Advanced Systems Topics

CST Part II, Lent 2007

4 lectures of 15

Scalable synchronization

Keir Fraser

keir.fraser@cl.cam.ac.uk

Course Aims and Structure

Part I: Scalable Synchronization [KAF, 4L]

- Architecture and Algorithms for Work Sharing
- Cache Coherency
- Implementing Mutual Exclusion
- Lock-Free Data Structures

Part II: Advanced Operating Systems [DM, 6L]

- Distributed & Persistent Virtual Memory
- Capability Systems & The CAP Computer
- Microkernels & Virtual Machine Monitors
- Extensible Operating Systems
- Database & Distributed Storage

Part III: Experimental Performance Analysis [SDK, 4L]

- Performance Metrics & Experimental Design
- Errors in Experimental Measurements & ANOVA
- Case Studies

Part IV: Worm Containment [MSC+MAC, 2L]

- Worm Attacks & Approaches to Containment
- Detection by Dynamic Data-Flow Techniques

First Four Lectures

Aims of this section of AST:

- to explore software techniques for developing applications for large multi-processor machines,
- to look at the hardware architecture of large shared-memory systems, and its impact on software performance,
- to describe how effective concurrency-control abstractions can be implemented,
- to introduce current research areas in mainstream concurrent programming.

Reference material:

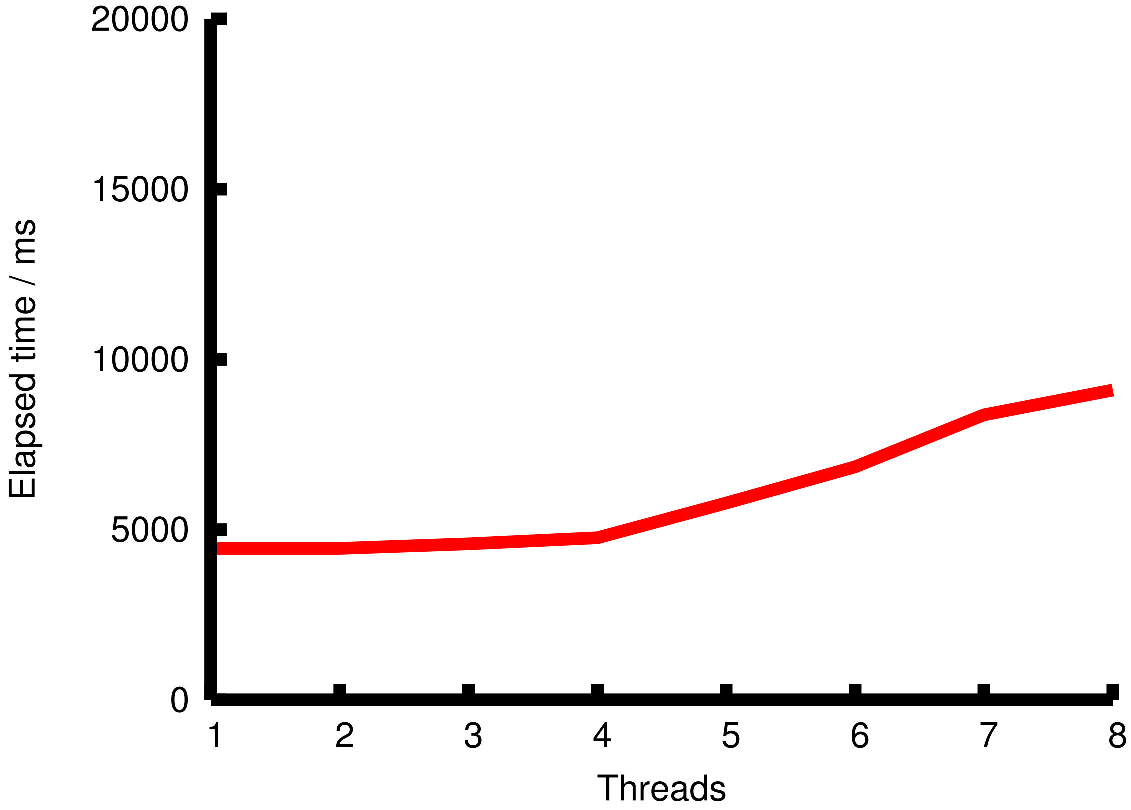
- `http://www.cl.cam.ac.uk/Teaching/2004/AdvSysTop/`
- Lea, D. (1999). *Concurrent Programming in Java*. Addison-Wesley (2nd ed.) – particularly Chapter 2
- Hennessy, J. and Petterson, D. *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann (3rd ed.) – particularly Chapter 6

Programming environment

- (i) hardware parallelism (chips/cores/threads)
  (ii) shared memory (*not* esoteric message-passing)
  (iii) cache-coherent (more on this later...)
- Modern uniprocessors use SMT (eg. Intel Hyperthreading)
- Multi-core is imminent, even on the desktop
- On x86, 2-way is common, 4-way and 8-way commodity
- SunFire 15k, 106-way:

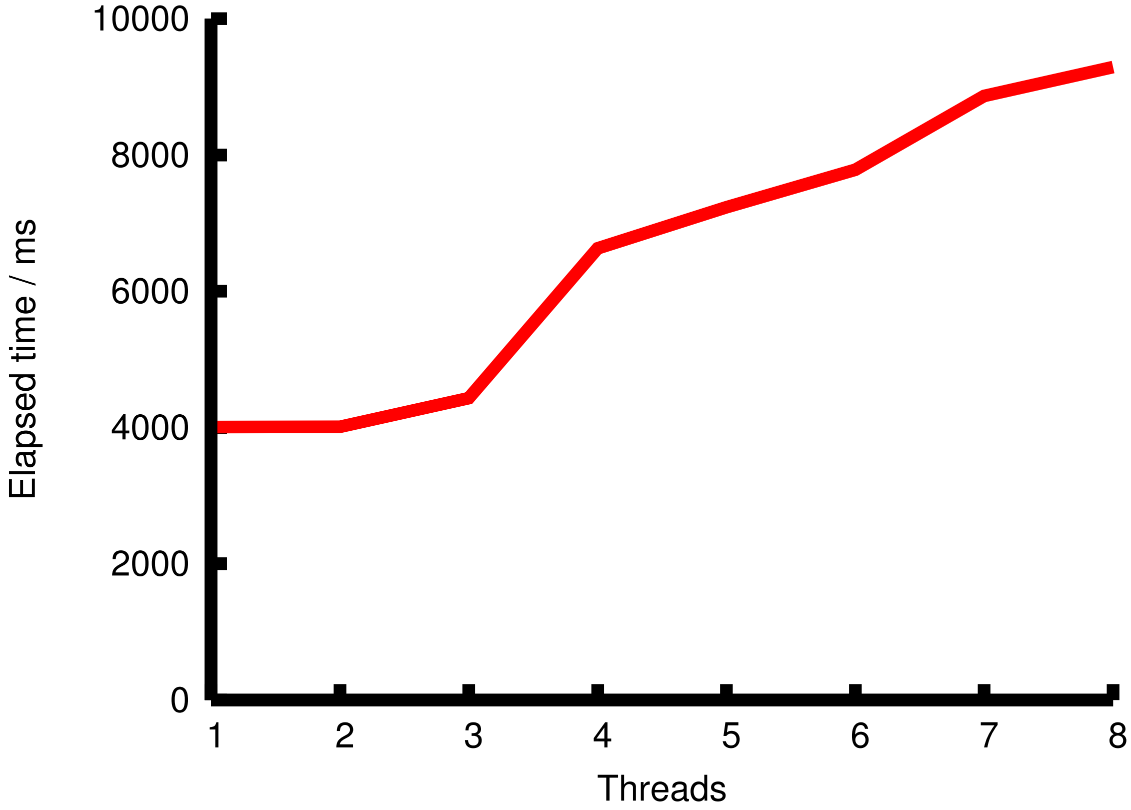
(191cm tall, 85cm across,
166cm deep, 1 000 kg)

A simple program



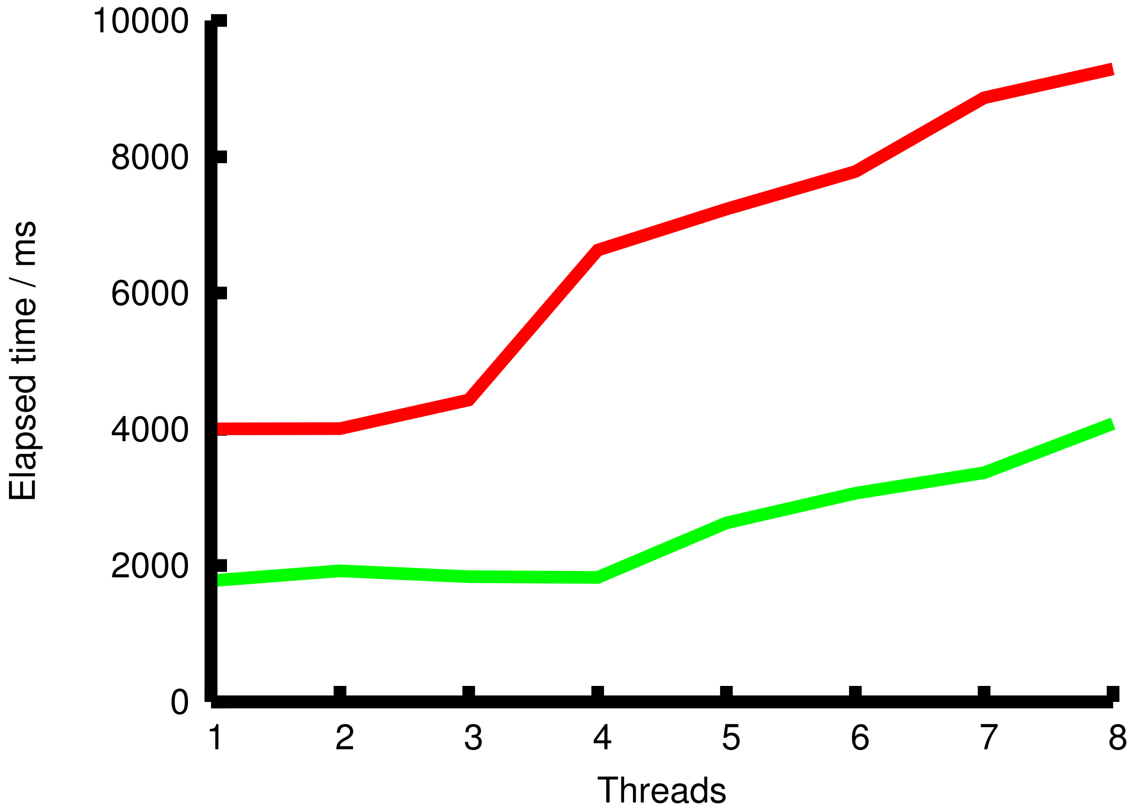
- 4-processor Sun Fire v480 server
- 1...8 threads run, each counting to 1 000 000 000
- Measure the wall-time it takes
- System load (number of runnable threads, in this case) grows 1..8

Another simple program



- Each thread has a reference to a shared hashtable
- Loops storing an entry in it (1 time in 1000) or reading entries from it (999 times in 1000)
- Using the built-in `java.util.Hashtable` class

Another simple program (2)



- This time using `ConcurrentHashMap` from Doug Lea's "`util.concurrent`" package
- Always faster. Scales better over 1..4 processors

This course is about designing software that performs gracefully as load increases and that makes effective use of all of the resources available e.g. on that 106-processor box

Motivating examples

Multi-threaded servers

- Improving server architecture over basic designs that use one thread per client (Zeus vs Apache)
- Implementing effective shared data structures (`ConcurrentHashMap` vs `Hashtable`)

Work sharing

- Tasks that can readily be partitioned between numbers of threads
- Coarse granularity
  - parallel ray tracing
  - little communication once started
- Fine granularity
  - parallel garbage collection
  - potential for lots of communication
- Usually 1 thread per available processor

General approach

1. Identify tasks that can usefully operate in parallel
   - Top/middle/bottom thirds of an image in the ray tracing example
   - Returning files to different clients
2. Ensure that they operate safely while doing so
   - e.g. basic locking strategies (as in `Hashtable` and CS&A last year)
3. Make it unlikely that they 'get in each others way'
   - Even though its safe to do so, there's no benefit from running 2 CPU-bound tasks on a simple uniprocessor
   - e.g. copying a shared read-only data structure so that locking is needed less often
4. Reduce the overheads introduced by their co-ordination
   - e.g. usually silly to start a new thread to do a tiny amount of work
   - Specialized algorithms exist for some cases, e.g. shared work queues operating correctly without any locks

General approach (2)

- In many cases step 3 comes down to avoiding *contention* for resources that cannot be shared effectively and encouraging *locality* to aid caching of those that can
- e.g. locks, CPU time, cache lines, disk
- In each case sharing them introduces costs which are not otherwise present
  - Waiting for resources held under mutual exclusion
  - Lock acquisition / release times
  - Context switch times
  - Management of scheduler data structures
  - Pollution of caches (both hardware and software-implemented)
  - Invalidation of remote hardware cache lines on write
  - Disk head movements
- Remember from CSM: it's the *bottleneck* resource which is ultimately important

Example: static-content web server

Single-threaded, one client at a time

Disc is the bottleneck. Thread is blocked on I/O most of the time. CPU time and disc bandwidth are *under-utilised*.

Thread per client

Threads can share CPU during each others idle periods. Can spread threads across multiple CPUs. Increased disc bandwidth due to scheduling of concurrent I/O requests.

But what happens when we hit a bottleneck: e.g., maximal disc throughput. If we continue to create a thread for every client request then the system is *overloaded*: service time increases; service rate remains constant or even **decreases**!

Thread pool

Could cap maximum number of threads simultaneously active. Extending this idea, we can avoid ongoing cost of thread creation and destruction by creating a *thread pool* during initialisation of the web server. Size the thread pool according to available system resources (statically or dynamically).

Java recap

- Encapsulate a thread's implementation inside a class that implements the `Runnable` interface
- Create a new thread by creating a `Thread` object and passing an implementation of `Runnable` to its constructor

```
public class Worker implements Runnable
{
   public void run()
   {
     ...
   }

   ...
}

public static void main()
{
   Worker w = new Worker();
   new Thread(w).start();
}
```

Java recap (2)

- Each object has an associated *mutual exclusion lock* (mutex) which can be held by at most one thread at any time
- If a thread calls a `synchronized` method then it blocks until it can acquire the target object's mutex
- No guarantee of fairness – FIFO or other queueing disciplines must be implemented explicitly if needed

```
public class Hashtable
{
   ...

   public synchronized Object get(Object key)
   {
     ...
   }

   public synchronized Object put
                     (Object key, Object value)
   {
     ...
   }
}
```

Java recap (3)

- Each object also has an associated *condition variable* (condvar) which can be used to control when threads acting on the object get blocked and woken up
  - `wait()` releases the mutex on `this` and blocks on its condition variable
  - `notify()` selects one of the threads blocked on `this`'s condvar and releases it – the woken thread must then re-acquire a lock on the mutex before continuing
  - `notifyAll()` releases all of the threads block on `this`'s condvar
- These operations can only be called *when holding* `this`*'s mutex* as well

Java recap (4)

```
class FairLock
{
  private int nextTicket = 0;
  private int currentTurn = 0;

  public synchronized void awaitTurn()
        throws InterruptedException
  {
    int me = nextTicket ++;
    while (currentTurn != me) {
      wait();
    }
  }

  public synchronized void finished()
  {
    currentTurn ++;
    notifyAll();
  }
}
```

Fine-grained parallelism

- Often operations on structured data can be executed in parallel, e.g.

```
public void sum(int a[], int b[]) {
  for (int i = 0; i < a.length; i ++) {
    a[i] += b[i];
  }
}
```

- How can the compiler identify such code? What assumptions are needed for parallel execution to be safe in this case? How can it be executed effecitvely?
- OpenMP provides a set of pragmas through which the programmer can indicate where concurrent execution is safe, e.g.

```
#pragma omp parallel
#pragma omp for
for (int i = 0; i < a.length; i ++) {
  a[i] += b[i];
}
```

- Notice how the program is still correct for single-threaded execution

Fine-grained parallelism (2)

- Non-iterative work-sharing:

```
#pragma omp sections
  {
#pragma omp section
    <block-1>
#pragme omp section
    <block-2>
  }
```

- Other pragmas indicate sections the must be executed by only one thread (`single`), by one thread at a time (`critical`), or *barrier* points which synchronize all threads in a team, or `reduction` clauses to avoid data dependencies

```
#pragma omp parallel for reduction(+: a) \
                         reduction(||: am)
for (i = 0; i < n; i ++) {
  a += b[i];
  am = am || b[i] == c[i];
}
```

- The implementation is responsible for creating a suitable number of threads and deciding how to assign work to them

Exercises

1-1 Discuss how `FairLock` could be made robust against interruption.

1-2 A particular application can either be structured on a 2-processor machine as a single thread processing n work items, or as a pair of threads each processing $n/2$ items.

Processing an item involves purely local computation of mean length l and a small period of mean length s during which it must have exclusive access to a shared data structure.

Acquiring a lock takes, on average a and releasing it takes r.

(i) Assuming that s is sufficiently small that the lock is never contended, derive an expression showing when the single-threaded solution is faster than the 2-threaded one.

(ii) Repeat your calculation, but take into account the possibility of a thread having to block. You may assume (unrealistically) that a thread encounters the lock held with probability $s/(s+l)$

(iii) Why is the previous assumption unrealistic?

Exercises (2)

1-3 Compare the performance of the `FairLock` class against that of a built-in Java mutex. How would you expect `FairLock`'s performance to scale as the number of threads increases and as the number of available processors increases?

1-4* An application is being written to transfer two files over HTTP from a remote well-provisioned server. Do you think the overall transfer of both would be faster (i) using two concurrent connections to the server and two threads performing the transfers or (ii) performing the transfers sequentially. Explain why and any additional assumptions that you have made.

'Starred' exercises are outside the syllabus of the course and are included as extensions or as topics for discussion

Deploying threads

To separate out reasonably independent tasks

- ➤ Examples from CS&A last year – dealing with different clients, updating the screen vs computing the data to display
- ✔ In modern systems each thread can block/unblock independently
- ✘ Uncontrolled thread creation creates contention on the CPU
- ✘ Sometimes a risk of +ve feedback effects – high load causes poor response times causes clients to retry causing higher load...

To make effective use of available processing resources

- ➤ Divide a task into 2 sections on a 2-processor machine, into 4 on a 4-processor etc; no benefit having more
- ➤ "Correct" number of threads not known until runtime (and dependent on other system load)

Threading models

Kernel threads

- ➢ Operating system manages the threads, usually as separate processes sharing the same address space.
- ✔ The OS can balance threads across multiple CPUs, and can individually block the threads when they submit I/O requests.
- ✘ Can be moderately expensive to create.

User/language threads

- ➢ Implemented within an application library or language runtime.
- ➢ The threads are multiplexed onto a single kernel-visible process.
- ✔ Fast creation and context-switch times.
- ✘ Not possible to exploit CPU parallelism.
- ✘ Without care, a blocked thread blocks the entire application.

Deploying threads (2)

Decouple the ideas of

- **commands** – things to be done
  - Each line of a scene to ray-trace
  - Each object to examine in a garbage collector
  - Each request received from a remote client
- **executors** – things responsible for arranging the execution of commands

Commands should be lightweight to create – perhaps a single object in an application, or an entry on a queue in a garbage collector

Executors can be more costly to create and likely to be long-lived – e.g. having an associated thread

> Terminology here varies greatly: this course aims to follow usage from the `util.concurrent` toolkit so you can see "real" examples alongside these notes

Commands & executors in util.concurrent

- A command is represented by an instance of `Runnable`, holding encapsulated state and supplying its code as a `run` method:

```
public interface Runnable {
  public abstract void run();
}
```

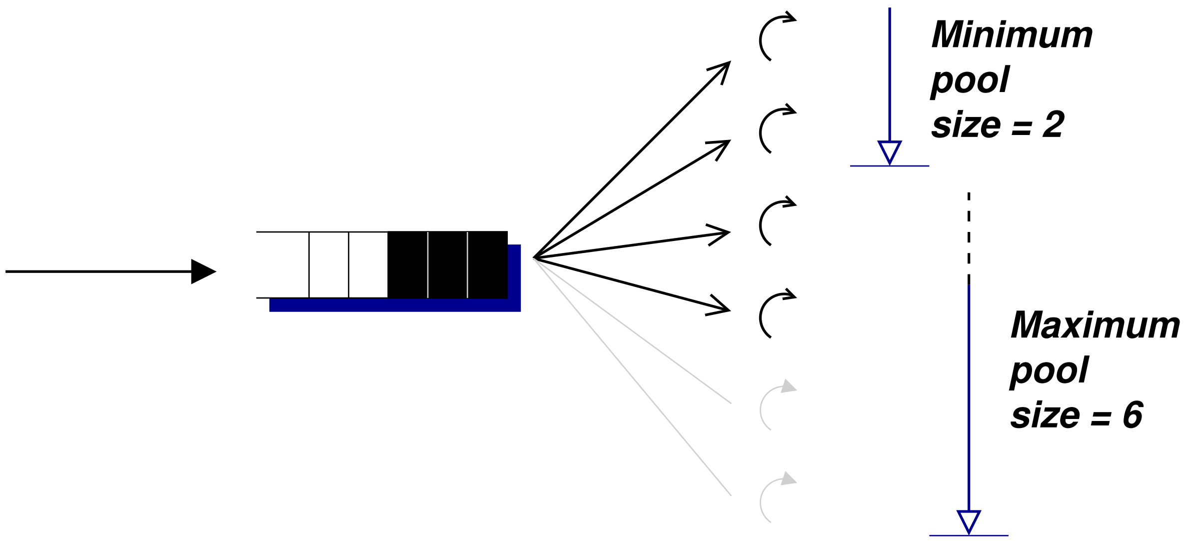
- An executor is interacted with through a similar interface by passing a command to its `execute` method:

```
public interface Executor {
  public void execute(Runnable command)
        throws InterruptedException;
}
```

Different implementations indicate common approaches

- `DirectExecutor` – synchronous
- `LockedExecutor` – synchronous, one at a time
- `QueuedExecutor` – asynchronous, one at a time
- `PooledExecutor` – asynchronous, bounded # threads
- `ThreadedExecutor` – asynchronous,
  unbounded # threads

Thread pools



- Commands enter a single queue
- Each thread taking commands from the queue, executing them to completion before taking the next
- Items are queued if 6 threads are already running
- New threads are created if fewer than 2 are running
- The queue size can be bounded or unbounded
- How to signal queue overflow?
  - Block the caller until there is space in the queue
  - Run the command immediately
  - Signal an error
  - Discard an item (which?) from the queue

Thread pools (2)

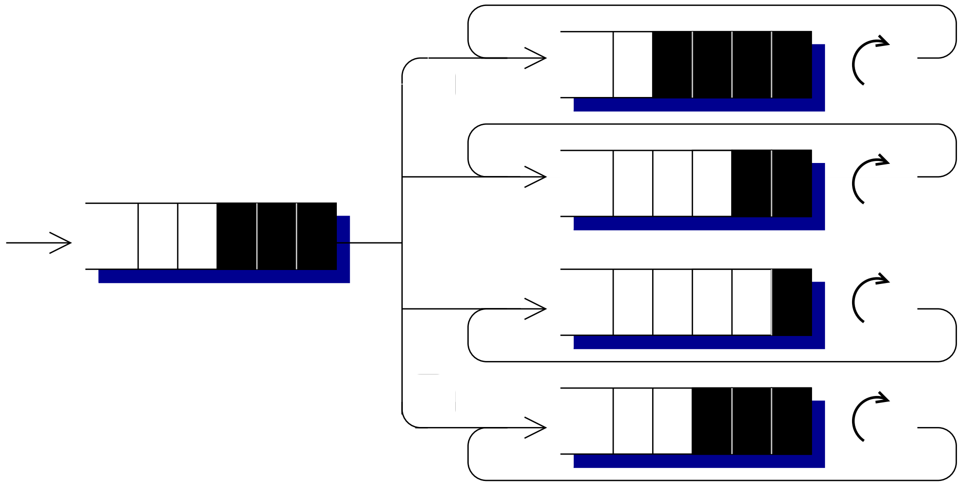
What about commands that are not simple things to run to completion?

- ✘ Ones that block (e.g. ongoing interaction with clients)
  - With suitable independence we could maybe just increase the maximum pool size
- ✘ Ones that generate other commands (e.g. parallel GC)
  - Could just add them to the queue, but...
  - ...may harm locality
  - ...also, what if the queue fills?

Good solutions depend on the application, but common approaches are:

- ➤ Use asynchronous I/O so that a 'command' is generated in response to one I/O completing
- ➤ That new command is then responsible for the next step of execution (e.g. replying to one client command) before issuing the next I/O
- ➤ Encourage affinity between particular series of commands and particular threads

Thread pools (3)



- Provide separate queues for each active **worker thread**
- If command C_1 generates command C_2 then place it on the queue C_1 came from
- ...at the head or at the tail?
- Note the analogy (and contrasts) with thread scheduling

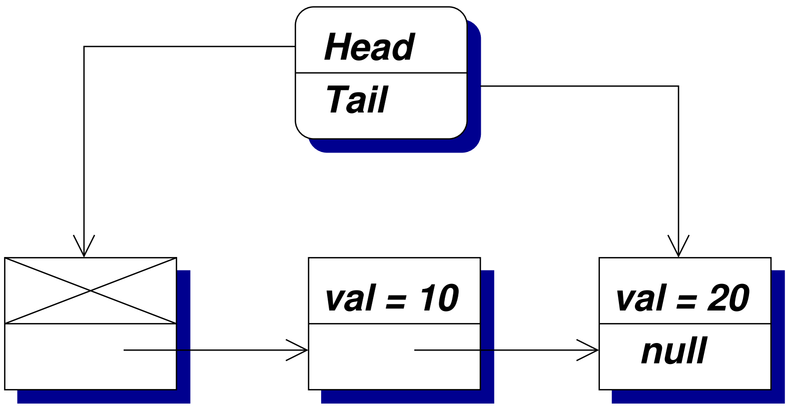
What happens if a queue runs empty?

- Take an item from another queue
- From the head or from the tail?
- One item or many?

Thread pools (4)

This motivates the design of specialized queues for **work stealing**

A basic version: Michael & Scott's 2-lock concurrent queue supporting concurrent `push_tail` and `pop_head` operations



- `Head` always points to a **dummy node**
- `Tail` points to the last node in the list
- Separate mutual exclusion locks protect the two operations – the dummy node prevents them from conflicting

More intricate designs provide one thread fast access to the head and stealing (of 1 item or 1/2 contents) from the tail

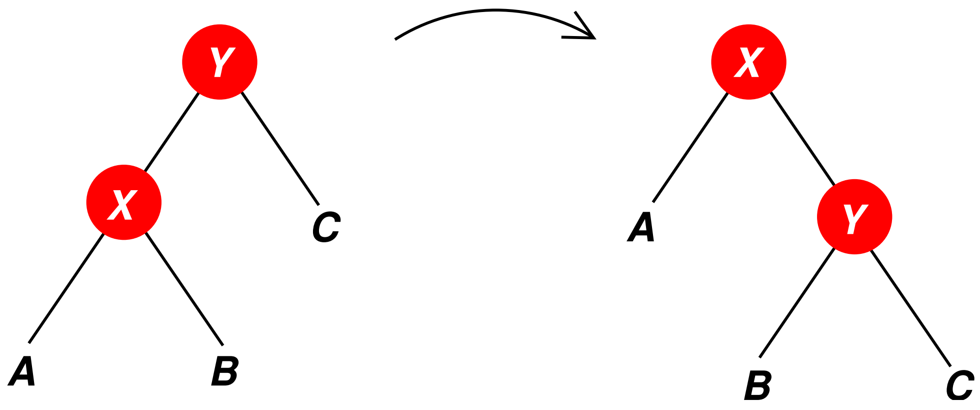
Reducing contention

We now turn to look at the shared data structures that are accessed during execution; what general techniques can we use (e.g. as on Michael & Scott's queue)?

- **Confinement**: guarantee that some objects must always remain thread-local → no need to lock/unlock them seperately
  - e.g. after locking some other 'controlling' object
  - e.g. per-thread copies of a read-only data structure
- **Accept stale/changing data**: particularly during reads → allow them to proceed without locking
  - What's the worst that can happen?
  - Can stale data be detected?
- **Copy-on-write**: access data through indirection and copy when updated → again, reads proceed without locking
  - Assumes writes are rare
  - e.g. lists of event recipients in Swing

Reducing contention (2)

- **Reduce locking granularity**: lots of 'small' locks instead of a few 'large' ones → operations using different locks proceed in parallel
  - Need to think about deadlock again
  - Not a magic solution
- Simple per-node locks in a red-black tree:



- Even read operations need to take out locks all of the way from the root
- Otherwise, suppose one thread is searching for **A** and has got to node **X**, another thread performs a rotation...
- We'll return to this in the context of lock-free designs in Lecture 10.

Reducing contention (3)

Example: `ConcurrentHashMap`

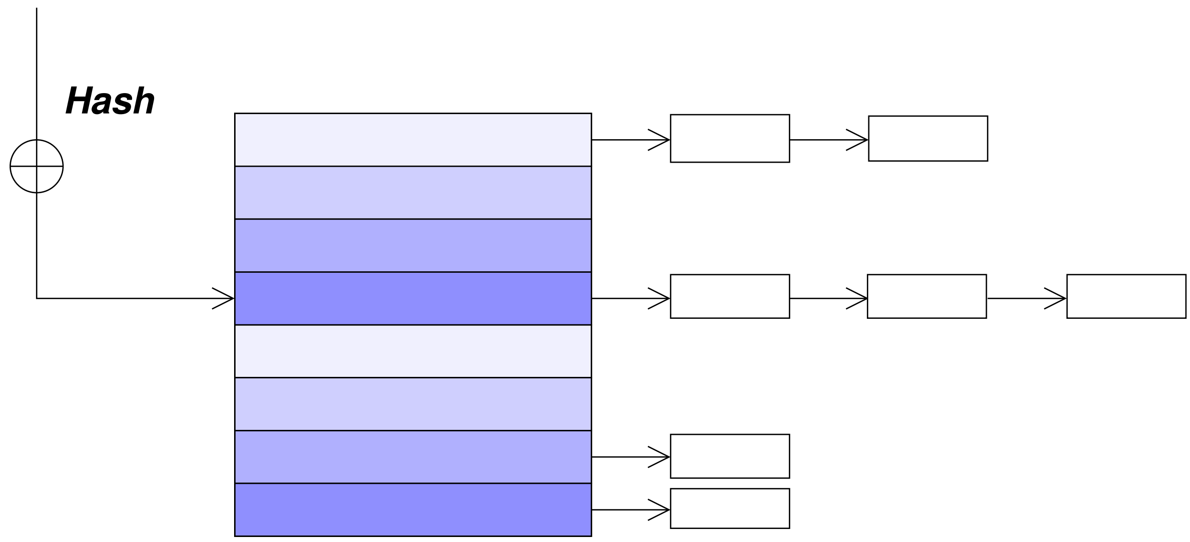


Table divided into *segments* (shown here as the same colour).
One update lock per segment.

Read operations:

- Proceed without locking
- If successful then return
- If failed then acquire segment lock and retry

Write operations:

- Acquire segment lock required
- If resizing then acquire all segment locks

Exercises

2-1 When might a thread pool be configured to create more threads than there are processors available?

2-2 Discuss the advantages and disadvantages of configuring a thread pool to use an unbounded input queue. Describe a situation in which each of the suggested overflow-management strategies would be appropriate.

2-3 A parallel garbage collector proceeds by taking objects to scan, one by one, off a per-thread queue. For each object it has to examine each of its fields and generate a new work item for each of the objects it encounters that has not been seen before.

Discuss the merits of placing these items on the head of the thread's queue versus the tail.

When the queue is empty, discuss the merits of stealing items from the head of another thread's queue versus the tail.

You do not need to consider the details of how a parallel garbage collector would work, but you may find it useful to consider how your algorithm would proceed with a number of common data structures such as lists and trees.

Exercises (2)

2-4 Design a double ended queue supporting concurrent `push_tail`, `pop_head` and `push_head` operations. As with Michael & Scott's design, you should allow operations on both ends to proceed concurrently wherever possible.

2-5* Now consider supporting `pop_tail`, `pop_head` and `push_head`. Why is this a much more difficult problem?

2-6* Examine either the `java.nio` features for asynchronous and non-blocking I/O in Java, or their equivalents in POSIX. Implement a simple single-threaded web server which can still manage separate clients.

Implementing mutual exclusion

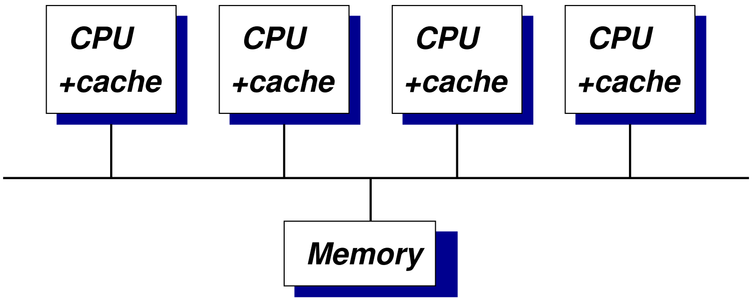
Many of the concurrency algorithms and language features we have seen so far depend on *mutual exclusion*.

- Implemented by mutual-exclusion locks (*mutexes*).
- Careful mutex implementation can have a huge impact on scalability.
- Locking protocol is also a design issue
  - A lock that admits multiple readers into their critical sections reduces serialisation, but complicates the locking protocol.
- The correct choice depends on:
  - workload
  - memory-access costs
  - scalability requirements

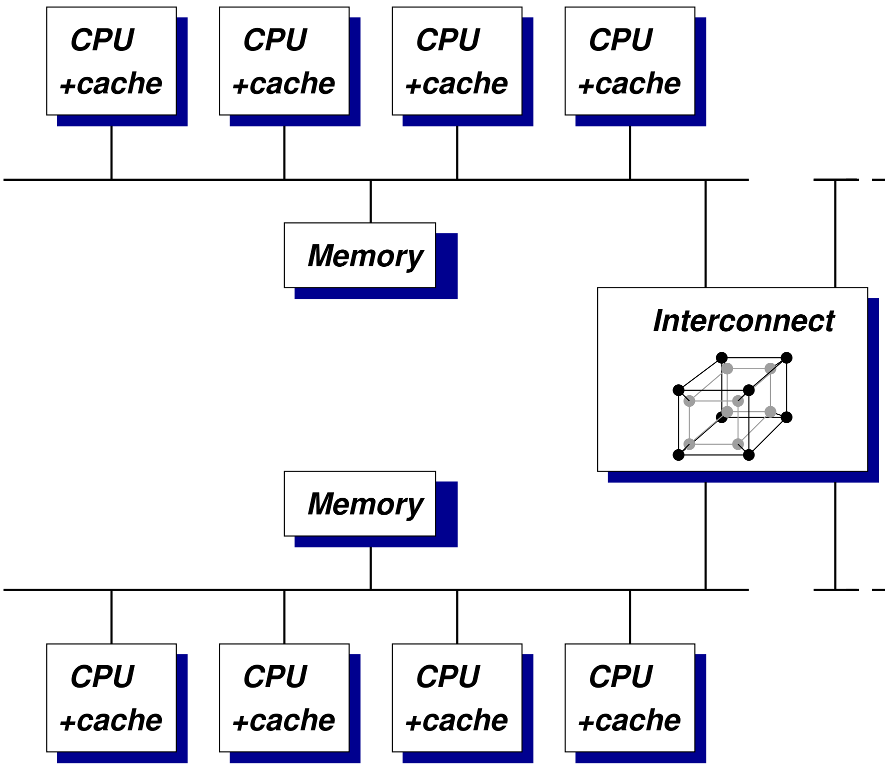
Before investigating lock designs and their tradeoffs, we must take a step back and look at the hardware architecture of the systems we are designing for...

Multi-processor system architectures

Cache-coherent shared memory multiprocessor, with either uniform memory access from each CPU:



or non-uniform access (ccNUMA):



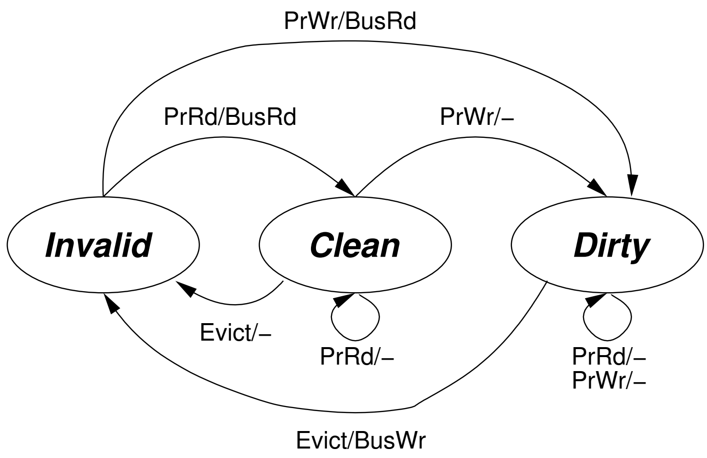
Recap: Uniprocessor cache coherence

- Modern CPUs implement a *write-back* policy
- Modified cache lines written back to memory only when they are replaced.
- Avoid writing back umodified cache lines on replacement.
- Associate a state with each data line in the cache:

Dirty Line is modified: data in main memory is stale.

Clean Line is clean: data in main memory is up to date.

Invalid Line is invalid.

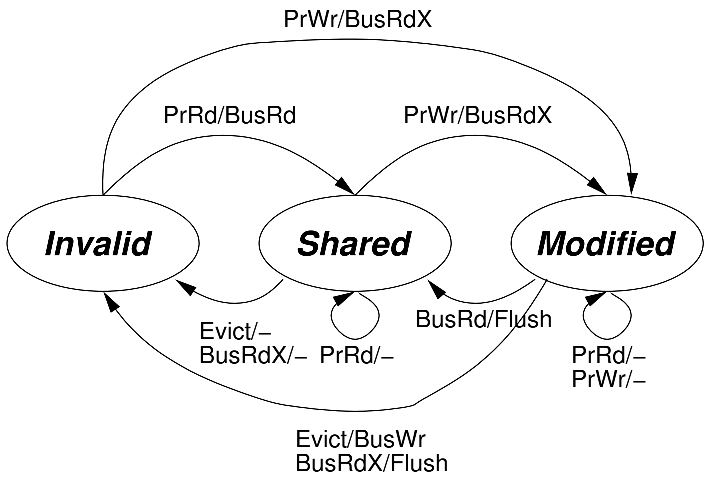


- A dirty cache line can be read or written by the CPU. It must be written back to main memory when replaced.
- A clean cache line need not be written back when it replaced. Upgraded to dirty state if it is modified.

SMP cache coherence

- Coherency requires that there is at most *one* dirty version of a cache line.
- But a line can be cached read-only by multiple processors
- SMP systems with a single memory bus ensure this by *snooping* requests on the bus.
- We can adapt the three-state uniprocessor state machine to create the classic **MSI** protocol:

Modified Line is modified (equiv. of 'dirty' state)
Shared Line is clean (equiv. of 'clean' state)
Invalid Line contains no data



- New read request **BusRdX**: read for exclusive ownership.
- BusRdX is snooped by other caches: invalidate their copies.
- New action **Flush**: typically abort the remote bus cycle and write the cache line back to memory.

More complex snoopy protocols

Invalidate bus message

- Replaces BusRdX in the Shared $\longrightarrow$ Modified transition.
- Avoids an unnecessary data fetch.

MESI protocol

- Extends MSI with an **Exclusive** state.
- Cache line is clean but owned exclusively.
- Exclusive $\longrightarrow$ Modified needs no bus transaction.
- Needs a more complex BusRd protocol to determine if other CPUs have the line cached.

MOESI protocol

- Extends MESI with an **Owned** state.
- Cache line may be modified even though other CPUs may have the line cached in Shared state.
- Modifications must be transmitted to those remote CPUs.
- Good if there is high bandwidth link between CPUs

There are many, many variations...

ccNUMA cache coherence

- CPUs and their local memory grouped into *nodes*.
- An interconnect allows message passing between nodes.
  - No shared bus to serialise writes.
  - No shared bus to snoop for memory accesses.
- **Directories** provide a serialisation point
  - One directory per node
  - Tracks the caching of memory blocks belonging to its node
- CPUs issue read/write requests to the *home node* of the memory block
  - Local node: The node the request originates from
  - Home node: Where that block of memory lives
  - Remote node: Any node that has that block cached
- Worst-case access latency in a ccNUMA system:
  - 1. Local node $\longrightarrow$ Home node
  - 2. Home node $\longrightarrow$ Remote node
  - 3. Remote node $\longrightarrow$ Local node
- **Three-hop miss**

Cache coherence: performance impact

Memory access *much* faster when satisfied by a cache, e.g. from Hennessy & Patterson for 17-64 processors:

	Processor cycles
Cache hit	1
Local memory	85
Remote, in home directory	150
Remote, cached elsewhere	170

- Locality between CPUs and local data is important
- Servicing cache misses will dominate execution time in poorly designed algorithms (see Comp Arch for uniprocessor examples)
- Stealing data cached elsewhere is usually worst of all
  - The ccNUMA 'three-hop miss'
- Know the cache block size to prevent false contention
- Consumption of interconnect bandwidth is also a concern

Memory consistency

As a further complication, modern processors execute instructions *out of order*! (recall from Comp Arch)

- What does this do to the programming model?

CPU A	**CPU B**
x = 1	if (y != 0)
y = 1	print x

- Can CPU B print any value other than 1?
  - Typically the answer is **yes**!
  - CPU B may predict the branch taken and read x from its cache *before* evaluating the predicate.
- What *can* we depend on?
  - Aliased accesses by a single CPU are ordered
  - Writes are eventually visible to other CPUs
  - Aliased writes by different CPUs are serialised
- Processors often guarantee more than this
  - Sequential Consistency: very strong, but expensive
  - Speculative Processor Ordering: modern x86
  - See `http://www.cl.cam.ac.uk/~kaf24/mem.txt`
- Insert **memory barriers** as necessary to enforce ordering
- Programs that use mutual exclusion generally do not need to worry about this: it can be hidden by the locking protocol.

Basic spin-locks

Assume that we've got an operation `CAS` (**compare and swap**) which acts on a single memory location

```
seen = CAS (&y, ov, nv);
```

- Look at the contents of `y`
- If they equal `ov` then write `nv` there
- Return the value seen
- Do all of this atomically

```
class BasicSpinLock {
  private boolean locked = false;

  void lock () {
    while (CAS(&locked,false,true) != false) {}
  }

  void unlock () {
    locked = false;
  }
}
```

- What are the problems here?

Basic spin-locks (2)

- `CAS` must acquire exclusive access to the cache block holding `locked`
- This block will ping-pong between all of the processors, probably with the worst case "three-hop miss" penalty
- The interconnect will probably be saturated
- This will harm the performance of other processes on the machine, including that of the thread holding the lock, delaying its release

Is this any better:

```
class ReadThenCASLock {
  private boolean locked = false;

  void lock () {
    while (CAS(&locked,false,true) != false) {
      while (locked == true) { /*2*/ }
    }
  }

  void unlock () { locked = false; }
}
```

Basic spin-locks (3)

- ✔ Threads now spin at /*2*/ and only go for the lock when they see it available
- ✔ Any number of threads can now spin without causing interconnect traffic
- ✘ They'll stampede for the lock when it becomes available

Several options exist:

- ➤ Use a lock that allows greater concurrency (e.g. build MRSW out of CAS)
- ➤ Introduce a purely-local delay between seeing the lock available and going for it
  - Count to a large random number
  - Exponentially increase this
  - Re-check the lock after counting
- ➤ Explicitly queue threads and arrange that the one at the head of the queue acquires the lock next

MRSW locks

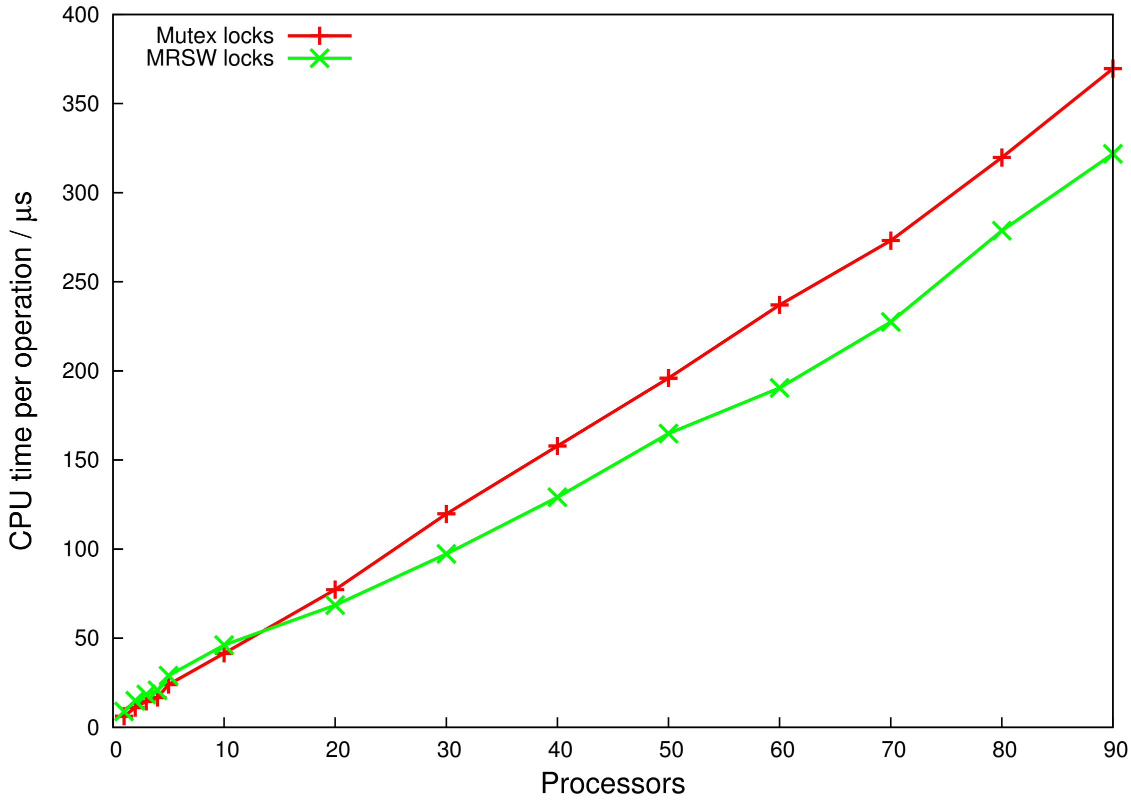
```
class MRSWLock {
  private int readers = 0; // -1 => writer

  void read_lock () {
    int seen;
    while ((seen = readers) == -1 ||
        CAS(&readers, seen, seen+1) != seen) { }
  }

  void read_unlock () {
    int seen = readers;
    while (CAS(&readers, seen, seen-1) != seen)
      seen = readers;
  }

  void write_lock () {
    while (readers != 0 ||
      CAS(&readers, 0, -1) != 0) { }
  }

  void write_unlock () {
    readers = 0;
  }
}
```

No silver bullet for performance

Red-black trees implemented over:

- A single spinlock (no concurrency)
- Per-node MRSW locks (parallel read accesses)

➤ The tree contains approximately half million nodes.
➤ Workload is 75% lookups, 25% insertions/deletions.
➤ Very low (real) contention.

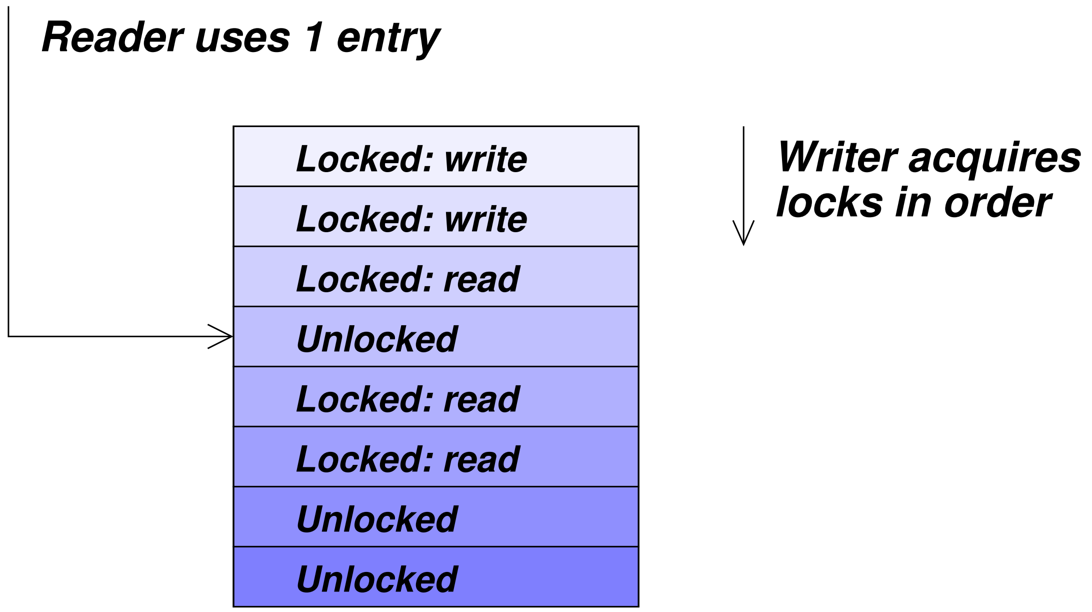


Poor scalability because of contention in the locking protocol!

➤ Always remember that performance is critically affected by the underlying primitives (locks, coherency protocol, ...), your choice of algorithm, and your workload

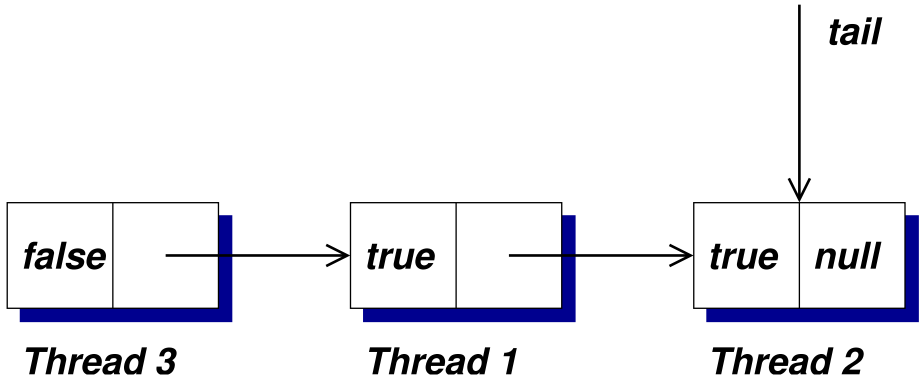
Linux "big reader" locks

- Supports `read_lock`, `read_unlock`, `write_lock`, `write_unlock` with usual MRSW semantics
- Assumes that read operations are *much* more common than write operations
- Built from per-CPU MRSW locks
- A reader just acquires the lock for that CPU
- A writer must acquire *all* of the locks



Queue-based spin locks

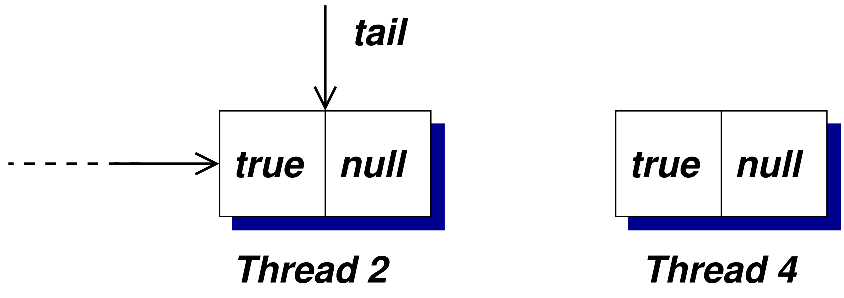
- Basic idea: each thread spins on an entirely separate location and keeps a reference to who gets the lock next:



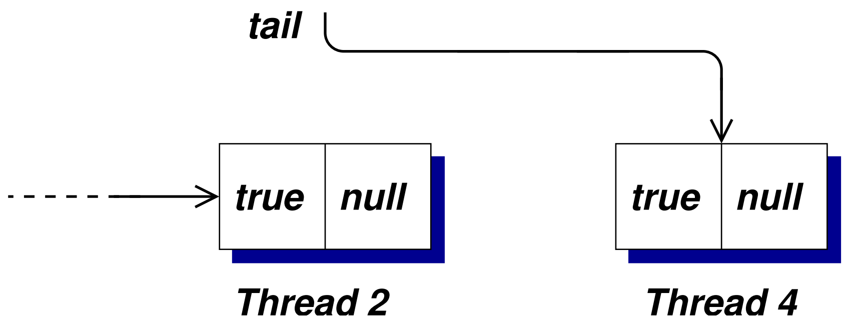
- Each **qnode** has a `next` field and a `blocked` flag
- In this case thread 3 holds the lock and will pass it to 1 and then to 2
- A shared `tail` reference shows which thread is last in the queue
- How do we acquire the lock (i.e. add a new node to the queue) safely without needing locks?
- How does one thread 'poke' the next one in the queue to get it to unblock?

Queue-based spin locks (2)

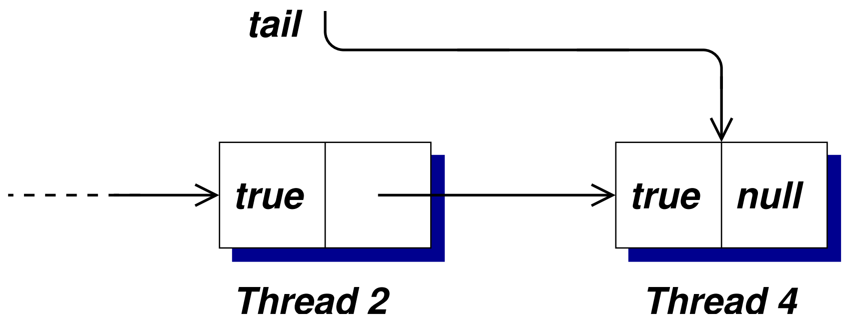
1. Suppose Thread 4 wants the lock. It prepares a new qnode in private storage:



2. It uses `CAS` to update `tail` to refer to its node:



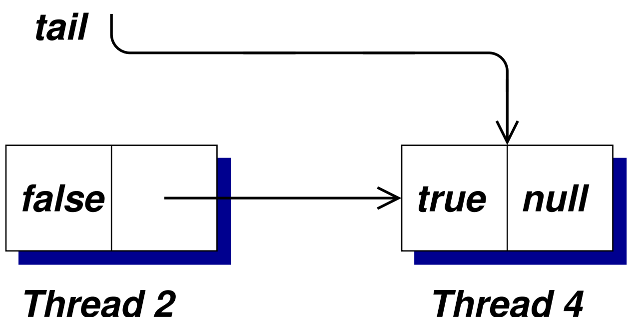
3. It writes to the `next` field of the previous tail:



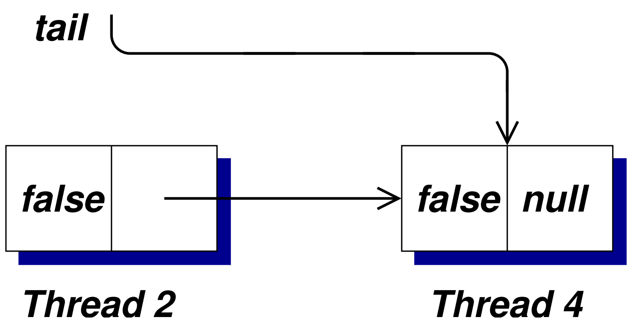
4. Thread 4 now spins watching the flag in its qnode

Queue-based spin locks (3)

Suppose Thread 2 now holds the lock:



If `next` is non-null (as here), wake the successor:



If `next` is null then either (i) there really isn't anyone waiting or (ii) another thread is between steps 2 and 3 on the previous slide:

- Thread 2 first tries to `CAS` the `tail` from itself to `null` (leaving no-one waiting)
- If that fails then someone must be waiting: spin watching `next` until the successor makes itself known

Queue-based spin locks (4)

- Note how the CAS used to update tail serves to define a total ordering between the threads that will acquire the lock
- It is critical that CAS returns the value that is seen when it makes its atomic update: this makes sure that each thread is aware of its immediate predecessor

This queue-based spin lock can be decomposed into two separate algorithms:

- The basic queue management using the next field:

  ```
  qnode push_tail (qnode q);
  qnode pop_head (qnode q);
  ```

  push_tail adds the qnode q to the tail of the queue and returns the previous tail

  pop_head removes the qnode q from the head of the queue and returns the new head
- ...and the actual blocking and unblocking

Exercises

3-1 The `BasicSpinLock` design is being used on a machine with n processors. Each processor wants to briefly acquire the lock and perform a small amount of computation before releasing it. Initially the lock is held and all processors are spinning attempting `CAS` operations.

Each access to the `locked` field takes, on average, 170 cycles and therefore vastly dominates the cost of executing other parts of the algorithm and indeed the work performed while holding the lock.

Estimate how many cycles will elapse between the lock first becoming available and all n processors having completed their work.

3-2 Explain why the `ReadThenCASLock` would be likely to perform better for even a moderate number of processors. Discuss the merits of rewriting the `lock` method to be:

```
void lock () {
  do {
    while (locked == true) { }
  } while (CAS(&locked,false,true) != false);
}
```

Exercises (2)

3-3 An implementation of Linux-style big reader locks for a 32-CPU machine uses the same basic scheme as the `MRSWLock` in these slides, but defines the array as:

```
int readers[] = new int[32];
```

Why is this a bad idea?

3-4 Develop a pseudo-code implementation of a queue-based spin lock, showing the memory accesses and `CAS` operations that are used.

3-5* To what extent is the queue developed for queue-based spin locks suitable as a general queue for work-stealing? Show how it can be extended to support an operation

```
void push_head(qnode prev_head,
               qnode new_head)
```

to push the qnode `new_head` onto the head of the queue, assuming that `prev_head` is currently the head

Disadvantages of mutual exclusion

Mutexes make it easy to ensure **safety** properties, but introduce concerns over **liveness**:

- Deadlock due to circular waiting
- Priority inversion problems
- Data shared between an interrupt handler and the rest of an OS
- Pre-emption or termination while holding locks

We've seen other performance concerns in this course:

- Programming with 'a few big locks' is easy, but may prevent valid concurrent operations (e.g. reads & writes on a hashtable using different keys)
- Programming with 'lots of little locks' is tricky (e.g. red-black trees) and juggling locks takes time
- Balancing these depends on the system's workload & configuration

Non-blocking data structures

A **non-blocking data structure** provides the following guarantee:

- The system as a whole can still make progress if any (finite) number of threads in it are suspended

Note that this generally precludes the use of locks: if a lock holder were to be suspended then the locked data structure remain unavailable for ever

We can distinguish various kinds of non-blocking design, each weaker than the one before:

- **Wait free** – per-thread progress bound
- **Lock free** – system wide progress bound
- **Obstruction free** – system wide progress bound if threads run in isolation

Theoretical results show that `CAS` is a **universal** primitive for building wait free designs – i.e. it can build anything

Non-blocking data structures (2)

A simple example of why `CAS` can easily implement any data structure:

- Suppose we have an complicated data structure without any concurrency control, e.g.

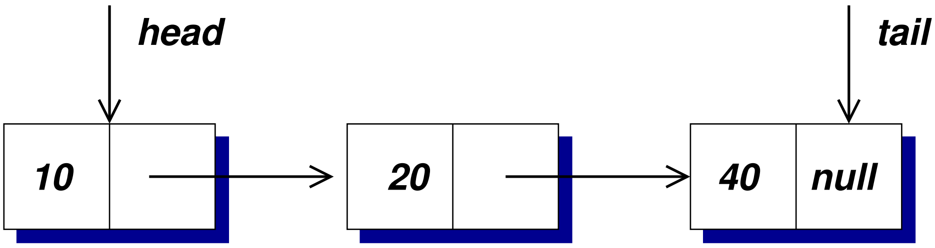
```
class FibonacciHeap {
   Object put(Object key, Object val) { ... }
   ...
}
```

- Access it through a level of indirection:

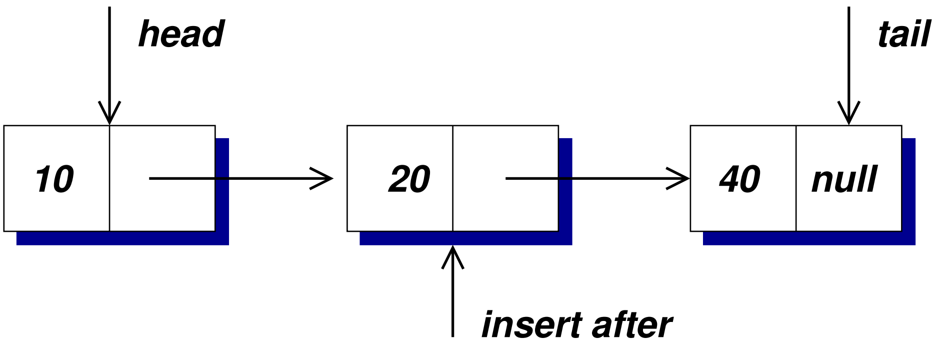
```
class LockFreeFibonacciHeap {
  private FibonacciHeap fh =
              new FibonacciHeap ();

  Object put(Object key, Object val) {
    FibonacciHeap copy, seen;
    do {
      seen = fh;
      copy = seen.clone ();
      copy.put (key, val);
    } while (CAS (&fh, seen, copy) != seen);
  }
  ...
}
```

Building directly from CAS

We'll now look at building data structures from scratch using `CAS`, e.g. consider inserting 30 into a sorted singly-linked list:

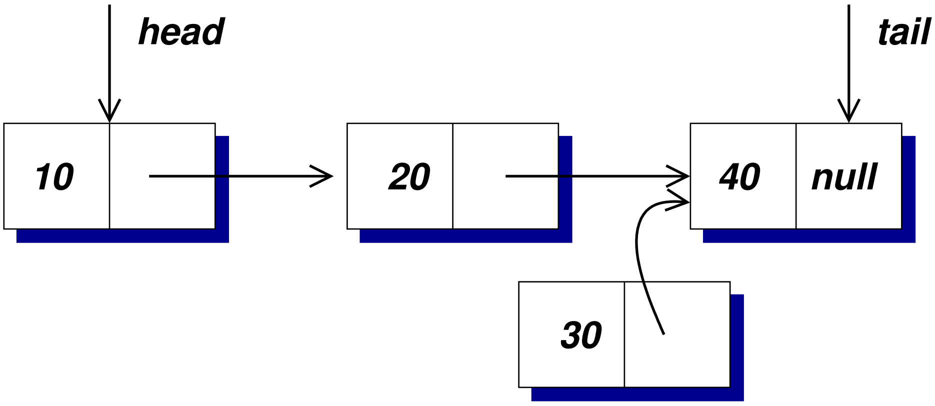


1. Search down the list for the node after which to make the insert:

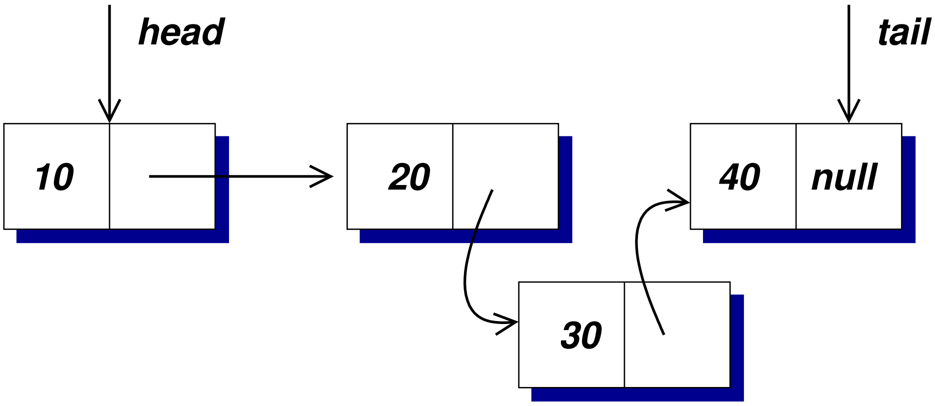


Building directly from CAS (2)

2. Create the new node in private storage:



3. Link it in using `CAS` on its predecessor:



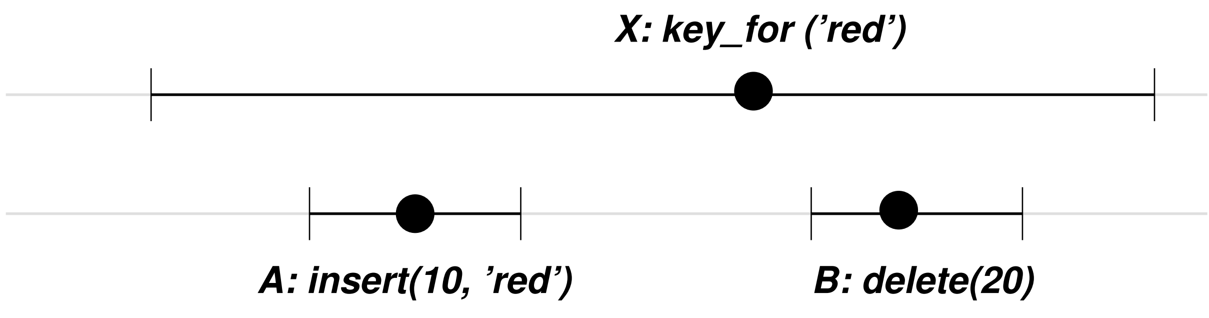
- The `CAS` will fail if the 20 node's successor is no longer the 40 node – e.g. if another thread has inserted a number there

Correctness

Suppose we now build a lookup table with lists of $(key,\ value)$ pairs. We want to also ask question such as "Which keys map to a particular value".

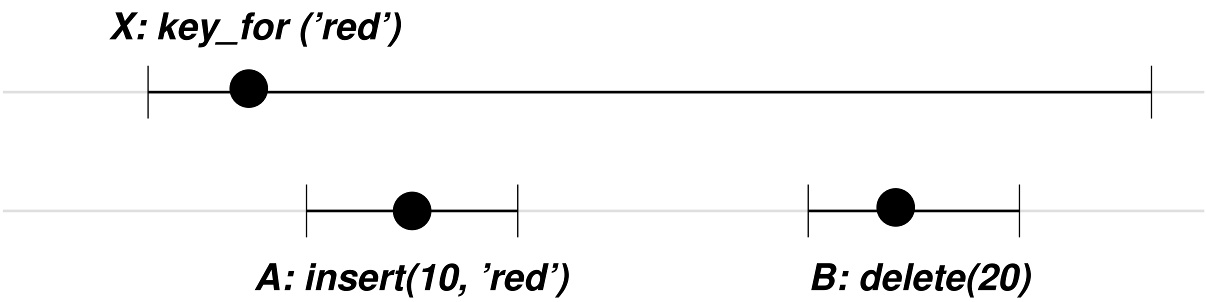
The table initially maps the key 20 to the colour 'red'

- One thread invokes `key_for('red')`
- Concurrently, a second thread invokes `insert(10, 'red')` then `delete(20)`
- What are the answers allowed to be?
- **A** OK, **B** OK, **X** returns $\{10, 20\}$ seems an intuitive option
- In that case, even though the operations take some time to run, they appear to occur atomically at the marked points:

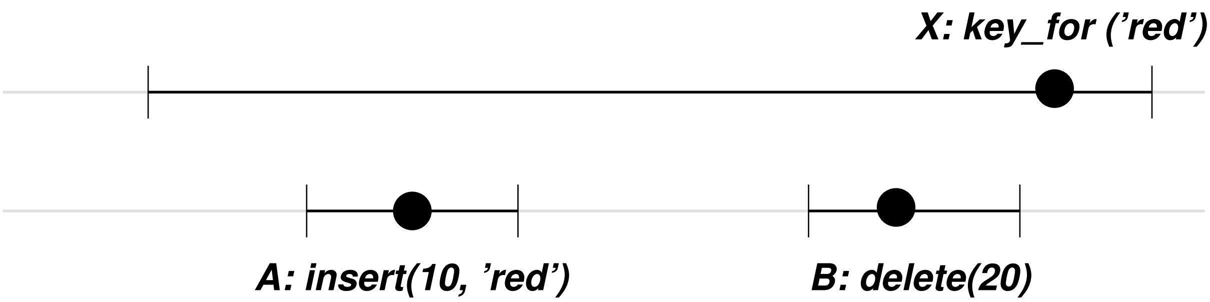


Correctness (2)

✔ **A** OK, **B** OK, **X** returns $\{20\}$ corresponds to:



✔ **A** OK, **B** OK, **X** returns $\{10\}$ corresponds to:



✘ **A** OK, **B** OK, **X** returns $\{\}$ doesn't correspond to any such execution – there's always some key associated with 'red'

- Suppose the keys are simply held in order and `CAS` is used to safely add and remove $(key,value)$ pairs
- The `key_for` implementation traverses down the list, gets to (say) 15, then **A** runs, then **B**, then **X** continues

Correctness (3)

This idea of correctness is known as **linearisability**:

- Each operation should appear to take place atomically at some point between when its invoked and when it returns
- Notice that this is more restrictive than serializability
- A linearizable non-blocking implementation can be used knowing only the operations it provides, not the detail of how they are implemented
- In many implementations this means identifying a single `CAS` operation (for updates) or a single memory read (for read-only operations) which atomically checks and/or updates all of the state that the result depends on
- Compound operations are still a problem – e.g. given two hashtables with linearizable operations, how to we do a 'transfer' that doesn't leave the item in both (or neither) in the middle...

Current research

- Programming with fine-grained locks is hard...
- programming without locks is even harder :-(
- A nice abstraction is a **software transactional memory** (STM) which holds ordinary values and supports operations such as

```
void STMStartTransaction();
word_t STMReadValue(addr_t address);
void STMWriteValue(addr_t address, word_t val);
boolean STMCommitTransaction();
```

- The STM implementation ensures that all of the accesses within a transaction appear to be executed in a linearizable manner
- We've developed a range of STMs supporting lock-free and obstruction-free updates
- In current research we're evaluating their performance and comparing 'simple' implementations of data structures, using them, to carefully engineered data structures (e.g. `ConcurrentHashMap`)

Current research (2)

- Another direction of research is exposing this to mainstream programmers as a language extension, e.g.

```
atomic {
  ...
}
```

- Anything within an `atomic` block would be implemented using the STM
- An extension to this is to allow threads to block mid-transaction until an update is made to an address that they are interested in, e.g.

```
do {
  atomic (!full) {
    full = true;
    value = new_val;
    done = true;
  }
} while (!done);
```

- This would block the current thread until `full` is false and then, atomically with that, perform the updates to `full`, to `value` and to the local variable `done`

Summary

We've seen a range of techniques for designing scalable concurrent applications for multiprocessors

The main points to remember:

- Lots of threads usually means lots of context switching: using a moderate number (e.g. #processors if they are CPU-bound) is often better
- Excessive contention and low locality will lead to poor performance: try to ensure threads can proceed using 'local' resources as often as possible
- Designing scalable shared data structures is hard and depends on the workload and the execution environment: higher level programming abstractions may help here

Exercises

4-1 Distinguish between the properties of a *wait-free* system, a *lock-free* one and an *obstruction-free* one. Which is most appropriate for a hard-real-time application? What other aspects of the system must be considered in order to guarantee meeting external deadlines?

4-2 Someone suggests performing deletions from a sorted singly linked list by finding the element to delete and using `CAS` to update the `next` pointer contained in its predecessor. Show why a solution based solely on this approach is incorrect.

4-3 A new processor supports a `DCAS` operation that acts as an atomic compare-and-swap on two arbitrary memory addresses. Outline how `DCAS` can be used to perform deletions from a sorted singly liked list.

4-4* Why would it be difficult to provide a wait free software transactional memory (STM)?