Lazy node removal

```java
class ValidatedNode<T> extends ReadWriteNode<T>
{
   private volatile boolean valid;

   boolean valid()    { return valid; }  // is node valid?
   void setValid()    { valid = true; }  // mark valid
   void setInvalid() { valid = false; } // mark invalid
}
```

```java
public class LazySet<T> extends OptimisticSet<T>
{
  public LazySet() {
    head = new ValidatedNode<>(Integer.MIN_VALUE);  // smallest key
    tail = new ValidatedNode<>(Integer.MAX_VALUE);  // largest key
    head.setNext(tail);
  }
```

```java
// is pred reachable from head, and does it point to curr?
protected boolean valid(Node<T> pred, Node<T> curr) {
  return pred.valid() && curr.valid() && pred.next() == curr;
}
```

```java
public boolean has(T item) {
    // find position without locking
  Node<T> pred, curr = find(head, item.key());
    // check validity and item without locking
  return curr.valid() && curr.key() == item.key();
}
```

Add works same as in OptimisticSet, but using overridden version of valid – which works in constant time.

```java
public boolean remove(T item) {
  do { Node<T> pred, curr = find(head, item.key()); // no locking
       pred.lock(); curr.lock();              // now lock position
       try { // if position still valid, while locking:
         if (valid(pred, curr)) {
           if (curr.key() != item.key())
             return false;       // item not in the set
           else { // item in the set at curr: remove it
             curr.setInvalid();          // logical removal
             pred.setNext(curr.next()); // physical removal
             return true;
           }
         }
       } finally { pred.unlock(); curr.unlock(); }// done: unlock
  } while (true);                          // if not valid: try again!
}
```

Lock-free access

```java
class AtomicMarkableReference<V> {
  V, boolean get();            // current reference and mark
   // if reference == expectRef set mark to newMark and return true
   // otherwise do not change anything and return false
  boolean attemptMark(V expectRef, boolean newMark);
   // if reference == expectRef and mark == expectMark,
   // set reference to newRef, mark to newMark and return true;
   // otherwise, do not change anything and return false
  boolean compareAndSet(V expectRef, V newRef,
                        boolean expectMark, boolean newMark);
}
```

```java
class LockFreeNode<T> extends SequentialNode<T> {

    // reference to next node and validity mark of current node
   private AtomicMarkableReference<Node<T>> nextValid;

    // return next and valid as a pair
   Node<T>, boolean nextValid() { return nextValid.get(); }

   Node<T> next()
    { Node<T> next, boolean valid = nextValid(); return next; }
   boolean valid()
    { Node<T> next, boolean valid = nextValid(); return valid; }
```

```java
 // try to set invalid; return true if successful
boolean setInvalid()
 { Node<T> next = next();
   return nextValid.compareAndSet(next, next, true, false); }

 // try to update to newNext if valid; return true if successful
boolean setNextIfValid(Node<T> expectNext, Node<T> newNext)
 { return nextValid.compareAndSet(expectNext, newNext, true, true); }
```

update next only if the node is valid

```java
public class LockFreeSet<T> extends SequentialSet<T>
{
  public LockFreeSet() {
    head = new LockFreeNode<>(Integer.MIN_VALUE);  // smallest key
    tail = new LockFreeNode<>(Integer.MAX_VALUE);  // largest key
    head.setNext(tail);  // unconditionally set next
                         // only in new nodes
  }
```

```java
public boolean remove(T item) {
  do { Node<T> pred, curr = find(head, item.key()); // not in set
       if (curr.key() != item.key() || !curr.valid()) return false;
        // try to invalidate; try again if node is being modified
       if (!curr.setInvalid()) continue;
        // try once to physically remove curr
       pred.setNextIfValid(curr, curr.next());
       return true;
  } while (true);  // changed during logical removal: try again!
}
```

```java
public boolean add(T item) {
  do { Node<T> pred, curr = find(head, item.key()); // already in set
       if (curr.key() == item.key() && curr.valid()) return false;
       // new node, pointing to curr
       Node<T> node = new LockFreeNode<>(item).setNext(curr);
       // if pred valid and points to curr, make it point to node
       if (pred.setNextIfValid(curr, node)) return true;
  } while (true);  // pred changed during add: try again!
}
```

```java
public boolean has(T item) {
   // find position (use plain search in SequentialSet)
  Node<T> pred, curr = super.find(head, item.key());
   // check validity and item
  return curr.valid() && curr.key() == item.key();
}
```

```java
protected Node<T>, Node<T> find(Node<T> start, int key) {
  boolean valid;              // is curr valid?
  Node<T> pred, curr, succ;   // consecutive nodes in iteration
  retry: do {
    pred = start; curr = start.next();  // from start node
    do {  // succ is curr's successor; valid is curr's validity
      succ, valid = curr.nextValid();
      while (!valid) { // while curr is not valid, try to remove it
          // if pred is modified while trying to redirect it, retry
        if (!pred.setNextIfValid(curr, succ)) continue retry;
          // curr has been physically removed: move to next node
        curr = succ; succ, valid = curr.nextValid();
      } // now curr is valid (and so is pred)
      if (curr.key() >= key) return (pred, curr);
      pred = curr; curr = succ;   // continue search
    } while (true);
  } while (true);
```

Parallel linked queues

```
class AtomicReference<V> {

  V get();                    // current reference
  void set(V newRef);         // set reference to newRef

  // if reference == expectRef, set to newRef and return true
  // otherwise, do not change reference and return false
  boolean compareAndSet(V expectRef, V newRef);
}
```

```
class QNode<T>
{ // value of node
  T value;
  // next node in chain
  AtomicReference<QNode<T>> next;
  QNode(T value)
  { this.value = value;
    next = new AtomicReference<>(null); }
}
```

```
class LockFreeQueue<T> implements Queue<T>
{
  // access to front and back of queue
  protected AtomicReference<QNode<T>> head, tail;

  // empty queue
  public LockFreeQueue() {
       // value of sentinel does not matter
    QNode<T> sentinel = new QNode<>();
    head = new AtomicReference<>(sentinel);
    tail = new AtomicReference<>(sentinel);
  }
```

The method dequeue removes the node at the head of a queue – where the sentinel points. Unlike enqueue, dequeueing only requires one update to the linked structure:

1. update head: make `head` point the node previously pointed to by the sentinel; the same node becomes the new sentinel and is also returned.

The update is atomic (it uses compare-and-set), but other threads may be updating the head concurrently:

- repeat update head until success,
- if you detect a “half finished” enqueue operation – with the tail pointing to the sentinel about to be removed – help by moving the tail forward.

```
public T dequeue() throws EmptyException {
 while (true) // nodes at front, back of queue
 { QNode<T> sentinel = head.get(), last = tail.get(),
            first = sentinel.next.get();
   if (sentinel == head.get()) // if head points to sentinel
   { // if tail also points to sentinel
     if (sentinel == last)
     { // empty queue: raise exception
       if (first == null)
         throw new EmptyException();
       // non-empty: update tail, repeat
       tail.compareAndSet(last, first); }
     else // tail doesn't point to sentinel
     { T value = first.value;
       // make head point to first (new sentinel); retry until success
       if (head.compareAndSet(sentinel, first)) return value; } } }
}
```

The method enqueue adds a new node to the back of a queue – where `tail` points. It requires two updates that modify the linked structure:

1. update last: make the last node in the queue point to the new node,
2. update tail: make `tail` point to the new node.

Each update is individually atomic (it uses compare-and-set), but another thread may interfere between the two updates:

- repeat update last until success;
- try update tail once;
- the implementation should be able to deal with a “half finished” enqueue operation (tail not updated yet), and finish the job – this technique is called helping.

```
public void enqueue(T value) {
 // new node to be enqueued
 QNode<T> node = new QNode<>(value);
 while (true) // nodes at back of queue
 { QNode<T> last = tail.get();
   QNode<T> nextToLast = last.next.get();
   // if tail points to last
   if (last == tail.get())
   { // and if last really has no successor
     if (nextToLast == null) {
       // make last point to new node
       if (last.next.compareAndSet(nextToLast, node))
       // if last.next updated, try once to update tail
       { tail.compareAndSet(last, node); return; }
     } else // last has valid successor: try to update tail and repeat
       { tail.compareAndSet(last, nextToLast); } } }
}
```

SequentialSet

```java
class SequentialNode<T> implements Node<T> {
   private T item;        // value stored in node
   private int key;       // hash code of item
   private Node<T> next;  // next node in chain
     // getters
   T item()       { return item; }
   int key()      { return key; }
   Node<T> next() { return next; }
     // setters
   void setItem(T item)        { this.item = item; }
   void setKey(int key)        { this.key = key; }
   void setNext(Node<T> next) { this.next = next; }
}
```

```java
// first position from 'start' whose key is no smaller than 'key'
protected Node<T>, Node<T> find(Node<T> start, int key) {
  Node<T> pred, curr; // predecessor and current node in iteratior
  curr = start;       // from start node
  do {
    pred = curr; curr = curr.next();    // move to next node
  } while (curr.key() < key);          // until curr.key >= key
  return (pred, curr);                  // return position
}
```

pseudo-code for: `new Position<T>(pred, curr)`

```java
// is 'item' in set?
public boolean has(T item) {
  int key = item.key();              // item's key
   // find position of key from head
  Node<T> pred, curr = find(head, key);
   // curr.key() >= key
  return curr.key() == key;          // item can only appear here!
}
public boolean add(T item) {
  Node<T> node = new Node<>(item);              // new node
  Node<T> pred, curr = find(head, item.key()); // curr.key >= item.key(
  if (curr.key() == item.key()) return false;  // item already in set
  else // item not already in set: add node between pred and curr
   { node.setNext(curr); pred.setNext(node); return true; }
}
public boolean remove(T item) {
  Node<T> pred, curr = find(head, item.key());
  // curr.key() >= item.key()
  if (curr.key() > item.key()) return false;  // item not in set
  else                          // item in set: remove node curr
   { pred.setNext(curr.next()); return true; }
}
```

Fine-grained locking: Lock individual nodes

```java
public class FineSet<T> extends SequentialSet<T>
{
  // empty set
  public FineSet() {
    head = new LockableNode<>(Integer.MIN_VALUE);  // smallest key
    tail = new LockableNode<>(Integer.MAX_VALUE);  // largest key
    head.setNext(tail);
  }
// find while locking pred and curr, return locked position
protected Node<T>, Node<T> find(Node<T> start, int key) {
  Node<T> pred, curr; // predecessor and current node in iteration
  pred = start; curr = start.next(); // from start node
  pred.lock(); curr.lock();          // lock pred and curr nodes
  while (curr.key < key) {
    pred.unlock();                   // unlock pred node
    pred = curr; curr = curr.next(); // move to next node
    curr.lock();                     // lock next node
  } // until curr.key >= key
  return (pred, curr);  // return position
}
```

pseudo-code for: `new Position<T>(pred, curr)`

```java
public boolean add(T item) {
  Node<T> node = new LockableNode<>(item);  // new node
  try { // find with hand-over-hand locking
      // the first position such that curr.key() >= item.key()
    Node<T> pred, curr = find(head, item.key()); // locking
    ... // add node as in SequentialSet, while locking
  } finally { pred.unlock(); curr.unlock(); } // done: unlocking
}

public boolean remove(T item) {
  try { // find with hand-over-hand locking
      // the first position such that curr.key >= item.key
    Node<T> pred, curr = find(head, item.key()); // locking
    ... // remove node as in SequentialSet, while locking
  } finally { pred.unlock(); curr.unlock(); } // done: unlocking
}
```

Coarse-Grained locking: At most one thread at a time operating on structure.

```java
class CoarseSet<T> extends SequentialSet<T>
{
  // lock controlling access to the whole set
  private Lock lock = new ReentrantLock();

  // overriding of add, remove, and has
```

Every method `add`, `remove`, and `has` simply works as follow

1. acquires the lock on the set
2. performs the operation as in `SequentialSet`
3. releases the lock on the set

```java
public boolean add(T item) {
  lock.lock();                 // lock whole set
  try {
    return super.add(item);   // execute 'add' while locking
  } finally {
    lock.unlock();             // done: release lock
  }
}
public boolean remove(T item) {
  lock.lock();                   // lock whole set
  try {
    return super.remove(item);  // execute 'remove' while locki
  } finally {
    lock.unlock();               // done: release lock
  }
}
public boolean has(T item) {
  lock.lock();              //lock whole set
  try {
    return super.has(item); // execute 'has' while locking
  } finally {
    lock.unlock();          // done: release lock
  }
```

Optimistic locking

```java
public class OptimisticSet<T> extends SequentialSet<T>
{
  public FineSet()
  { head = new ReadWriteNode<>(Integer.MIN_VALUE);  // smallest key
    tail = new ReadWriteNode<>(Integer.MAX_VALUE);  // largest key
    head.setNext(tail); }

  // is (pred, curr) a valid position?
  protected boolean valid(Node<T> pred, Node<T> curr) // ...
public boolean add(T item) {
  Node<T> node = new ReadWriteNode<>(item);          // new node
  do { Node<T> pred, curr = find(head, item.key()); // no locking
       pred.lock(); curr.lock();                    // now lock position
       try { // if position still valid, while locked:
         if (valid(pred, curr)) { ... }    // physically add node
       } finally { pred.unlock(); curr.unlock(); }// done: unlock
  } while (true);                        // if not valid: try again!
}
public boolean remove(T item) {
  do { Node<T> pred, curr = find(head, item.key()); // no locking
       pred.lock(); curr.lock();                    // now lock position
       try { // if position still valid, while locked:
         if (valid(pred, curr)) { ... } // physically remove node
       } finally { pred.unlock(); curr.unlock(); }// done: unlock
  } while (true);                        // if not valid: try again!
}
public boolean has(T item) {
  do { Node<T> pred, curr = find(head, item.key()); // no locking
       pred.lock(); curr.lock();                    // now lock position
       try { // if position still valid, check key while locked
         if (valid(pred, curr)) return curr.key() == item.key();
       } finally { pred.unlock(); curr.unlock(); }// done: unlock
  } while (true);                        // if not valid: try again!
}
// is pred reachable from head, and does it point to curr?
protected boolean valid(Node<T> pred, Node<T> curr) {
  Node<T> node = head;  // start from head
  while (node.key() <= pred.key()) { // does pred point to curr?
    if (node == pred) return pred.next() == curr;
    node = node.next(); // continue to the next node
  } // until node.pred > pred.key
                                                        curr
  return false; // pred could not be reached or does not point tc
```

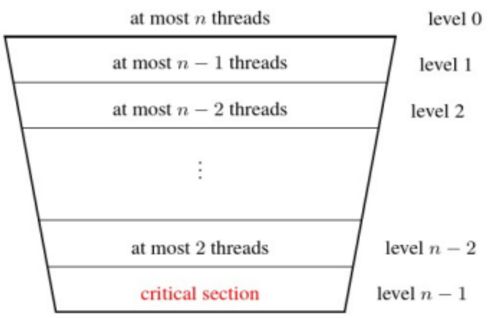
```java
public boolean has(T item) {
  try { // find with hand-over-hand locking
        // the first position such that curr.key() >= item.key()
    Node<T> pred, curr = find(head, item.key()); // locking
    ... // check node as in SequentialSet, while locking
  } finally { pred.unlock(); curr.unlock(); } // done: unlocking
}
```

Peterson's algorithm for N threads

```
int[] enter = new int[n]; // n elements, initially all 0s
  int[] yield = new int[n]; // use n - 1 elements 1..n-1
```

thread x

```
while (true) {
  // entry protocol
  for (int i = 1; i < n; i++) {
    enter[x] = i;   // want to enter level i
    yield[i] = x;   // but yield first
    await (∀ t != x: enter[t] < i     wait until all other
            || yield[i] != x);         threads are in lower levels
  }
  critical section { ... }            or another thread
  // exit protocol                     is yielding
  enter[x] = 0;  // go back to level 0
```



Every thread goes through $n-1$ levels to enter the critical section:

- when a thread is at level 0 it is outside the entry region;
- when a thread is at level $n-1$ it is in the critical section;
- Thread `t` is in level `i` when it has finished the loop at line 6 with `enter[t]=i`;
- `yield[l]` indicates the *last* thread that wants to enter level `l` last;
- to enter the next level, wait until there are no processes in higher levels, or another process (which entered the current level last) is yielding;
- mutual exclusion: at most $n-\ell$ processes are in level ℓ, thus at most $n-(n-1)=1$ processes in critical section.

Barriers with n threads (single use)

```
int nDone = 0; // number of done threads
Lock lock = new Lock(); // mutual exclusion for nDone
Semaphore open = new Semaphore(0); // 1 iff barrier is open
```

thread t_k

```
// code before barrier                       Total number of
lock.lock();              // lock nDone       expected threads
nDone = nDone + 1;        // I'm done
if (nDone == n) open.up(); // I'm the last: we can go!   Can we switch
lock.unlock();            // unlock nDone                these? Yes,
open.down();              // proceed when possible       in this
open.up();                // let the next one go         single use.
// code after barrier
```

```
public class SemaphoreBarrier implements Barrier {
  int nDone = 0; // number of done threads
  Semaphore gate1 = new Semaphore(0);// first gate
  Semaphore gate2 = new Semaphore(1);// second gate
  final int n;

  // initialize barrier for `n` threads
  SemaphoreBarrier(int n) {
    this.n = n;
  }

  // number of threads expected at the barrier
  int expect() {
    return n;
  }

  public void wait() { approach(); leave(); }

void approach() {
  synchronized (this) {
    nDone += 1;          // arrived
    if (nDone == n) {    // if last in:
      gate1.up();        // open gate1
      gate2.down();      // close gate2
    }
  }
  gate1.down(); // pass gate1
  gate1.up();   // let next pass
}
void leave() {
  synchronized (this) {
    nDone -= 1;          // going out
    if (nDone == 0) {    // if last out:
      gate2.up();        // open gate2
      gate1.down();      // close gate1
    }
  }
  gate2.down(); // pass gate2
  gate2.up();   // let next pass
}
```

Reusable barriers: First attempt

```
public class NonBarrier1 implements Barrier {
  int nDone = 0; // number of done threads
  Semaphore open = new Semaphore(0);
  final int n;

  // initialize barrier for `n` threads
  NonBarrier1(int n) {
    this.n = n;
  }

  // number of threads expected at the barrier
  int expect() {
    return n;
  }

  public void wait() {
    synchronized(this) {
      nDone += 1;         // I'm done
    }
    if (nDone == n)
      open.up();          // I'm the last arrived: All can go!

    open.down()           // proceed when possible
    open.up()             // let the next one go
    synchronized(this) {
      nDone -= 1;         // I've gone through
    }
    if (nDone == 0)
      open.down();        // I'm the last through: Close barrier!
  }
}
```

More than one thread may open the barrier (the first `open.up()`): this was not a problem in the non-reusable version, but now some threads may be executing `wait` again before the barrier is closed again!

What if n threads block here until nDone == n?

What if n threads block here until nDone == 0?

More than one thread may try to close the barrier (last `open.down()`): Deadlock!


```
public class SemaphoreBarrier implements Barrier {
  int nDone = 0; // number of done threads
  Semaphore gate1 = new Semaphore(0);// first gate
  Semaphore gate2 = new Semaphore(1);// second gate
  final int n;

  // initialize barrier for `n` threads
  SemaphoreBarrier(int n) {
    this.n = n;
  }

  // number of threads expected at the barrier
  int expect() {
    return n;
  }

  public void wait() { approach(); leave(); }

void approach() {
  synchronized (this) {
    nDone += 1;          // arrived
    if (nDone == n) {    // if last in:
      gate1.up();        // open gate1
      gate2.down();      // close gate2
    }
  }
  gate1.down(); // pass gate1
  gate1.up();   // let next pass
}
void leave() {
  synchronized (this) {
    nDone -= 1;          // going out
    if (nDone == 0) {    // if last out:
      gate2.up();        // open gate2
      gate1.down();      // close gate1
    }
  }
  gate2.down(); // pass gate2
  gate2.up();   // let next pass
}
```


Counter with mutual exclusion

```
public class LockedCounter extends CCounter
{
    @Override
    public void run() {           Entry
      lock.lock();                protocol
      try {
        // int cnt = counter;
        // counter = counter + 1;      Critical
        super.run();                   section
      }
      finally {                   Exit
        lock.unlock();            protocol
      }
    }
    // shared by all threads working
    // on this object
    private Lock lock = new ReentrantLock();
}
```

The philosophers

```
Table table; // table shared by all philosop
```

philosopher$_k$

```
while (true) {
  think();            // think
  table.getForks(k); // wait for forks
  eat();              // eat
  table.putForks(k); // release forks
}
```

```
// in classes implementing Table:

// fork to the left of philosopher k
public int left(int k) {
  return k;
}

// fork to the right of philosopher k
public int right(int k) {
  // N is the number of philosophers
  return (k + 1) % N;
}
```

```
public class AsymetricTable implements Table {
  Lock[] forks = new Lock[N];

public void getForks(int k) {
      if (k == N) { // right before left
            forks[right(k)].lock();
            forks[left(k)].lock();
      } else {     // left before right
            forks[left(k)].lock();
            forks[right(k)].lock();
      }
}
```

Readers-writes board: Synched but starvation

```
public class SyncBoard<T> implements Board<T> {
  int nReaders = 0; // # readers on board
  Lock lock = new Lock(); // for exclusive access to nReaders
  Semaphore empty = new Semaphore(1); // 1 iff no active threads
  T message; // current message

 public T read() {
   lock.lock();          // lock to update nReaders
   if (nReaders == 0)    // if first reader,
      empty.down();      //            set not empty
   nReaders += 1;        // update active readers
   lock.unlock();        // release lock to nReaders

   T msg = message;      // read (critical section)

   lock.lock();          // lock to update nReaders
   nReaders -= 1;        // update active readers
   if (nReaders == 0)    // if last reader
      empty.up();        //            set empty
   lock.unlock();        // release lock to nReaders
   return msg;
 }

 public void write(T msg) {
   // get exclusive access
   empty.down();
   message = msg; // write (cs)
   // release board
   empty.up();
 }
```

`invariant` { nReaders == 0 ⟸ empty.count() == 1 }

`Count()` becomes 1 after executing `empty.up()` and it happens that `nReaders = 0`

Synched and no starvation

```
public class FairBoard<T> extends SyncBoard<T> {
  // held by the next thread to go
  Semaphore baton = new Semaphore(1, true); // fair binary sem.

  public T read() {
    // wait for my turn
    baton.down();
    // release a waiting thread
    baton.up();
    // read() as in SyncBoard
    return super.read();
  }

  public void write(T msg) {
    // wait for my turn
    baton.down();
    // write() as in SyncBoard
```

Reusable barriers: Correct solution


```
public class SemaphoreBarrier implements Barrier {
  int nDone = 0; // number of done threads
  Semaphore gate1 = new Semaphore(0);// first gate
  Semaphore gate2 = new Semaphore(1);// second gate
  final int n;

  // initialize barrier for `n` threads
  SemaphoreBarrier(int n) {
    this.n = n;
  }

  // number of threads expected at the barrier
  int expect() {
    return n;
  }

  public void wait() { approach(); leave(); }

void approach() {
  synchronized (this) {
    nDone += 1;          // arrived
    if (nDone == n) {    // if last in:
      gate1.up();        // open gate1
      gate2.down();      // close gate2
    }
  }
  gate1.down(); // pass gate1
  gate1.up();   // let next pass
}
void leave() {
  synchronized (this) {
    nDone -= 1;          // going out
    if (nDone == 0) {    // if last out:
      gate2.up();        // open gate2
      gate1.down();      // close gate1
    }
  }
  gate2.down(); // pass gate2
  gate2.up();   // let next pass
}
```


Buffers:

```
interface Buffer<T> {
    // add item to buffer; block if full
    void put(T item);

    // remove item from buffer; block if empty
    T get();

    // number of items in buffer
    int count();
}
```

Unbounded buffer

```
public class UnboundedBuffer<T> implements Buffer<T> {
  Lock lock = new Lock(); // for exclusive access to buffer
  Semaphore nItems = new Semaphore(0); // number of items in buffer
  Collection storage = ...; // any collection (list, set, ...)
  invariant { storage.count() == nItems.count() + at(5,15-17); }
}

public void put(T item) {
  lock.lock(); // lock
  // store item
  storage.add(item);
  nItems.up();    // update nItems
  lock.unlock(); // release
}

public T get() {
  // wait until nItems > 0
  nItems.down();
  lock.lock(); // lock
  // retrieve item
  T item =storage.remove();
  lock.unlock(); // release
  return item;
}
```

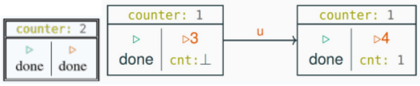
Signals to consumers waiting in **get** that they can proceed

Bounded buffer

Size of buffer

```
public class BoundedBuffer<T> implements Buffer<T> {
 Lock lock = new Lock(); // for exclusive access to buffer
 Semaphore nItems = new Semaphore(0); // # items in buffer
 Semaphore nFree = new Semaphore(N);  // # free slots in buffer
 Collection storage = ...; // any collection (list, set, ...)
 invariant { storage.count() == nItems.count() +
            + at(6,13-15) == N - nFree.count() - at(4-6,16) ; }
```

State/transition diagrams: We capture essential elements of concurrent programs using these. **States** in a diagram capture possible program states. **Transitions** connect states according to execution order.



Reasoning about program properties: The **structural properties** of a diagram capture the semantic properties of a program.

* **Mutual exclusion:** There are no states where two threads are in the cs.
* **Deadlock freedom:** For every non-final state, there is an outgoing transition.
* **Starvation freedom:** There is no (looping) path such that a thread never enters its critical section while trying to do so.
* **No race conditions:** All the final states have the same (correct) result.

Mutual exclusion with strong fairness Bounded waiting (also called bounded bypass). Peterson's algorithm guarantees freedom from starvation, but threads may get access to their CS before "older" threads.

Finite waiting (starvation freedom): When a thread t is waiting to enter its CS, it will eventually enter it.

Bounded waiting: when a thread t is waiting to enter its CS, the maximum number of times other arriving threads are allowed to enter their CS before t, is bounded by a function of the number of contending threads.

r-bounded waiting: -.-, -.- before t, is less than r + 1.

First-come-first-served: 0-bounded waiting.

Lamport's bakery algorithm achieves mutex, deadlock freedom and first-come-first-served fairness and is based on the idea of waiting threads getting a ticket number.

* Because of lack of atomicity, two threads may end up with the same ticket number. In that case, their thread identifier number is used to force an order.
* The tricky part is evaluating multiple variables (ticket #s of all other waiting processes) consistently.
* Idea: A thread raises a flag when computing the number; other threads then wait to compute the numbers.

The main drawback (compared to Peterson's algo, original version of bakery algo) is that the algorithm may use arbitrarily large integers (ticket numbers) in shared variables.

Instruction execution order: When we designed and analysed concurrent algorithms, we implicitly assumed that threads execute instructions in textual program order. This is not guaranteed by the Java language – or, for that matter, by most programming languages – when threads access shared fields. **Compilers** may reorder instructions based on static analysis, which does not know about threads. **Processors** may delay the effect of writes to when the cache is committed to memory. All of this adds to the complications of writing low-level concurrent software correctly.

Volatile fields: Accessing a field (attribute) declared as **volatile** forces synchronization, and thus prevents any optimization from reordering instructions in a way that alters the "happens before" relationship defined by a program's textual order. So, when accessing a shared variable that is accessed concurrently, declare the variable as **volatile** OR guard access to the variable with **locks** (or other synchronization primitives).

General semaphores using binary semaphores

Barz's solution (pseudocode, capacity > 0)

```
BinarySemaphore mutex = 1;  // protects access to count
BinarySemaphore delay = 1;  // blocks threads in down until count >0
int count = capacity;       // value of general semaphore
void up()
{  mutex.down();                 // get exclusive access to count
   count = count + 1;            // increment count
   if (count == 1) delay.up();   // release threads blocking on down
   mutex.up(); }                 // release exclusive access to count
void down()
{ delay.down();                  // block other threads starting down
  mutex.down();                  // get exclusive access to count
  count = count - 1;             // decrement count
  if (count > 0) delay.up();     // release threads blocking on down
  mutex.up(); }                  // release exclusive access to count
```

Mutual exclusion with only atomic reads and writes

Busy waiting: `await(c) ≜ while (!c) {}`.

Three failed attempts:

1) Using Boolean flags. Does not guarantee mutex. Threads can be in CS at the same time, problem is **await** executed *before* enter is set. Both threads can proceed in parallel.

```
boolean[] enter = {false, false};
thread t0                          thread t1
while (true) {                     while (true) {
  // entry protocol                  // entry protocol
  await (!enter[1]);                 await (!enter[0]);
  enter[0] = true;                   enter[1] = true;
  critical section { ... }           critical section { ... }
  // exit protocol                   // exit protocol
  enter[0] = false;                  enter[1] = false;
}                                  }
```

2) Using Boolean flags and waiting for the other one. Sets enter[k] true, waits until other thread not trying to enter cs. Does achieve mutex, but may deadlock. Threads can end up waiting for each other to proceed. Problem is the enter[k] are accessed independently.

```
boolean[] enter = {false, false};
thread t0                          thread t1
while (true) {                     while (true) {
  // entry protocol                  // entry protocol
  enter[0] = true;                   enter[1] = true;
  await (!enter[1]);                 await (!enter[0]);
  critical section { ... }           critical section { ... }
  // exit protocol                   // exit protocol
  enter[0] = false;                  enter[1] = false;
}                                  }
```

Producer-Consumer: Producers and consumers exchange items through a shared buffer. Producers asynch produce items and store them in buffer. Consumers asynch consume items after removing them from buffer. **nItems.up()** can be called after **lock.unlock(),** leads to temporary broken invariant. Executing **nItems.down()** after **lock.lock()** can lead to deadlock.

```
public class UnboundedBuffer<T> implements Buffer<T> {
  Lock lock = new Lock(); // for exclusive access to buffer
  Semaphore nItems = new Semaphore(0); // number of items in buffer
  Collection storage = ...; // any collection (list, set, ...)
  invariant { storage.count() == nItems.count() + at(5,15-17); }
}

public void put(T item) {                public T get() {
  lock.lock(); // lock                     // wait until nItems > 0
  // store item                            nItems.down();
  storage.add(item);                       lock.lock(); // lock
  nItems.up();   // update nItems          // retrieve item
  lock.unlock(); // release                T item =storage.remove();
}                                          lock.unlock(); // release
                                           return item;
public int count() {                     }
  return nItems.count(); // locking here?
}
```

Bounded version just adds another semaphore **nFree** with capacity **N**. **put()** needs to call **nFree.down()** first to ensure there is free space in buffer. Consumer calls **nFree.up()** after it has removed an item from the buffer to free up space.

Barriers with n threads (single use) variant:

```
int nDone = 0; // number of done threads
Lock lock = new Lock(); // mutual exclusion for nDone
Semaphore open = new Semaphore(0); // 1 iff barrier is open

thread tk
// code before barrier
lock.lock();                     // lock nDone
nDone = nDone + 1;               // I'm done
if (nDone == n) open.up();       // I'm the last: we can go!
lock.unlock();                   // unlock nDone
open.down();                     // proceed when possible
open.up();                       // let the next one go
// code after barrier
```

Total number of expected threads

Can we swit these?

Can switch tho in general reading a shared var outside a lock may give incosistent value. In this case, only after last thread has arrived can any thread read nDone == n, becase nDone is only incremented.

Reusable barriers (multiple use) variant:

Reusable barriers: Correct solution

```
public class SemaphoreBarrier implements Barrier {    void approach() {
  int nDone = 0; // number of done threads              synchronized (this) {
  Semaphore gate1 = new Semaphore(0);// first gate        nDone += 1;          // arrived
  Semaphore gate2 = new Semaphore(1);// second gate       if (nDone == n) {    // if last in:
  final int n;                                              gate1.up();        // open gate1
                                                            gate2.down();      // close gate2
  // initialize barrier for `n` threads                   }
  SemaphoreBarrier(int n) {                             }
    this.n = n;                                         gate1.down(); // pass gate1
  }                                                     gate1.up();   // let next pass
                                                      }
  // number of threads expected at the barrier        void leave() {
  int expect() {                                        synchronized (this) {
    return n;                                             nDone -= 1;          // going out
  }                                                       if (nDone == 0) {    // if last out:
                                                            gate2.up();        // open gate2
  public void wait() { approach(); leave(); }               gate1.down();      // close gate1
                                                          }
                                                        }
                                                        gate2.down(); // pass gate2
                                                        gate2.up();   // let next pass
                                                      }
```

Readers-writers concurrently accessing shared data. Readers may execute concurrently with other readers but need to exclude writers. **Writers** need to exclude both readers and other writers. Captures common problems in databases and filesystems. **Invariant** (#WRITERS = 0 OR (#WRITESR = 1 AND #READERS = 0).

3) Use one single integer variable yield. Thread Tk waits for its turn while yield is k, when it is done with its cs. It yields to the other thread k by setting yield = k. Guarantees mutex and deadlock freedom but not starvation freedom. A thread in CS may crash and never again yield, as such the other thread is waiting forever.

```
int yield = 0 || 1; // initialize to either value
thread t0                          thread t1
while (true) {                     while (true) {
  // entry protocol                  // entry protocol
  await (yield != 0);                await (yield != 1);
  critical section { ... }           critical section { ... }
  // exit protocol                   // exit protocol
  yield = 0;                         yield = 1;
}                                  }
```


3) Use one single integer variable yield. Thread t_k waits for its turn while yield is k, when it is done with its cs. It yields to the other thread k by setting yield = k. Guarantees mutex and deadlock freedom but not starvation freedom. A thread in CS may crash and never again yield, as such the other thread is waiting forever.

```
int yield = 0 || 1; // initialize to either value
thread t0                          thread t1
while (true) {                     while (true) {
  // entry protocol                  // entry protocol
  await (yield != 0);                await (yield != 1);
  critical section { ... }           critical section { ... }
  // exit protocol                   // exit protocol
  yield = 0;                         yield = 1;
}                                  }
```


Peterson's algorithm, the algorithm which solves the problem. Combine the ideas behind 2nd and 3rd attempts. Thread Tk first **sets enter[k] to true** then **lets other thread go** by setting **yield**.

```
boolean[] enter = {false, false};   int yield = 0 || 1;
thread t0                          thread t1
while (true) {                     while (true) {
  // entry protocol                  // entry protocol
  enter[0] = true;                   enter[1] = true;
  yield = 0;                         yield = 1;
  await (!enter[1]     ||            await (!enter[0]     ||
         yield != 0);                       yield != 1);
  critical section { ... }           critical section { ... }
  // exit protocol                   // exit protocol
  enter[0] = false;                  enter[1] = false;
}                                  }
```

Peterson's algorithm for **n threads** uses O(n) shared memory locations (two n-element arrays). It is possible to prove that this is the minimum amount of shared memory needed to have mutual exclusion **if only atomic reads and writes** are available. This is one of the reasons why synchronization through only atomic reads and writes is impractical. We need more powerful primitive operations.

One way of solving the n-process mutex problem using a single boolean variable is with the **test-and-set** operation and **busy-waiting**.

```
public class TASLock implements Lock {
  AtomicBoolean held = new
AtomicBoolean(false);

  public void lock() {
    while (held.getAndSet(true)) {
    }  // await (!testAndSet());
  }

  public void unlock() {
    held.set(false); // held = false;
  }
}
```

A thread trying to acquire the lock tries to continously get and set the lock. It tries to get the lock by obtaining the AtomicBoolean held and setting it to "true", only when it succeeds with that it has the locks. The "thing" here that makes this work is that "getANdSet(..)" is an atomic operation. So now only one flag is used instead of n.

Monitors: Semaphores provide powerful, concise mechanism for synch and mutex… but have shortcomings. They are global and unstructured, difficult to understand behavior. They are prone to deadlocks or other incorrect behavior. They do not support well different conditions at once. They are a low-level synchronization primitive, we will raise abstraction. **Monitors** provide a structured synchronization mechanism that is built on top of object oriented constructs, classes, objects and encapsulation. In a monitor class attributes are shared private variables and methods are executed in mutual exclusion. The methods themselves define (are) critical sections. At most one thread is active on a monitor at any time. Threads trying to access a monitor queue for entry; as soon as the active thread leaves the monitor the next thread in the entry queue gets exclusive access to the monitor.

Momotr do's and don'ts: What happens if a method monitor M calls a method n in monitor N (with condition variable cN)? Different rules are possible:
1) Prohibit nested calls
2) Release a lock on M before acquiring lock on N
3) Hold locn on M while also locking N
 3.1) When waiting on cN release both locks on N and M
 3.2) When waiting on cN release only lock on N
Rules 3 are prone to deadlock, especially 3.2 because deadlocks often occur when trying to acquire multiple locks.
Pros:
* Monitors provide a structured approach to concurrent programming, which builds atop the familiar notions of objects and encapsulation.
* This raises the level of abstraction of concurrent programming compared to semaphores.
* Monitors introduce seperation of concerns when programming concurrently, mutex is implicit in the use of monitors and condition variables provide a clear means of synchronization.
Cons: Monitors generally larger performance overhead than semaphores, perf traded agaisnt error proneness. The different signaling disicplines are a source of confusion with tarnishes the clarity of the monitor abstraction. For complex synchronization patterns, nested monitor calls are another source of complications.

Erlang:
* **Eight primitive types**: Integers, atoms, floats, references, binaries (sequences of bytes), pids (process identifiers), ports (for communication) and funs (function closures).
* **Three + two compound types:** Tuples, Lists, Maps, Strings, Records. Integer division (DIV), integer remainder (REM), float division (/).

OPERATOR	MEANING
not	negation
and	conjunction (evaluates both arguments/eager)
or	disjunction (evaluates both arguments/eager)
xor	exclusive or (evaluates both arguments/eager)
andalso	conjunction (short-circuited/lazy)
orelse	disjunction (short-circuited/lazy)

true or (10 + false) (error: type mismatchin second arg)
true orelse (10 + false) (true: only evalutes first arg)

OPERATOR	MEANING
<	less than
>	greater than
=<	less than or equal to
>=	greater than or equal to
=:=	equal to
=/=	not equal to
==	numeric equal to
/=	numeric not equal to

```
3 =:= 3    % true: same value, same type
3 =:= 3.0  % false: same value, different type
3 == 3.0   % true: same value, type not checked
```

When different types are compared, following order applies:
$number < atom < reference < fun < port < pid < tuple < map < list$

Concurrency is fundamental in Erlang:
* Processes are strongly isolated
* Process creation and destructions is a lightweight operation
* Message passing is the only way for processes to interact
* Processes have unique names
* If you know the name of ap rocess you can send it a emssage
* Proecsses share no resources
* Error handling is non-local
* Processes do what they are supposed to do or fail

For more complex synchronization patterns than mutual exclusion, monitors provide conditions variables:

```
interface Condition {
  void wait();        // block until signal
  void signal();      // signal to unblock
  boolean isEmpty();  // is no thread waiting on this condition?
}
```

A monitor class can declare condition variables as attributes (private, only callable by methods of the monitor). Every condition variable includes a FIFO queue blocked.
* c.wait() blocks the running thread, appends to blocked and releases lock on monitor.
* c.signal() removes one thread from blocked (if not empty) and unblocks it.
* c.isEmpty() returns true iff blocked is empty.

More signaling disciplines:
* **URGENT SIGNAL AND CONTINUE**: s continues executing; u is mvoed to the front of the entry queue.
* **URGENT SIGNAL AND WAIT:** s is moved to the front of the entry queue; u resumes executing.
An urgent thread gets ahead of "regular" threads, but may have to queue behind other urgent threads that are waiting for entry.

Writing correct programs: Programming means writing instructions that achieve a ceratin functionality. How do we know if a program is correct? And what does it even mean that a program is correct? To this end, we distinguish between **implementation** and **specification**. The **implementation** is the code that is written, compiled and executed. The **specification** is a description of what the program should do, usually at a more abstract level than the implementation.

Functional specifications: In sequential programming, we are mainly interested in functional – or input/output – specifications of individual methods. Such specifications consist of two parts. (1) **Precondition:** A constraint that defines the method's valid inputs. (2) **Postcondition:** A functional description of the expected output after executing the method. In some object-oriented programs, the input and output of a method also include the object state before anda fter executing the method.

Specifications of concurrent programs: The specification of concurrent programs should cover two parts: (1) A **functional** specification defines the correct input/output behavior. (2) A **temporal** specification defines the **absence** of **undesired** behavior, such as no race conditions, deadlock and starvation. Functional specification techniques such as pre- and postconditions and class invariants are also applicable to concurrent programs. Class invariants are particularly useful for shared-memory concurrency, where invariants characterize the valid states of shared objects. Temportal specifications require new notations and techniques. Temporal logic is a notation to specify behavior over time. More precisely, it formally defines properties of traces of states, like those that originate from the execution of a (concurrent) program).

Verification: is the process of checking that a program is correct. This means that, in addition to the implementation, there is also some form of specification (possibly only informal). Two main techniques to do verification: (1) **Testing**: Run the program using many different inputs and check that every run satisfies the specification. Testing ALL inputs is usually not possible, too big of a test-space. (2) **Formal verification:** Mathematically prove that every possible run of the program satisfies the expectation.

Erlang's runtime provides weak guarantees of message delivery order:
* If a process S sends some messages to another proecss R, then R will receive the messages in the same order S sent them
* If a proecss S sends some messages to two (or more) other processes R and Q, there is no guarantee about the order in which the messages sent by S are received by R relative to when they are received by Q. In practice, pretty much all Erlang code one writes dose not rely on any assumptions about message delivery order.

Signaling disciplines: When a thread s calls signal() on a condition varuiable, it is executing inside the monitor. Since no more than one thread may be active on a monitor at any time, the thread u unblocked by s cannot enter the monitor immedaitely. The **signaling discipline** determines what happens to a signaling thread s after it unblock another thread u by signaling. Two main choices:
* **signal and continue:** s continues executing; u is moved to the entry queue of the monitor. Under this discipline the signaled condition may no longer hold when the unblocked thread u resumes execution, because other threads may change state while continuing. There for the blocked threads need to recheck after waiting. (while-loop). Simpler to implement than S&W, so it is more commonly used.
* **signal and wait:** s is moved to the entry queue of the monitor; u resumes executing (silently gets monitor's lock). Under this disicpline the signaled condition is guaranteed to hold when the unblocked thread resumes execution, because it immediately follows the signal.

Transactions: The notion of transaction, which comes from database research, supports a general approach to lock-free programming: A **transaction** is a sequence of steps executed by a single thread which are executed atomically. A transaction may:
 * **Succeed:** All changed made by the transaction are committed to shared memory; they appear as if they happened instantaneously.
 * **Fail:** The partial changes are **rolled back**, and the shared memory is in the same state it would be if the transaction had never executed.
Therefore, a transaction either executed completely and successfully or it does not have any effect at all.

The notion of transaction supports a general approach to lock-free programming: (*) Define a transaction for every access to shared memory (**) If the transaction succeeds, there was no interference. (**) If the transaction failed, retry until it succeeds. Transactional atomic blocks are similar to monitor's methods with implicit locking, but they are much more flexile. (*) Transactions do not lock => no overhead (*) Parallelism is achieved without risks of race conditions. (*) Since no locks acquired => no problem of deadlocks (starvation may still occur though with big contention (*) Transactions compose easily.

Linear Temporal Logic (LTL)

FORMULA	MEANING
p	p is true
$\neg p$	p is not true (i.e., false)
$p \wedge q$	p and q are true
$p \vee q$	p or q is true (or both)
$p \Rightarrow q$	p true implies that q true (if p then q too)

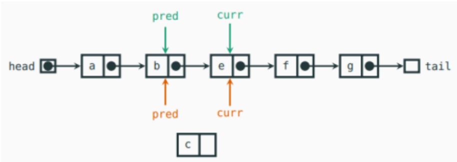
FORMULA	MEANING
$\Diamond p$	p is eventually true (from now on)
$\Box p$	p is always true (from now on)
$p \, \mathsf{U} \, q$	p is true (from now on) until q is true
$\mathsf{X} p$	p is true in the next step

PROPOSITION	STATE PROPERTY
c_t	thread t is in its critical section
c_u	thread u is in its critical section
e_t	thread t is trying to enter its critical section
n_t	thread t has terminated

Parallelization: risks and opportunities
* Concurrent programing introduces (+) the potential for parallel execution (faster, better resource usage) (-) the risk of race conditions (incorrect, unpredictable computations). Main challenge is introducing parallelism without affecting correctness. A number of factors challenge designing correct and efficient parallelizations:
* **Sequential dependencies:** Some steps in a task computation depends on the result of other steps. One task must wait for another task to run, limits the amount of parallelism that can be achieved. The synch problems (producer-consumer, dining philosophers, etc.) capture kinds of sequentuial dependencies.
* **Synchronization costs**, **spawning costs**
* **Error proneness and composability**: For example two thread unsafe methods executed inside a thread safe method does not work! Can't just create compositions like that and expect it to work.

Two classes of lock-free algorithms, collectively called non-blocking:
* **Lock-free:** Guarantee system-wide progress: infinitely often, some process makes progress.
* **Wait-free**: Guarantee per-process progress: every process eventually makes progress. This is stronger. Lock-free algos are free from deadlock while wait-free algos are free from both deadlock AND starvation.

Sequential set does not work under concurrency:

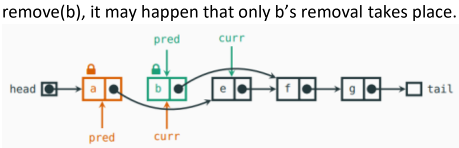


If thread t runs remove(e) while thread u runs add(c), in some interleavings remove can be reverted. In some add can be reverted.

Concurrent set with coarse-grained locking: A straight forward way to make the SequentialSet work correctly under concurrency is to use a lock mechanism so an operating thread has exclusive acces to when operating on the structure. This works and avoids race conditions and deadlocks. If the lock is fair, so is the access to the set and if contention is low (not many threads are accessing the set concurrently) then it is quite efficient too. But this solution practically makes the set sequential, missing the point of parallelization. If contention is high, it can easily become slow too.

Can we reduce the size of the critical sections by executing find without locking and only locking when a thread has found the elements and wants to operate on them? No we can't, this is because the list may be modified between when a thread performs find and when it acquires the lock. A thread u wanting to add some element might end up working on a set that is in an inconsistent state because of some other thread that had operated on it just previously.

Fine grained locking Now we try and add a lock to each node, then the threads only lock the individual nodes on which they are operating. This will not work because say a thread t (standing on b) runs remove(e) while u runs remove(b), it may happen that only b's removal takes place.



So we must lock both PRED and CURR at once. This lock acquisition protocol is called "**hand-over-hand"** locking or **"lock coupling"**. Locking two nodes at once is sufficient to prevent problems with conflicting operations; threads proceed along the linked list in order, without one thread "overtaking" another thread that is further out. The protocol ensures that locks are acquired by all threads in the same order, thus avoiding deadlocks. If the locks are fair, so is access to the set. Threads operating on disjoint portions of the list may be able to operate in parallel. It is though still possible that a thread prevents another from operating in parallel on disjoint portions of the set. If one thread operates early in the set while another sets wants to operate late in the set, the early one will block the late one. The hand-over-hand locking protocol may also be quite slow as it requires a lot of locking operations.

Fork join paralellism: Recursive subdivision of a task that assigns new processes to smaller tasks. **Forking**: spawning child processes and assigning them smaller tasks. **Joining**: Waiting for the child processes to complete and combining their results.

Let us now try and build a concurrent set with optimistic locking. Previously the problems occurred between when a thread finds a position and when it acquires the locks on that position, the list could've been modified between that. Instead we validate a position after finding it and while the nodes are locked, this to verify that no interference took place.
1. Find the item's pos withotu locking, as in SeqSet.
2. Lock the position's nodes pred and curr
3. Validate pos while nodes ocked
  3.1) If valid, perform operation while nodes locked then release locks.
  3.2) If invalid, release locks and repeat the operation from scratch. (go back to 2).

What can happen between the time when a thread finds a position (pred, curr) and when it locks nodes pred and curr?
* Node pred is removed, validation fails because pred is not reachable.
* Node curr is removed, validations fails because pred does not point to curr.
* A node is added between pred and curr, validation fails because pred does not point to curr.
* Any other modification of the set, validation succeeds because operations leave the set in a consistent state.

What happens if the set is being modified while a thread is validating a locked position (pred, curr)?
* If a node following curr is modified, validation is not affected because it only goes up until curr.
* If a node n before pred is removed, validation succeeds even if it goes through n, since n still leads back to pred.
* If a node n is added before pred, validation succeeds even if it skips over n.

Pros: Threads operating on disjoint portions of the list can operate in parallel. When validation often succeeds, there is much less locking involed than in FineSet.

Cons: OptimisticSet is not starvation free, a thread t may fail validation forever if other threads keep removing and adding pred/curr between when t performs find and when it locks pred and curr. If traversing the list twice without locking is not significantly faster than traversing it once with locking, OptimisticSet dose not have a clear advantage over FineSet.

Testing membership without locking: In many applications, operation ahs is executed many more times than add and remove. Can has work correctly without locking? Problems may occur if another thread removes curr between find and has's check, since remove is not atomic without locking, if has does not acquire locks it may not notice that curr is being removed. **As such** we need a way to atomically share the information that a node is being removed, but without locking. So we use ValidatedNode with a flag valid that is either true or false.

In lazy-set: Validation only needs to check the mark valid, operation remove marks a node invalid before removing it, operation has is lock free, operation add works as in optimistic set. Validation now becomes a constant-time operation. Node pred is reachable from the head iff it has not been removed iff it is marked valid. Now curr follows pred in the list iff pred.next() == curr and curr.isValid().

Pros: Validation constant time, membership checking does not require any locking it's also wait free (traverses list without locking), physical removal of logically removed nodes could be batched and performed when convenient thus reducing the number of times physical chain of nodes is changed, in turn reducing expensive propagation of info between threads.

Cons: Operations add and remove still require locking (as in OptimisticSet), which may reduce amount of parallelism.

Pools: ForkJoinPools take care of efficiently dispatching work to threads. The framework introduces a layer of abstraction between computational tasks and actually running threads that execute the tasks. The fork/join model simplifies parallelizing computations.

Good practies:
* After forking childen tasks, keep some work for the parent task before it joins the children.
* For the same reason, use invoke and invokeAll only at the top level as a norm
* Perform small enough tasks sequentailly in the parent task, and fork children tasks only when there is a substantial chunk of work left.
* Make sure different tasks can prcoeed independently, minimize data dependencies.

Pools and work stealing: A pool creates a number of worker processes upon initialization. As long as more work is available, the pool deals a work assignment to a worker that is available. The pool collects the results of the workers' computations. When all work is completed, the pool terminates and returns the overall result. This kind ofa pool is called a **dealing pool** because it actviely deals work to workers. They work well when workload can be split in even chunks and the workload does not change over time. Under these conditions, the workload is balanced evenly between workers, so as to maximize the amount of parallel computation.

Stealing pools: associate a queue to every worker process, the pool distributes new tasks by adding them to the workers' queues. When a worker becomes idle, it first gets the next task from its own queue. If it's empty, it can directly steal tasks from the queue of another worker that is currently busy. With this approach workers adjust dynamically to the current working conditions without requiring a supervisor that can reliably predict the workload required by each task. With stealing, the pool may even send all tasks to one default thread letting the other idle threads steal directly from it. This simplifies the pool and reduces synchronization costs it incurs.

Lock-free acces: Completely lock free concurrent set. We need to rely on more powerful synchronization primitives than just reading and writing shared variables. We use the **compare-and-set** operation. compareAndSet(expected, new) works if reference/value == expected, set to new and return true otherwise do not change ref/value and return false.

```
public boolean remove(T item) {
  boolean done;
  do {
    Node<T> pred, curr = find(head, item.key());
    if (curr.key() >= item.key()) return false;  // item not in set
    else
      // try to remove curr by setting pred.next using compareAndSet
      done = pred.next().compareAndSet(pred.next(), curr.next());
  } while (!done); return true;
}
```

pred.next may have changed
when compareAndSet() executes

Does not work, similar problem to before. Need to have control of both pred and curr.

In a lock-free set: Operation remove marks a node invalid (using attemptMark (similar to compareAndSet, now setNextIfValid()) before removing it. Operations that modify nodes complete successfully ONLY IF the nodes are valid and not concurrently modified by another thread. Failed operations are repeated until success (no interference).

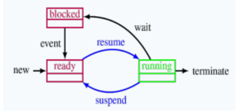
```
public boolean remove(T item) {
  do { Node<T> pred, curr = find(head, item.key()); // not in set
       if (curr.key() != item.key() || !curr.valid()) return false;
        // try to invalidate; try again if node is being modified
       if (!curr.setInvalid()) continue;
        // try once to physically remove curr
       pred.setNextIfValid(curr, curr.next());
       return true;
  } while (true);  // changed during logical removal: try again!
}
```

Has does not modify set, traverses safely valid and invalid without changing node structures. Methods add and remove PHYSICALLY REMOVE all logically removed nodes encountered by find.

Pros: No operations require locking, maximum potential for parallelism. Membership checking does not require any locking, it's even wait free.

Cons: Implementation needs test-and-set-like synhronization primitives, need to be supported and come with their own performance costs. Operations add and remove lock free but not wait free, they may have to repeat operations and they may be delayed while they physically remove invalid nodes with the risk of introducing contention on nodes that have been already previously logically deleted.

Abstraction: Separating tasks without worrying when to execute them.
Responsiveness: Providing a responsible UI, diff tasks executing independently
Performance: Splitting complex tasks.
Process/Thread = independent unit of execution.
Runtime/OS = schedules processes for execution.
Ready: to be executed, not allocated to any CPU
Blocked: waiting for an event to happen
Running: on some CPU



You can have concurrency without physical parallelism.

Amdahl's law: We have *n processors* in parallel, how much *speedup* can we achieve?
S = seq.ex.time/par.ex.time
Max_speedup = 1/(1-p) + p/n.
(1-p) is seq part, p/n par part.

Shared memory vs message passing
SM: comm by writing to shared memory, e.g., multi-core systems.
MP: comm by message passing, e.g., distributed systems.

Sequence of states gives **execution trace** of concurrent program. A trace is an *abstraction* of concrete executions:
Atomic/linearized: The effects of each thread appears as if they happened instantaneously, when the trace snapshot is taken, in the thread's sequential order.
Complete: The trace includes all intermediate atomic states.
Interleaved: The trace is an interleaving of each thread's linear trace (in particular, no simultaneity).

Concurrent programs are nondeterministic:
* Executing same conc.prog multiple times with same inputs may lead to diff exec traces.
* Are a result of the nondeterministic interleaving of each thread's trace to determine overall program trace.
* In turn the interleaving is a result of the scheduler's decisions
Not every *race condition* is a *data race*: Race conditions can occur even when there is no shared memory access, e.g., filesystems or network access.
Not every *data race* is a *race condition*: The data race may not affect the result, e.g., if two threads write the same value to shared memory.

Race condition: A RC is a situation where the correctness of a concurrent program depends on the specific execution. (Different interleavings during execution may lead to different end results). RCs can complicate debugging.
Data race: Race conditions are typically caused by a lack of synchronization between threads that access shared memory which introduce *data races*. A **data race** occurs when two concurrent threads:
* Access a shared memory location
* At least one access is a *write*
* The threads use no explicit *synchronization mechanism* to protect the shared data.

Concurrent programming introduces:
* **The potential for parallel execution** (faster, better resource usage)
* **The risk of race conditions** (incorrect, unpredictable computations)
The main challenge of conc.prog is thus introducing parallelism without introducing RCs. This requires restricting the amount of nondeterminism by synchronizing processes/threads that access shared resources. **Correctness** more important than performance.

Critical section: Part of a program that accesses the shared resource (ex: shared var)
Mutual Exclusion Property: No more than 1 thread is in its CS at any given time.
Mutual Exclusion Problem: Devise a protocol for accessing a shared resource that satisfies the **mutual exclusion property.**

What's a good solution to the mutual exclusion problem?
Achieves three properties:
* **Mutual exclusion:** at most one thread is in its critical section at any given time.
* **Freedom from deadlock:** if one or more threads try to enter the critical section, some thread will eventually succeed. A **deadlock** is the situation where a group of threads wait forever because each of them is waiting for resources that are held by another thread in the group (circular dependency).
* **Freedom from starvation:** every thread that tries to enter the critical section will eventually succeed. Starvation is the situation where a thread is *perpetually denied access* to a resource it requests. If thread t is in its critical section, then thread u can reach its critical section without requiring thread t's collaboration after it executes the exit protocol. **NOTE!!! FREEDOM FROM STARVATION IMPLIES FREEDOM FROM DEADLOCK … BUT NOT THE OPPOSITE.** A good solution should also work for an arbitrary number of threads sharing the same memory.

Weak fairness: if a thread continuously requests (that is, without interruptions) access to a resource, then access is granted eventually (or infinitely often).
Strong fairness: if a thread requests access to a resource infinitely often, then access is granted eventually (or infinitely often).

Semaphores: A (general/counting) semaphore is a data structure with interface (count(), acquire(), release()). **A semaphore** is often used to regulate access permits to a **finite** number of resources. Several threads share the same *sem* object
* Initially count is set to a nonnegative value (capacity C)
* a call to sem.acquire() atomically increments count by one
* a call to sem.release() waits until count is positive, then atomically decrements count by one.

Weak vs. strong semaphores
Every implementation of semaphores should **guarantee** 1) the atomicity of the acquire and release operations and 2) deadlock freedom)
Fairness is optional though:
* **Weak semaphore:** Threads waiting to perform acquire are scheduled nondeterministically.
* **Strong semaphore:** threads waiting to perform acquire are scheduled fairly in FIFO order.

Binary semaphores vs locks
Binary semaphores are very similar to locks but differ in one important thing. In a lock, only the holding thread can increment it back to 1 when releasing. In a semaphore though, any thread that is sharing the same semaphore object may decrement and/or increment it. Not only the holding thread.

Barrier synchronization
A **barrier** is a form of synchronization where there is a *point* (the **barrier**) in a program's execution that all threads in a group have to reach **before any of them** are allowed to continue.

Dining philosophers, A Classic synchronization problem:
* Five philosophers sit at dinner table, fork between each pair of adjacent philosophers.
* Each philosopher alternates between thinking (non-cs) and eating (cs).
* In order to eat, must pick up two forks (left and right).
* Since forks are shared, requires synchronization.

Failed attempt at solving:

```
entry   () {
  left_fork.acquire(); // pick up left fork
  right_fork.acquire();// pick up right forl
}
critical section { eat(); }
exit    () {
  left_fork.release(); // release left fork
  right_fork.release();// release right forl
}
```

The protocol deadlocks if all philosophers get their left forks and wait forever for their right forks to become available. (Circular dependency!)

Breaking a circular wait
A solution to the problem that avoids deadlock by breaking circular wait. Pick up first the fork with the lowest ID number. It avoids circular wait since not every philosopher picks up their left fork first.

```
entry  () {
  if (left_fork.id()< right_fork.id())
  { left_fork.acquire();
    right_fork.acquire();
  }

  else
  { right_fork.acquire();
    left_fork.acquire();
  }
}
critical section { eat(); }
exit   () { /* ... */ }
```

Now at most two people at a time can eat. Ordering shared resoruces and forcing threads to acquire the resources in

Invariants: An object's invariant is a property that always holds between calls to the object's methods:
* The invariant holds **initially** (when object is created)
* Every method call **starts** in a state that satisfies the invariant
* Every method call **ends** in a state that satisfies the invariant.
Ex: A **bank account** that cannot be overdrawn has an invariant *balance >= 0.*

```
class BankAccount {
  private int balance = 0;
  void deposit(int amount)
    { if (amount > 0) balance += amount; }
  void withdraw(int amount)
    { if (amount > 0 && balance > amount) balance -= amount; }
}
```

There are more solutions too, for example bounding resources. Allow only M < N to sit down, pick up both forks then leave. This is the "bounded-resource solution).
* At most M philosopher are active at the table
* The other N – M are waiting on seats.down()
* The first of the M philosophers that finishes eating releases a seat
* The philosopher P that has been waiting on seats.down() proceeds
* Similarly to the assymetric solution, P also eventually gets the forks. (No starvation => no deadlock).

The Coffman conditions, necessary conditions for a deadlock to occur:
1) **Mutual exclusion:** threads may have exclusive access to the shared resources
2) **Hold and wait:** a thread may request one resource while holding another one.
3) **No pre-emption:** resources cannot forcibly be released from threads that hold them.
4) **Circular wait:** two or more threads form a circular chain where each thread waits for a resource that the next thread in the chain is holding.
* Avoiding deadlocks requires to *break* **one or more** of these conditions.

Another solution to the problem that avoids deadlocking by now breaking **hold and wait** (and thus **circular wait**).

```
entry  () {
  forks.acquire(); // pick up left
and right fork, atomically
}
critical section { eat(); }
exit    () {
  forks.release(); // release left
and right fork,atomically
}
```

Here they pick up boths forks at once (atomic op.) and release when done. This avoids **deadlock**, but may introduce **starvation**, a philosopher may never get a chance to pick up the forks.

Sequential philosophers:
There is another solution which avoids deadlocks and starvation by using a fair waiter which decides which philosopher eats, gives permission to one philosopher at a time.

```
entry  () {
  while (!waiter.can_eat(k)) {
  // wait for permission to eat
  }
  left_fork.acquire();
  right_fork.acquire();
}
critical section { eat(); }
exit   () { /* ... */ }
```

This works, but is not really concurrent programming because a waiter who only gives permission to one thread at a time obliges to follow a sequential order.