Parallel Programming
Readers/Writers Lock, Lock Granularity:
Coarse Grained, Fine Grained, Optimal, and Lazy Synchronization

Source: xkcd.com

How the Boeing 737 Max Disaster Looks to a Software Developer
Design shortcuts meant to make a new plane seem like an old, familiar one are to blame

By Gregory Travis
Administrivia

- **Book for (the second part of) the lecture series:**
  - Herlihy & Shavit: The Art of Multiprocessor Programming
    
    *Contains more concepts (e.g., linearizability) than we use here!*

- **Locking objects – terminology clarification**
  - When we say “we lock an object” then I mean “we acquire the lock associated with the object” – data cannot be locked directly in Java (cf. advisory filesystem locks)

- **Starvation-freedom:**
  - *every* thread that tries to make progress, makes progress *eventually*
Last week time

- **Deadlock**
  - Cause (cyclic dependencies)
  - Avoidance (acquire resources in global order)

- **Semaphores**
  - Generalization of locks, can count (enables producer/consumer)

- **Barriers**
  - Multi-process synchronization, important in parallel programming
  - More examples for complexity of parallel programming (trial and error impossible)

- **Producer/Consumer in detail**
  - Queues, implementation

- **Monitors**
  - Condition variables, wait, signal, etc. (continued today)
Learning goals today

- Finish condition variables and interface locks
  - Producer-consumer examples

- More on locks (essentially a bag of tricks)
  - Reader/writer
  - Coarse-grained vs. fine-grained
  - Optimistic synchronization
  - Lazy synchronization

- Conflict-minimizing structures
  - Example: skip lists
Producer / Consumer queues

```java
public synchronized void enqueue(long item) {
    while (isFull())
        ; // wait
    doEnqueue(item);
}

public synchronized long dequeue() {
    while (isEmpty())
        ; // wait
    return doDequeue();
}
```

→ Blocks forever
infinite loops with a lock held ...

Do you see the problem?

```java
public void doEnqueue(long item) {
    buffer[in] = item;
    in = next(in);
}

public boolean isFull() {
    return (in+1) % size == out;
}

public long doDequeue() {
    long item = buffer[out];
    out = next(out);
    return item;
}

public boolean isEmpty() {
    return in == out;
}
```
Producer / Consumer queues using sleep()

```java
public void enqueue(long item) throws InterruptedException {
    while (true) {
        synchronized(this) {
            if (!isFull()) {
                doEnqueue(item);
                return;
            }
        }
        Thread.sleep(timeout); // sleep without lock!
    }
}
```

What is the proper value for the timeout? Ideally we would like to be notified when the change happens! When is that?
import java.util.concurrent.Semaphore;

class Queue {
    int in, out, size;
    long buf[];
    Semaphore nonEmpty, nonFull, manipulation;

    Queue(int s) {
        size = s;
        buf = new long[size];
        in = out = 0;
        nonEmpty = new Semaphore(0); // use the counting feature of semaphores!
        nonFull = new Semaphore(size); // use the counting feature of semaphores!
        manipulation = new Semaphore(1); // binary semaphore
    }
}

Producer / Consumer queues with semaphores
Producer / Consumer queues with semaphores, correct?

```java
void enqueue(long x) {
    try {
        manipulation.acquire();
        nonFull.acquire();
        buf[in] = x;
        in = (in+1) % size;
    } catch (InterruptedException ex) {} 
    finally {
        manipulation.release();
        nonEmpty.release();
    }
}
```

```java
long dequeue() {
    long x=0;
    try {
        manipulation.acquire();
        nonEmpty.acquire();
        x = buf[out];
        out = (out+1) % size;
    } catch (InterruptedException ex) {} 
    finally {
        manipulation.release();
        nonFull.release();
    }
    return x;
}
```
Deadlock (nearly the same as before, actually)!

Diagram:

- **Consumer**
  - Requires `nonEmpty`
  - Owned by `Producer`
- **Producer**
  - Requires **Consumer**
  - Owned by **Manipulation**
- **Manipulation**
  - Requires `nonEmpty`
  - Owned by **Consumer**
Producer / Consumer queues with semaphores

void enqueue(long x) {
    try {
        nonFull.acquire();
        manipulation.acquire();
        buf[in] = x;
        in = next(in);
    } catch (InterruptedException ex) {} 
    finally {
        manipulation.release();
        nonEmpty.release();
    }
}

long dequeue() {
    long x=0;
    try {
        nonEmpty.acquire();
        manipulation.acquire();
        x = buf[out];
        out = next(out);
    } catch (InterruptedException ex) {} 
    finally {
        manipulation.release();
        nonFull.release();
    }
    return x;
}
Why are semaphores (and locks) problematic?

Semaphores are unstructured. Correct use requires high level of discipline. Easy to introduce deadlocks with semaphores.

We need: a lock that we can temporarily escape from when waiting on a condition.
Monitors
Monitors

Monitor:
abstract data structure equipped with a set of operations that run in mutual exclusion.

Monitors vs. Semaphores/Unbound Locks
Producer / Consumer queues

```java
public void synchronized enqueue(long item) {
    "while (isFull()) wait"
    doEnqueue(item);
}
```

```
public long synchronized dequeue() {
    "while (isEmpty()) wait"
    return doDequeue();
}
```

The mutual exclusion part is nicely available already. But: while the buffer is full we need to give up the lock, how?
Monitors

Monitors provide, in addition to mutual exclusion, a mechanism to check conditions with the following semantics:

If a condition does not hold
- Release the monitor lock
- Wait for the condition to become true
- Signaling mechanism to avoid busy-loops (spinning)
Monitors in Java

Uses the intrinsic lock *(synchronized)* of an object

+ **wait / notify / notifyAll:**
  - `wait()` – the current thread waits until it is signaled (via notify)
  - `notify()` – wakes up *one* waiting thread (an arbitrary one)
  - `notifyAll()` – wakes up *all* waiting threads
Producer / Consumer with monitor in Java

class Queue {
    int in, out, size;
    long buf[];

    Queue(int s) {
        size = s;
        buf = new long[size];
        in = out = 0;
    }
    ...
}
Producer / Consumer with monitor in Java

```java
synchronized void enqueue(long x) {
    while (isFull())
        try {
            wait();
        } catch (InterruptedException e) { }
    doEnqueue(x);
    notifyAll();
}
```

```java
synchronized long dequeue() {
    long x;
    while (isEmpty())
        try {
            wait();
        } catch (InterruptedException e) { }
    x = doDequeue();
    notifyAll();
    return x;
}
```

Wouldn't an if be sufficient?

(Why) can't we use notify()?
IMPORTANT TO KNOW JAVA MONITOR IMPLEMENTATION DETAILS
Thread States in Java

- **NEW**: thread has not yet started
- **RUNNABLE**: thread is runnable, may or may not be currently scheduled by the OS
- **TIMED_WAIT**: waiting state with specified waiting time, e.g., `sleep`
- **WAITING**: thread is waiting for a condition or a join
- **BLOCKED**: thread is waiting for entry to monitor lock
- **TERMINATED**: thread has finished execution

**Spaced repetition**

Spaced repetition is a learning technique that incorporates increasing intervals of time between subsequent review of previously learned material in order to exploit the psychological spacing effect. Alternative names include spaced rehearsal, expanding rehearsal, graduated intervals, repetition spacing, repetition scheduling, spaced retrieval, and expanded retrieval. 

*From Wikipedia, the free encyclopedia*
Monitor Queues

- Method call
- Waiting entry
- Waiting condition
- Monitor
- Notification
- Wait
Various (exact) semantics possible

Important to know for the programmer (you): what happens upon notification? Priorities?
signal and wait
  signaling process exits the monitor (goes to waiting entry queue)
  signaling process passes monitor lock to signaled process

signal and continue
  signaling process continues running
  signaling process moves signaled process to waiting entry queue

other semantics: signal and exit, signal and urgent wait ...
Why is this important? Let's try this implementing a semaphore:

class Semaphore {
    int number = 1; // number of threads allowed in critical section

    synchronized void acquire() {
        if (number <= 0)
            try {
                wait();
            } catch (InterruptedException e) { };
        number--;
    }

    synchronized void release() {
        number++;
        if (number > 0)
            notify();
    }
}

Looks good, doesn't it? But there is a problem. Do you know which?
Java Monitors = signal + continue

synchronized void acquire() {
    if (number <= 0)
        try {
            wait();
        } catch (InterruptedException e) { };
    number--;
}

synchronized void release() {
    number++;
    if (number > 0)
        notify();
}

Scenario:
1. Process P has previously acquired the semaphore and decreased number to 0.
2. Process Q sees number = 0 and goes to waiting list.
3. P is executing release. In this moment process R wants to enter the monitor via method acquire.
4. P signals Q and thus moves it into wait entry list (signal and continue!). P exits the function/lock.
5. R gets entry to monitor before Q and sees the number = 1
6. Q continues execution with number = 0!

Inconsistency!
The cure – a while loop.

synchronized void acquire() {
    while (number <= 0)
        try {
            wait();
        } catch (InterruptedException e) {
        }
    number--;
}

synchronized void release() {
    number++;  
    if (number > 0)
        notify();
}

If, additionally, different threads evaluate different conditions, the notification has to be a notifyAll. In this example it is not required.
Something different: Java Interface Lock

Intrinsic locks ("synchronized") with objects provide a good abstraction and should be first choice

Limitations

▪ one implicit lock per object
▪ are forced to be used in blocks
▪ limited flexibility

Java offers the Lock interface for more flexibility (e.g., lock can be polled).

```java
final Lock lock = new ReentrantLock();
```
Condition interface

Java Locks provide *conditions that can be instantiated*

```java
Condition notFull = lock.newCondition();
```

Java conditions offer

- `.await()` – the current thread waits until condition is signaled
- `.signal()` – wakes up one thread *waiting on this condition*
- `.signalAll()` – wakes up all threads *waiting on this condition*
Condition interface

→ Conditions are always associated with a lock
  lock.newCondition()

.await()
  – called with the lock held
  – atomically releases the lock and waits until thread is signaled
  – when returns, it is guaranteed to hold the lock
  – thread always needs to check condition

.signal{,All}() – wakes up one (all) waiting thread(s)
  – called with the lock held
Producer / Consumer with explicit Lock

class Queue {
    int in=0, out=0, size;
    long buf[];
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    Queue(int s) {
        size = s;
        buf = new long[size];
    }

    ...
}

void enqueue(long x) {
    lock.lock();
    while (isFull())
        try {
            notFull.await();
        } catch (InterruptedException e) {

    }
    doEnqueue(x);
    notEmpty.signal();
    lock.unlock();
}

long dequeue() {
    long x;
    lock.lock();
    while (isEmpty())
        try {
            notEmpty.await();
        } catch (InterruptedException e) {

    }
    x = doDequeue();
    notFull.signal();
    lock.unlock();
    return x;
}
The Sleeping Barber Variant (E. Dijkstra)

Disadvantage of the solution: notFull and notEmpty signal will be sent in any case, even when no threads are waiting.

Seemingly simple solution (in barber analogy)

1. Barber cuts hair, when done, check waiting room, if nobody left, sleep
2. Client arrives, either enqueues or wakes sleeping barber

What can go wrong (really only in a threaded world)?

Sleeping barber requires additional counters for checking if processes are waiting:

\[ m \leq 0 \iff \text{buffer full } \& \ (-m \text{ producers (clients) are waiting}) \]
\[ n \leq 0 \iff \text{buffer empty } \& \ (-n \text{ consumers (barbers) are waiting}) \]
class Queue {
    int in=0, out=0, size;
    long buf[];
    final Lock lock = new ReentrantLock();
    int n = 0; final Condition notFull = lock.newCondition();
    int m; final Condition notEmpty = lock.newCondition();

    Queue(int s) {
        size = s; m=size-1;
        buf = new long[size];
    }
    ...
}
Producer Consumer, Sleeping Barber Variant

```java
void enqueue(long x) {
    lock.lock();
    m--; if (m<0)
        while (isFull())
            try { notFull.await(); } catch(InterruptedException e){}
    doEnqueue(x);
    n++;
    if (n<=0) notEmpty.signal();
    lock.unlock();
}

long dequeue() {
    long x;
    lock.lock();
    n--; if (n<0)
        while (isEmpty())
            try { notEmpty.await(); } catch(InterruptedException e){}
    x = doDequeue();
    m++;
    if (m<=0) notFull.signal();
    lock.unlock();
    return x;
}
```
Guidelines for using condition waits

• Always have a condition predicate
• Always test the condition predicate:
  ▪ before calling wait
  ▪ after returning from wait
• Always call wait in a loop
• Ensure state is protected by lock associated with condition
Check out java.util.concurrent

Java (luckily for us) provides many common synchronization objects:

- Semaphores
- Barriers (CyclicBarrier)
- Producer / Consumer queues
- and many more... (Latches, Futures, ...)

Check out java.util.concurrent
Reader / Writer Locks

Literature: Herlihy – Chapter 8.3
Reading vs. writing

Recall:
- Multiple concurrent reads of same memory: *Not* a problem
- Multiple concurrent writes of same memory: Problem
- Multiple concurrent read & write of same memory: Problem

So far:
- If concurrent write/write or read/write *might* occur, use synchronization to ensure one-thread-at-a-time

But this is unnecessarily conservative:
- Could still allow multiple simultaneous readers!
Example

Consider a hashtable with one coarse-grained lock
- So only one thread can perform operations at a time

But suppose:
- There are many simultaneous lookup operations
- insert operations are very rare

Note: Important that lookup does not actually mutate shared memory, like a move-to-front list operation would

Average views per day: ~200,000,000
→ 0.12% write rate
Another Example

Shared use of text, e.g., in an IDE
writers: editor(s), copy&paste agents, syntax highlighter
readers: compiler, editor(s), text viewers, copy&paste agents, search tools
Reader/writer locks

A new abstract data type for synchronization: The reader/writer lock

This lock's states fall into three categories:
- “not held”
- “held for writing” by one thread
- “held for reading” by one or more threads

\[ 0 \leq \text{writers} \leq 1 \]
\[ 0 \leq \text{readers} \]
\[ \text{writers} \times \text{readers} = 0 \]
Reader/writer locks

**new:** make a new lock, initially “not held”

**acquire_write:** block if currently “held for reading” or “held for writing”, else make “held for writing”

**release_write:** make “not held”

**acquire_read:** block if currently “held for writing”, else make/keep “held for reading” and increment **readers count**

**release_read:** decrement readers count, if 0, make “not held”
Pseudocode example

```java
class Hashtable<K,V> {
    ...
    // coarse-grained, one lock for table
    RWLock lk = new RWLock();
    ...

    V lookup(K key) {
        int bucket = hashval(key);
        lk.acquire_read();
        ...
        read array[bucket] ...
        lk.release_read();
    }

    void insert(K key, V val) {
        int bucket = hashval(key);
        lk.acquire_write();
        ...
        write V to array[bucket] ...
        lk.release_write();
    }
    ...
}
```
A Simple Monitor-based Implementation

class RWLock {
    int writers = 0;
    int readers = 0;

    synchronized void acquire_read() {
        while (writers > 0)
            try { wait(); }
        catch (InterruptedException e) {}
        readers++;
    }

    synchronized void release_read() {
        readers--;
        notifyAll();
    }

    synchronized void acquire_write() {
        while (writers > 0 || readers > 0)
            try { wait(); }
        catch (InterruptedException e) {}
        writers++;
    }

    synchronized void release_write() {
        writers--;
        notifyAll();
    }
}

Is this lock fair?
The simple implementation gives priority to readers:
• when a reader reads, other readers can enter
• no writer can enter during readers reading

Exercise: come up with a better performing version using condition variables!
Strong priority to the writers

class RWLock {
    int writers = 0;
    int readers = 0;
    int writersWaiting = 0;

    synchronized void acquire_read() {
        while (writers > 0 || writersWaiting > 0)
            try { wait(); } catch (InterruptedException e) {};
        readers++;
    }

    synchronized void release_read() {
        readers--;
        notifyAll();
    }

    synchronized void acquire_write() {
        writersWaiting++;
        while (writers > 0 || readers > 0)
            try { wait(); } catch (InterruptedException e) {};
        writersWaiting--;
        writers++;
    }

    synchronized void release_write() {
        writers--;
        notifyAll();
    }
}

synchronized void acquire_write() {
    writersWaiting++;
    while (writers > 0 || readers > 0)
        try { wait(); } catch (InterruptedException e) {};
    writersWaiting--;
    writers++;
}

synchronized void release_write() {
    writers--;
    notifyAll();
}

Is this lock now fair?
(this was just to see of you’re awake)
A fair(er) model

What is fair in this context?

For example

- When a writer finishes, a number $k$ of currently waiting readers may pass.
- When the $k$ readers have passed, the next writer may enter (if any), otherwise further readers may enter until the next writer enters (who has to wait until current readers finish).
A fair(er) model

class RWLock{
    int writers = 0; int readers = 0;
    int writersWaiting = 0; int readersWaiting = 0;
    int writersWait = 0;

    synchronized void acquire_read() {
        readersWaiting++;
        while (writers > 0 ||
            (writersWaiting > 0 && writersWait <= 0))
            try { wait(); } catch (InterruptedException e) {};
        readersWaiting--;
        writersWait--;
        readers++;
    }

    synchronized void release_read() {
        readers--;
        notifyAll();
    }

    synchronized void acquire_write() {
        writersWaiting++;
        while (writers > 0 || readers > 0 || writersWait > 0)
            try { wait(); } catch (InterruptedException e) {};
        writersWaiting--;
        writers++; };

    synchronized void release_write() {
        writers--;
        writersWait = readersWaiting;
        notifyAll();
    }
}

When a writer finishes, the number of currently waiting readers may pass.

Writers have to wait until the waiting readers have finished.

Writers are waiting and the readers don’t have priority any more.

Writers have to wait until the waiting readers have finished.

Exercise: come up with a better performing version using condition variables! Introduce an upper bound of k readers!
Reader/writer lock details

A reader/writer lock implementation ("not our problem") usually gives priority to writers:

- Once a writer blocks, no readers arriving later will get the lock before the writer
- Otherwise an insert could starve

Re-entrant?

- Mostly an orthogonal issue
- But some libraries support upgrading from reader to writer
In Java

Java’s \texttt{synchronized} statement does not support readers/writer

Instead, library
\texttt{java.util.concurrent.locks.ReentrantReadWriteLock}

Different interface: methods \texttt{readLock} and \texttt{writeLock} return objects that themselves have \texttt{lock} and \texttt{unlock} methods

Does \texttt{not} have writer priority or reader-to-writer upgrading

- Always read the documentation
LOCK GRANULARITY

Literature: Herlihy – Chapter 9
The Five-Fold Path

- Coarse-grained locking
- Fine-grained locking
- Optimistic synchronization (locking)
- Lazy synchronization (locking)

- Next lecture: Lock-free synchronization
Running Example: Sequential List Based Set

Add, Remove, and Find unique elements in a sorted linked list.

- add(c)

- remove(c)
Set and Node

```java
public class Set<T> {

    private class Node {
        T item;
        int key;
        Node next;
    }

    private Node head;
    private Node tail;

    public boolean add(T x) {...};
    public boolean remove(T x) {...};
    public boolean contains(T x) {...};
}
```

Note that the list is not "in place" but provides references to its items.
Coarse Grained Locking

```java
public synchronized boolean add(T x) {...};
public synchronized boolean remove(T x) {...};
public synchronized boolean contains(T x) {...};
```

Simple, but a bottleneck for all threads.
Fine grained Locking

Often more **intricate** than visible at a first sight
- requires careful consideration of special cases

Idea: split object into pieces with separate locks
- no mutual exclusion for algorithms on disjoint pieces
Let's try this

remove(c)

Is this ok?
Let's try this

Thread A: remove(c)
Thread B: remove(b)

c not deleted! 😞
What's the problem?

- When deleting, the next field of next is read, i.e., next also has to be protected.
- A thread needs to lock both, predecessor and the node to be deleted (hand-over-hand locking).
public boolean remove(T item) {
    Node pred = null, curr = null;
    int key = item.hashCode();
    head.lock();
    try {
        pred = head;
        curr = pred.next;
        curr.lock();
        try {
            // find and remove
        } finally {
            curr.unlock();
        }
    } finally {
        pred.unlock();
    }

    while (curr.key < key) {
        pred.unlock();
        pred = curr; // pred still locked
        curr = curr.next;
        curr.lock(); // lock hand over hand
    }

    if (curr.key == key) {
        pred.next = curr.next; // delete
        return true;
    }

    return false;
}
Disadvantages?

- Potentially long sequence of acquire / release before the intended action can take place
- One (slow) thread locking "early nodes" can block another thread wanting to acquire "late nodes"
Idea

Find nodes without locking,
• then lock nodes and
• check that everything is ok (validation)

e.g., add(c)
Validation: what could go wrong?

Thread A: add(c)
A: find insertion point

Thread B: remove(b)
A: lock
A: validate: rescan
A: b not reachable
\(\rightarrow\) return false
Validation: what could go wrong?

Thread A: add(c)

A: find insertion point

Thread B: insert(b')

A: lock
A: validate: rescan
A: d != succ(b)
→ return false
private Boolean validate(Node pred, Node curr) {
    Node node = head;
    while (node.key <= pred.key) { // reachable?
        if (node == pred)
            return pred.next == curr; // correct?
        node = node.next;
    }
    return false;
}
Correctness (remove c)

If

• nodes b and c both locked
• node b still reachable from head
• node c still successor to b

then

• neither is in the process of being deleted

⇒ ok to delete and return true
Correctness (remove c)

If

• nodes b and d both locked
• node b still reachable from head
• node d still successor to b

then

• neither is in the process of being deleted, therefore a new element c must appear between b and d
• no thread can add between b and d: c cannot have appeared after our locking

⇒ ok to return false
Optimistic List

Good:
- No contention on traversals.
- Traversals are wait-free.
- Less lock acquisitions.

Bad:
- Need to traverse list twice
- The contains() method needs to acquire locks
- Not starvation-free

Wait-Free:
Every call finishes in a finite number of steps (NEVER waits for other threads).

Is the optimistic list starvation-free? Why/why not?
LAZY SYNCHRONISATION

Laziness

The quality that makes you go to great effort to reduce overall energy expenditure [...] the first great virtue of a programmer.

Larry Wall, Programming Perl
(emphasis mine)
Lazy List

Like optimistic list but

• Scan only once
• Contains() never locks

How?

• Removing nodes causes trouble
• Use deleted-markers → invariant: every unmarked node is reachable!
• Remove nodes «lazily» after marking
Lazy List: Remove

Scan list (as before)
Lock predecessor and current (as before)
Logical delete: mark current node as removed
Physical delete: redirect predecessor's next

e.g., remove(c)
Invariant

If a node is not marked then
• It is reachable from head
• And reachable from its predecessor

A: remove(c)

lock
check if b or c are marked
not marked? ok to delete:
  mark c
  delete c
public boolean remove(T item) {
    int key = item.hashCode();
    while (true) { // optimistic, retry
        Node pred = this.head;
        Node curr = head.next;
        while (curr.key < key) {
            pred = curr;
            curr = curr.next;
        }
        pred.lock();
        try {
            curr.lock();
            try {
                // remove or not
            } finally {
                curr.unlock();
            }
        } finally {
            pred.unlock();
        }
    }
}

if (!pred.marked &amp;&amp; !curr.marked &amp;&amp; pred.next == curr) {
    if (curr.key != key)
        return false;
    else {
        curr.marked = true; // logically remove
        pred.next = curr.next; // physically remove
        return true;
    }
}
public boolean contains(T item) {
    int key = item.hashCode();
    Node curr = this.head;
    while (curr.key < key) {
        curr = curr.next;
    }
    return curr.key == key && !curr.marked;
}
More practical:
Lazy Skiplists
Skiplist

- Collection of elements (without duplicates)
- Interface:
  - add \hspace{1em} // add an element
  - remove \hspace{1em} // remove an element
  - find \hspace{1em} // search an element

- Assumption:
  - Many calls to find()
  - Fewer calls to add() and much fewer calls to remove()
How about balanced trees?

- AVL trees, red-black trees, treaps, ...
  - rebalancing after add and remove expensive
  - rebalancing is a *global* operation (potentially changing the whole tree)
  - particularly hard to implement in a lock-free way.

- → SkipList
Skip Lists

- Sorted multi-level list
- Node height probabilistic, e.g., $\mathbb{P}(\text{height} = n) = 0.5^n$, no rebalancing
Skip List Property

- Sublist relationship between levels: higher level lists are always contained in lower-level lists. Lowest level is entire list.
Searching

- Logarithmic Search (with high probability)
- Example: Search for 8
Sequential Find

- // find node with value x
- // return -1 if not found, node level otherwise
- // pre = array of predecessor node for all levels
- // succ = array of successor node for all levels
- int find(T x, Node<T>[] pre, Node<T>[] succ)

- e.g., x = 8
- returns 0
Sequential Find

- // find node with value x
- // return -1 if not found, node level otherwise
- // pre = array of predecessor node for all levels
- // succ = array of successor node for all levels
- int find(T x, Node<T>[] pre, Node<T>[] succ)

- e.g., x = 6
- returns -1
add (6) – with four levels!

- Find predecessors (lock-free)
- Lock predecessors
- Validate (cf. Lazy Synchronisation)
add (6)

- Find predecessors (lock-free)
- Lock predecessors
- Validate (cf. Lazy Synchronisation)

- Splice
- mark fully linked
- Unlock
remove(5)

- find predecessors
- lock victim
- logically remove victim (mark)

- Lock predecessors and validate
remove(5)

- find predecessors
- lock victim
- logically remove victim (mark)

- Lock predecessors and validate
- physically remove
- unlock
contains(8)

- sequential find() & not logically removed & fully linked
- even if other nodes are removed, it stays reachable
- contains is *wait-free* (while add and remove are not)
Skiplist

- Practical parallel datastructure
- Code in book (latest revision!) – 139 lines
  - Too much to discuss in detail here
- Review and implement as exercise