count.getAndDecrement()==1) {
   count.set(size);
  } else {
   while (count.get() != 0);
  }
 }
}
```



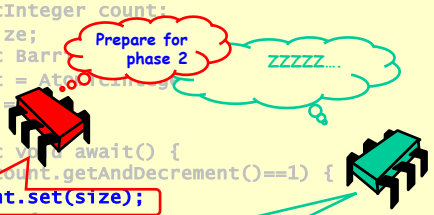
Barriers

```
public class Barrier {
 AtomicInteger count;
 int size;
 public Barrier(int n) {
  count = AtomicInteger(n);
  size = n;
 }
 public void await() {
  if (count.getAndDecrement()==1) {
   count.set(size);
  } else {
   while (count.get() != 0);
  }
 }
}
```



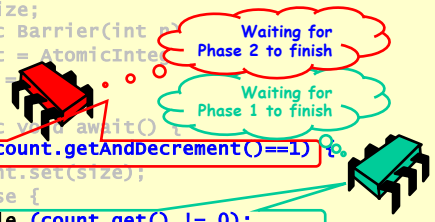
Barriers

```
public class Barrier {
 AtomicInteger count;
 int size;
 public Barrier(int n) {
  count = AtomicInteger(n);
  size = n;
 }
 public void await() {
  if (count.getAndDecrement()==1) {
   count.set(size);
  } else {
   while (count.get() != 0);
  }
 }
}
```



Uh-Oh

```
public class Barrier {
 AtomicInteger count;
 int size;
 public Barrier(int n) {
  count = AtomicInteger(n);
  size = n;
 }
 public void await() {
  if (count.getAndDecrement()==1) {
   count.set(size);
  } else {
   while (count.get() != 0);
  }
 }
}
```



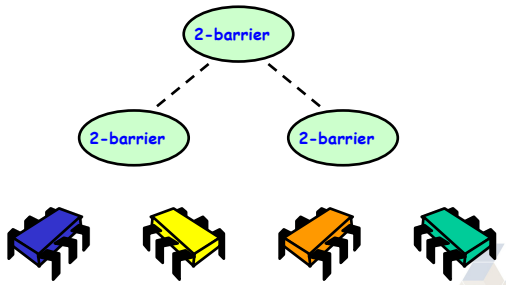
Basic Problem

- One thread "wraps around" to start phase 2
- While another thread is still waiting for phase 1

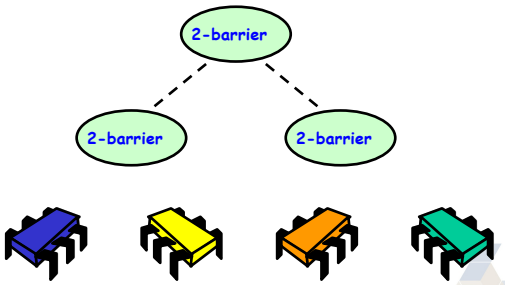
Sense-Reversing Barriers

```
public class Barrier {
 AtomicInteger count;
 int size;
 boolean sense = false;
 threadSense = new ThreadLocal<boolean>…

 public void await {
  boolean mySense = threadSense.get();
  if (count.getAndDecrement()==1) {
   count.set(size);
   sense = !mySense
  } else {
   while (sense != mySense) {}
  }
  threadSense.set(!mySense)
 }
}}
```

Combining Tree Barriers



Combining Tree Barriers



Combining Tree Barrier

```
public class Node{
 AtomicInteger count;
 int size;
 Node parent;
 volatile boolean sense;

 public void await() {…
  if (count.getAndDecrement()==1) {
   if (parent != null)
    parent.await()
    count.set(size);
    sense = mySense
   } else {
   while (sense != mySense) {}
  }
…}}}
```


Combining Tree Barrier

- No sequential bottleneck
  - Parallel getAndDecrement() calls
- Low memory contention
  - Same reason
- Cache behavior
  - Local spinning on bus-based architecture
  - Not so good for NUMA

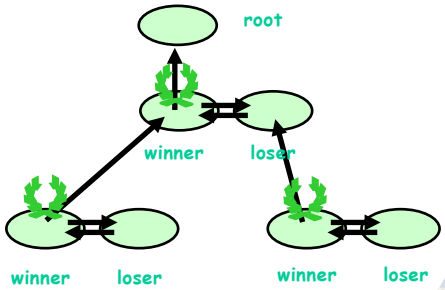
Remarks

- Everyone spins on sense field
  - Local spinning on bus-based (good)
  - Network hot-spot on distributed architecture (bad)
- Not really scalable

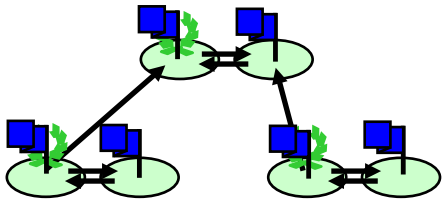
Tournament Tree Barrier

- If tree nodes have fan-in 2
  - Don't need to call getAndDecrement()
  - Winner chosen statically
- At level i
  - If i-th bit of id is 0, move up
  - Otherwise keep back

Tournament Tree Barriers

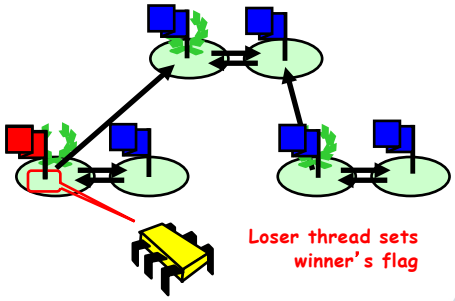


Tournament Tree Barriers



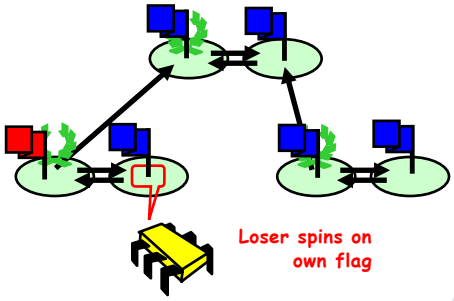
All flags blue

Tournament Tree Barriers



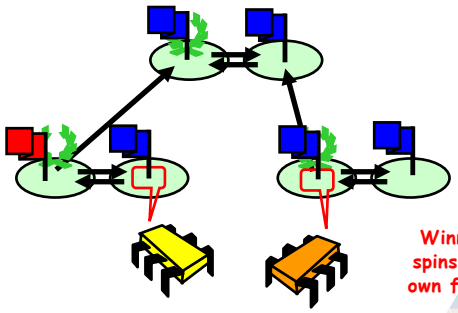
Loser thread sets winner's flag

Tournament Tree Barriers



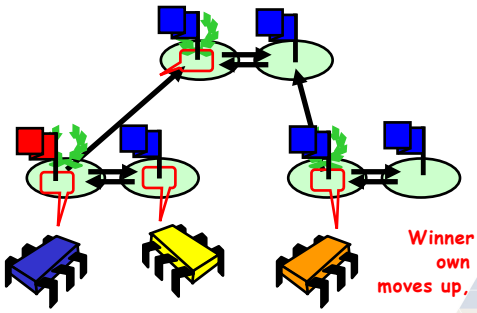
Loser spins on own flag

Tournament Tree Barriers



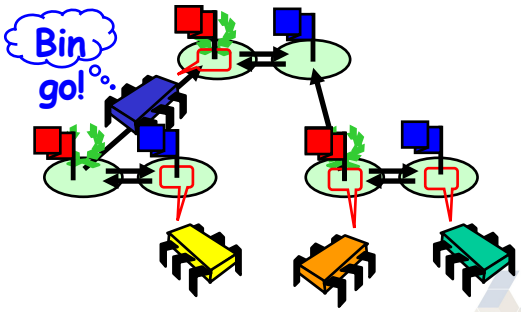
Winner spins on own flag

Tournament Tree Barriers

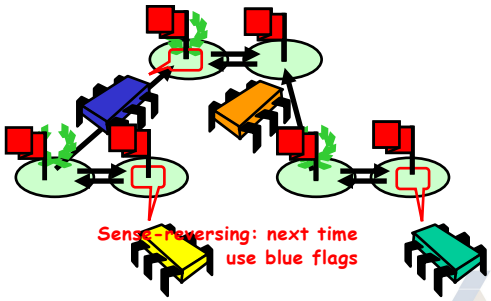


Winner sees own flag, moves up, spins

Tournament Tree Barriers



Tournament Tree Barriers



Tournament Barrier

```
class TBarrier {
  boolean flag;
  TBarrier partner;
  TBarrier parent;
  boolean top;
  …
}
```


Tournament Barrier

```
void await(boolean mySense) {
 if (top) {
  return;
 } else if (parent != null) {
  while (flag != mySense) {};
  parent.await(mySense);
  partner.flag = mySense;
 } else {
  partner.flag = mySense;
  while (flag != mySense) {};
}}}
```

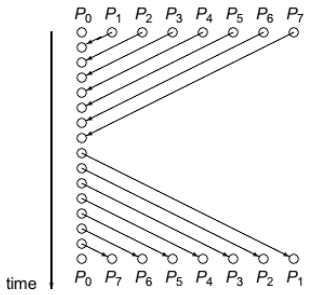

Remarks

- No need for read-modify-write calls
- Each thread spins on fixed location
  - Good for bus-based architectures
  - Good for NUMA architectures

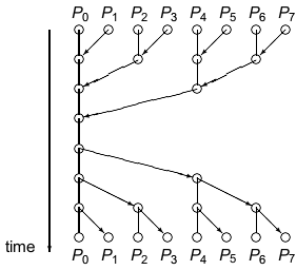
Dissemination Barrier

- At round i
  - Thread A notifies thread $A+2^i$ (mod n)
- Requires log n rounds

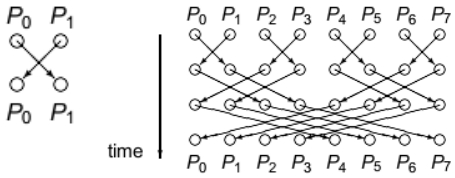
Linear Barrier



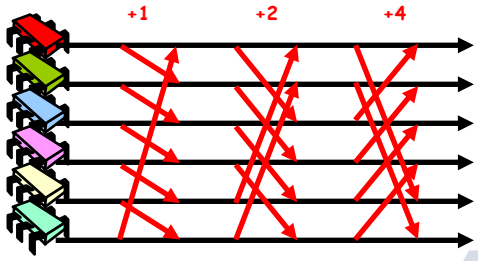
Tree Barrier



Butterfly Barrier



Dissemination Barrier



Remarks

- Great for lovers of combinatorics
- Not so much fun for those who track memory footprints

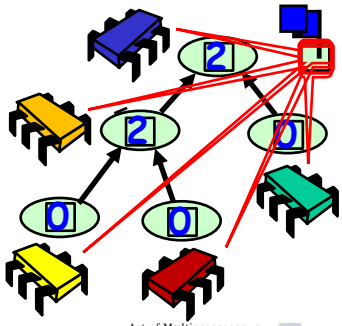
Ideas So Far

- Sense-reversing
  - Reuse without reinitializing
- Combining tree
  - Like counters, locks …
- Tournament tree
  - Optimized combining tree
- Dissemination barrier
  - Intellectually Pleasing (matter of taste)

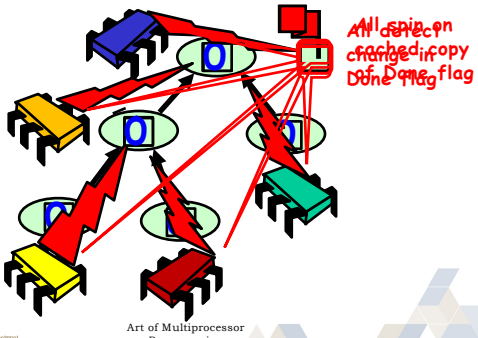
Which is best for Multicore?

- On a cache coherent multicore chip: none of the above…
- Use a static tree barrier!

Static Tree Barrier



Static Tree Barrier



CountDownLatch

- A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.
- A CountDownLatch is initialized with a given *count*
- The await methods block until the current count reaches zero due to invocations of the countDown method, after which all waiting threads are released and any subsequent

CountDownLatch Example

```
class Driver {
  void main(int N) throws InterruptedException {
    final CountDownLatch startSignal = new CountDownLatch(1);
    final CountDownLatch doneSignal = new CountDownLatch(N);


    for (int i = 0; i < N; ++i) // Make threads
      new Thread() {
        public void run() {
          try {
             startSignal.wait();
             doWork();
             doneSignal.countDown();
          }
          catch(InterruptedException ie) {}
        }}.start();

  . . .
```

CountDownLatch Example

```
  . . .
    initialize();
    startSignal.countDown();  // Let all threads proceed
    doSomethingElse();
    doneSignal.await();       // Wait for all to complete
     cleanup();
  }
}
```

Barriers (Cyclic Barrier)

- A synchronization aid that allows a set of threads to all reach a common point before proceeding
- Useful in programs that involve a fixed number of threads that must wait for each other at some point
- Called cyclic because barrier can be reused
- A thread signals it has reached the barrier by calling CyclicBarrier.await()
- The threads blocks until all other threads have reached the barrier (which signal this the same way)
- Can specify a timeout at which time the thread will stop blocking

Barriers (Cyclic Barrier)

```
class cyclicbarriertest {
void compute(Task t, int num){
   //two inputs,num_threads,runnable tasks
   final CyclicBarrier barrier =
            new CyclicBarrier(num, new Runnable() {
               for (int i = 0; i < nThreads; ++i) {
                  //split computation betweeen threads
                  final int id = i;
                  Runnable worker = new Runnable() { // new worker
                       final Segment segment = t.createSegment(id);
                       public void run() {
                          try {
                             segment.update();
                             barrier.await();
                          }
                          catch(Exception e) { //Take appr. action}
                       }
                  };
                  new Thread(worker).start();
               }
            };
}}
```


Exchangers

- A synchronization point at which two threads can exchange objects
- Can be thought of as a CyclicBarrier with a count of two plus allowing an exchange of state at the barrier
- On calling the exchange() method the thread provides some object as input
- The exchange() methods returns the object entered as input by the second thread to the calling thread

Exchangers

- Useful for example in the case where one thread is filling a buffer (filler thread) and the other emptying (emptying thread)
- On calling exchange() the filler thread is passed an empty buffer
- The emptying thread obtain the newly full buffer from the filler thread

Remarks

- Very little cache traffic
- Minimal space overhead
- Can be laid out optimally on NUMA
  - Instead of Done bit, notification must be performed by passing sense down the static tree