**Lecture1b multiprocessor**

**Amdahl’s law**: speed up = 1/(1-p+p/n). p: num of concurrent process, n: num of processors.

**The moral**: make use of multiple processors(cores), minimize sequential part, reduce thread idle time.

**Lecture1c concurrency**

**Mimick multiple CPU with 1**: scheduling program one by one.

**Process**: abstraction representing program execution; *switch* when timer interrupt goes off, I/O, etc; CPU needs to store *process context*: program counter, stack pointer&register.

**State of process**: new ->ready, ready->running, waiting->ready, running->ready/waiting/terminated

**Address space**: unit of management of process’s virtual memory have: stack-temp.data extensible to lower virtual address; heap-memory allocated dynamically extensible to higher V.address; data section: global variable; text region: program code.

**Process creation**: require allocation: choosing host of the process, and execution environment: an address space with initialized content&open files, can be static/dynamic. Interprocess communication(IPC): require costly context switch

**Context switch**: p1:change memory map in MMU and flush TLB-> swap process from disk to main memory->p2:kernel-user.

**Thread**: smallest unit of CPU utilization, has program counter& stack(temp), but no data&code&files(share between thread). Communication between thread always through memory->no need IPC and context switch.

**User lvl thread lib**: cheap to create&destroy, blocking system call freezes process and other thread, cheap context switch.

**Kernel scheduled thread**: opposite of above.

**LWP**: run as process;concurrent thread in table; once find runnable thread: lock table in user mode to update and switch thread to user mode; doesn’t block system calls by pass 2 kernel.

**ProcessvsThread**: process isolated and inefficient, thread oppo.

**Thread lifecycle**: new->ready, ready->running, sleeping->ready, waiting->ready, running->sleeping/waiting/terminated.

**Thread safety**: solution: make variable thread-specific/ make state variable immutable/ use synchronization.

**Lecture2 mutual exclution**

**Model summary:** multiple threads, single shared memory with objects, and unpredictable asynchronous delays.

**Safety:** nothing bad(crash) happens ever, **mutual exclusion**(not both thread visit same variable)

**Liveness:** something good(wanted) happens eventually(if one wants in, it gets in, if both want in, one gets in), **no starvation**

**Mutual exclusion:** can’t be solved by transient communication (cell phone, maybe not there), and interrupt(cans, may run out of bits). It can be solved by one-bit shared variables(flag), that can be read or written

**Deadlock:** a state when all threads are waiting for another thread to give out lock, leading to program hangs&starvation.

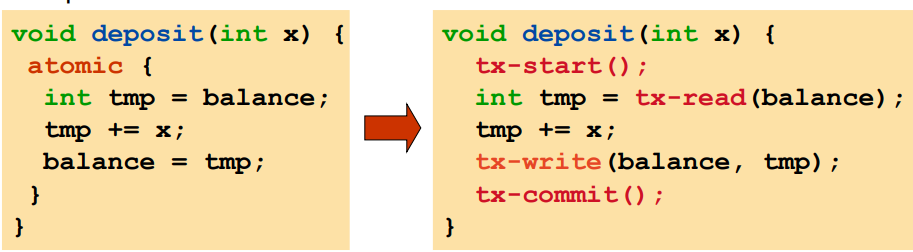
**Starvation:** one thread is unable to execute its useful part.

**Producer/consumer:** pet never enter pond if no food, bob never produce food if there is food left(safety property) , this can be solved by the can protocol(while(canisup){}, dotask(), resetcan()) to make sure alice is notified when there is food produced.

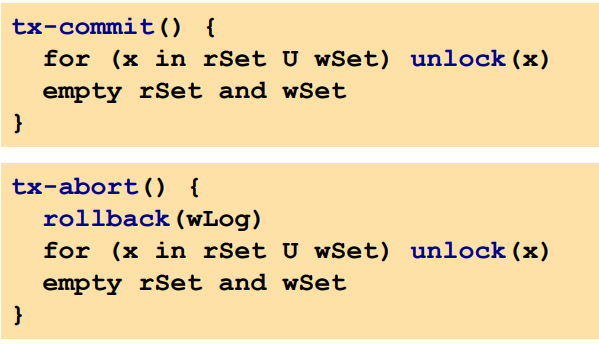
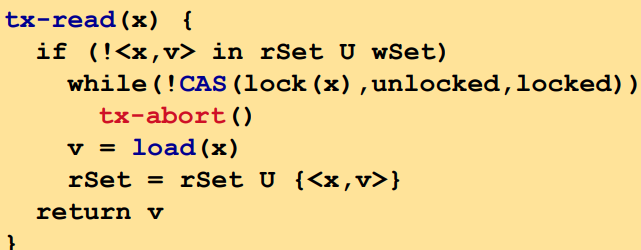
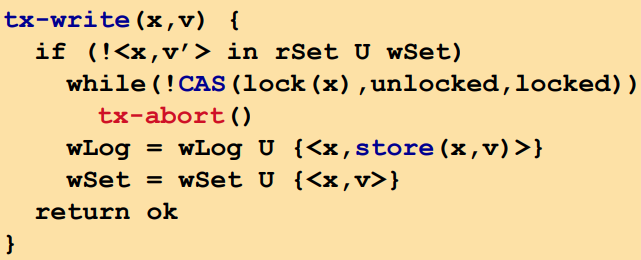
**Waiting thread:** if 1 participant is delayed, so is every thread else. But delays are common and unpredictable.

**Lecture3 transactional memory**

**Deadlock example:** when thread needs 2 locks to execute, but t1 got lockA, t2 got lockB, so they can never get next lock. Multiple locks acquired in arbitrary order may cause deadlock, multiple locks acquired in single shared order is safe(eg lexicographical order, memory address order).

**Transactional memory:** ‘atomic’ has a wrapper to act like a lock. ensures safety(atomicity)&liveness(no deadlock).

**Transaction:** a sequence of instructions executed atomically. Good for software engineering(simple, no deadlock&race) and performance(when no conflict, high parallelism and no locking overhead. ‘universal sync’

**Implementation eg:** tx-read(x)/tx-write(x,v): if unlocked, then acquire lock of x, else tx-abort();;tx-commit(): release all locks, clean up. 

**Starvation:** may if transactions keep aborting each other.

**Contention manager:** external module by transactional memory to arbitrate between conflict transactions.

**Irrevocability:** an action that can’t be undo. Usual trans. Can’t execute irrevocable actions. *Privatization* ensures a trans. is only one to access data, therefore guarantee it will commit, then it can execute irrevocable code safely.

**Expressiveness:** trans.use a balanced open-close block so locks can be ‘interleaved’(atomic{a(),b(),c()} = {lock()a() unlock(),lock()b()unlock(),lock()c()unlock()}. *Relaxed trans*.helps bridge gap between regular one and locks.

**Reusability:** simplify programming by extension and composition, not in traditional concurrent environment, but works with transactional memory.

**Software TM:** STM overhead was too high to speed up sequential performance, and it was not scaling to 8 threads because 4 hyperthreaded cores != 8 cores.

**Property of TM:** safe: ensure atomicity of trans. live:ensure a trans commits if exec in isolation for long enough and sometimes every trans eventually commits;; A trans can encapsulate multiple ones so no knowledge is priori necessary, which makes concurrent programming modular;;concurrent with TM is simpler, safer, faster.

**Lecture 4 synchrobench**

**Method for concurrency:** *lock*-block other thread access resource, slow thread may slow others;;*CopyonWrite(cow)*- writers create&update a resource copy, reader use concu- rrent copy without noticing;;*CompareandSwap(CAS)*-switch from 1 version to another, allow system to always make progress: if I can’t switch, someone else can, otherwise, I proceed;;transaction-threads try to access resource: if conf licts can’t be resolved, then abort, otherwise proceed.

**Contention**: 争夺, problem: growing core count induces con tention;;idea: relax structural invariants during load bursts eg: level distribution, Ologn depth, load factor;;minimize contention, postpone restructuring.

**New conflict detection:** *read-copy-update*:readers proceed uninterrupted, writers copy data to be modified, shares similarity with CopyOnWrite;;*PolymorphicTM*:multiple transaction semantics, cohabiting in same application.

**Synchrobench**: to evaluate synchronization technique and data structure algorithm on a common ground;;abstraction set, queue, map;;Why: macro-bench cant help find cause of data structure bottleneck. Profiling tools are architecture specific. Micro-bench is appealing but customized-does 50% update mean 25% modification? How does it speed up sequential performance?

**Status**: ~40 concurrent structure in java&c&c++, 1/3 are recent, 4 require change threading model, most open sourced and few from scratch;;architectures: SPARC, x86, Tilera;;sync tech: cas, lock, cow, read-copy-update, TM.

**Parameter**: t-threadnum, i-initsize, r-valuerange, u-updateratio, f-effectiveupdate, A-alt.values, U-unbalancerate, d-duration(ms), a-ratioOfWriteAll, s-ratioOfSnapshot, W-warmup(s), n-iterations

**Thread pinning**: COMP: ok for rapic peak performance, SCAT: ok for scability in TM. RCU: may not tolerate contention as in TM

**Reusability**: COW not tolerate contention as in TM.

**Conclusion**: CAS helps boost performance, but difficult to use efficiently. RCU/COW suffer from contention than TM, good for read only ops;;concurrent alg.can reach higher performance over energy ratio on manycore than on multicore.

**Week5 consistency criteria**

**Sequential specification**: object state is meaningful only between method calls, each method described in isolation, can add new methods without change description of old method.

**Concurrent specification**: method call is an interval and takes time, object may never between calls, method are not isolate, everything may interact with anything when adding method.

**Consistency model**: a contract between process(thread) and data store(memory) such that if process obey certain rules, data store promise to work correctly.

**Coherence vs consistency**: one memory location vs all threads. Coherent execution may not be consistent.

**Sequential consistency**: too strong due to expensive to imple. Alt: violate by default, or honored by explicit request . to bypass: read-load data into cache, write-update cache copy, write cache back into memory;;most I/O not sync, only sync explicit ones.

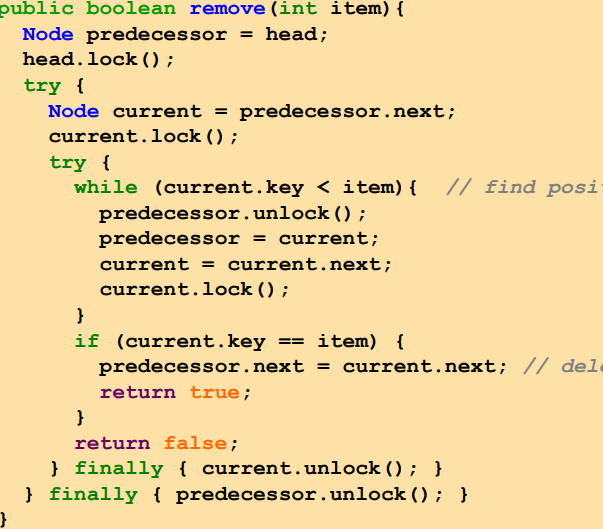
**DB transaction:** atomic-transaction take indivisibly, consistent- trans.doesnt violate system invariants, isolated-concurrent trans don’t interfere each other, durable-once trans commit, change is permenant.

**Memory trans:** only ACI, but another property prevent other trans from observing the transient state produced by a trans that will abort.

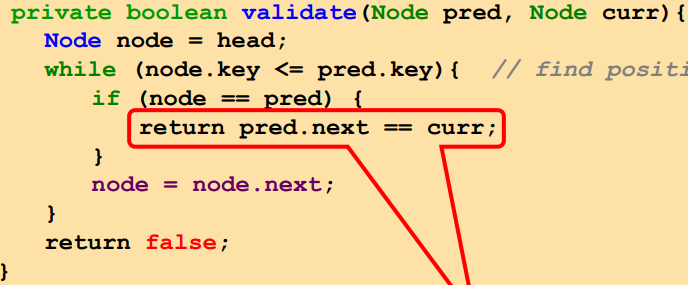
**Week6 coarse&hand over hand locking**

**Coarsegrained lock:** execution is sequential, simple implement, no starvation,but has bottleneck and don’t leverage concurrency

**finegrainedLock:** each lock protect a subpart of list.

**Handoverhand:** stary by lock 2 first node, then traverse, lock next and release previous until find correct node(prev, target).but long chain of acquire&release thus inefficient. 

**Week7 optimistic, lazy, versioned locking**

**Optimistic:** find node without lock, lock node, validation eg if b,c locked&&b still there&&b still point to c, then its ok to delete c. on exit from loop, if item present: curr=item, if item absent, curr = first>item. PRO: limited hot-spots, traversal is wait-free, fewer lock ops. CONs:need to traverse twice, read-only ops(contain) need lock.

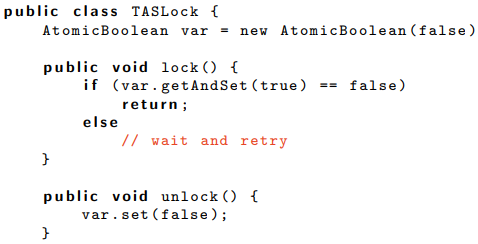
**Lazy lock:** like optimistic, but: scan once, contain(x)never locks. Logical delete: mark target as removed, then physical delete. *Validation*: no need to rescan list, only check pred&curr not marked, and pred->curr. 2 locks per update ops, lock->validate

**Process:**traverse;lock;validate;try-update;unlock. Why valid after lock: between find&lock, node may change. Problem: read only method need lock, and produce contention.

**Versioned:** traversal don’t need versioned lock, once item found, validate by read version and grab lock only if value is absent and version is unchanged. Traverse;validate;try-lock;update;unlock.

**Lazy vs version:** all update need lock vs only effective ops lock.

**Week8 spinlock**

**Spin:** immediate try to retake lock, waiting: stop executing until someone unlocks. 

**Locality principle:** if a thread access a byte, its likely to access it and its next byte again.

**Cache MSI protocol:** 1 line can have multi copies: original in memory, multi in cache. If one copy modified, verify: invalid: line is not in cache, shared: line is in cache and not modified, modified: line is in cache and modified;;if a line modified in 1 cache, its invalid in all caches, if a line shared in 1 cache, then its invalid or shared in other caches.

**Java memory model:** allow compiler to maximize performance by reordering program, and skipping program. Harmless in single thread, but dangerous in multi thread.

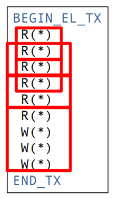
**Happensbefore:** limitation on compiler: if an action happen before another, they look ordered to observer;;each action in a thread happen before subsequent action;;the end of sync block happen before subsequent sync;;write on volatile variable happen before subsequent read;;all action before end of sync block or write to volatile, happens before it ->memory fence. The happens-before relationship is transitive(可以连锁)

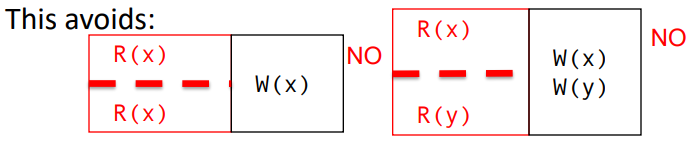
**Memory fence:** prevent optimization at byte compiler, just-in-time bytecode to native compiler, out-of-order execution engine of processer, write-buffer cache of processor;;but memory fence is mandatory in all synchronization types.

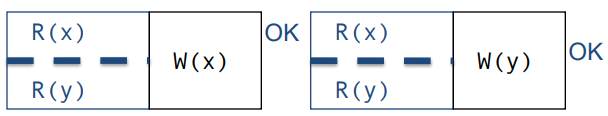
**Week10 elastic transactions**

**Goal:** easiness-simple adaptation of sequential programming, extensibility-code reusable,efficiency-high concurrency.

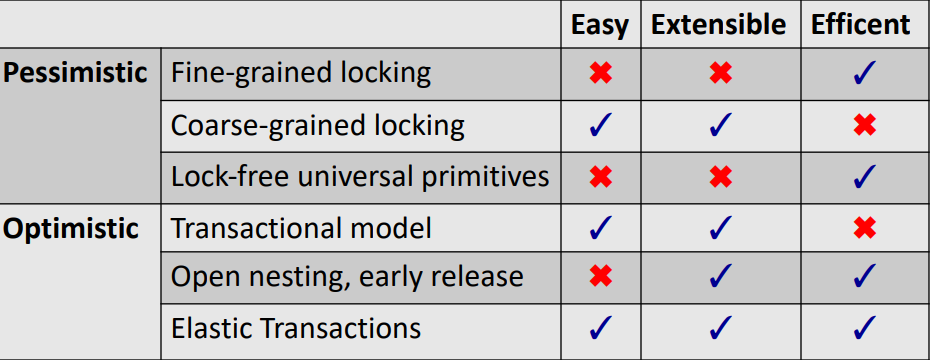
**Pessimistic vs optimistic:**valid(wait)->exec (locks)vs exec->valid (rollback) (transaction)

**Elastic transaction:** cutting trans.dynamically, abort if no way to cut. A cut e is well formed if: 1.all subsequence contain >=2 ops. 2. All writes of e belong to same subsequence3.all subsequence start with a read(unless e start write).

A cut e is consistent if common elements access by 2 consecutive sub-seq are not written. 



**Elastic opacity:** if there’re consistent cuts that replacing each elastic tx e by its resulting sub-seq form an opaque history.



**Polymorphic trans:** provide multi form of trans.: 1. Hand over had trans-elastic opacity, ensure atomicity of consecutive access 2. Snapshot trans-atomic snapshot sementics, use multi version concurrency control. 3. Irrevocable-use a global R/W lock to disallow other trans from running concurrently.4. opaque trans.

**Poly 2:** semantic of each tx form is preserved at runtime, deadlock free with lazy update. Makes concurrent program reusable&efficient, reusable package is developed in java, reusability has cost but not hamper scalability. **Week11 no hotspot skiplist and speculation free tree**

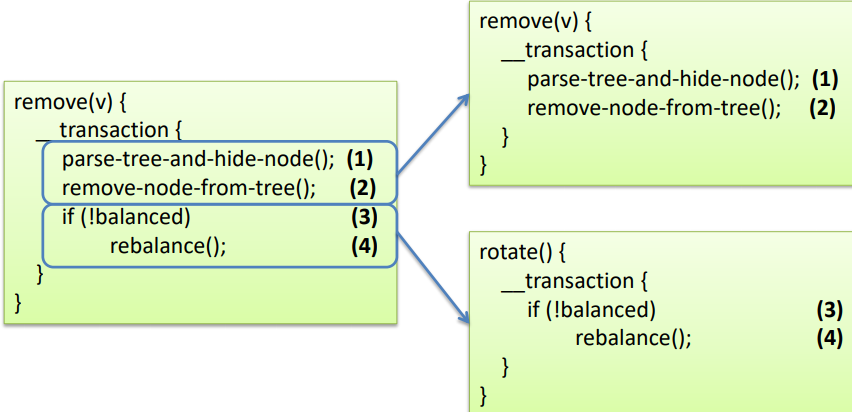
**Hotspot:** contention scale high. Idea to reduce: 1. Relaxing invariant(balanced tree) when burst.2.re-ensure invariant in the absence of contention.

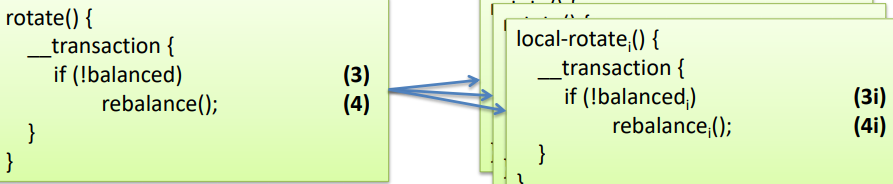
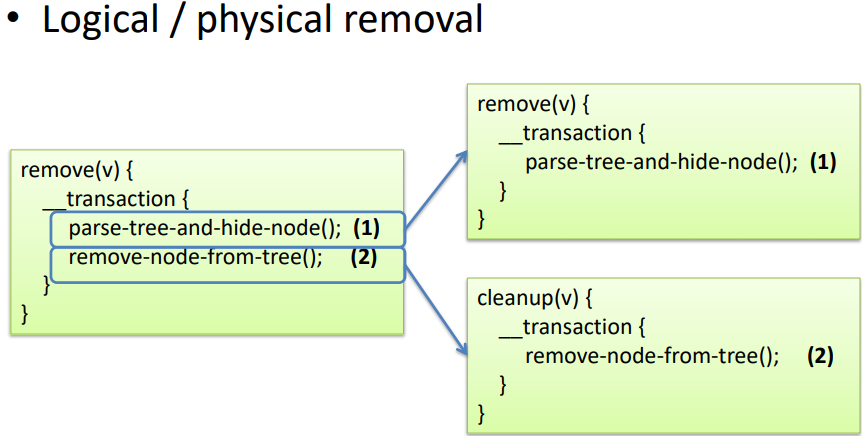
**Update decoupling:** eager abstract modification: update-return after updating at bottom lvl;;lazy and selective adaption: update-postpone adaption at high lvls, remove- chooses the least likely contended towers.

**No hotspot skiplist:** use method above,sync only at bottom

**Conclusion:** datastructure is bottleneck for multi-core;;skiplist is appealing for in-mem database;;non-block-ness for fault tolerance&heterogeneity support;;contention friendliness is most effective at high lvl of concurrency& high contention.

**Speculate:** eg, TM: wrap ops to ensure atomicity, shared memory access is wrapped, hope for no contention, if conflict can’t be resolved, abort the ops.

**Speculation friendly tree:** 1. Rotation decoupling

2. Deletion decoupling, 

3. Further optimisations. 1. Lightweight read(forget past read&limit conflict)-slight speedup but broken TM interface 2. Conflict tree rotation/removal: operation backtrack.

**Hardware transactional memory(HTM):** faster than STM, but red-black tree doesn’t scale in high contention.

**Week12 reproducibility**

**Reproducable artifact:** consistent, complete, well documented, easy to reuse.

**Need:** computing->aggregating->visualizing;;needs backup .

**Reproducing concurrency:** experiment of concurrent programs cant be reproduced – multi-threading is non – deterministic,thread tied OS,memory tied to architecture.

**To overcome:** take existing benchmark: list parameter& explain experiment setting.

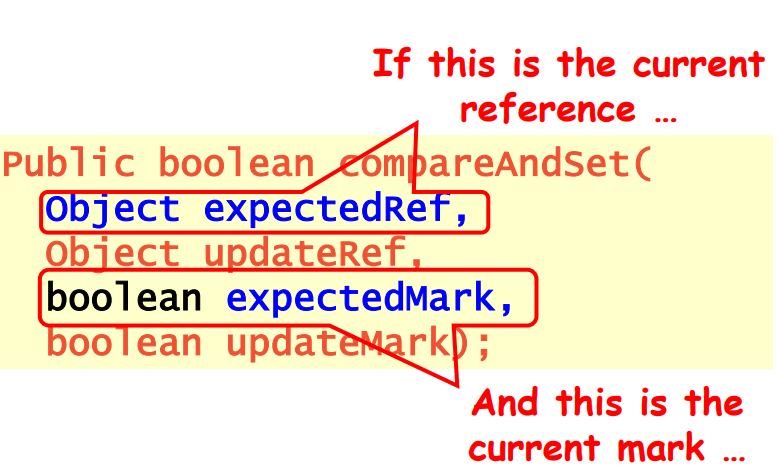
**Simultaneous multi-**threading: allow to have threads share pipeline, cpu, cache and exec multi instructions per cycle.

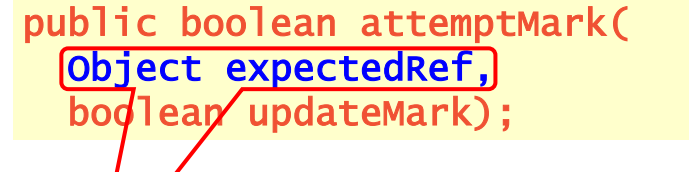
**Result:** lazy list is overly conservative(no scaling) , and versioned list is concurrency-optimal.

**To reproduce:** 1. Use well documented benchmark tool. 2. Gather sources(code, script, log, aggregated dataset), list setting(OS, architecture, environment, runtime obs, workloads), 3. Package everything, share code, submit.

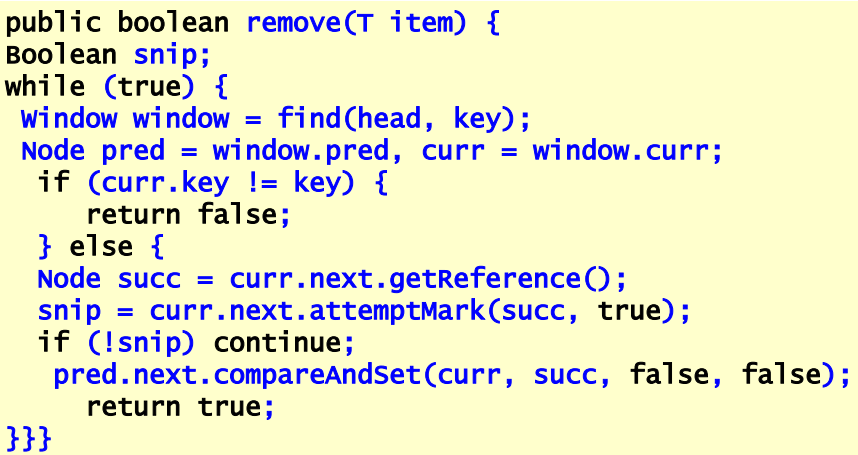
**Week9 lockfreedom**

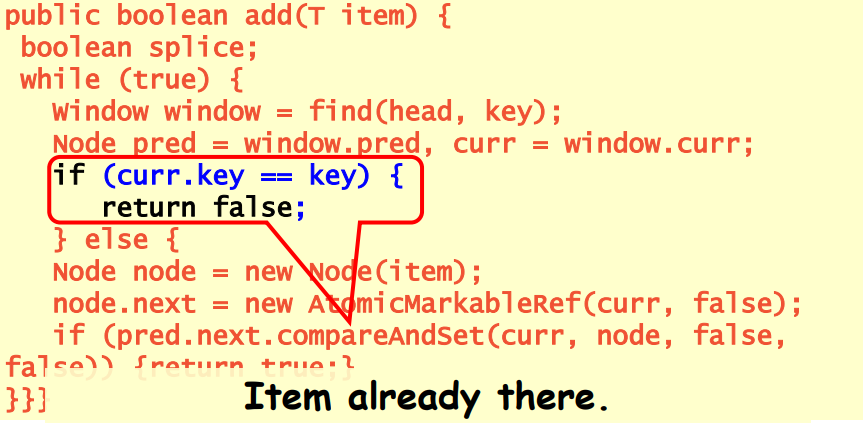
**Cas:** compare and set, need to prevent manipulation of removed node’s pointer, use AtomicMarkableReference to atomically swing reference&update flag. Remove is in 2 steps: set mark bit in next field, and redirect pred’s pointer.

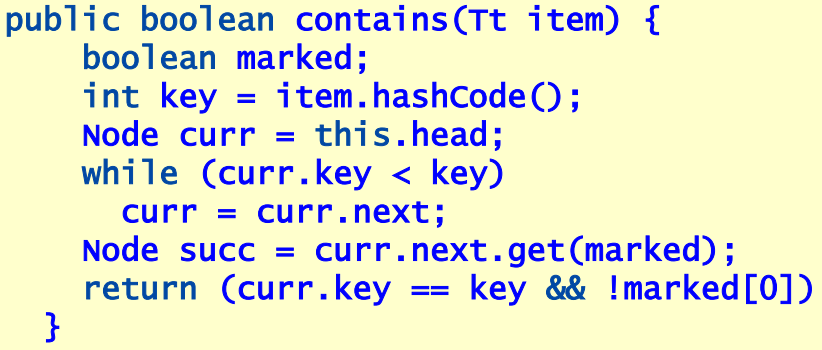
**Markable reference:** a Boolean mark bit next to address, return mark at array index 0. 

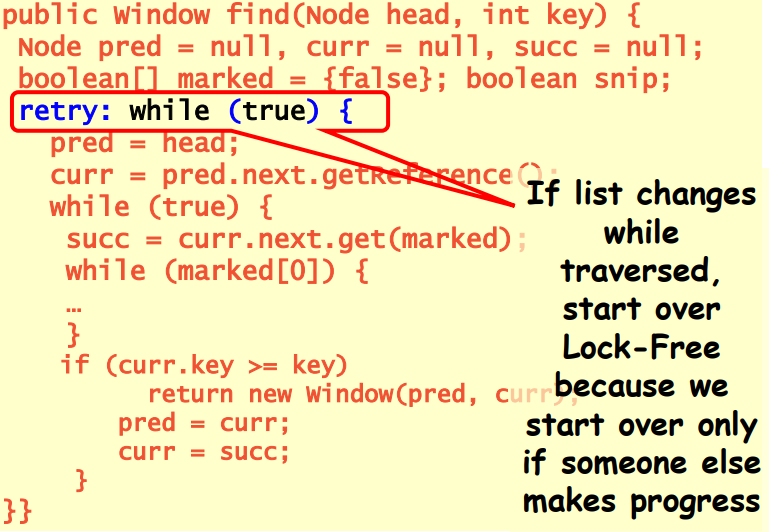


When find a logically deleted node, CAS pred’s next field and proceed.

Remove: 

Add: 

Wait-free contains: 

Lock-free find: 

**lock vs lockfree:** try combine both. Eg: versioned list combine blocking add()&remove(), and wait-free contains().

**ABA problem:** **CAS** can be used to check that a value at a reference is still the same before updating it;;The **ABA problem** arises when the same location is read twice and returns the same results even though it was modified in the meantime;;A **possible solution** is to make sure that the location always changes. – This is what we done with the versioned linked list – You can use StampedLock in Java for this