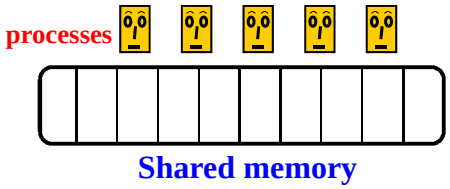
Fences and RMRs Required for Synchronization

Hagit Attiya
CS, Technion

Standard Shared-memory Model

Asynchronous processes

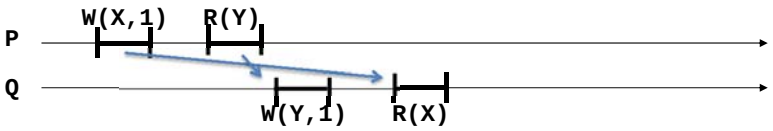
Apply primitive operations to shared memory



Per-Thread Ordering is Crucial

Generic mutex algorithm (e.g., Dekker)

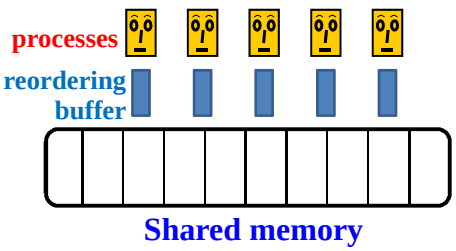
Process P:	Process Q:
Write(X,1)	Write(Y,1)
Read(Y)	Read(X)



Out-of-Order Execution

Compensate for slow writes

Most common: **issue reads before following writes**, if they access different locations



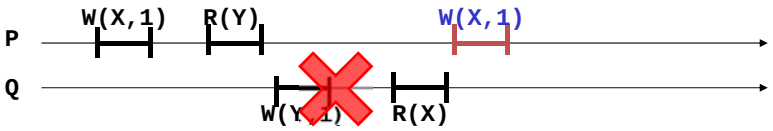
Read-After-Write (RAW) Reordering Leads to Inconsistency

Process P:

```
Write(X,1)
Read(Y)
```

Process Q:

```
Write(Y,1)
Read(X)
```



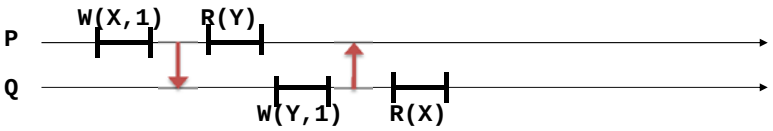
Avoiding RAW Reordering: Fences

Process P:

```
Write(X,1)
FENCE
Read(Y)
```

Process Q:

```
Write(Y,1)
FENCE
Read(X)
```



Avoiding RAW Reordering: Atomic Operations

Atomic-write-after-read (**AWAR**)

E.g., CAS, TAS, Fetch&Add,…

```
atomic{
   read(Y)
   …
   write(X,1)
}
```

RAW fences / AWAR are slow

But they cannot be avoided

One fence is necessary

Any mutex algorithm must have read-after-write unless it has atomic-write-after-read

Result also holds for concurrent data types

- **Strongly non-commutative** operations (queues, counters…)
- Linearizable solo-terminating implementations

[Attiya, Guerraoui, Hendler, Kuznetsov, Michael, Vechev: POPL 2011]

Scalable commutativity rule

[Clements, Kaashoek, Zeldovich, Morris, SOSP 2013☼, ACM TOCS 2015]

Whenever interface operations commute, they can be implemented in a way that scales

- **Commuter** tests commutativity of high-level interfaces Applied to POSIX file and virtual memory system operations, highlights Linux kernel problems
- A new POSIX-like operating system scaling for many tests, which has linear scalability in specific situations

Back to Theory : Is Lower Bound Tight?

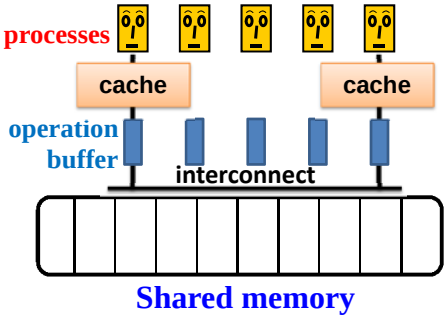
- Lamport's Bakery algorithm needs only O(1) fences (one for every write)
- But many reads
- And they are **remote**

Not All Memory Accesses are Equal

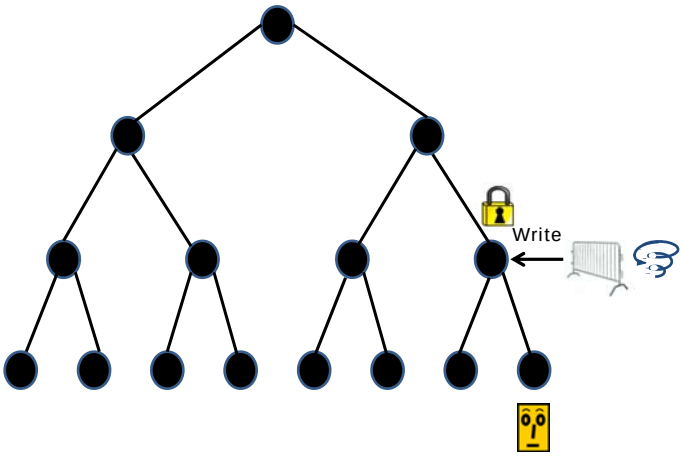
Cache coherent (**CC**) model
- accesses served from cache: "cheap"
- Remote Memory References (**RMRs**): "expensive"

Similar **DSM** model

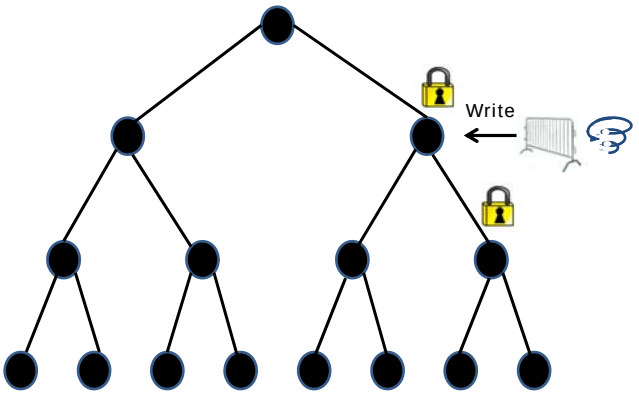
RAW fence still ~60 slower than RMR



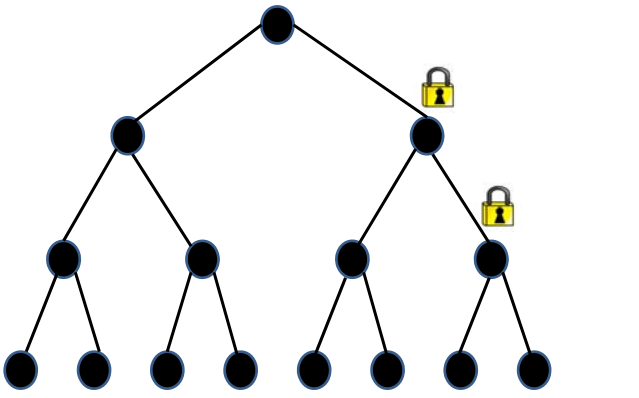
Tournament-tree: entry section



Tournament-tree: entry section



Tournament-tree: entry section



Fences vs. RMRs

	RMRs	Fences
Tournament [Yang, Anderson]	Θ(log n)	Θ(log n)
Bakery [Lamport]	Θ(n)	O(1)
Without write reordering [Attiya, Hendler, Levy, PODC 2013]	Θ(log n)	O(1)
With write reordering	NO	NO

- **Ω(log n) RMRs** lower bound for (deterministic) mutex

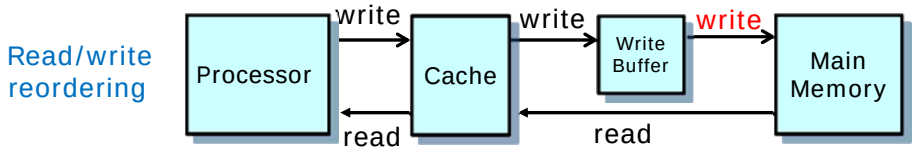
  [Attiya, Hendler and Woelfel, STOC 2008]

- Can we get the best of both worlds?

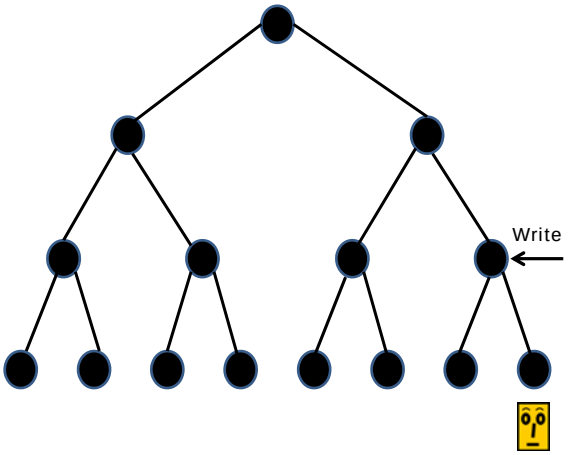
Total Store Ordering (TSO) Model

Just read-after-write reordering, e.g., Intel x86

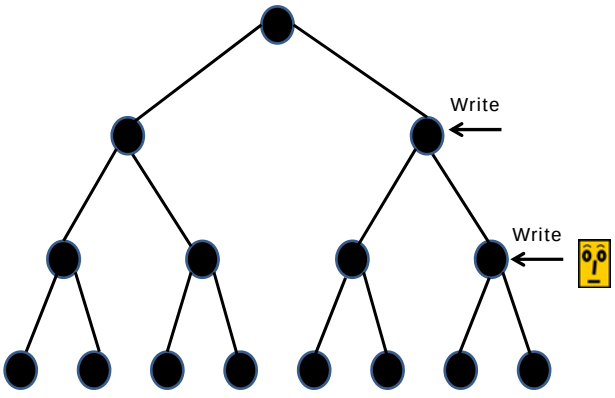
A fence flushes the write buffer, **in-order**



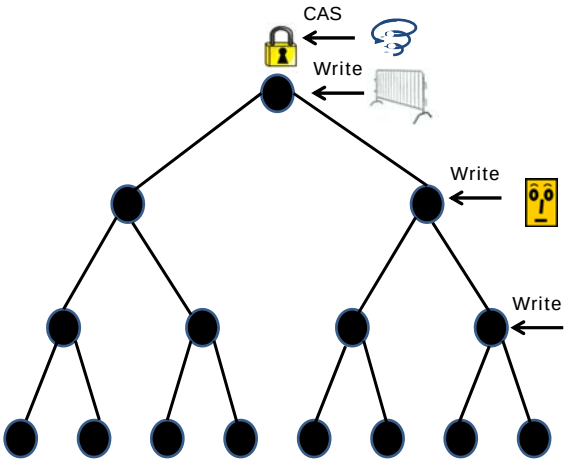
One-Fence Mutex: Entry Section



One-Fence Mutex: Entry Section



One-Fence Mutex: Entry Section



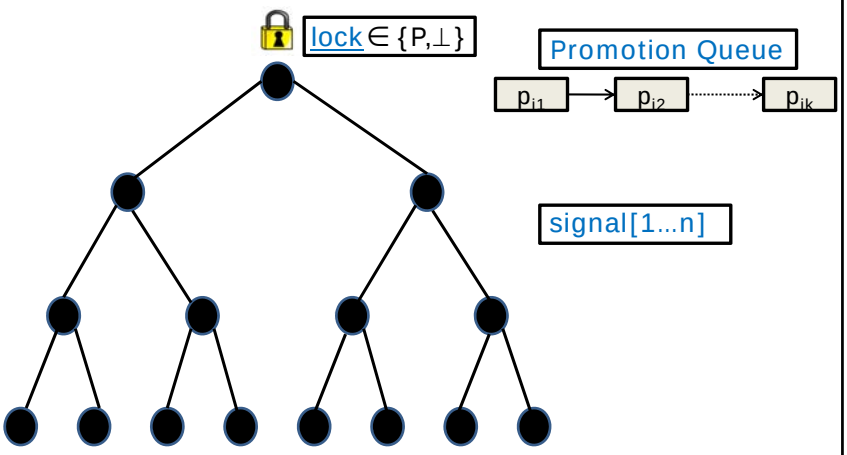
Our Algorithm: Promoting Your Peers

When exiting the critical section, processes look around to see who's waiting and promote them

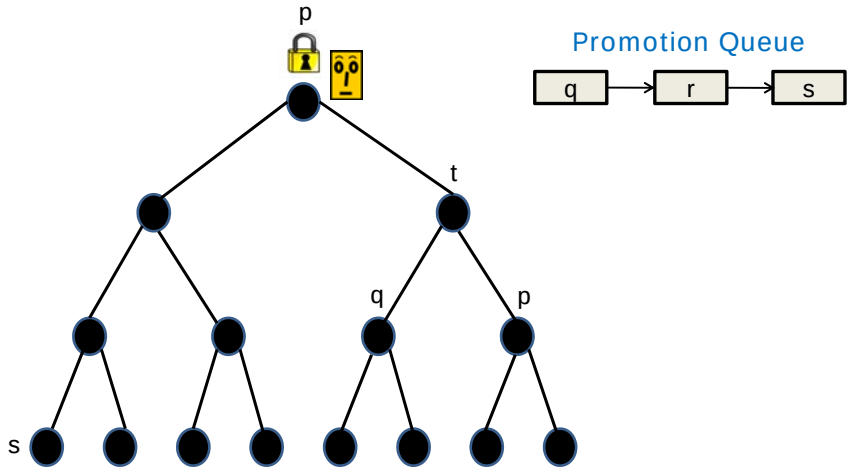
Place in a queue of waiting processes

Ensure that waiting processes are promoted, and hence, not starved

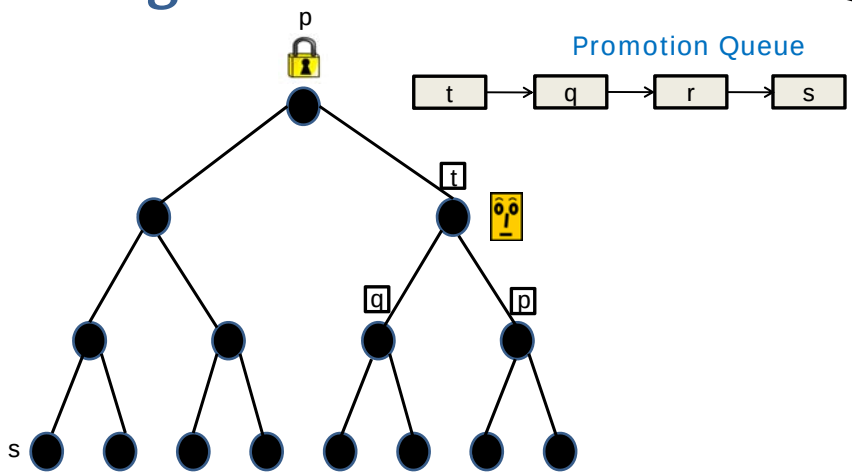
One-Fence Mutex: Variables



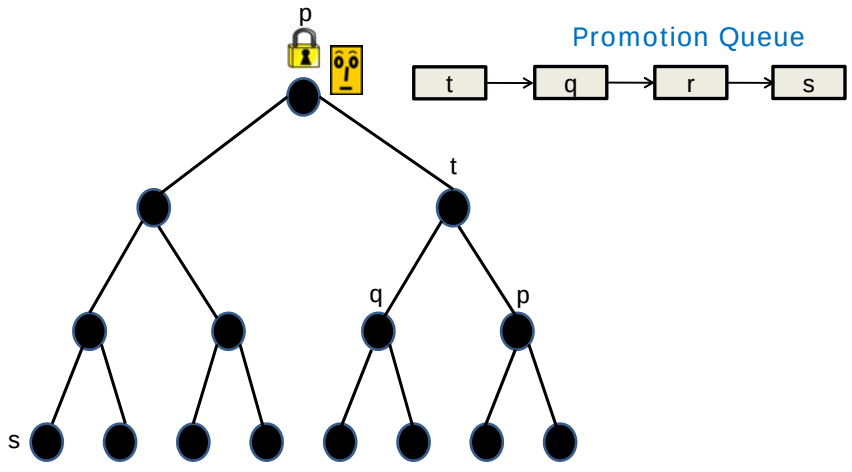
One-Fence Mutex: Exit Section



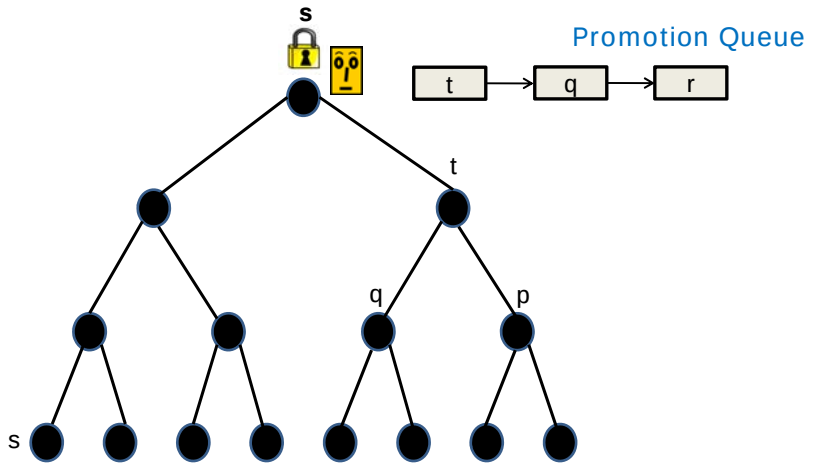
New Algorithm: Exit section



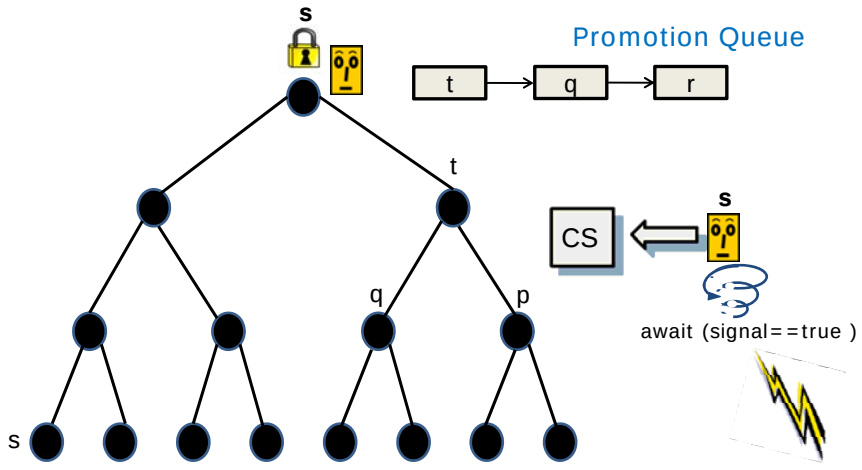
Exit section: Scenario 1



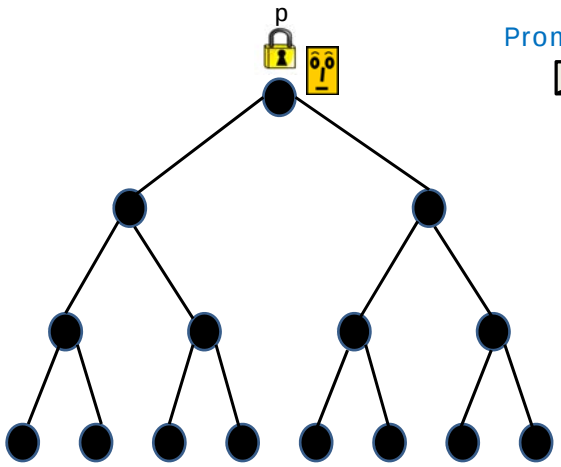
Exit section: Scenario 1



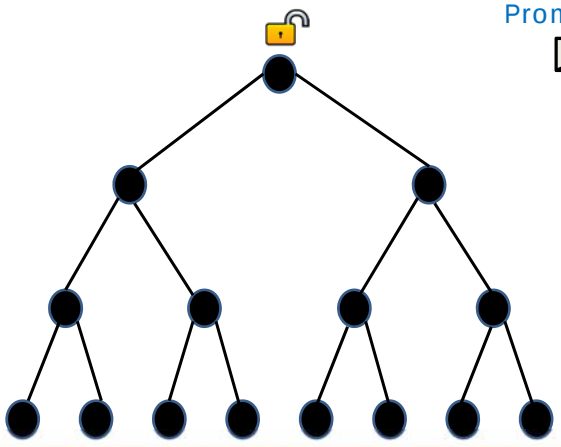
Exit section: Scenario 1



Exit section: Scenario 2



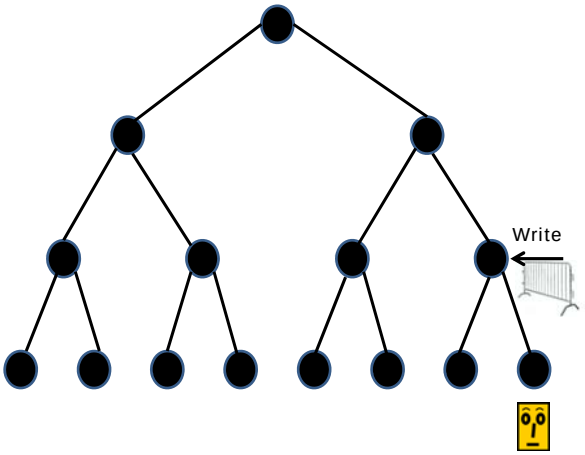
Exit section: Scenario 2



Proving correctness relies on writes occurring in order (along the leave-to-root path)

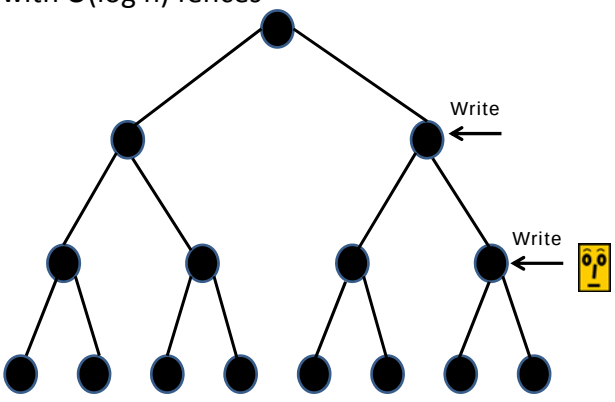
When Writes can be Reordered

Insert a fence in each node



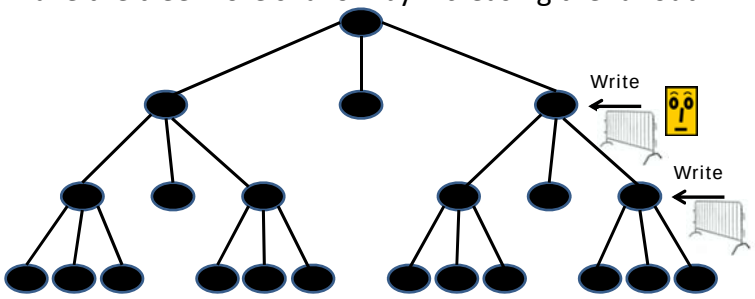
When Writes can be Reordered

Back with O(log n) fences



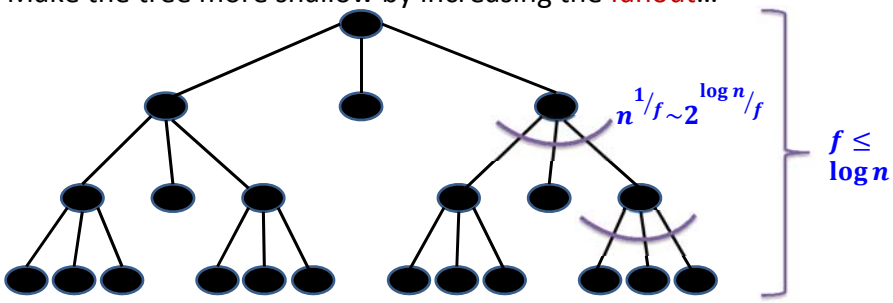
Fewer Fences?

Make the tree more shallow by increasing the fanout…



How Shallow?

Make the tree more shallow by increasing the fanout…



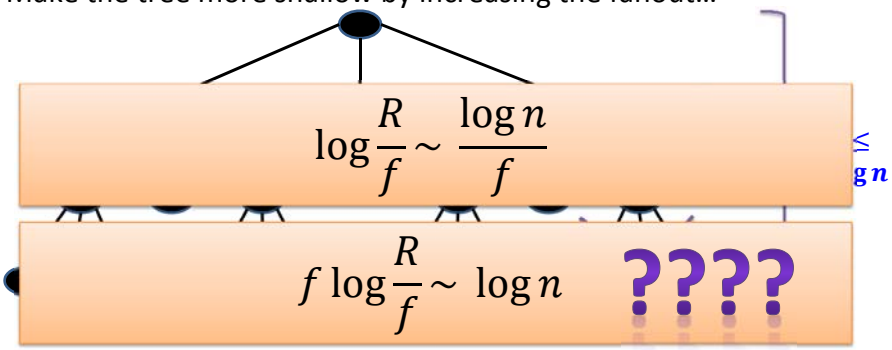
Per passage (entering and exiting the critical section):

Number of fences $O(f)$

Number of RMRs $\mathrm{R} = O\left(f \cdot 2^{\log n / f}\right)$

How Shallow?

Make the tree more shallow by increasing the fanout…



Per passage (entering and exiting the critical section):

Number of fences $O(f)$

Number of RMRs $R = O\left(f \cdot 2^{\log n / f}\right)$

A Tradeoff Between Fences and RMRs

For an execution α with one passage per process

F_α = total number of fences

R_α = total number of RMRs

When writes can be reordered, every mutex algorithm has an execution α, in which

$$F_\alpha \log \frac{R_\alpha}{F_\alpha} \in \Omega(n \log n)$$

[Attiya, Hendler and Woelfel, PODC 2015]

Proof Strategy I

Given a permutation $\pi = [q_1, q_2, \ldots, q_n]$
construct a unique execution α_π, in which

- processes enter the CS ordered according to π
  ⇨Can reconstruct π from α_π:
  processes concatenate their id's once in the CS

q_1, q_2, q_3

- if $j > i$, then q_i does not see q_j

Proof Strategy II

Encode α_π with m commands

$$C_1(k_1), C_2(k_2), \ldots, C_m(k_m)$$

such that

$m \sim$ number of fences in α_π (F_π)

$k_1 + \ldots + k_m \sim$ number of RMRs in α_π (R_π)

⇨ Requires at least $F_\pi \log \frac{R_\pi}{F_\pi}$ bits

Proof Strategy III

$n!$ permutations & can find π from α_π

⇨ for some π, $\Omega(n \log n)$ bits are needed to encode α_π

⇨ $F_\pi \log \frac{R_\pi}{F_\pi} \in \Omega(n \log n)$ ✓

Constructing α_π for $\pi = [q_1, \ldots, q_n]$

For each process q_i, put all writes in a reordering buffer until there is a fence

Then, encode together all writes in the reordering buffer with ≤ 4 commands:

- Wait for earlier processes to cover your writes
- Wait for earlier processes to finish so they won't read your writes
- Proceed
- Commit

Each read gives one RMR

Each covering gives one RMR

A fence every ~4 commands

Must Be Non-Adaptive

[Ben-Baruch, Hendler, PODC 2015]

No adaptive mutex has RMR complexity depending on # participants and constant fence complexity

– Regardless of RMR complexity

Ω(log log n) fences needed for mutex with linear (or sub-linear) dependency on # participants

– With compare-and-swap, read and write

Fence-free implementations on bounded TSO processors

[Morrison, Afek, ASPLOS 2014]

In many x86 and SPARC TSO systems, a load bypasses a bounded # stores (bounded TSO)

⇨Work stealing with fence-free worker

[Morrison, Afek, ASPLOS 2015]

In TSO, store buffer often flushed in bounded time

⇨ fence-free algorithms for asymmetric problems

– hazard pointers memory reclamation scheme

– biased lock

Fence Complexity of STMs

[Kuznetsov, Ravi, OPODIS 2011]

- In permissive STMs, the number RAW/AWARs is linear in the transaction's read-set size
- Progressive STMs can be implemented with less than one RAW/AWAR per transaction
- Still, lock or RAW/AWAR on data that is linear in the transaction's write-set size

Wrap-Up

- More accurate models for shared-memory multiprocessors, and their costs, e.g. # fences
  - RAW/AWARs in randomized mutex

[Alistarh, Aspnes, DISC 2011]

- Relate to semantic properties of objects & implemented operations
- Optimal fence insertion