Synchronizing Data Structures
Overview

- caches and atomics
- list-based set
- memory reclamation
- Adaptive Radix Tree
- B-tree
- Bw-tree
- split-ordered list
- hardware transactional memory
modern CPUs consist of multiple CPU cores and

- per-core registers
- per-core write buffers
- per-core caches (L1, L2)
- shared cache (L3)
- shared main memory
Cache Organization

- caches are organized in fixed-size chunks called *cache lines*
- on Intel CPUs a cache line is 64 bytes
- data accesses go through cache, which is transparently managed by the CPUs
- caches implement a replacement strategy to evict pages
- associativity: how many possible cache locations does each memory location have?
Cache Coherency Protocol

- although cores have private caches, the CPU tries to hide this fact
- CPU manages caches and provides the illusion of a single main memory using a *cache coherency protocol*
- example: MESI protocol, which has the following states:
  - *Modified*: cache line is only in current cache and has been modified
  - *Exclusive*: cache line is only in current cache and has not been modified
  - *Shared*: cache line is in multiple caches
  - *Invalid*: cache line is unused
- Intel uses the MESIF protocol, with an additional *Forward* state
- *Forward* is a special *Shared* state indicating a designated “responder”
Optimizations

• both compilers and CPUs reorder instructions, eliminate code, keep data in register, etc.
• these optimizations are sometimes crucial for performance
• for single-threaded execution, compilers and CPUs guarantee that the semantics of the program is unaffected by these optimizations (as if executed in program order)
• with multiple threads, however, a thread may observe these “side effects”
• in order to write correct multi-threaded programs, synchronization primitives must be used
Example

```c
int global(0);

void thread1() {
    while (true) {
        while (global % 2 == 1); // wait
        printf("ping\n");
        global ++;
    }
}

void thread2() {
    while (true) {
        while (global % 2 == 0); // wait
        printf("pong\n");
        global ++;
    }
}
```
C++11 Memory Model

- accessing a shared variable by multiple threads where at least thread is a writer is a race condition
- according to the C++11 standard, race conditions are *undefined behavior*
- depending on the compiler and optimization level, undefined behavior may cause *any* result/outcome
- to avoid undefined behavior when accessing shared data one has to use the `std::atomic` type\(^1\)
- atomics provide atomic load/stores (no torn writes), and well-defined ordering semantics

\(^1\) `std::atomic` is similar to Java's `volatile` keyword but different from C++'s `volatile`
Atomic Operations in C++11

- compare-and-swap (CAS): bool std::atomic_compare_exchange_strong(T& expected, T desired)
- there is also a weak CAS variant that may fail even if expected equals desired, on x86-64 both variants generate the same code
- exchange: std::exchange(T desired)
- arithmetic: addition, subtraction
- logical: and/or/xor
Naive Spinlock (Exchange)

```cpp
struct NaiveSpinlock {
    std::atomic<int> data;

    NaiveSpinlock() : data(0) {}

    void lock() {
        while (data.exchange(1) == 1);
    }

    void unlock() {
        data.store(0); // same as data = 0
    }
};
```
Naive Spinlock (CAS)

```cpp
struct NaiveSpinlock {
    std::atomic<int> data;

    NaiveSpinlock() : data(0) {}  

    void lock() {
        int expected;
        do {
            expected = 0;
        } while (!data.compare_exchange_strong(expected, 1));
    }

    void unlock() {
        data.store(0); // same as data = 0
    }
};
```
Sequential Consistency and Beyond

- by default, operations on `std::atomic` types guarantee *sequential consistency*
- non-atomic loads and stores are not reordered around atomics
- this is often what you want
- all `std::atomic` operations take one or two optional `memory_order` parameter(s)
- allows one to provide less strong guarantees (but potentially higher performance), the most useful ones on x86-64 are:
  - `std::memory_order::memory_order_seq_cst`: sequentially consistent (the default)
  - `std::memory_order::memory_order_release` (for stores):
    may move non-atomic operations before the store (i.e., the visibility of the store can be delayed)
  - `std::memory_order::memory_order_relaxed`:
    guarantees atomicity but no ordering guarantees

- nice tutorial: https://assets.bitbashing.io/papers/lockless.pdf

---

2 sometimes useful for data structures that have been built concurrently but are later immutable
Spinlock

```cpp
struct Spinlock {
    std::atomic<int> data;
    Spinlock() : data(0) {} 

    void lock() {
        for (unsigned k = 0; !try_lock(); ++k)
            yield(k);
    }

    bool try_lock() {
        int expected = 0;
        return data.compare_exchange_strong(expected, 1);
    }

    void unlock() {
        data.store(0, std::memory_order::memory_order_release);
    }

    void yield();
};
```
Yielding

// adapted from Boost library
void Spinlock::yield(unsigned k) {
    if (k < 4) {
    }
    else if (k < 16) {
        _mm_pause();
    }
    else if ((k < 32) || (k & 1)) {
        sched_yield();
    }
    else {
        struct timespec rqtp = { 0, 0 };
        rqtp.tv_sec = 0;
        rqtp.tv_nsec = 1000;
        nanosleep(&rqtp, 0);
    }
}
Lock Flavors

- there are many different lock implementations
- C++: `std::mutex`, `std::recursive_mutex`
- `pthreads`: `pthread_mutex_t`, `pthread_rwlock_t`
- on Linux blocking locks are based on the `futex` system call
- [https://www.threadingbuildingblocks.org/docs/help/tbb_userguide/Mutex_Flavors.html](https://www.threadingbuildingblocks.org/docs/help/tbb_userguide/Mutex_Flavors.html)

<table>
<thead>
<tr>
<th>TBB type</th>
<th>Scalable</th>
<th>Fair</th>
<th>Recursive</th>
<th>Long Wait</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex</td>
<td>OS dependent</td>
<td>OS dependent</td>
<td>no</td>
<td>blocks</td>
<td>≥ 3 words</td>
</tr>
<tr>
<td>recursive_mutex</td>
<td>OS dependent</td>
<td>OS dependent</td>
<td>yes</td>
<td>blocks</td>
<td>≥ 3 words</td>
</tr>
<tr>
<td>spin_mutex</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yields</td>
<td>1 byte</td>
</tr>
<tr>
<td>speculative_spin_mutex</td>
<td>HW dependent</td>
<td>no</td>
<td>no</td>
<td>yields</td>
<td>2 cache lines</td>
</tr>
<tr>
<td>queuing_mutex</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yields</td>
<td>1 word</td>
</tr>
<tr>
<td>spin_rwlock</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yields</td>
<td>1 word</td>
</tr>
<tr>
<td>speculative_spin_rwlock</td>
<td>HW dependent</td>
<td>no</td>
<td>no</td>
<td>yields</td>
<td>3 cache lines</td>
</tr>
<tr>
<td>queuing_rwlock</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yields</td>
<td>1 word</td>
</tr>
<tr>
<td>null_mutex</td>
<td>moot</td>
<td>yes</td>
<td>yes</td>
<td>never</td>
<td>empty</td>
</tr>
<tr>
<td>null_rwlock</td>
<td>moot</td>
<td>yes</td>
<td>yes</td>
<td>never</td>
<td>empty</td>
</tr>
</tbody>
</table>
Atomics on x86-64

- atomic operations only work on 1, 2, 4, 8, or 16 byte data that is aligned
- atomic operations use lock instruction prefix
- CAS: `lock cmpxchg`
- exchange: `xchg` (always implicitly locked)
- read-modify-write: `lock add`
- memory order can be controlled using fences (also known as barriers): `_mm_lfence()`, `_mm_sfence()`, `_mm_mfence()`
- locked instructions imply full barrier
- fences are very hard to use, but atomics generally make this unnecessary
x86-64 Memory Model

- x86-64 implements Total Store Order (TSO), which is a strong memory model
- this means that x86 mostly executes the machine code as given
- loads are not reordered with respect to other loads, writes are not reordered with respect to other writes
- however, writes are buffered (in order to hide the L1 write latency), and reads are allowed to bypass writes
- a fence or a locked write operations will flush the write buffer (but will not flush the cache)
- important benefit from TSO: sequentially consistent loads do not require fences
Weakly-Ordered Hardware

- many microarchitectures (e.g., ARM) are weakly-ordered
- on the one hand, on such systems many explicit fences are necessary
- on the other hand, the CPU has more freedom to reorder
- ARMv8 implements acquire/release semantics in hardware (ldaq and stx instructions)
- https://en.wikipedia.org/wiki/Memory_ordering:

<table>
<thead>
<tr>
<th></th>
<th>Alpha</th>
<th>ARM v7</th>
<th>IBM POWER</th>
<th>zArch</th>
<th>SPARC RMO</th>
<th>SPARC PSO</th>
<th>SPARC TSO</th>
<th>Intel x86</th>
<th>Intel x86-64</th>
<th>Intel IA-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads reord. after loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loads reord. after stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stores reord. after stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stores reord. after loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomic reord. with loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atomic reord. with stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dependent loads reord.</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incoh. instr. cache pipel.</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Concurrent List-Based Set

- operations: insert(key), remove(key), contains(key)
- keys are stored in a (single-)linked list sorted by key
- head and tail are always there ("sentinel" elements)
Why CAS Is Not Enough

- thread A: remove(7)
- thread B: insert(9)
Coarse-Grained Locking

- use a single lock to protect the entire data structure
  - very easy to implement
  - does not scale at all
Synchronizing Data Structures

Concurrent List-Based Set

Lock Coupling

- also called “hand-over-hand locking” or “crabbing”
- hold at most two locks at a time
- interleave lock acquisitions/release pair-wise
- may use read/write locks to allow for concurrent readers

+ easy to implement
+ no restarts
− does not scale
Synchronizing Data Structures
Concurrent List-Based Set

Optimistic

- traverse list optimistically without taking any locks
- lock 2 nodes (predecessor and current)
- validate: traverse list again and check that predecessor is still reachable and points to current
- if validation fails, unlock and restart

+ lock contention unlikely
- must traverse list twice
- readers acquire locks
Optimistic Lock Coupling

- general technique that can be applied to many data structures (e.g., ART, B-tree)
- associate lock with update counter
- write:
  - acquire lock (exclude other writers)
  - increment counter when unlocking
  - do not acquire locks for nodes that are not modified (traverse like a reader)
- read:
  - do not acquire locks, proceed optimistically
  - detect concurrent modifications through counters (and restart if necessary)
  - can track changes across multiple nodes (lock coupling)

+ easy to implement
+ scalable
- restarts
Non-Blocking Algorithms

- killing a thread at any point of time should not affect consistency of the data structure (this precludes locks)
- non-blocking data structures may be beneficial for (soft) real-time applications
- classification:
  - wait-free: every operation is guaranteed to succeed (in a constant number of steps)
  - lock-free: overall progress is guaranteed (some operations succeed, while others may not finish)
  - obstruction-free: progress is only guaranteed if there is no interference from other threads
Read-Optimized Write Exclusion (Lazy)

- contains is wait-free
- add/remove traverse list only once (as long as there is no contention)
- add marker to each node for logically deleting a key
- invariant: every unmarked node is reachable
- contains: no need to validate; if a key is not found or is marked, the key is not in the set
- add/remove:
  1. lock predecessor and current
  2. check that both are unmarked and that predecessor points to current
  3. remove marks first, then updates next pointer of predecessor

+ no restarts
+ scalable
- insert/remove lock
Lock-Free List

- insert and remove are lock-free, contains is wait-free
- remove: marks node for deletion, but does not physically remove it
- marker is stored within the next pointer (by stealing a bit of the pointer)
- insert and remove:
  - do not traverse marked node, but physically remove it during traversal using CAS
  - if this CAS fails, restart from head
- contains traverses marked nodes (but needs same check as Lazy variant)

+ contains always succeeds
+ scalable
- insert/remove may restart
Synchronization Paradigms

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
<th>Scalability</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse-Grained Locking</td>
<td>+</td>
<td>--</td>
<td>+</td>
</tr>
<tr>
<td>Lock Coupling</td>
<td>+</td>
<td>-</td>
<td>~</td>
</tr>
<tr>
<td>Optimistic</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Optimistic Lock Coupling</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>ROWEX</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Lock-Free</td>
<td>--</td>
<td>++</td>
<td>--</td>
</tr>
</tbody>
</table>
Memory Reclamation for Lock-Free Data Structures

- after deleting a node in a lock-free data structure, readers might still be accessing that node
- freeing/reusing that node immediately can cause correctness problems and/or crashes
- one must not physically delete a node unless it is ensured that no threads can access that node
- how long should one wait until the node is physically deleted?
- garbage-collected languages usually do not have this particular problem
Reference Counting

- associate every node with an atomic counter
- effectively results in similar behavior as locks

+ easy to use
- does not scale
Hazard Pointers

- observation: most lock-free operations only require a bounded number of node pointers at any point in time
- during traversal, one can store these hazard pointers into thread-local locations
- before physically removing a node, check if any thread has a hazard pointer referencing that node

+ non-blocking
  - high overhead due to required fences
  - error-prone (requires invasive changes to data structure)
Epoch-Based Memory Reclamation

- global epoch counter (incremented infrequently)
- per-thread, local epoch counters
- before every operation: enter epoch by setting local epoch to global epoch
- after every operation: set local epoch to $\infty$ indicating that this thread is not accessing the data structure
- tag nodes to be deleted with current global epoch
- defer physically deleting nodes
- it is safe deleting nodes are older than the minimum of all local epochs

+ low overhead
+ no need to change data structure code
- a single slow/blocking thread may prevent any memory reclamation
ABA Problem

- A compare-and-swap on a pointer structure may succeed even though the pointer has been changed in the mean time (from A to B back to A).
- This is a correctness issue for some lock-free data structures (e.g., queues).
- Whether this problem occurs depends on data structure and the memory reclamation strategy.
Adaptive Radix Tree (ART)

**Header**
(in front of each node)

<table>
<thead>
<tr>
<th>prefixCount</th>
<th>count</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>N4</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Node4**

<table>
<thead>
<tr>
<th>key</th>
<th>child pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>13 129 130</td>
<td></td>
</tr>
</tbody>
</table>

**Node48**

<table>
<thead>
<tr>
<th>child index</th>
<th>child pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>255</td>
</tr>
</tbody>
</table>

**Node16**

<table>
<thead>
<tr>
<th>key</th>
<th>child pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 8 9</td>
<td>255 TID</td>
</tr>
</tbody>
</table>

**Node256**

<table>
<thead>
<tr>
<th>child pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>255 TID</td>
</tr>
</tbody>
</table>
Path Compression and Lazy Expansion

Path compression: merge one-way node into child node.

Lazy expansion: remove path to single leaf.
Lock Coupling

- easy to apply to ART
- modifications only change 1 node and (potentially) its parent
- can use read/write locks to allow for more concurrency
Optimistic Lock Coupling

- add lock and version to each node
- how to detect that root node changed?
  1. extra optimistic lock for root pointer (outside of any node)
  2. always keep the same Node256 as root node (similar to static head element in list-based set)
  3. after reading the version of the current root, validate that the root pointer still points to this node
Optimistic Lock Coupling (2)

**traditional**

1. lock node A
2. search node A
3. lock node B
4. unlock node A
5. search node B
6. lock node C
7. unlock node B
8. search node C
9. unlock node B

**optimistic**

1. lock node A
   - A
2. search node A
   - B
3. lock node B
4. unlock node A
5. search node B
6. lock node C
7. unlock node B
8. search node C
9. unlock node B
Optimistic Lock Coupling (2)

**traditional**

1. lock node A
2. search node A
3. lock node B
4. unlock node A
5. search node B
6. lock node C
7. unlock node B
8. search node C
9. unlock node B

**optimistic**

1. lock node A
2. search node A
3. lock node B
4. unlock node A
5. search node B
6. lock node C
7. unlock node B
8. search node C
9. unlock node B
Optimistic Lock Coupling (2)

**traditional**

1. lock node A
2. search node A
3. lock node B
4. unlock node A
5. search node B
6. lock node C
7. unlock node B
8. search node C
9. unlock node B

**optimistic**

1. read version v3
2. search node A

A

B

C
Optimistic Lock Coupling (2)

**traditional**

1. lock node A
2. search node A
3. lock node B
4. unlock node A
5. search node B
6. lock node C
7. unlock node B
8. search node C
9. unlock node B

**optimistic**

1. read version v3
2. search node A
3. read version v7
4. re-check version v3
5. search node B
6. search node C

Optimistic Lock Coupling (2)

**traditional**

1. lock node A
2. search node A
3. lock node B
4. unlock node A
5. search node B
6. lock node C
7. unlock node B
8. search node C
9. unlock node B

**optimistic**

1. read version v3
2. search node A
3. read version v7
4. re-check version v3
5. search node B
6. read version v5
7. re-check version v7
8. search node C
9. re-check version v5
Synchronizing Data Structures

Adaptive Radix Tree

Lock Coupling Vs. Optimistic Lock Coupling

```java
lookup(key, node, level, parent)
    readLock(node)
    if parent != null
        unlock(parent)
    // check if prefix matches, may increment level
    if !prefixMatches(node, key, level)
        unlock(node)
        return null // key not found
    // find child
    nextNode = node.findChild(key[level])
    if isLeaf(nextNode)
        value = getLeafValue(nextNode)
        unlock(node)
        return value // key found
    if nextNode == null
        unlock(node)
        return null // key not found
    // recurse to next level
    return lookup(key, nextNode, level+1, node)
```

### Optimistic Lock Coupling

```java
lookupOpt(key, node, level, parent, versionParent)
    version = readLockOrRestart(node)
    if parent != null
        readUnlockOrRestart(parent, versionParent)
    // check if prefix matches, may increment level
    if !prefixMatches(node, key, level)
        readUnlockOrRestart(node, version)
        return null // key not found
    // find child
    nextNode = node.findChild(key[level])
    checkOrRestart(node, version)
    if isLeaf(nextNode)
        value = getLeafValue(nextNode)
        readUnlockOrRestart(node, version)
        return value // key found
    if nextNode == null
        readUnlockOrRestart(node, version)
        return null // key not found
    // recurse to next level
    return lookupOpt(key, nextNode, level+1, node, version)
```

Insert using Optimistic Lock Coupling

```
insertOpt(key, value, node, level, parent, parentVersion)
version = readLockOrRestart(node)
if !prefixMatches(node, key, level)
    upgradeToWriteLockOrRestart(parent, parentVersion)
    upgradeToWriteLockOrRestart(node, version, parent)
    insertSplitPrefix(key, value, node, level, parent)
writeUnlock(node)
writeUnlock(parent)
return
nextNode = node.findChild(key[level])
checkOrRestart(node, version)
if nextNode == null
    if node.isFull()
        upgradeToWriteLockOrRestart(parent, parentVersion)
        upgradeToWriteLockOrRestart(node, version, parent)
        insertAndGrow(key, value, node, parent)
        writeUnlockObsolete(node)
        writeUnlock(parent)
    else
        upgradeToWriteLockOrRestart(node, version)
        readUnlockOrRestart(parent, parentVersion, node)
        node.insert(key, value)
        writeUnlock(node)
    return
if parent != null
    readUnlockOrRestart(parent, parentVersion)
    if isLeaf(nextNode)
        upgradeToWriteLockOrRestart(node, version)
        insertExpandLeaf(key, value, nextNode, node, parent)
        writeUnlock(node)
    return
// recurse to next level
insertOpt(key, value, nextNode, level+1, node, version)
return
```
ART with ROWEX

- **local modifications:**
  - make key and child pointer accesses atomic (`std::atomic`)
  - make `Node4` and `Node16` unsorted and append-only

- **grow/shrink a node:**
  - lock node and its parent
  - create new node and copy entries
  - set parent pointer to the new node
  - set old node to obsolete
  - unlock nodes

- **path compression:**
  - modify 16-byte prefix atomically
  - add `level` field to each node

- much more complicated than OLC
Path Compression with ROWEX

- `insert("AS")`:

1. **Install new node**

2. **Update prefix**

   - **level prefix**
     - 0
     - 1
     - 2

   - **prefix level**
     - 0
     - 1
     - 2

   - **nodes**
     - `R`, `S`
     - `AS`
     - `ARE`, `ART`

   - **edges**
     - `A`, `I`, `E`, `T`
Lookup (50M 8B Integers)

- no sync.
- lock coupling
- Opt. Lock Coupling
- ROWEX
- HTM
- Masstree
Insert (50M 8B Integers)
Contention

![Graph showing contention for lookup and insert + remove operations across different Zipf parameters.

**Lookup**
- ROWEX
- Opt. Lock Coupling
- HTM
- Masstree

**Insert + Remove**
- HTM
- ROWEX
- OLC
- Masstree

The graphs illustrate the performance of various data structures under different Zipf parameters (skew) for M operations per second and lock coupling.]
Synchronizing B-trees

- we assume B+-trees (inner nodes only store pointers, not values)
- for insert/delete there are two cases:
  - single-node change (normal, frequent case)
  - structure modification operation (during split/merge, infrequent)
Lock Coupling

- lock coupling can be applied
- need additional lock for root pointer
- however, structure modification operations may propagate up multiple levels (locking 2 levels)
- solutions:
  - eagerly split full inner nodes during traversal (good solution for fixed-size keys)
  - restart operation from the root holding all locks
Optimistic Lock Coupling

- optimistic lock coupling can be applied (after solving the problem discussed on the previous slide)
- obsolete marker for each node
- additional issues:
  - infinite loops: one has to ensure that the intra-node search may not terminate due to concurrent modifications
  - segmentation faults/invalid behavior: a pointer read from a node may be invalid
- both issues can be solved, making OLC a very good technique for B-trees
B-Link Tree

- B-tree synchronization with only a single lock at a time
- observation: as long as there are only splits (no merges), the key that is being searched may have moved to a right neighbor
- solution: add *next* pointers to inner and leaf nodes
- operations may have to check neighbor(s)
- fence keys may help determining whether it is necessary to traverse to neighbor
B-trees ROWEX?

- use B-Link tree idea to enable readers to handle concurrent splits
- lock-less per-node operations can be implemented using a design similar to slotted pages
- not possible to keep keys sorted (must scan all keys)
- not clear whether this is a useful design
References

- *The art of multiprocessor programming*, Herlihy and Nir Shavit, Morgan Kaufmann, 2008
- *The adaptive radix tree: ARTful indexing for main-memory databases*, Leis and Kemper and Neumann, ICDE 2013
- *The Bw-Tree: A B-tree for new hardware platforms*, Levandoski and Lomet and Sengupta, ICDE 2013
- *The ART of practical synchronization*, Leis and Scheibner and Kemper and Neumann, DaMoN 2016