Lock-free Shared Data Structures

10th Estonian Summer School on Computer and Systems Science

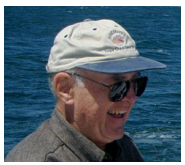
Eric Ruppert
DisCoVeri Group
York University
Toronto, Canada

August, 2011

Introduction and Motivation

Moore's Law

Gordon Moore [1965]:
The number of components in integrated circuits doubled each year between 1958 and 1965.



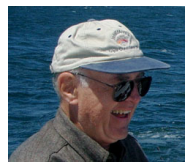
Continued doubling every 1 to 2 years since.
This yielded exponential increases in performance:

- clock speeds increased
- memory capacity increased
- processing power increased

Moore's Law for Automobiles

"If the automobile industry advanced as rapidly as the semiconductor industry, a Rolls Royce would get a million miles per gallon, and it would be cheaper to throw it away than to park it."

– Gordon Moore



Limits of Moore's Law

Exponential improvements cannot go on forever.

Components will soon be getting down to the scale of atoms.

Even now, increases in computing power come from having more processors, not bigger or faster processors.

Big performance gains in the future will come from parallelism.

Multicore Machines

Multicore machines have multiple processor chips.

Each chip has multiple cores.

Each core can run multiple programmes (or processes).

Today: Machines can run 100 processes concurrently.

Future: Many more processes.

The Big Challenge for Computer Science

Most algorithms are designed for one process.

We must design efficient parallel solutions to problems.

Amdahl's Law

Suppose we only know how to do 75% of some task in parallel.

# Cores	Time required	Speedup
1	100	1.0
2	63	1.6
3	50	2.0
4	44	2.3
⋮	⋮	⋮
10	32	3.1
⋮	⋮	⋮
100	26	3.9
⋮	⋮	⋮
∞	25	4.0

Better parallelization is needed to make use of extra cores.

One Challenge of Parallelization

Many processes must cooperate efficiently to solve a problem. Cooperation often requires communication between processes and sharing data.

- load-balancing requires transferring tasks between processes
- distributing input to processes
  (inputs may arrive online at different processes)
- aggregating results of different processes' computations
- communicating partial results from one process to another

What These Lectures Are About

Designing implementations of shared data structures.

Desirable properties:

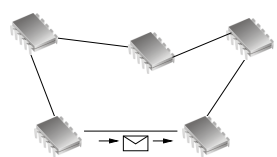
- (reasonably) easy to use
- can be accessed concurrently by several processes
- efficient (in terms of time and space)
- scalable
- (fault-tolerant)

Formalizing the Problem

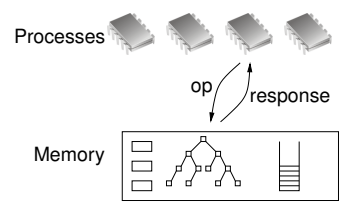
Models of Distributed Systems

Two main models of communication in distributed computing.

Message Passing



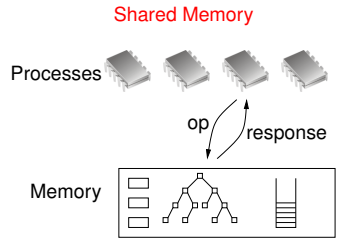
Shared Memory



Shared-Memory Model

We will focus on the shared-memory model.
(It closely resembles multicore machines.)

- Shared memory contains objects of various types.
- Each process can perform operations on the objects.
- The operation can change the state of the object and return a response.



Types of Shared Objects

Read-write register

Each register stores a value and provides two operations:

- READ(): returns current value stored and does not change the value
- WRITE(v): changes the stored value to v and returns *ack*

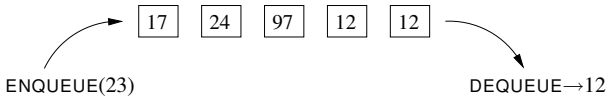
The most basic type of shared object.
Provided in hardware.

Another Type

FIFO Queue

Stores a list of items and provides two operations:

- ENQUEUE(x): adds x to the end of the list and returns *ack*
- DEQUEUE(): removes and returns one item from the front of the list



Useful for dividing up work among a collection of processes.

Not usually provided in hardware; must build it in software.

Formal Definition of an Object Type

Sequential specification describes how object behaves when accessed by a single process at a time.

- Q is the set of states
- OP is the set of operations that can be applied to the object
- RES is the set of responses the object can return
- $\delta \subseteq Q \times OP \times RES \times Q$ is the transition relation

If $(q, op, res, q') \in \delta$, it means that
when op is applied to an object in state q,
the operation can return response res and
the object can change to state q'.

Example: Read/Write Register

Object stores a natural number and allows processes to write values and read current value.

$$
\begin{aligned}
Q &= \mathbb{N} \\
OP &= \{\text{WRITE}(v) : v \in \mathbb{N}\} \cup \{\text{READ}\} \\
RES &= \mathbb{N} \cup \{\text{ACK}\} \\
\delta &= \{(v, \text{WRITE}(v'), \text{ACK}, v') : v, v' \in \mathbb{N}\} \cup \\
&\quad \{(v, \text{READ}, v, v) : v \in \mathbb{N}\}
\end{aligned}
$$

Example: Queue

A FIFO queue of natural numbers.

$$
\begin{aligned}
Q &= \text{set of all finite sequences of natural numbers} \\
OP &= \{\text{ENQUEUE}(v) : v \in \mathbb{N}\} \cup \{\text{DEQUEUE}()\} \\
RES &= \mathbb{N} \cup \{\text{NIL}, \text{ACK}\} \\
\delta &= \{(\sigma, \text{ENQUEUE}(v), \text{ACK}, \langle v \rangle \cdot \sigma) : \sigma \in Q, v \in \mathbb{N}\} \cup \\
&\quad \{(\sigma \cdot \langle v \rangle, \text{DEQUEUE}(), v, \sigma) : \sigma \in Q, v \in \mathbb{N}\} \cup \\
&\quad \{(\langle\rangle, \text{DEQUEUE}(), \text{NIL}, \langle\rangle)\}
\end{aligned}
$$

How to Share an Object

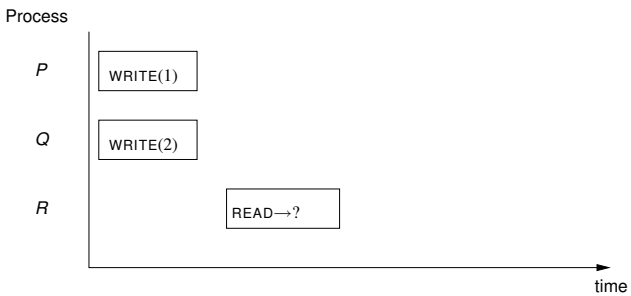
In reality, performing an operation is not instantaneous; it takes some interval of time to complete.

What happens when processes access an object concurrently?

How should the object behave?

An Example: Read-Write Register

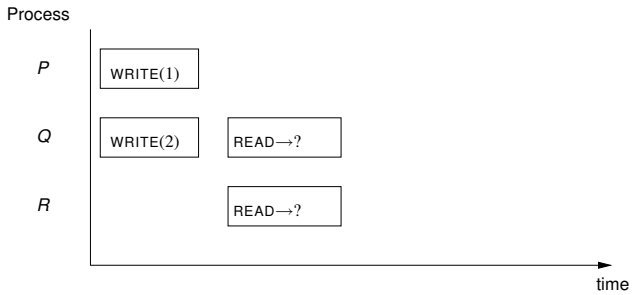
Suppose we have a read-write register initialized to value 0.



READ can output 1 or 2.

An Example: Read-Write Register

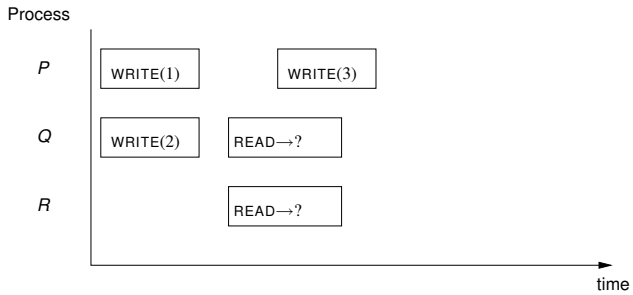
Suppose we have a read-write register initialized to value 0.



Both READS can output 1 or 2. For linearizability, they should both output the same value.

An Example: Read-Write Register

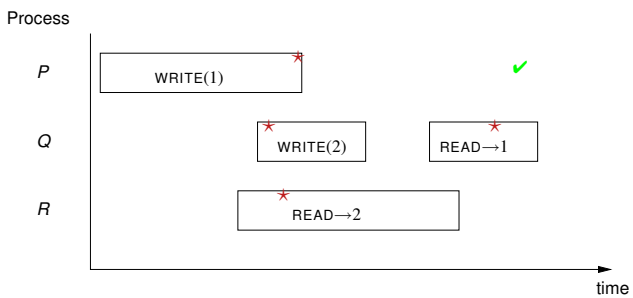
Suppose we have a read-write register initialized to value 0.



Both READS can output 1, 2 or 3. For linearizability, they should not output 1 and 2.

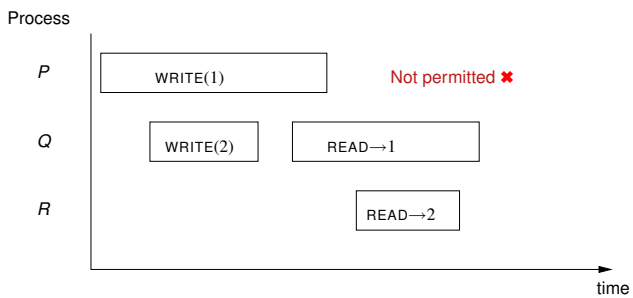
An Example: Read-Write Register

Suppose we have a read-write register initialized to value 0.



An Example: Read-Write Register

Suppose we have a read-write register initialized to value 0.



Linearizability

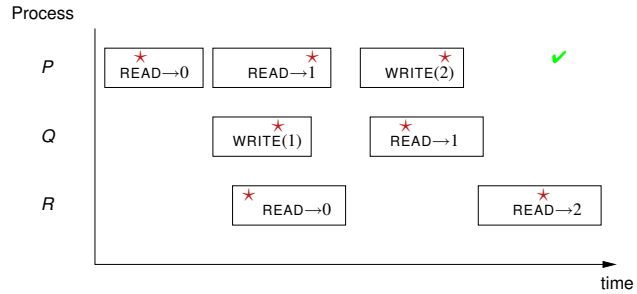
An object (built in hardware or software) is linearizable if its operations seem to occur instantaneously.

More formally:

For every execution that uses the objects,
we can choose a $\star$ inside each operation's time interval so that all operations would return the same results if they were performed sequentially in the order of their $\star$s.

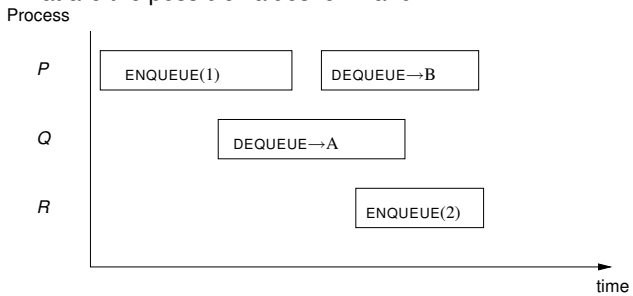
The $\star$ is called the linearization point of the operation.

A More Complicated Execution



Linearizability Example with a Queue

Consider an initially empty queue.
What are the possible values for A and B?



$A = 1, B = 2$ OR $A = 2, B = 1$ OR
$A = \text{NIL}, B = 1$ OR $A = 1, B = \text{NIL}$

Alternative Definition of Linearizability

For every execution that uses the objects,
there exists a sequential order of all the operations such that

1. all operations return the same results in the sequential order, and
2. if op_1 ends before op_2 begins, then op_1 precedes op_2 in the sequential order.

Original (and equivalent) definition of Herlihy and Wing [1990].

Consistency Conditions

Linearizability is a consistency condition.

There are many other weaker ones:

- sequential consistency
- processor consistency
- causal consistency
- etc.

Weaker conditions are easier to implement but harder to use.
We will focus on linearizable objects.

$\Rightarrow$ When using linearizable objects,
can assume operations happen atomically.

Specifying Correctness: Summary

How to specify correctness of a shared data structure?

Safety = sequential specification + linearizability
(Q, OPS, RES, δ)

Progress: operations terminate (discussed later)

What are These Talks About?

Goal: Design shared data structure implementations using basic objects provided by system (like registers).

- Assume basic objects are linearizable.
- Prove implemented objects are also linearizable.
- Study complexity of implementations.
- Prove some implementations are impossible.

Challenges of Implementing Shared Objects

- Asynchrony (processes run at different, variable speeds).
- Concurrent updates should not corrupt data.
- Reading data while others update it should not produce inconsistent results.
- Other unpredictable events (e.g., failures).

Traditional Approach: Locks

Each object has a lock associated with it.
Only one process can hold the lock at any time.

1. obtain lock
2. access object
3. release lock

Pros:

- simple
- avoids problems of concurrent accesses

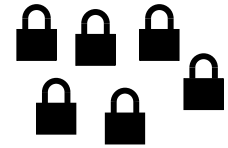
Cons:

- slow
- no real parallelism
- no fault-tolerance

Fine-Grained Locking

Each part of an object has a lock.

1. Obtain locks for parts you want to access
2. Access those parts of object
3. Release locks



Pros:

- some parallelism
- reduces problems of concurrent accesses

Cons:

- limited parallelism
- no fault-tolerance
- danger of deadlock, livelock

Alternative Approach: Lock-Free Implementations

Avoid use of locks altogether.
Programmer is responsible for coordination.

Pros:

- permits high parallelism
- fault-tolerance
- slow processes don't block fast ones
- unimportant processes don't block important ones

Cons:

- difficult to design

Lock-free Implementations

"The shorter the critical sections, the better. One can think of lock-free synchronization as a limiting case of this trend, reducing critical sections to individual machine instructions."

– Maurice Herlihy

Lock-Free Progress Guarantees

Non-blocking Progress Property

If processes are doing operations, one eventually finishes.

- No deadlock
- No fairness guarantee: individual operations may starve

Wait-free Progress Property

Each operation eventually finishes.

- Stronger guarantee
- Hard to implement efficiently

The Goal

Design lock-free, linearizable shared data structures.

Snapshot Objects

Snapshot Objects

Definition

A (single-writer) snapshot object for n processes:

- Stores a vector of n numbers (one per process)
- Process i can UPDATE(i, v) to stores v in component i
- Any process can SCAN to return the entire vector

1	2	3		n
0	0	0	$\cdots$	0

Original definition: Afek, Attiya, Dolev, Gafni, Merritt, Shavit [1993]; Anderson [1993]; Aspnes, Herlihy [1990]

Why Build Snapshots?

- Abstraction of the problem of reading consistent view of several variables
- Making backup copy of a distributed database
- Saving checkpoints for debugging distributed algorithms
- Used to solve other problems (timestamps, randomized consensus, ...)

Snapshot Algorithm: Attempt #1

Use an array $A[1..n]$ of registers.

```
UPDATE(i,v)
  A[i] ← v
  return ACK
```

```
SCAN
  for i ← 1..n
    r_i ← A[i]
  end for
  return (r_1, r_2, ..., r_n)
```

Convention for pseudocode

- Shared objects start with capital letters (e.g., A)
- Local objects start with lower-case letters (e.g., i, r_i, v).

Algorithm is simple, but wrong. ✖

A Bad Execution

Shared Registers			Processes		
$A[1]$	$A[2]$	$A[3]$	$P1$	$P2$	$P3$
0	0	0			SCAN:
			UPDATE(1,7):		READ $A[1] \to 0$
			$A[1] \leftarrow 7$		
7	0	0		UPDATE(2,9):	
				$A[2] \leftarrow 9$	
7	9	0			
					READ $A[2] \to 9$
					READ $A[3] \to 0$
					return $(0, 9, 0)$

Not linearizable. ✖

Snapshot Algorithm: Attempt #2

Problem

SCAN can read inconsistent values if concurrent UPDATES occur.

Solution

If SCAN detects a concurrent UPDATE, it tries again.

A better snapshot implementation:

```
UPDATE(i,v)
  A[i] ← v
  return ACK
```

```
SCAN
  READ A[1..n] repeatedly until the
       same vector is read twice
  return that vector
```

Does this work?

Another Bad Execution

$A[1]$	$A[2]$	$A[3]$	$P1$	$P2$	$P3$
0	0	0			READ $A[1] \rightarrow 0$
			UPDATE(1,7)		
7	0	0			
				UPDATE(2,9)	
7	9	0			READ $A[2] \rightarrow 9$
				UPDATE(2,0)	READ $A[3] \rightarrow 0$
7	0	0			
			UPDATE(1,0)		
0	0	0			READ $A[1] \rightarrow 0$
			UPDATE(1,7)		
7	0	0			
				UPDATE(2,9)	
7	9	0			READ $A[2] \rightarrow 9$
					READ $A[3] \rightarrow 0$
					return (0,9,0)

Not linearizable $\Rightarrow$ Algorithm is still wrong. ✖

Snapshot Algorithm: Attempt #3

ABA Problem

Reading the same value twice does not mean the register has not changed in between.

Solution

Attach a counter to each register so that scanner can really tell when a register has changed.

UPDATE(i,v)
  $A[i] \leftarrow (v, counter_i{+}{+})$
  return ACK

SCAN
  READ $A[1..n]$ repeatedly until the same vector is read twice
  return that vector (without counters)

Does this work?

An example

$A[1]$	$A[2]$	$A[3]$	$P1$	$P2$	$P3$
(0,0)	(0,0)	(0,0)			READ $A[1] \rightarrow (0,0)$
			UPDATE(1,7)		
(7,1)	(0,0)	(0,0)			
				UPDATE(2,9)	
(7,1)	(9,1)	(0,0)			READ $A[2] \rightarrow (9,1)$
				UPDATE(2,0)	READ $A[3] \rightarrow (0,0)$
(7,1)	(0,2)	(0,0)			
			UPDATE(1,0)		
(0,2)	(0,2)	(0,0)			READ $A[1] \rightarrow (0,2)$
			UPDATE(1,7)		
(7,3)	(0,2)	(0,0)			
				UPDATE(2,9)	
(7,3)	(9,3)	(0,0)			READ $A[2] \rightarrow (9,3)$
					READ $A[3] \rightarrow (0,0)$
					keep trying...

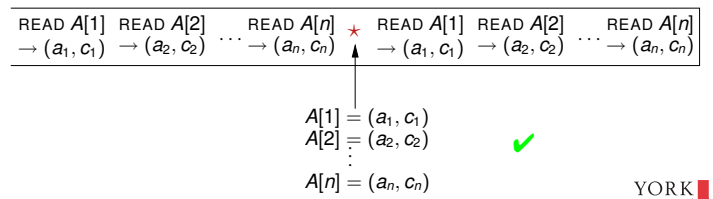
Problem fixed. Is algorithm linearizable now?

How to Prove Snapshot Algorithm is Linearizable?

Choose a linearization point ($\star$) for each operation.

Show that operations would return same results if done in linearization order.

1. UPDATE operation linearized when it does its WRITE.
   $\Rightarrow$ Invariant: $A[1..n]$ contains true contents of snapshot object (ignoring counters).
2. SCAN operation linearized between last two sets of READS.



✔

What About Progress?

Non-blocking Progress Property

If processes are doing operations, one eventually finishes.

Clearly, UPDATES always finish.

SCANS could run forever, but only if UPDATES keep happening.

⇒ Algorithm is non-blocking. ✔

Measuring Complexity of Lock-Free Algorithm

Space: n registers (of reasonable size).

Worst-case running time:

- $O(1)$ per UPDATE
- ∞ per SCAN

Worst-case amortized running time:

- $O(n^2)$ per UPDATE
- $O(n)$ per SCAN

This means the total number of steps taken in an execution with u UPDATES and s SCANS be at most $O(un^2 + sn)$.

Wait-Free Snapshots

Non-blocking Progress Property

If processes are doing operations, one eventually finishes.

Wait-free Progress Property

Each operation will eventually finish.

Can we make the snapshot implementation wait-free?

How to Make Non-blocking Algorithm Wait-free

The Problem

Fast operations (UPDATES) prevent slow operations (SCANS) from completing.

Solution

Fast operations help slow operations finish.

Some kind of helping mechanism is usually needed to achieve fair progress conditions.

A Wait-Free Snapshot Algorithm

Main Idea

- An UPDATE must perform an embedded SCAN and write the result in memory.
- If a SCAN sees many UPDATES happening, it can return the vector that was found by one of the embedded SCANS.

```
UPDATE(i,v)
  s⃗ ← SCAN
  A[i] ← (v, counter_i++, s⃗)
  return ACK
```

```
SCAN
  READ A[1..n] repeatedly
  if same vector is read twice then
      return that vector (values only)
  if some A[i] has changed twice then
      return the vector saved in A[i]
```

Note: uses BIG registers.

Why Is This Algorithm Wait-Free?

If a SCAN reads $A[1..n]$ $n+2$ times then either

- two sets of reads will be identical or
- one component will have changed twice.

Thus, each operation will terminate. ✔

Example ($n = 4$)

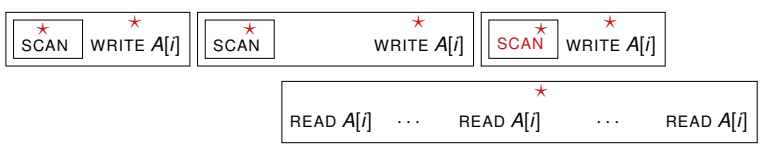
0000	0100	0110	1110	1111	1121

Space: n BIG registers.
Worst-case time: $O(n^2)$ shared-memory accesses per operation.

Can build BIG registers from small registers, but number of registers and time will increase.

Why Is This Algorithm Linearizable?

- UPDATE linearized at the time it writes (as before).
- SCAN that sees same values twice is linearized as before.
- What about a SCAN that returns result of an embedded SCAN?
  Linearize it at the same time as the embedded SCAN.
  Why is this okay?



If a SCAN returns the result of an embedded SCAN, then the ⋆ of the embedded SCAN is inside the SCAN. ✔
(This is why we wait for a component to change twice.)

What Have We Learned from Snapshots?

- Accessing a data structure while others are updating it can be difficult.
- Use repeated reads for consistency.
- How to avoid the ABA problem.
- Helping is useful for making algorithms wait-free.
- How to talk about complexity of implementations.

Further Work on Snapshots

- Multi-writer snapshots: Any process can write to any component.
- Better complexity: Implementations with better time and space requirements.
- Adaptive snapshots: Running time depends on number of active processes.
- Lower bounds: Understand how much time is really required to perform a snapshot.
- Immediate snapshots: UPDATE and SCAN combined into a single operation.
- Partial snapshots: Implement (efficiently) SCANS of a few components of a large object.

Shared Counters

Another Example: Unit Counters

Definition

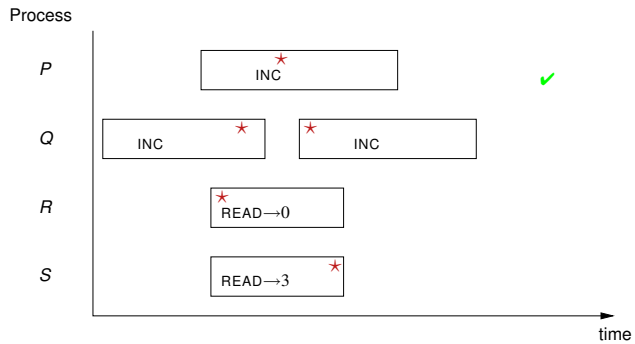
A unit counter stores an integer and provides two operations:

- INC adds one to the stored value (and returns ACK)
- READ returns the stored value (and does not change it)

How to implement a wait-free unit counter from registers?

Exercise: Linearizability

Consider a counter that is initially 0.



Unit Counter: Attempt #1

Try the obvious thing first.

Idea

Use a single shared register C to store the value.

```
INC
    C ← C + 1
    return ACK
```

```
READ
    return C
```

Is this correct?

Note: $C \leftarrow C + 1$ really means
$temp \leftarrow$ READ C
WRITE($temp + 1$) into C

A Bad Execution

Consider two interleaved INC operations:

C	P1	P2
0	INC:	INC:
	READ $\rightarrow$ 0	
		READ $\rightarrow$ 0
	WRITE(1)	
1		WRITE(1)

Two INCS have happened, but a READ will now output 1.

So the simple algorithm is wrong. ✖

Unit Counters from Snapshots

Idea

Use a snapshot object.
Let num_i be the number of INCS process i has done.

- Each process stores num_i in its component.
- Use a SCAN to compute $\sum_{i=1}^{n} num_i$.

```
INC
    UPDATE(i, num_i++)
    return ACK
```

```
READ
    SCAN and return sum of components
```

Trivial to check that this is linearizable. ✔
Same progress property as snapshot object. ✔

But SCANS are expensive. Can we do it more efficiently?

Back to Square One

Try using the simplest idea we tried for snapshots.

Idea

Use an array $A[1..n]$.
Process i stores the number of increments it has done in $A[i]$.

```
INC
    A[i] ← A[i] + 1
    return ACK
```

```
READ
    sum ← 0
    for i ← 1..n
        sum ← sum + A[i]
    return sum
```

Clearly wait-free (and more efficient). ✔
Is it linearizable?

Linearizability Argument

- Linearize INC when it WRITES.
- Prove there exists a linearization point for the READS.

Let $A[i]_{start}$ and $A[i]_{end}$ be the value of $A[i]$ at start and end of some READ of the counter.
Let $A[i]_{read}$ be the value of $A[i]$ when it is actually read.

$$\begin{array}{ccccc} A[i]_{start} & & \leq A[i]_{read} \leq & & A[i]_{end} \\ \sum_{i=1}^{n} A[i]_{start} & & \leq \sum_{i=1}^{n} A[i]_{read} \leq & & \sum_{i=1}^{n} A[i]_{end} \\ \text{counter value at start} & & \leq \text{sum READ returns} \leq & & \text{counter value at end} \end{array}$$

So, at some time during the READ,
the counter must have had the value the READ returns. ✔

General Counters

Definition

A general counter stores an integer and provides two operations:

- INC(v) adds v to the stored value (and returns ACK)
- READ returns the stored value (and does not change it)

Can use the same wait-free implementation from snapshots as for unit counters.

Does the more efficient implementation from registers work?

A Bad Execution

Execution of simpler register-based implementation.

$A[1]$	$A[2]$	$A[3]$	$P1$	$P2$	$P3$
0	0	0			READ $A[1] \to 0$
			INC(1)		
1	0	0			
				INC(2)	
1	2	0			READ $A[2] \to 2$
					READ $A[3] \to 0$
					return 2

Not linearizable: counter has values $0, 1$ and 3 but never 2. ✖

Fetch&Increment Counter

Definition

A fetch&increment counter stores an integer and provides one operation:

- FETCH&INC adds 1 to the value and returns the old value.

Fetch&increment counters are useful for implementing timestamps, queues, etc.

How can we implement it from registers?

Ideas for Implementing Fetch&Increment Object

Idea 1

Use single register C.
A FETCH&INC performs $C \leftarrow C + 1$.

Two concurrent FETCH&INCS might only increase value by 1 (and return same value). ✖

Idea 2

Use array of registers or snapshot object.

No obvious way to combine reading total and incrementing it into one atomic action. ✖

People tried some more ideas...

A Problem!

A lock-free implementation of FETCH&INC from registers is impossible!

Proved by Loui and Abu-Amara and by Herlihy in 1987.

How can we prove a result like this?

Consensus Objects

Definition

A consensus object stores a value (initially $\perp$) and provides one operation, PROPOSE(v).

- If value is $\perp$, PROPOSE(v) changes it to v and returns v.
- Else, PROPOSE(v) returns the current value without changing it.

Essentially, the object returns first value proposed to it.

Very useful for process coordination tasks.
Consensus is very widely studied.

Cannot Implement FETCH&INC From Registers

Proof idea:

1. There is a lock-free implementation of consensus using FETCH&INC and registers for two processes.
   → Easy.
2. There is no lock-free implementation of consensus using only registers for two processes.
   → Hard, but we have lots of tools for doing this.
3. There is no lock-free implementation of FETCH&INC from registers for two (or more) processes.
   → Follows easily from previous two steps.

Lock-Free Consensus Using FETCH&INC and Registers for Two Processes

Idea:

- First process to access FETCH&INC object F wins.
- Write proposed value in shared memory so loser can return winner's value.

```
PROPOSE(v)
  A[i] ← v                    % Announce my value
  if F.FETCH&INC = 0 then     % I won
    return v
  else                        % I lost
    return A[3 − i]           % Return winner's value
```

Exercise: Prove this works.

Registers Cannot Implement Consensus

Proof Sketch:

- There must eventually be a step by some process when the final decision is actually made.
- Both processes must be able to determine what the decision was.
- No step that accesses a register can do this.
  - A process will not notice a READ by the other process.
  - A WRITE can be overwritten by the other process.

  Either way, second process cannot see evidence of the decision made by a step of the first process.

Based on classic result of Fischer, Lynch and Paterson.

Chaos and Confusion

Computers equipped with different primitive objects may or may not be able to implement certain data structures.

Example

Registers can implement	Registers cannot implement
snapshot objects	consensus
counters	fetch&inc objects
...	queues
	stacks
	...

Very different from classical models of computing.

To implement a data structure, which primitives do I need?

Hardware Implications

Tell hardware designers:

- Registers are not good enough.
- Build machines with stronger primitive objects.

But which primitives should we ask for?

Universal Objects to the Rescue

Herlihy proved that some object types are universal:
They can be used to implement every other type.

Basically, if you can solve consensus, then you can implement every type of object.
So, tell hardware designers to include a universal object.
(They do now.)

Herlihy also classified objects:

- registers, snapshot objects are weakest
- fetch&inc objects, stacks are a little bit stronger
- ...
- compare&swap objects, LL/SC are most powerful

Universal Constructions

Compare&Swap Objects

Compare-And-Swap (CAS) Object

Stores a value and provides two operations:

- READ(): returns value stored in object without changing it
- CAS(*old*, *new*): if the value currently stored is *old*, change it to *new* and return *old*;
  otherwise return current value without changing it.

CAS objects are universal.
Multicore systems have hardware CAS operations.

Example: Non-blocking Fetch&Inc from CAS

Idea

Use a single CAS object to store the value of the counter.
To increment, try CAS($v, v+1$) until you succeed.

```
INC
loop
    READ CAS object and store value in v
    if CAS(v, v + 1) = v then return v
end loop
```

Note this algorithm is non-blocking, but not wait-free.

There are much more efficient ways to do this, but at least this shows it is possible.

Wait-free Consensus from CAS

Idea

Use a CAS object with initial value $\perp$.
The first process to access object wins and stores its value.

```
PROPOSE(v)
    return CAS(⊥, v)
```


The Universal Construction

Goal

Given sequential specification of any object,
show there is a non-blocking (or wait-free) implementation of it using CAS objects (and registers).

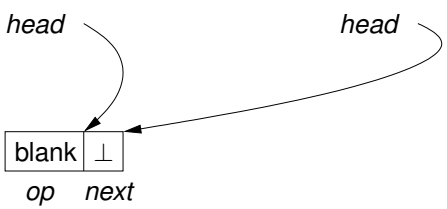
Idea

The implementation must be linearizable.
So, processes agree on the linearization of all operations.
To perform an operation, add it to the end of a list.

A Non-blocking Universal Construction

Example: Implementing a stack.

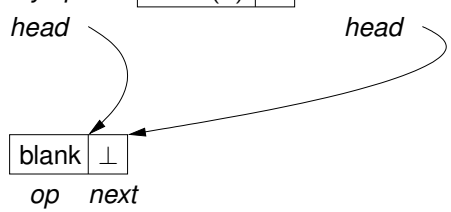
Process P: $state = \langle\rangle$ Process Q: $state = \langle\rangle$



A Non-blocking Universal Construction

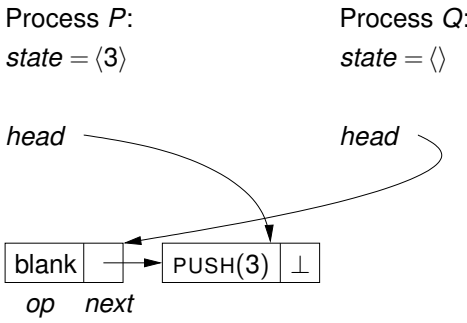
Example: Implementing a stack.

Process P: $state = \langle\rangle$ Process Q: $state = \langle\rangle$



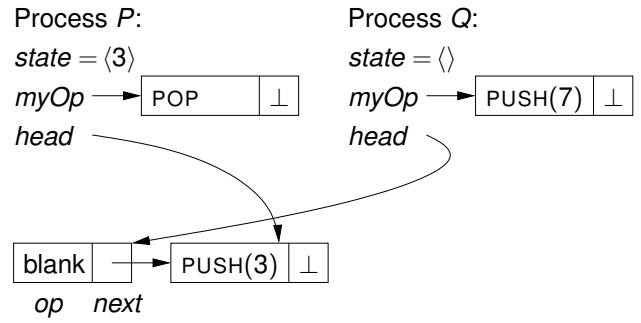
A Non-blocking Universal Construction

Example: Implementing a stack.



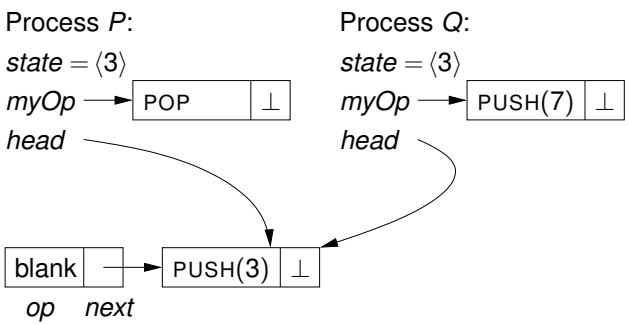
A Non-blocking Universal Construction

Example: Implementing a stack.



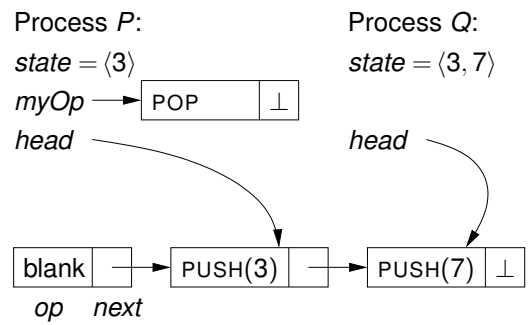
A Non-blocking Universal Construction

Example: Implementing a stack.



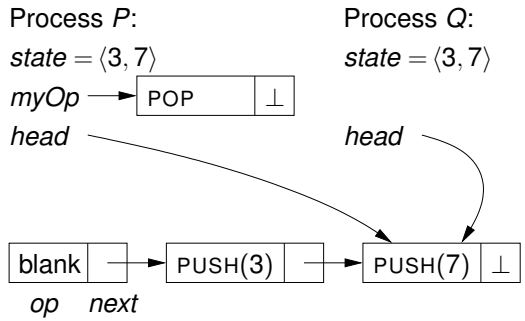
A Non-blocking Universal Construction

Example: Implementing a stack.



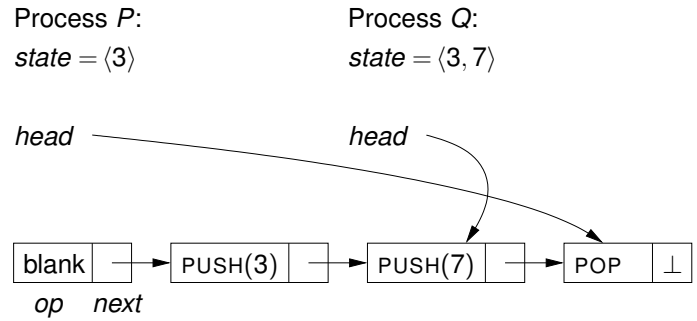
A Non-blocking Universal Construction

Example: Implementing a stack.



A Non-blocking Universal Construction

Example: Implementing a stack.



The Details

head initially contains first node on list
state initially contains initial state of the implemented object

DO-OPERATION(*op*)
  myOp ← new node containing *op* and null *next* pointer
  loop
    head ← *CAS*(*head.next*, ⊥, *myOp*)
    if *head* = ⊥ then *head* = *myOp*
    (*state*, *result*) ← δ(*state*, *head.op*) % apply transition function
    if *head* = *myOp* then return *result*
  end loop

Correctness

Linearization ordering: the order of operations on the list.

Progress:
Some process eventually reaches end of list
and adds its operation.
⇒ Non-blocking.

Efficiency: terrible (but it shows any object can be built).

A Wait-Free Universal Construction

Can we make the algorithm wait-free?

Idea

Add helping.
Fast processes help slow ones add their operations to the list.

- Processes announce their operations in shared memory.
- Each position on the list is reserved for one process.
- When you reach a position that belongs to process P:
  - If P has a pending operation, help P add it.
  - Otherwise, try adding your own operation.
- Be careful not to add same operation to list twice.

Details

```
DO-OPERATION(op)
  Announce[id] ← new node containing op and null next pointer
                 and added flag = false
  loop
    if Announce[position mod n].added = false then
      myOp ← Announce[position mod n]
    else
      myOp ← Announce[id]
    head ← CAS(head.next, ⊥, myOp)
    if head = ⊥ then head = myOp
    head.added ← true
    position ← position + 1
    (state, result) ← δ(state, head.op) % apply transition function
    if head = Announce[id] then return result
  end loop
```


Correctness

Linearization ordering: the order of operations on the list.

Progress:
After you announce your operation,
at most one other process's operation will be put in one of your reserved positions.
⇒ Your operation will be added within the next $O(n)$ nodes.
⇒ Wait-free.

Efficiency

Still terribly inefficient (but shows CAS is universal).

In fact, time to complete an operation is unbounded (but finite).

Can make it polynomial by sharing information about *head*:
⇒ If a process falls behind, he can read up-to-date *head* value.

Lots of later work on more efficient universal constructions.

(There is also the transactional memory approach.)

However, handcrafted data structures are still needed for high efficiency shared data structures (e.g., for libraries).

Stacks

Stack

Definition

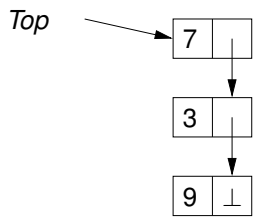
A stack stores a sequence of values and provides two operations

- PUSH(x) adds x to the beginning of the sequence
- POP removes and returns the first value from the sequence

Treiber's Stack

Idea

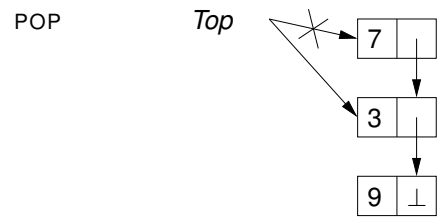
Store the sequence as a linked list (pointers go top to bottom)
Use CAS to update pointers.



Treiber's Stack

Idea

Store the sequence as a linked list (pointers go top to bottom)
Use CAS to update pointers.

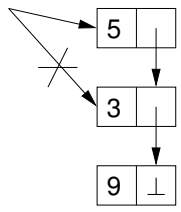


Treiber's Stack

Idea

Store the sequence as a linked list (pointers go top to bottom)
Use CAS to update pointers.

PUSH(5) *Top*



Treiber's Stack: Details

Each operation tries its CAS until it succeeds.

```
PUSH(x)
  create new node v containing
      x and next pointer ⊥
  loop until a CAS succeeds
    v.next ← Top
    CAS(Top, v.next, v)
  end loop
```

```
POP
  loop
    t ← Top
    if t = ⊥ then return "empty"
    if CAS(Top, t, t.next) = t then
      return t.value
  end loop
```


Correctness

Linearizability

Linearize each operation at its successful CAS.
(Except: POP that returns "empty" is linearized at READ of *Top*.)

Invariant: Contents of linked list are true contents of stack.

Non-blocking property

Eventually some operation successfully changes *Top*
(or returns "empty" if only POPS are happening on empty stack).

Efficiency

Problem

Every operation must wait its turn to change *Top*
$\Rightarrow$ no concurrency, no scalability

Solution

Use a back-off scheme.

While waiting, try to find matching operation.

If a PUSH(x) finds a POP, they can cancel out:

- PUSH(x) returns ACK
- POP returns x

Linearize PUSH(x) and then POP at the moment they meet.

See Hendler, Shavit, Yerushalmi [2010]

ABA Problem Again

Question

Suppose we POP a node v off the stack.
Is it safe to free the memory used by v?

Could cause ABA problem if another process has a pointer to v.

Process P wants to POP.
P reads $Top = v$ and $Top.next = u$.
Process Q POPS v and u and frees them.
⋮
Later, Top again points to v, and u is in the middle of the stack.
P performs CAS(Top, v, u). ✖

Solving the ABA Problem

Solution 1

Do garbage collection more carefully.
Do not free a node if a process holds a pointer to it.

Makes garbage collection expensive.

Solution 2

Store a pointer together with a counter in the same CAS object.
E.g. Java's AtomicStampedReference.

Makes pointer accesses slower.
Technically, requires unbounded counters.

Sets
(Using Binary Search Trees)

Set Objects

Definition

A set object stores a set of items and provides three operations:

- FIND(x): returns *true* if x is in the set, or *false* otherwise
- INSERT(x): adds x to set and returns *true* (or *false* if x was already in the set)
- DELETE(x): removes x from the set and returns *true* (or *false* if x was not in the set)

One of the most commonly used types of objects.

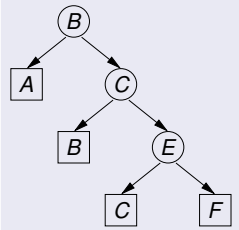
Ways of Implementing Set Object

- Sorted linked list
  - Well-studied
  - Slow
- Skip list
  - Provided in Java standard library
  - Hard to guarantee good running time
- Hash table
  - They exist
  - Harder to guarantee good running time
- Search tree
  - Can be implemented efficiently in Java
  - Hard to balance (but maybe soon...)

Binary Search Trees

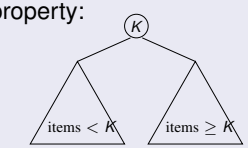
Example

BST storing items $\{A, B, C, F\}$



Definition

- One square node (leaf) for each item in the set
- Round nodes (internal nodes) used to route find operations to the right leaf
- Each internal node has a left child and a right child
- BST property:



Non-blocking BST

A non-blocking implementation of BSTs from single-word CAS.

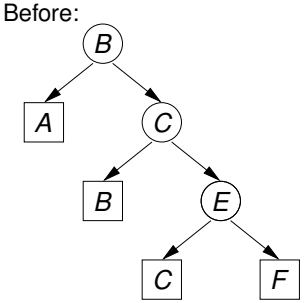
Some properties:

- Conceptually simple
- Fast searches
- Concurrent updates to different parts of tree do not conflict
- Technique seems generalizable
- Experiments show good performance
- But unbalanced

Ellen, Fatourou, Ruppert, van Breugel [2010].

Insertion (non-concurrent version)

Before:

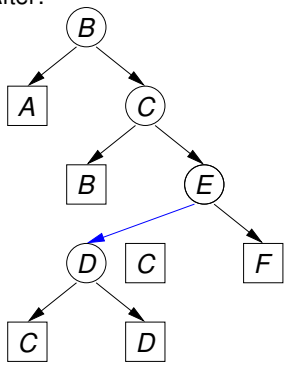


INSERT(D)

1. Search for D
2. Remember leaf and its parent
3. Create new leaf, replacement leaf, and one internal node
4. Swing pointer

Insertion (non-concurrent version)

After:

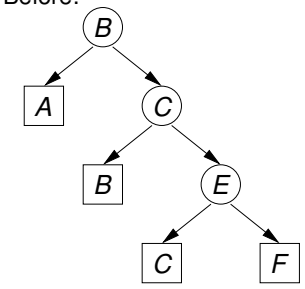


INSERT(D)

1. Search for D
2. Remember leaf and its parent
3. Create new leaf, replacement leaf, and one internal node
4. Swing pointer

Deletion (non-concurrent version)

Before:

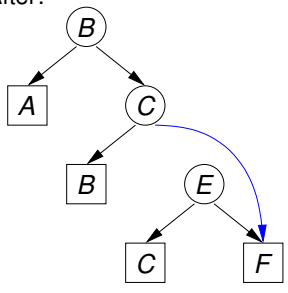


DELETE(C)

1. Search for C
2. Remember leaf, its parent and grandparent
3. Swing pointer

Deletion (non-concurrent version)

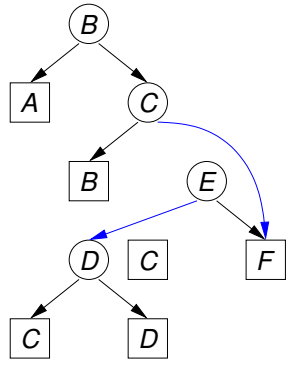
After:



DELETE(C)

1. Search for C
2. Remember leaf, its parent and grandparent
3. Swing pointer

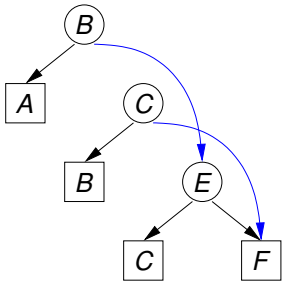
Challenges of Concurrent Operations (1)



Concurrent DELETE(C) and INSERT(D).

$\Rightarrow$ D is not reachable!

Challenges of Concurrent Operations (2)



Concurrent DELETE(B) and DELETE(C).

$\Rightarrow$ C is still reachable!

Coordination Required

Crucial problem: A node's child pointer is changed while the node is being removed from the tree.

Solution: Updates to the same part of the tree must coordinate.

Desirable Properties of Coordination Scheme

- No locks 
- Maintain invariant that tree is always a BST
- Allow searches to pass unhindered
- Make updates as local as possible
- Algorithmic simplicity

Flags and Marks

An internal node can be either flagged or marked (but not both). Status is changed using CAS.

Flag

Indicates that an update is changing a child pointer.

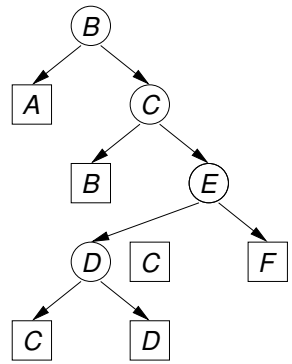
- Before changing an internal node's child pointer, flag the node.
- Unflag the node after its child pointer has been changed.

Mark

Indicates an internal node has been (or soon will be) removed from the tree.

- Before removing an internal node, mark it.
- Node remains marked forever.

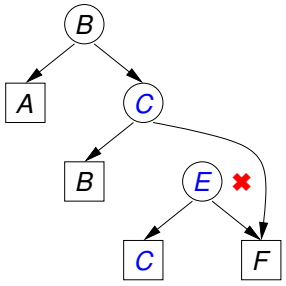
Insertion Algorithm



INSERT(D)

1. Search for D
2. Remember leaf and its parent
3. Create three new nodes
4. Flag parent (if this fails, retry from scratch)
5. Swing pointer (using CAS)
6. Unflag parent

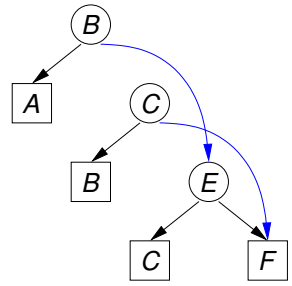
Deletion Algorithm



DELETE(C)

1. Search for C
2. Remember leaf, its parent and grandparent
3. Flag grandparent (if this fails, retry from scratch)
4. Mark parent (if this fails, unflag grandparent and retry from scratch)
5. Swing pointer (using CAS)
6. Unflag grandparent

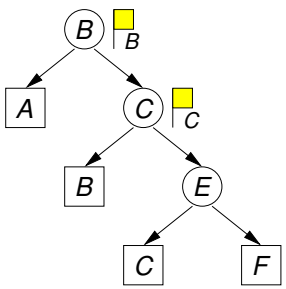
Recall: Problem with Concurrent DELETES



Concurrent DELETE(B) and DELETE(C).

$\Rightarrow$ C is still reachable!

Conflicting Deletions Now Work



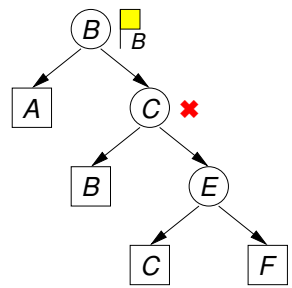
Concurrent DELETE(B) and DELETE(C)

Case I: DELETE(C)'s flag succeeds.

$\Rightarrow$ Even if DELETE(B)'s flag succeeds, its mark will fail.

$\Rightarrow$ DELETE(C) will complete
DELETE(B) will retry

Conflicting Deletions Now Work



Concurrent DELETE(B) and DELETE(C)

Case II: DELETE(B)'s flag and mark succeed.

$\Rightarrow$ DELETE(C)'s flag fails.

$\Rightarrow$ DELETE(B) will complete
DELETE(C) will retry

Locks

Can think of flag or mark as a lock on the child pointers of a node.

- Flag corresponds to temporary ownership of lock.
- Mark corresponds to permanent ownership of lock.

If you try to acquire lock when it is already held, CAS will fail.

Wait a second . . .

Problem

Implementation is supposed to be lock-free!

Solution

Whenever "locking" a node, leave a key under the doormat.

A flag or mark is actually a pointer to a small record that tells a process how to help the original operation.

If an operation fails to acquire a lock, it helps complete the update that holds the lock before retrying.

Thus, locks are owned by operations, not processes.

Searching

SEARCHES just traverse edges of the BST until reaching a leaf.

They can ignore flags and marks.

This makes them very fast.

But, this means SEARCHES can

- go into marked nodes,
- travel through whole marked sections of the tree, and
- possibly return later into old, unmarked sections of the tree.

How do we linearize such searches?

Linearizing Searches

Lemma

Each node visited by SEARCH(K) *was on the search path for* K

- *after the* SEARCH *started, and*
- *before the* SEARCH *entered the node.*

Suppose SEARCH(K) visits nodes $v_1, v_2, \ldots, v_m$.

Base: Dummy root v_1 never changes $\Rightarrow$ always on search path.

Inductive step: Assume v_i is on the search path at time T_i.
The SEARCH read v_{i+1} as a child of v_i after entering v_i.
Case 1: v_{i+1} was already v_i's child at T_i.
$\Rightarrow$ v_{i+1} was on search path at T_i. ✔
Case 2: v_{i+1} became v_i's child after T_i.
When v_{i+1} was added, v_i was still on search path (because nodes never get new ancestors).
So v_{i+1} was on search path then. ✔

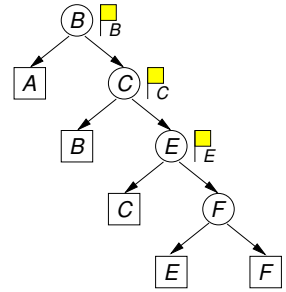
Progress

Goal: Show data structure is non-blocking.

- If an INSERT successfully flags, it finishes.
- If a DELETE successfully flags and marks, it finishes.
- If updates stop happening, SEARCHES must finish.

One CAS fails only if another succeeds.
⇒ A successful CAS guarantees progress, except for a DELETE's flag.

Progress: The Hard Case



A DELETE may flag, then fail to mark, then unflag to retry.

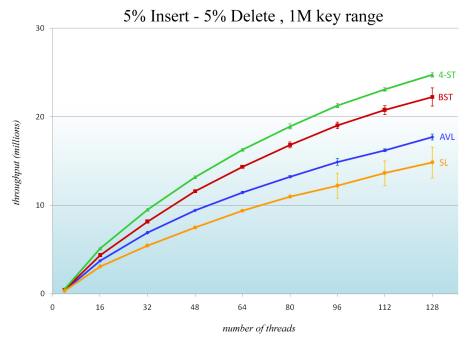
⇒ The DELETE's changes may cause other CAS steps to fail.

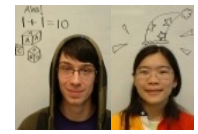
However, lowest DELETE will make progress.

Warning

Some details have been omitted.
The proof that this actually works is about 20 pages long.

Experimental Results





2 UltraSPARC-III CPUs, each with 8 cores running at 1.2GHz.
8 threads per core.
Each thread performs operations on random keys.

More Work Needed on Lock-free Trees

- Balancing the tree
- Proving worst-case complexity bounds
- Can same approach yield (efficient) wait-free BSTs? (Or at least wait-free FINDS?)
- Other tree data structures

Conclusions

Lock-Free Data Structures: A Summary

- Studied for 20+ years
- Research is important for new multicore architectures
- Universal constructions [1988–present]
  Disadvantage: inefficient
- Array-based structures [1990–2005]
  snapshots, stacks, queues
- List-based structures [1995–2005]
  singly-linked lists, stacks, queues, skip lists
- Tree-based structures [very recent]
  BSTs, some work based on B-trees, ...
- A few others [1995–present]
  union-find, ...

Techniques

- Use repeated reads for consistency.
- Methods to avoid the ABA problem.
- Helping is useful for making algorithms wait-free.
- Pointer-swinging using CAS.
- "Locks" owned by operations can be combined with helping.
- Elimination filters

Other Directions

- Other data structures
- Memory management issues
- Weaker progress guarantees (e.g., obstruction freedom)
- Using randomization to get better implementations
- Software transactional memory

YORK UNIVERSITY

Final Words

Lots of work on lock-free data structures
needed for multicore machines of the future.

YORK UNIVERSITY

Photo credits

Gordon Moore: Steve Jurvetson
Gene Amdahl: Pkivolowitz at en.wikipedia

YORK UNIVERSITY