Parallel DS

goal: (one or more instances) DS supporting operations in parallel on asynchronous processes (threads, ..) over a shared memory

aim: consistent, reliable, memory management, fault-tolerance, realistic HW assumptions, independent on OS

hierarchy of parallel DS:

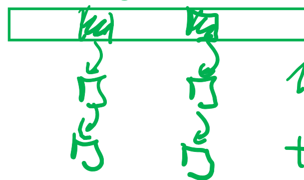
- blocking (using locks)
- obstruction-free: "if others halt, I will succeed"
- lock-free: "some process will succeed in finite time"
- wait-free: "every process will succeed in finite time"
- bounded wait-free, moreover, we have an upper bound (in #process)

Locks (mutex)

synchronization objects (locked/unlocked) forcing a sequential access

use: 1 per instance
1 per item
1 per some part

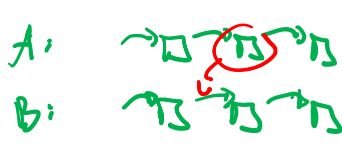
ex: hashing with chains



1 per bucket
+ global R/W for rehashing

Deadlock problem:

ex: linked lists, 1 lock per DS, move operation



moves from A to B

process 1:	process 2:
A.lock	B.lock
B.lock	A.lock
.....	...
B.unlock	A.unlock
A.unlock	B.unlock

moves from B to A

AB-BA deadlock

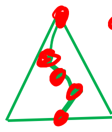
solution: global ordering of locks, always lock in this order, but not always possible (e.g. if the list B where to move depends on the value of the item in A)

parallel (a,2a)-trees with locks (in general $b \geq 2a$)

$a-1 \leq$ #keys $\leq 2a-1$

(except root)

motivation: BSTs based on edge rotations need to lock the entire path



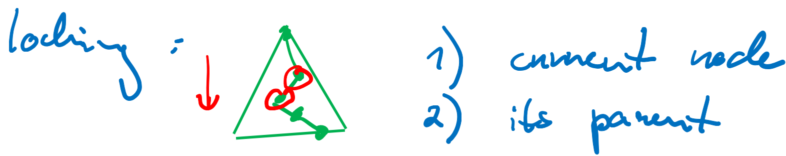
⇒ effectively locking entire DS

INSERT: top-down strategy of preemptive splitting

invariant: current node has $\leq 2a-2$ keys (at least 1 can be added)

#keys $2a-1 \xrightarrow{split} a-1 + a-1 + 1$ to parent ✓

locking:



1) current node
2) its parent

DELETE: similarly with preemptive merging

problems with locking

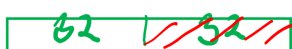
- risk of deadlocks
- fairness – who will take a released lock? (queues, ...)
- priority inversion – faster with higher priority has to wait for slow with lower priority ⇒ need to increase its priority (overhead for OS)
- performance – considerable slowdown with multiple locks
- not fault-tolerant (e.g. when process is ended forcefully)

→ we need other way to assure consistency without locks

Lock-free DS

need some atomic operations (supported by HW + programming language)

ex: int64 x on 32-bit processor



1. process: write
2. process: read ← may read neither old nor new value

Assumptions:

always avail. {
- atomic registers: read, write (+ sometimes exchange)
  ↖ performed atomically (at once) for all observers
- test & set bit

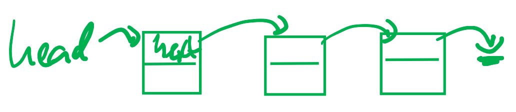
one of this {
- LL/SC – load locked / store conditional
  LL – reads and "locks" memory cell, SC – writes only if noone was working with it since LL
- CAS – compare & swap (can be implemented by LL/SC)

```
CAS(address, x, y) :    atomic {
                           old ← [address]
                           if old == x: [address] ← y
                           return old
                        }
```

x = expected, y = new value

- other operations (fetch and add, ...) (not assumed)

ex: Lock-free stack



node: atomic node * next
.....
global: atomic node * head

```
Push(node *n):
  loop { h ← head
         n.next ← h
         head ← n      (crossed out)
         if CAS(head, h, n) == h:
            return
  }
```

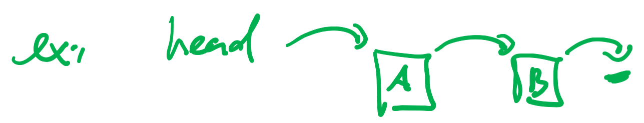
does not work in parallel ⇒ head ← n

we "check" that nobody else modified head

```
Pop:
  loop { h ← head
         n ← h.next
         if CAS(head, h, n) == h:
            return h
  }
```

⚠: if CAS fails, someone else succeeded ⇒ lock-free, but not wait-free (may infinite loop)

Problem 1: equality of pointers head and h does not guarantee that nothing happened on the stack

ex.: 

"ABA-problem"

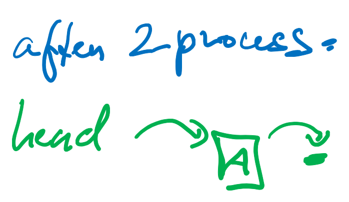
1. process: Pop

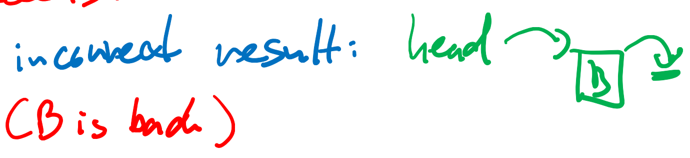
```
loop { h <- head
       n <- h.next
       if CAS(head, h, n) == h:
           return h
}
```

but this succeeds!

2. process: Pop → A, Pop → B, Push(A)

after 2 process:



incorrect result: 

(B is back)

solution:

1) use LL/SC instead of CAS
2) double CAS (CAS2) (check also h.next)

   if CAS2(<head, h.next>, <h, n>, <n, n>) == <h, n>: return

   but not available in practice
3) wide CAS (WCAS, DCAS) ... double CAS on consecutive cells

   ⇒ versioning of pointers.

   if WCAS(<head, version>, <h, ver>, <n, ver++>) == <h, ver> return

   (ver: stored version)

   (potential problem with overflow of versions)
4) restricted memory recycling: if someone refers to an address we will not allow to use it anymore (in Push(A)). How to implement it?

Problem 2: accessing a node that someone may have popped (and already freed from the memory)

ex: Pop: $h \leftarrow$ head
$n \leftarrow h.next$

other process may succeed with Pop $\Rightarrow$ memory security fault

solution: memory management: freelist – list of nodes to be freed when no one uses them

But how do we know?

a) synchronization points (e.g. main loop of the process)
If all processes pass through, we can empty the freelist.
(but not always possible to have such point)

b) reference counting – keep #references to a node in each node, remove from freelist if counter is at 0, restricted regarding

c) hazard pointers   [r slots]   HP[pid]
" before you start to use anything, make sure it appears in HP"

Pop:
```
loop { h ← head
       HP ← h
       if h ≠ head : continue
       n ← h.next
       if CAS(head, h, n) == h:
           return h
       HP ← ∅
}
```
new

if someone else succeeded with Pop here, I had my HP already set $\Rightarrow$ it is not removed from the free list

Amortization of scanning the freelist: #processes p, #referenced items per process $\leq r$ $\Rightarrow$ scan only if $\geq 2pr$ items $\Rightarrow$ removes at least pr items, scan accounted to adding to freelist for $O(1)$ amortized cost

Problem 3: compilers (usually) assume sequential semantics

ex: $x = 1$
⋮
$a = x$

← replaces by $a=1$ (but other process may change the value of x in between)

solution: volatile int x (in C)

Problem 4: HW proceeds with consecutive reads and writes (to different cells) in parallel ⇒ may reorder the operations as observed by different processes

solution: use barriers (full, read/write only, acquire/release) ⇒ prevents reordering

ex (mutex): ↑ in its implementation

```
m.lock()       ← acquire barrier:
                 no read after the barrier
  .... reads ...   may happen before the barrier
  .... writes ...
m.unlock()     ← release barrier:
                 no write before the barrier
                 may happen after the barrier
```