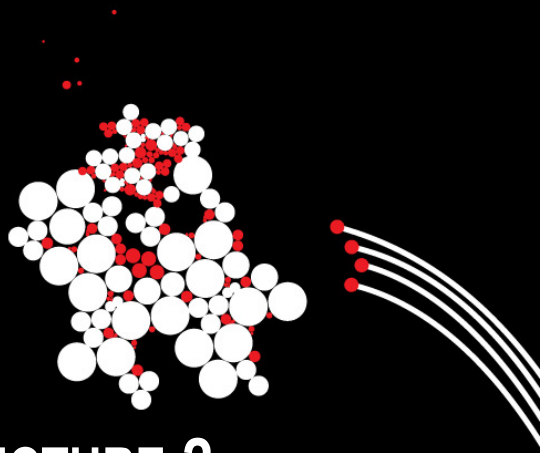
UNIVERSITY OF TWENTE.

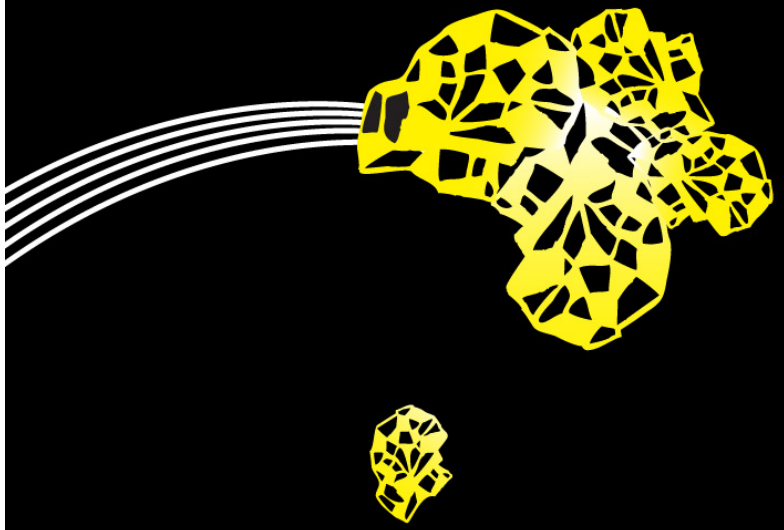


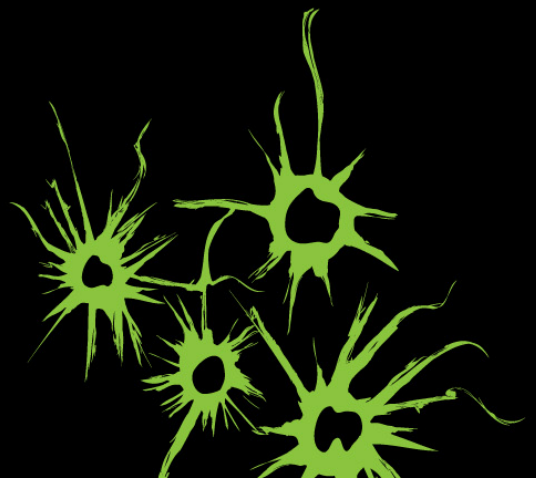
CONCURRENT PROGRAMMING – LECTURE 2

SYNCHRONISATION

MODULE 8: PROGRAMMING PARADIGMS

8 MAY 2018

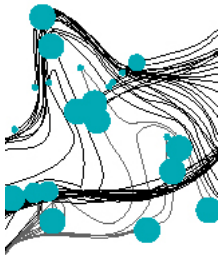




CONCURRENT PROGRAMMING

CONTENTS

2	Tue 30/4	Basics of concurrency, thread safety, testing concurrent systems	Ch. 1-3, 12 (+ SS material)
3	Tue 7/5	Synchronisation	Ch. 4, 5, 13, 14.1-14.4
4	Wed 15/5	Liveness, performance, and fairness	Ch. 10,11
5	Thu 23/5	Homogeneous threading (OpenMP, OpenCL)	Papers
6	Mon 3/6	Safe concurrency (Software Transactional Memory, Rust)	Papers
7	Fri 7/6	Fine-grained concurrency, memory models	Ch. 14.5-6, 15, 16



CONTENTS

- Quiz: basics of concurrency: thread safety, race conditions and data races
- Critical sections
- Locks in Java
- Condition variables
- Other synchronisers
- Synchronised collections

Literature:
JCIP: Chapters 4, 5, 13, 14.1-14.4

THREAD SAFETY

From Block 1

- Thread-safe code fragment:
  - shared data structures safely executed by multiple threads at same time
  - Parallel execution should not introduce errors
- Ensuring thread-safety: coordinating access to shared, mutable state

From Block 1

COORDINATING ACCESS TO SHARED STATE

- Thread confinement: thread local data
- Restricted volatile: single thread read-modify-writes, all others just read
- Stack confinement: local variables are local, and are not leaked

  API class: ThreadLocal (conceptually: Map<Thread, Value>)
- Immutable objects
- Safe publication: store reference in a place protected by a synchroniser
  - from static initialiser into volatile or final field
  - into field guarded by a lock

Programmer's responsibility!

QUESTION 1

```
class CP11 {
        public static final int N = 500;
        public static int[] a = new int[N];
        public static void main(String [] args) {
                (new Thread1()).start();
                (new Thread2()).start();
        }}
```

```
class Thread1 extends Thread {
    public void run() {
        int i = 0;
        while (i < CP11.N) { CP11.a[i] = CP11.a[i] + 2;  i ++;}
}}
```

```
class Thread2 extends Thread {
    public void run() {
        int i = 0;
        while (i < CP11.N) { CP11.a[i] = CP11.a[i] + 4; i ++; }
}}
```

Is this thread-safe?

a. Yes
b. No
c. Arrays are never thread safe
d. If you change the loop in Thread2 to `i = N; while (i >= 0){ a[i] = …; i--;}`

QUESTION 2

```
class CP12 {
        public static final int N = 500;
        public static int[] a = new int[2*N];
        public static void main(String [] args) {
                (new Thread1()).start();
                (new Thread2()).start();
        }}
```

```
class Thread1 extends Thread {
    public void run() {
        int i = 0;
        while (i < CP12.N) { CP12.a[i] = CP12.a[i] + 2;  i ++; }
}}

class Thread2 extends Thread {
    public void run() {
        int i = CP12.N;
        while (i < 2 * CP12.N) { CP12.a[i] = CP12.a[i] + 2; i ++;    }
}}
```

Is this thread-safe?

a. Yes
b. No
c. This cannot be done in parallel
d. Only if you add synchronisation

QUESTION 3

```
class CP13 {
    public static int x = 0;
    public static void main(String [] args) {
        (new Thread1()).start();
        (new Thread2()).start();
    }}
```

```
class Thread1 extends Thread {
    public void run() {
        CP13.x = CP13.x + 2;
}}

class Thread2 extends Thread {
    public void run() {
        CP13.x = CP13.x * 2;
}}
```

Does this have data races or race conditions?

a. None
b. Only data races
c. Only race conditions
d. Both

QUESTION 4

```
class CP14 {
    public static int x = 0;
    public static int y = 0;
    public static void main(String [] args) {
        (new Thread1()).start();
        (new Thread2()).start();
    }}
```

```
class Thread1 extends Thread {
   public void run() {
      int  r1 = CP14.x;
      if  (r1 >  0)  {   CP14.y =  1; }
}}

class Thread2 extends Thread {
   public void run() {
      int r1 = CP14.y;
      if  (r1 > 0) { CP14.x  =  1; }
}}
```

Does this have data races or race conditions?

a. None
b. Only data races
c. Only race conditions
d. Both

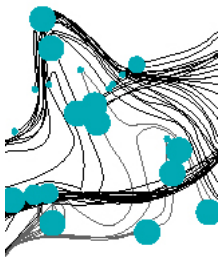
QUESTION 5

```
class CP15 {
    public static volatile long x = 0;
    public static void main(String [] args) {
        (new Thread1()).start();
        (new Thread2()).start();
    }}
```

```
class Thread1 extends Thread {
    public void run() {
        while (CP15.x < Integer.MAX_VALUE) {}
        System.out.println("ready");
    }}

class Thread2 extends Thread {
    public void run() {
        CP15.x = Integer.MAX_VALUE;
}}
```

Why will this always print “ready”?

a. It won’t
b. Because x is greater than Integer.MAX_VALUE
c. Because x is volatile, so the long is always properly written
d. Because the update to x will always be visible to the other thread



CRITICAL SECTION PROBLEM

CRITICAL SECTION PROBLEM

Each of *N* processes is executing in an infinite loop a sequence of statements that can be divided into two subsequences: the critical section and the non-critical section

```
while (true) {
  non-critical section
  pre protocol
  critical section
  post protocol
}
```

Only within a critical section, we want to access or modify shared data

How to ensure that at most one process is executing its **critical section** at a given time?

CRITICAL SECTION PROPERTIES

CORRECTNESS

A solution to the critical section problem should have the following properties:

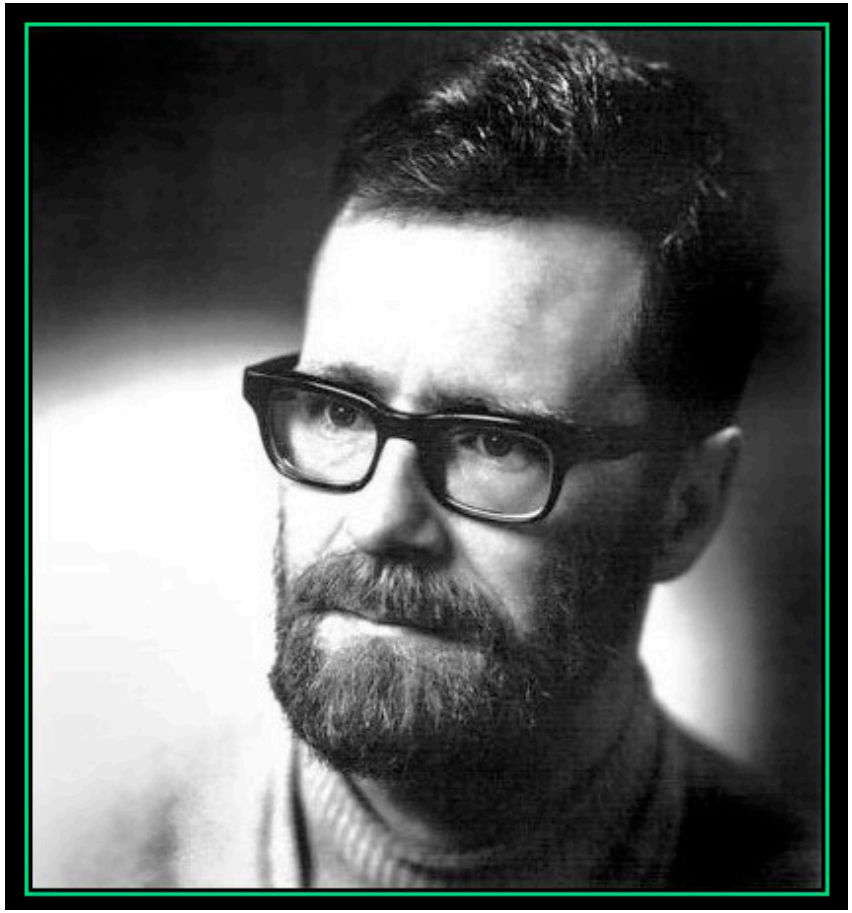
- mutual exclusion: statements from the critical sections of two or more processes must not be interleaved
- freedom from deadlock: if some processes are trying to enter their critical sections, then one of them must eventually succeed
- freedom from (individual) starvation: if any process tries to enter its critical section, then that process must eventually succeed

EDSGER DIJKSTRA (1930 – 2000)

Semaphores
- P: Passeren
- V: Vrijgeven

Semaphore: distribute access to *N* resources.

Mutex (lock) similar to binary semaphore.



LOCKS AND DATA

The association between locks and data ultimately exists only in a programmer's mind, and may be documented only in comments.

Example from the Linux kernel [fs/buffer.c]:

> *When a locked buffer is visible to the I/O layer* BH_Launder *is set. This means before unlocking we must clear* BH_Launder*,* mb() *on* alpha *and then clear* BH_Lock*, so no reader can see* BH_Launder *set on an unlocked buffer and then risk to deadlock.*

SYNCHRONISATION POLICY

- Synchronisation needs to protect all accesses to a variable, not only when a variable is written
- Document synchronisation policy
- Relation between multiple variables (invariants and constraints):
  - impact on level of atomicity
  - other threads should not be able to observe intermediate states
  - should be protected by same synchroniser

Guardedby annotation

Beware: coarse-grained locking can have impact on performance

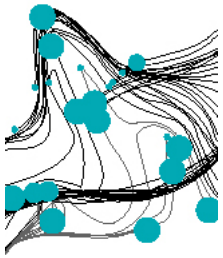
LOCKING-RELATED CHALLENGES

- Not robust: If a thread holding a lock is delayed, other threads cannot make *progress*
- Hard to use: Even a simple queue based on *fine-grained locking* is a tour de force
- Deadlock: Can occur if threads attempt to lock the same objects in different orders (*lock-order reversal*)
- Relies on conventions: Nobody really knows how to organize and maintain large systems that rely on locking
- Not composable: Managing concurrent locks to, e.g., atomically delete an item from one table and insert it in another table requires full program knowledge
- Impact on performance

CONDITIONAL LOCKING

```
static void updateCount(Vector vec) {
   int count = 0;
   if (vec.size() > 0) {
          synchronized (lock) {
            for (item : vec)
              count = count + item;
           }
         }
   }
}
```

Will this work?



SYNCHRONISATION IN JAVA

No guarantee waiting thread will ever get the lock

- Every object implicitly is a lock
- Usage: `synchronized (obj) {` critical section`}`

`Thread t` arrives at synchronized code block for Object `obj`

- If no other thread holds the lock on object `obj`, this thread will get the lock
- If another thread already has the lock, this thread will start waiting for the lock

`Thread t` leaves synchronized code block for Object `obj`

- Lock of object `obj` is passed to an arbitrary thread waiting for it

LOCK REENTRANCY

```
class Point {
    private int x;
    private int y;
    public synchronized void setX(int x) {
        this.x = x;
    }
    public synchronized void setY(int y) {
        this.y = y;
    }
    public synchronized void setPoint(int x, int y) {
        setX(x); setY(y);
    }
```

If locks are not reentrant, this code would deadlock

Useful for library code

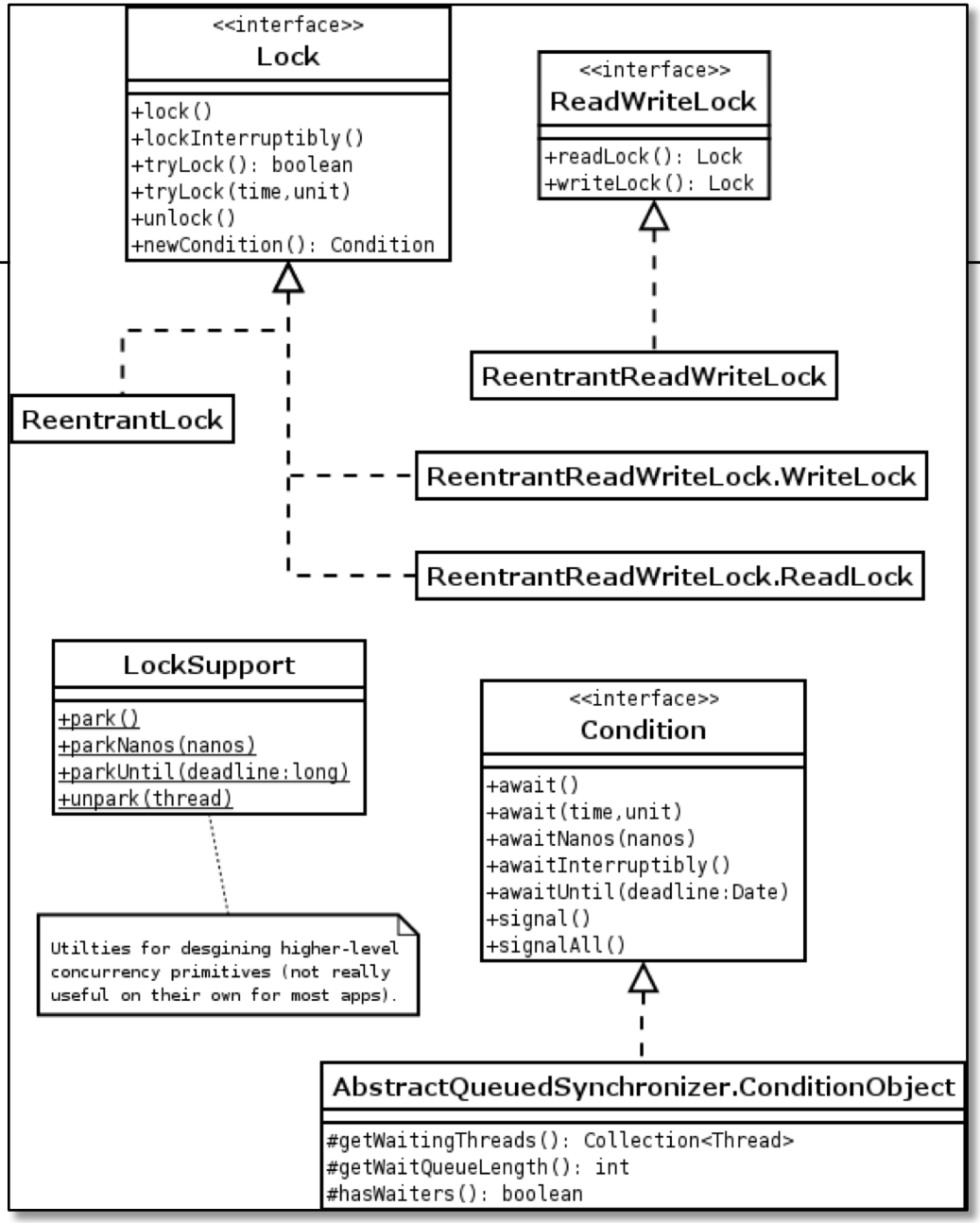
JAVA 5

2004

Introduced:

- explicit locks
- various other locks

Synchronised: intrinsic lock



LOCK INTERFACE

```
interface Lock {
  public void    lock();
  public void    unlock();
  public boolean tryLock();
  public boolean tryLock(long time, TimeUnit unit);
  Condition newCondition();
  /* … */
}
```

EXPLICIT VS INTRINSIC LOCKS

INTRINSIC LOCKS

```
class ConcurrentQueue<T> {
    int head = 0;
    int tail = 0;
    T[] items;

  public ConcurrentQueue(int capacity) {
      items = (T[]) new Object[capacity];
  }

  public synchronized void enq(T x)
          throws FullExc  { ... }

  public synchronized T deq()
          throws EmptyExc { ... }
}
```

Bounded, capacity > 0

Implicitly locked on this

EXPLICIT VS INTRINSIC LOCKS

INTRINSIC LOCKS

```
public synchronized void enq(T x) throws FullExc {
  if (tail - head == items.length)
    throw new FullExc();

  items[tail % items.length] = x;
  tail++;
}

public synchronized T deq() throws EmptyExc {
  if (tail == head)
    throw new EmptyExc();

  T x = items[head % items.length];
  head++;
  return x;
}
```

Array uses a ring buffer

`tail` is index of first free array slot

EXPLICIT VS INTRINSIC LOCKS

```
class LockBasedQueue<T> {
  int head = 0;
  tail = 0;
  T[] items;

  final Lock lock = new ReentrantLock();

  public LockBasedQueue(int capacity) {
    items = (T[]) new Object[capacity];
  }

  public void enq(T x) throws FullExc  { ... }
  public T    deq()    throws EmptyExc { ... }
}
```

Reentrant: thread holding a lock can acquire it again without blocking.

Final: lock cannot be changed

methods are not synchronized

EXPLICIT VS INTRINSIC LOCKS

This implementation mimics the built-in synchronization, but you do not have to lock/unlock in the same method!

```
public void enq(T x) throws FullExc {
  lock.lock();
  try {
    if (tail - head == items.length)
      throw new FullExc();

    items[tail % items.length]= x;
    tail++;

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

```
public T deq() throws EmptyExc {
  lock.lock();
  try {
    if (tail == head)
      throw new EmptyExc();

    T x = items[head % items.length];
    head++;
    return x;

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

WHY FINALLY CLAUSES FOR UNLOCKING?

```
public synchronized void foo(int bar)
        throws InterruptedException {
    ...
    Thread.sleep(1000);
    ...
}
```

If the thread gets interrupted, the JVM ensures that the intrinsic lock is given back.

When you use an explicit lock, you are responsible for releasing the lock.

```
Lock lock = new ReentrantLock();
...
lock.lock();
try {
   // update object state
   // catch exceptions and restore invariants
} finally {
     lock.unlock();
}
```

EXPLICIT VS INTRINSIC LOCKS

COMPARISON

- Visibility: same memory visibility rules for both locks
- Features of explicit locks:
  - Flexibility in ordering of lock/unlock operations
  - More possibilities for different locks (non-reentrant, non-exclusive)
  - Possibility to use fair/unfair locks
  - … but no guarantee that lock is always released (in particular in case of exceptions); so use unlock in finally clause
- Performance:
  - In Java 5, explicit locks had better performance
    - which led to better implementation of `synchronized`
  - Performance advantage of explicit locks is diminished now

TYPICAL USAGES OF EXPLICIT LOCKS

- Synchronisation over code block that is not within a single method
- Timed or interruptible attempts to acquire a lock
- Lock coupling

See exercises

Explicit locking:
- More flexilible usage
- More ways to make mistakes

Only use them when necessary!

EXPLICIT LOCK WITH TIMEOUT

EXAMPLE

```
public class TimedLocking {
  private Lock lock = new ReentrantLock();

  public boolean trySendOnSharedLine(String message,
                     long timeout, TimeUnit unit)
                     throws InterruptedException {

    long nanosToLock = unit.toNanos(timeout) - estimated(message);
    if (!lock.tryLock(nanosToLock, NANOSECONDS))
      return false;
    try {
      return sendOnSharedLine(message);
    } finally {
      lock.unlock();
    }
} }
```

Sending a message over a channel might fail. In that case, you do not want to block on it forever

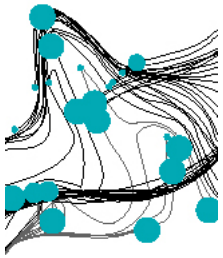
READ-WRITE LOCK

- Coupled locks: read lock (non-exclusive) and a write lock (exclusive)
- Either one or multiple threads can hold the read lock, or one thread can hold the write lock
- Read and write lock never held simultaneously
- Useful for read-mostly data structures
- Can improve performance (but not always - examine usage carefully)

```
public interface ReadWriteLock {
  Lock readLock();
  Lock writeLock();
}
```

EXAMPLE: READ WRITE MAP

```
public class ReadWriteMap <K,V> {
    private final Map<K, V> map;
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock r = lock.readLock();
    private final Lock w = lock.writeLock();
    public V put(K key, V value) {
        w.lock();
        try {return map.put(key, value);} finally { w.unlock(); } }
    public V get(Object key) {
        r.lock();
        try { return map.get(key); } finally { r.unlock(); } }
    }
```



CONDITION VARIABLES

BACK TO THE BOUNDED QUEUE

```
public void enq(T x) throws FullExc {
  lock.lock();
  try {
    if (tail - head == items.length)
      throw new FullExc();

    items[tail % items.length]= x
    tail++;

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

Exceptions should be handled at the client side

```
public T deq() throws EmptyExc {
  lock.lock();
  try {
    if (tail == head)
      throw new EmptyExc();

    T x = items[head % items.length];
    head++;
    return x;

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

Can we get rid of the exceptions?

AVOIDING THE EXCEPTIONS

```
public void enq(T x) throws FullExc {
  lock.lock();
  try {
    while (tail - head == items.length)
      {}

    items[tail % items.length]= x
    tail++;

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

Wait until we can continue

```
public T deq() throws EmptyExc {
  lock.lock();
  try {
    while (tail == head) {}

    T x = items[head % items.length];
    head++;
    return x;

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

Problem: deadlock!
Spinning while lock is held!

CONDITION VARIABLES

```
final Condition notFull
    = lock.newCondition();
final Condition notEmpty
    = lock.newCondition();
```

```
public void enq(T x) throws FullExc {
  lock.lock();
  try {
    while (tail - head == items.length)
      {notFull.await();}

    items[tail % items.length]= x
    tail++;
    notEmpty.signal();

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

Wait until we can continue

```
public T deq() throws EmptyExc {
  lock.lock();
  try {
    while (tail == head) {
        notEmpty.await();}

    T x = items[head % items.length];
    head++;
    notFull.signal();
    return x;

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

Condition false: wait
Condition true: woken up

CONDITION INTERFACE

```
interface Lock {
   ...
   Condition newCondition();
}
```

```
interface Condition {
  void     await();
  boolean  await(long time, TimeUnit unit);
  boolean  awaitUntil(Date deadline);
  long     awaitNanos(long nanosTimeout);

  void     awaitUninterruptibly();

  void     signal();
  void     signalAll();
}
```

might throw an `InterruptedException`

wakes up one waiting thread

wakes up all waiting threads

BOUNDED QUEUE

WITH WAIT/NOTIFY

Instead of an explicit lock we use Java's monitor pattern to synchronize on the `Queue`

```
public synchronized
            void enq(T x) {

  while (tail - head ==
                items.length {
    try { wait(); }
    catch { ... }
  }

  items[tail % items.length]
    = x;
  tail++;
  notify();
}
```

```
public synchronized T deq() {


  while (tail == head) {
    try { wait(); }
    catch { ... }
  }

  T x =
    items[head % items.length];
  head++;
  notify();
  return x;
}
```

WAIT/NOTIFY VS CONDITIONS

```
class Object {
  void wait();
  void notify();
  void notifyAll();
}
```

- synchronization on the implicit lock
  - wait/notify
  - single waiting room

```
interface Condition {
  void await();
  void signal();
  void signalAll();
}
```

- `Condition` interface
  - associated to a `lock`
  - multiple conditions possible
  - slightly more variety in wait-methods

Beware: a Condition object also has **wait** and **notify(All)** methods

In both cases, you should have the (implicit or explicit) lock, to call these methods.

OPTIMISATION?

```
final Condition notFull
    = lock.newCondition();
final Condition notEmpty
    = lock.newCondition();
```

```
public void enq(T x) throws FullExc {
  lock.lock();
  try {
    while (tail - head == items.length)
      {notFull.await();}

    items[tail % items.length]= x
    tail++;
    notEmpty.signal();

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

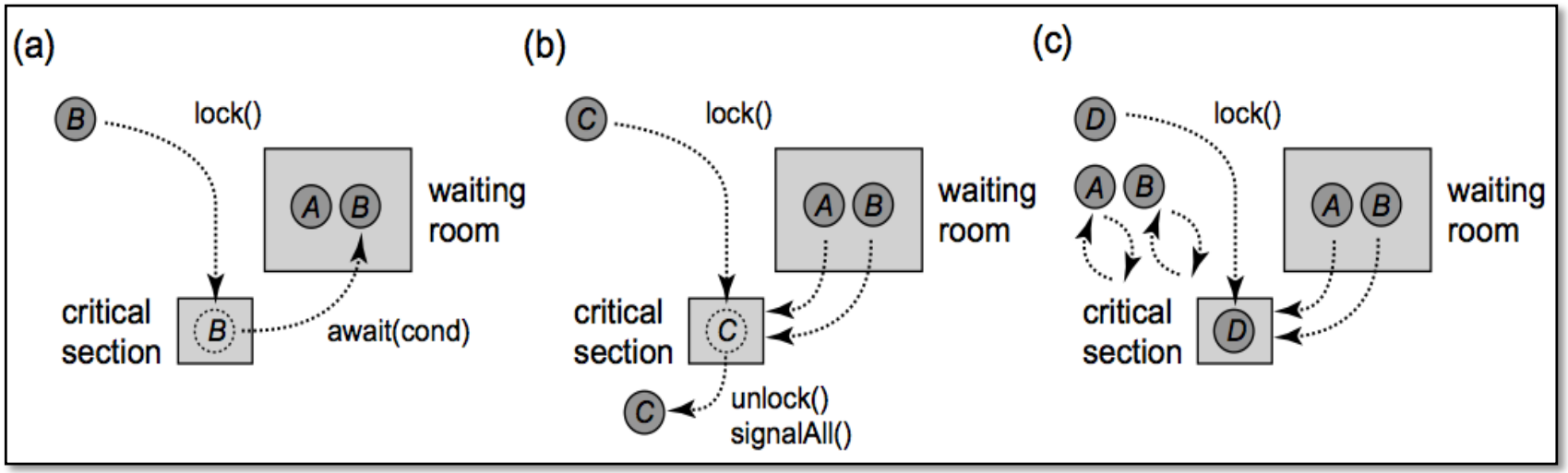
Can we replace while by if?

```
public T deq() throws EmptyExc {
  lock.lock();
  try {
    while (tail == head) {
        notEmpty.await();}

    T x = items[head % items.length];
    head++;
    notFull.signal();
    return x;

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

No, by time thread can continue, Condition might no longer hold

due to 'unfair' policy!

TYPICAL SCENARIO



a) A has lock and calls `await`, releases lock and goes to waiting room; B does the same.

b) C leaves critical section, calls `signalAll`, A & B both attempt to reacquire lock.

c) D wins lock over A & B, hence A & B 'spin' until D leaves critical section.

OPTIMISATION?

Can we reduce signals, only signal when it is useful?

```
if (tail - head == 1) {
    notEmpty.signal();
}
```

```
public void enq(T x) throws FullExc {
  lock.lock();
  try {
    while (tail - head == items.length)
      {notFull.await();}

    items[tail % items.length]= x
    tail++;
    notEmpty.signal();

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

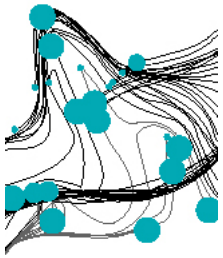
```
public T deq() throws EmptyExc {
  lock.lock();
  try {
    while (tail == head) {
          notEmpty.await();}

    T x = items[head % items.length];
    head++;
    notFull.signal();
    return x;

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

No, chance of missed signal

MISSED SIGNAL SCENARIO

- A & B waiting in dequeue
- C enqueuing
- A gets signalled
- D enqueus before A takes lock
- B not signalled by D because queue was not empty

- SignalAll will fix this!



OTHER SYNCHRONIZERS

Synchronizers we have seen so far

- Locks (intrinsic/explicit)
- Thread creation
- Thread termination

But there are many more

- Counting semaphore
- Exchanger
- Future
- Latch
- Barrier
- Phaser

COUNTING SEMAPHORE

- Provides at most *n* permits to access some resource
- Acquire blocks if *n* permits have been given out
- Binary semaphore: mutual exclusion
- Thread can also release a permit that it did not acquire
- Typical usage: resource pool

- Semaphore can implement synchronized + wait/notify

EXCHANGER

- Synchronous channel to swap two resources
- Common use case: buffer exchange
  - Thread 1 puts new data in buffer, retrieves result
  - Thread 2 processes data and puts result back in buffer

- Symmetric protocol, needs to consider failure
- Generalisations to more parties and arbitrary functions possible

BUFFER EXCHANGE

```
void main() {
    new Thread(new FillingLoop()).start();
    new Thread(new EmptyingLoop()).start();
}
```

```
class FillAndEmpty {
  Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>();
  DataBuffer initialEmptyBuffer = ... a made-up type
  DataBuffer initialFullBuffer = ...

  class FillingLoop implements Runnable {
    public void run() {
      DataBuffer currentBuffer = initialEmptyBuffer;
      try {
        while (currentBuffer != null) {
          addToBuffer(currentBuffer);
          if (currentBuffer.isFull())
            currentBuffer = exchanger.exchange(currentBuffer);
        }} catch (InterruptedException ex) { ... handle ... }}}
```

EmptyingLoop symmetric

FUTURE

- Handle to result that is computed by separate thread (asynchronous computation)
- Lookup blocks if result is not available yet

Pattern

```
public run() {
    //…
    result = …
    }
```

```
t.start();
//…
t.join();
Read t.result
```

- Common pattern captured by Callable

```
interface Callable<V> {
    V call();
}
```

Similar to Runnable, but non-void method
Uses executor framework

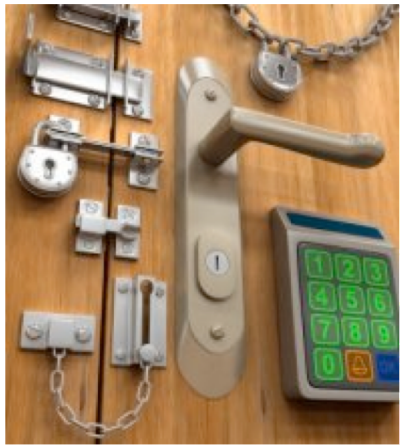
FUTURE EXAMPLE

```
ExecutorService executor = ...
Future<String> future =
    executor.submit(new Callable<String>() {
        public String call() {
            return searcher.search(target);
        }
    });
    displayOtherThings(); // do other things while searching
    try { displayText(future.get()); // use future
        }
    catch (ExecutionException ex) {
        cleanup(); return;
    }
```

LATCHING VARIABLE

- One-time synchronization, no reset possible
- Single release permits all previous (and future) acquires to proceed

- Variation: count-down latch:
  fixed number of releases necessary



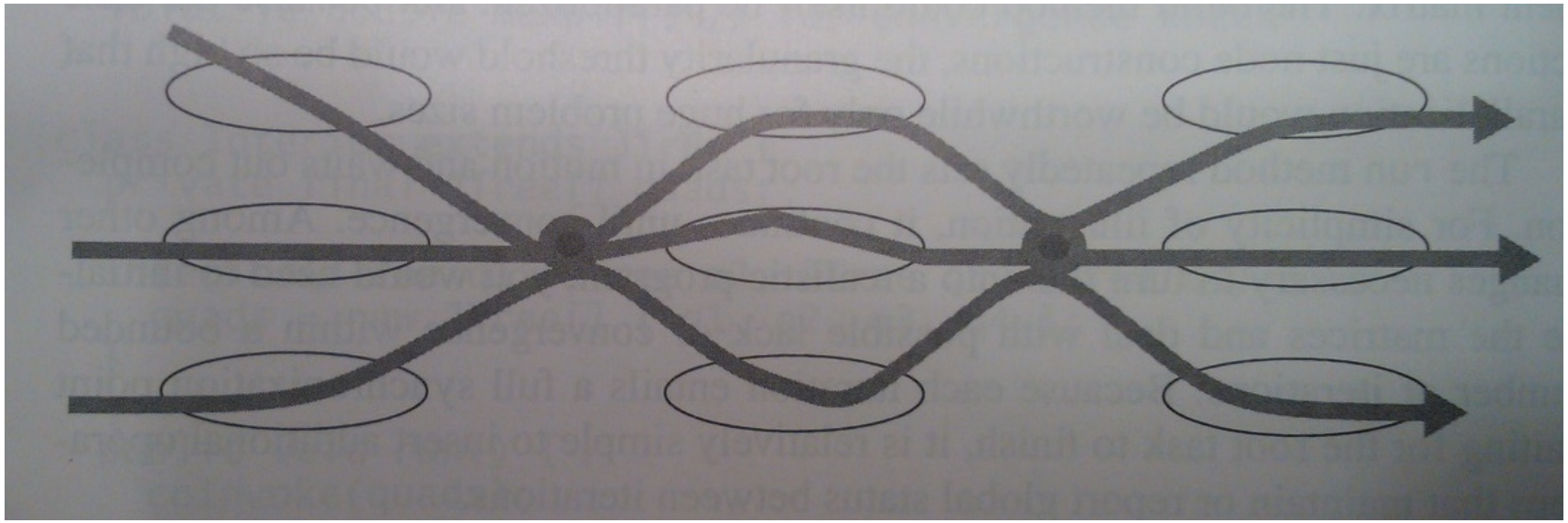
- Typical use cases:
  - Initialisation
  - Completion
  - Event indication
  - Error indication to proceed with global shutdown tasks

LATCH FOR INITIALISATION

```
class Player implements Runnable {
   protected final Latch startSignal;
   Player(Latch l) { startSignal = l; }

   public void run() {
      try { startSignal.acquire(); play();}
      catch(InterruptedException ie) {..}

   void play() {}
}
```

```
class Game {
   void begin(int nplayers) {
   Latch startSignal = new Latch();

   for (int i = 0; i < nplayers; ++i) {
     new Thread(
        new Player(startSignal)).start();
   }
   startSignal.release();
  }
}
```

BARRIERS



Threads wait at barrier for other threads to reach this point

GENERAL BARRIER USAGE PATTERN

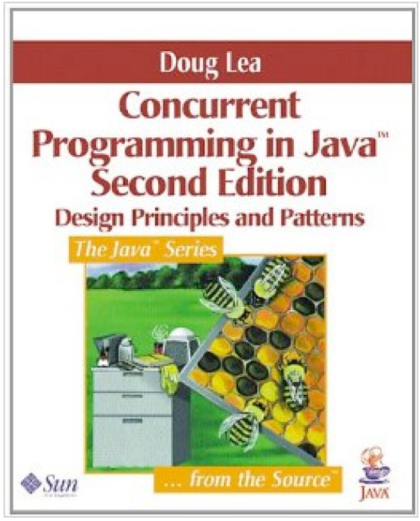
```
class Segment implements Runnable  {
  final CyclicBarrier bar; // shared by all segments
  Segment(CyclicBarrier b) { bar = b; }

  void update() {  }

  public void run() {
    // ...
    try {
      for (int i = 0; i < iterations; ++i) {
        update();
        bar.awaitBarrier();
      }}}
```

PHASER

- Similar to CyclicBarrier and CountDownLatch but more flexible usage
- Number of parties registered to synchronize on a phaser may vary over time
- Explicit phase number
- Tree-structure of phasers may be used to reduce contention
- Can be terminated
- Possiblity to monitor the state of the phaser

SYNCHRONISED COLLECTIONS

Doug Lea



SYNCHRONIZED COLLECTIONS

SYNCHRONIZE EVERY PUBLIC METHOD

- serialise all accesses (i.e. methods are synchronized)
- examples:
  - `Vector, Hashtable`
  - synchronized wrapper classes created by `Collections.synchronizedXXX` factory methods
- Beware: only individual actions are synchronized, not compound actions.

```
Vector v = ...
if (prop(v.getLast()) {
  v.removeLast();
}
```

This might remove an element that does not satisfy `prop`.

But a sychronized collection is always protected by this, thus sequences can be easily protected by an extra synchronized block

CONCURRENT COLLECTIONS

- Maps and Lists especially designed for concurrent use
- Much better scalability, while thread safe

- ConcurrentHashMap (includes put-if-absent operations)
- CopyOnWriteArrayList
- LinkedBlockingQueue

CONCURRENT HASH MAP

- Same functional specification as Hashtable
- All operations thread-safe
- No support for locking the entire table
- Full concurrency of retrievals (no locking)
- Restricted number of parallel writers
- Lookup and update may overlap
- Retrievals reflect the results of the most recently completed update operations holding upon their onset
- Iterators, enumerations and aggregate operations such as putAll and clear may reflect the state of the hash table at some point
- Semantics of size, isEmpty weakened

COPY-ON-WRITE ARRAYLIST

- Mutative operations (add, set, and so on) make a fresh copy of the underlying array
- May be more efficient when traversal operations vastly outnumber mutations
- No need to synchronize traversals, while concurrent threads cannot interfere
- Iterator uses a reference to the state of the array at the point that the iterator was created, which is unchanged during the lifetime of the iterator
- No ConcurrentModificationException.
- Iterator does not reflect updates to the lists

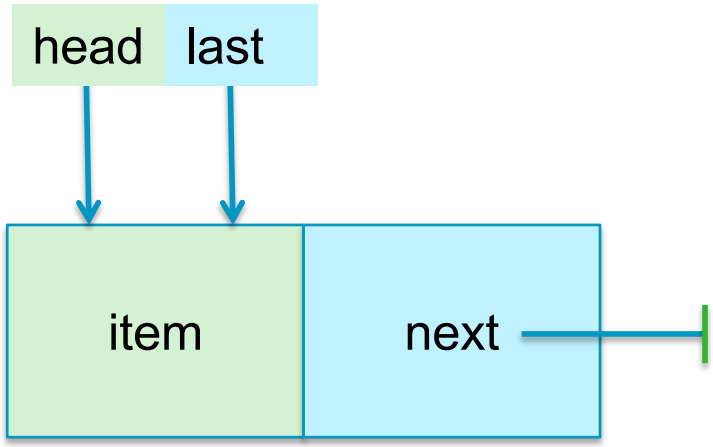
LINKED BLOCKING QUEUE

- A variant of the two lock queue algorithm
- putLock protects put operations
- takeLock protects take operations
- count field volatile
- Optional bound
- Operations like remove (in the middle of the queue) and iteration require both locks

```
private void fullyLock() {
    takeLock.lock();
    putLock.unlock()
}
```

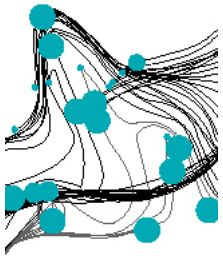
PUT AND TAKE OPERATIONS

```
putLock.lock();
last = last.next = new Node<E>(x);
c = count.getAndIncrement();
putLock.unlock();
```




```
takeLock.lock();
Node<E> first = head.next;
head = first;
E x = first.item;
first.item = null;
c = count.getAndDecrement();
takeLock.unlock();
return x;
```

last, last.next guardedby putLock
head,head.item guardedby takeLock



IMPORTANT POINTS

- Thread safety
  - shared data is accessed in a manner that guarantees safe execution by multiple threads at the same time
  - synchronization important means to achieve thread safety
- Synchronisation policy
  - what data protected by what lock
  - should be documented
- Intrinsic vs. explicit locking
- Condition variables: communication and synchronisation
- Many other forms of synchronisation
- Synchronised collections