So, a computing scientist entered a store....

They want $2,700 for the server and $100 for the iPod.

I will get both and pay only $2,240 altogether!

Ma'am you cannot take the arithmetic average of percentages!

But... I just came from at top CS conference in San Jose where they do it!

But the average of 10% and 50% is 30% and 70% of $3,200 is $2,240.

Scientific Benchmarking: The fallacies of summarizing (Rules 3+4)

<table>
<thead>
<tr>
<th></th>
<th>System A</th>
<th>System B</th>
</tr>
</thead>
<tbody>
<tr>
<td>torture</td>
<td>I</td>
<td>II</td>
</tr>
<tr>
<td>floating-point operations (GFlop)</td>
<td>10,000</td>
<td>15,000</td>
</tr>
</tbody>
</table>
**Review of last lecture**

- Directory-based cache coherence
  - Simple working with presence/dirty bits
  - Case study with Xeon Phi: Illustrates performance impact of the protocol and its importance!

- Memory models
  - Ordering between accesses to different variables
  - Sequential consistency – nice but unrealistic
  - Demonstrate how it prevents compiler and architectural optimizations

- Practical memory models
  - Overview of various models (TSO, PSO, RMO, … existing CPUs)
  - Case study of x86 (continuing today)

**Goals of this lecture**

- Recap: Correctness in parallel programs
  - Covered in PP: here a slimmed down version to make the DPHPC lecture self-contained
  - Watch for the green bar on the right side

- Languages and Memory Models
  - Java/C++ definition

- Recap sequential consistency from the programmer’s perspective
  - Reads (now in practice)
  - Synchronization variables (now in practice)

- Mutual exclusion
  - Recap – simple lock properties
  - Proving correctness in SC and memory models (x86)
  - Locks in practice – performance overhead of memory models!

- Fast (actually practical) locks
  - CLH – queue locks
  - MTS – “cache coherence optimal” queue locking

**Administrivia**

- First project presentation: 10/29 (two weeks from now!)
- First presentation to gather feedback
  - You already know what your peers are doing, now let us know

- Some more ideas what to talk about:
  - What tools/programming language/parallelization scheme do you use?
  - Which architecture? (we only offer access to Xeon Phi, you may use different)
  - How to verify correctness of the parallelization?
  - How to argue about performance (bounds, what to compare to?)

- (Somewhat) realistic use-cases and input sets?
- What are the key concepts employed?
- What are the main obstacles?
Principle 1 and 2

Reads are not reordered with other reads. (R→R)
Writes are not reordered with other writes. (W→W)

If P3 observes ordering P1:xchg, then P4 will also observe the same ordering.

No allowed: r1 = 0, r2 = 1

No allowed: r3 = 1, r4 = 0, r5 = 1, r6 = 0

If P3 observes P1's write before P2's write, then P4 will also observe the same ordering.

Principle 3

Reads are not reordered with other reads. (R→R)
Writes are not reordered with other writes. (W→W)

Order: from left to right

Principle 4

Reads may be reordered with older writes to different locations but not with older writes to the same location.

Principle 5

In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

Principle 6

In a multiprocessor system, writes to the same location have a total order (implied by cache coherence).

Principle 7

In a multiprocessor system, locked instructions have a total order. (enables synchronized programming)
Reads and writes are not reordered with locked instructions. (enables synchronized programming)

Notions of Correctness

- We discussed so far:
  - Read/write of the same location
    - Cache coherence (write serialization and atomicity)
  - Read/write of multiple locations
    - Memory models (visibility order of updates by cores)

- Now one level up: objects (variables/fields with invariants defined on them)
  - Invariants "tie" variables together
  - Sequential objects
  - Concurrent objects

Sequential Queue

- Each object has a type
- A type is defined by a class
  - Set of fields forms the state of an object
  - Set of methods (or free functions) to manipulate the state

- Remark
  - An interface is an abstract type that defines behavior
  - A class implementing an interface defines several types

Sequential Queue

- Insert elements at tail
- Remove elements from head
  - Initial: head = tail = 0
  - enq(x)
  - enq(y)
  - deq() [x]
  - ...

Running Example: FIFO Queue

- Insert elements at tail
- Remove elements from head
  - Initial: head = tail = 0
  - enq(x)
  - enq(y)
  - deq() [x]
  - ...

Sequential Queue

An Alternative View: x86-TSO

- Sewell et al.: "x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors", CACM May 2010
  "... real multiprocessors typically do not provide the sequentially consistent memory that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, in which different hardware threads may have only loosely consistent views of a shared memory. Second, the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for loose specifications), leading to widespread confusion. [...] We present a new x86-TSO programmer’s model that, to the best of our knowledge, suffers from none of these problems. It is mathematically precise (rigorously defined in HOL4) but can be presented as an intuitive abstract machine which should be widely accessible to working programmers. [...]"
Sequential Queue

class Queue {
public:
void enq(Item x) {
    if (tail + 1) % items.size() == head {
        throw FullException;
    }
    items[tail] = x;
    tail = (tail + 1) % items.size();
    Item item = items[head];
    head = (head + 1) % items.size();
    return item;
}
};

head tail
0 2
1 5 4
3
yx
capacity = 8

Sequential Execution

{P
enq(x) enq(y) deq()}

Design by Contract!

• Preconditions:
  Specify conditions that must hold before method executes
  Involves state and arguments passed
  Specify obligations a client must meet before calling a method

• Example: enq()
  Queue must not be full

class Queue {
public:
void enq(Item x) {
    assert((tail + 1) % items.size() != head);
    // ...
}
};

head tail
0 2
1
5 4
3
yx
capacity = 8

Postconditions:

• Specify conditions that must hold after method executed
  Involves old state and arguments passed

• Example: enq()
  Queue must contain element!

class Queue {
public:
void enq(Item x) {
    assert((tail + 1) % items.size() != head);
    // ...
}
};

head tail
0 2
1
5 4
3
yx
capacity = 8

Sequential specification

• If precondition
  Object is in a specified state

• Then postcondition
  The method returns a particular value or throws a particular exception
  Leaves the object in a specified state

• Invariants
  Specified conditions (e.g., object state) must hold anytime a client could invoke an objects method!

Advantages of sequential specification

• State between method calls is defined
  Enables reasoning about objects
  Interactions between methods captured by side effects on object state

• Enables reasoning about each method in isolation
  Contracts for each method
  Local state changes global state

• Adding new methods
  Only reason about state changes that the new method causes
  If invariants are kept: no need to check old methods

  Modularity!
Concurrent execution - State

- Concurrent threads invoke methods on possibly shared objects
  - At overlapping time intervals

<table>
<thead>
<tr>
<th>Property</th>
<th>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>Meaningful only between method executions</td>
<td>Overlapping method executions → object may never be &quot;between method executions.&quot;</td>
</tr>
</tbody>
</table>

Each method execution takes some non-zero amount of time.

Concurrent execution - Method addition

- Reasoning must now include all possible interweavings
  - Of changes caused by methods themselves

<table>
<thead>
<tr>
<th>Property</th>
<th>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Method</td>
<td>Without affecting other methods; inservatives on state before/after execution</td>
<td>Everything can potentially interact with everything else</td>
</tr>
</tbody>
</table>

Consider adding a method that returns the last item enqueued:

- If peek() and enq() run concurrently: what if tail has not yet been incremented?
- If peek() and deq() run concurrently: what if last item is being dequeued?

Lock-based queue

- Class question: how is the lock ever unlocked?

Concurrent objects

- How do we describe one?
  - No pre/postconditions
- How do we implement one?
  - Plan for quadratic or exponential number of interactions and states
  - How do we tell if an object is correct?
    - Analyze all quadratic or exponential interactions and states

Is it time to panic for (parallel) software engineers? Who has a solution?

Lock-based queue

- One of C++'s ways of implementing a critical section

We can use the lock to protect Queue’s fields.
C++ Resource Acquisition is Initialization

- RAII – suboptimal name
- Can be used for locks (or any other resource acquisition)
  - Constructor grabs resource
  - Destructor frees resource
- Behaves as if
  - Implicit unlock at end of block!
- Main advantages
  - Always unlock/free lock at exit
  - No "lost" locks due to exceptions
  - or strange control flow (points 12)
  - Very easy to use

Example execution

```cpp
ttail items
```
Java and C++ High-level overview

- Relaxed memory model
  - No global visibility ordering of operations
  - Allows for standard compiler optimizations
- But
  - Program order for each thread (sequential semantics)
  - Partial order on memory operations (with respect to synchronizations)
  - Visibility function defined

- Correctly synchronized programs
  - Guarantee sequential consistency
- Incorrectly synchronized programs
  - Java: maintain safety and security guarantees (Type safety etc., require behavior bounded by causality)
  - C++: undefined behavior

Java and C++ High-level overview

Communication between threads: Intuition

- Not guaranteed unless by:
  - Synchronization
  - Volatile/atomic variables
  - Specialized functions/classes (e.g., java.util.concurrent, ...)

- Synchronization variables
  - Similar to synchronization variables

  • All memory accesses before an unlock ...
  • are ordered before and are visible to ...
  • any memory access after a matching lock!

- Variables can be declared volatile (Java) or atomic (C++)

  • Reads and writes to synchronization variables
    - Are totally ordered with respect to all threads
    - Must not be reordered with normal reads and writes
  
  • Compiler
    - Must not allocate synchronization variables in registers
    - Must not swap variables with synchronization variables
    - Must need to issue memory fences/barriers
Memory model semantics of synchronization variables

- Write to a synchronization variable
  - Similar memory semantics as unlock (no process synchronization)
- Read from a synchronization variable
  - Similar memory semantics as lock (no process synchronization)

Case Study: Implementing locks - lecture goals

- Among the simplest concurrency constructs
- Yet, complex enough to illustrate many optimization principles
- Goal 1: You understand locks in detail
  - Requirements / guarantees
  - Correctness / validation
  - Performance / scalability
- Why you do not want to use them in many cases!
- Goal 2: Acquire the ability to design your own locks (and other constructs)
  - Understand techniques and weaknesses/traps
  - Extend to other concurrent algorithms
  - Issues are very much the same
- Goal 3: Understand the complexity of shared memory!
  - Memory models in realistic settings

Recap Concurrent Updates

```
const int m=1000;
volatile int a=0;
for(int i=0; i<n; ++i) a++;

const int m=1000;
std::atomic<int> a;
a=0;
for(int i=0; i<n; ++i) a++;
```

Multi-threaded execution!

- Demo: value of a for p=1?
- Demo: value of a for p>1?

Preliminary Comments

- All code examples are in C/C++ style
  - Neither C (<11) nor C++ (<11) had a clear memory model
  - C++ is one of the languages of choice in HPC
- Consider source as exemplary (and pay attention to the memory model)
  - Usually given on x86 (easy to enforce)
- Number of threads/processes is p, tid is the thread id

Intuitive memory model rules

- Java/C++: Correctly synchronized programs will execute sequentially consistent
  - Correctly synchronized = data-race free
  - All sequentially consistent executions are free of data races
- Two accesses to a shared memory location form a data race in the execution of a program if
  - The two accesses are from different threads
  - At least one access is a write and
  - The accesses are not synchronized

One instruction less! Performance!

- run with larger n (10^8)
- Compiler: gcc version 4.9.2 (enabled c++11 support, -O3)

```
const int n = 100;
volatile int a=0;
for(int i=0; i<n; ++i) a++;
```

```
const int n = 100;
std::atomic<int> a;
a=0;
for(int i=0; i<n; ++i) a++;
```

```
const int n = 100;
for(int i=0; i<n; ++i) a++;
```

References: Bonce et al. "Evaluating the Cost of Atomic Operations on Modern Architectures", ACM PACT'15
Some Statistics

- Nondeterministic execution
  - Result depends on timing (probably not desired)
  - What do you think are the most significant results?
  - Running two threads on Core i5 dual core
- Some subsets of memory accesses (issued by the same process) need to happen at the same time

Conflicting Accesses

- (recap) two memory accesses conflict if they can happen at the same time (in happens-before) and one of them is a write (store)
- Such a code is said to have a “race condition”
  - Also data race
  - True around races:
    - The Iberia 25 killed three people due to a race
    - A date race lead to a large blackout in 2003, leaving 55 million people without power causing $1bn damage
  - Can be avoided by critical regions
    - Mutually exclusive access to a set of operations

Fixing it with locks

- What must the functions lock and unlock guarantee?
  - #1: preventing two threads from simultaneously entering CR
  - #2: ensure consistent memory
    - i.e., stores must be globally visible before new lock is granted
  - Any performance guesses (remember, 0.23s -> 0.78s for atomics)

More formal: Mutual Exclusion

- Control access to a critical region
  - Memory accesses of all processes happen in program order (a partial order, many interleavings)
  - An execution history defines a total order of memory accesses
  - Some subsets of memory accesses (issued by the same process) need to happen atomically (thread’s memory accesses may not be interleaved with other thread’s accesses)

To achieve linearizability?

- We need to restrict the valid executions

- Reqs. synchronisation of some sort
  - Many possible techniques (e.g., TM, CAS, T&S, …)

- We first discuss locks which have wait semantics

Lock Overview

- Lock/unlock or acquire/release
  - Lock/acquire: before entering CR
  - Unlock/release: after leaving CR

- Lock/unlock pairs have to match

- Between lock/unlock, a thread holds the lock
Desired Lock Properties
- Mutual exclusion
  - Only one thread is on the critical region
- Consistency
  - Memory operations are visible when critical region is left
- Progress
  - If any thread a is not in the critical region, it cannot prevent another thread b from entering
- Starvation-freedom (implies deadlock-freedom)
  - A thread that requested access to a critical region before thread b, did is also granted access to this region before b?
- Performance
  - Scaling to large numbers of contending threads

Peterson's Two-Thread Lock (1981)
- Combines the first lock (request access) with the second lock (grant access)

```c
volatile int flag[2];
volatile int victim;

void lock() {
  int j = -1; tid;
  flag[tid] = 1; // I'm interested
  victim = tid; // other goes first
  while (flag[j] && victim == tid) {} // wait
}

void unlock() {
  flag[tid] = 0; // I'm not interested
}
```

Proof Correctness
- Intuition:
  - Victim is written once
  - Pick thread that wrote victim last
  - Show thread must have read flag==0
  - Show that no sequentially consistent schedule permits that

Proof Starvation Freedom
- Intuition:
  - Threads can only wait/starve in while()
  - Unit(flag==0 or victim==other)
  - Other thread enter(s lock()) -> sets victim to other
  - Will definitely "unstuck" (int thread)
  - So other thread can only be stuck in lock()
  - Will wait for victim==other, victim cannot block both threads -> one must leave!
Case Study: Automatic Reasoning about Semantics

- Spin tells us quickly that it found a problem
- It's not always that easy
- More in the recitation session!

Back to Peterson in Practice ... on x86

- Implement and run our little counter on x86
- 100000 Iterations
- 1.6-10% errors
- What is the problem?

```c
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid; // I'm interested
    victim = tid; // other goes first
    asm("fence");
    while ((flag[j] && victim == tid)) { // wait
        j = 1 - j;
    }
    flag[tid] = 0; // I'm not interested
}

void unlock() {
    flag[tid] = 0; // I'm not interested
}
```

Peterson in Practice ... on x86

- Implement and run our little counter on x86
- Many Iterations
- 1.6-10% errors
- What is the problem?

```c
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1; // I'm interested
    victim = tid; // other goes first
    asm("fence");
    while ((flag[j] && victim == tid)) { // wait
        j = 1 - j;
    }
    flag[tid] = 0; // I'm not interested
}
```

Peterson in Practice ... on x86

- Implement and run our little counter on x86
- 100000 Iterations
- 1.6-10% errors
- What is the problem?

```c
volatile int flag[2];
volatile int victim;

void lock() {
    int j = 1 - tid;
    flag[tid] = 1; // I'm interested
    victim = tid; // other goes first
    asm("fence");
    while ((flag[j] && victim == tid)) { // wait
        j = 1 - j;
    }
    flag[tid] = 0; // I'm not interested
}
```

Case Study: Automatic Reasoning about Semantics

- Is the proposed algorithm correct?
- We may proof it manually
  Using tools from the last lecture
  • reason about the state space of \( H \)
  • Or use automated proofs (model checking)
  E.g., SPIN (Promela syntax)

```c
bool want[2];
bool turn;
byte cnt;
proctype P(bool i)
{
    want[i] = 1;
do
    :: (turn == false) && victim == tid {}; // wait
}
void unlock() {
    flag[tid] = 0; // I'm not interested
}
```

- More in the recitation session!

- Implement and run our little counter on x86
- 100000 iterations
- 1.6 \( \times \) 10\(^{-6}\)% errors
- ... (flag[j] && victim == tid) {}; // wait
}
void unlock() {
    flag[tid] = 0; // I'm not interested
}
Peterson in Practice … on x86

- Implement and run our little counter on x86
- Many iterations
  - 1.6 \cdot 10^{-6}\% errors

![Image of Peterson's solution](spcl.inf.ethz.ch)

```c
volatile int flag[];
volatile int victim;

void lock() {
    int j = tid;
    flag[j] = 1; // I'm interested
    victim = tid; // other goes first
    if (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    flag[tid] = 0; // I'm not interested
}
```

The compiler may inline this function 

Correct Peterson Lock on x86

- Unoptimized (naïve sprinkling of mfences)
- Performance:
  - No mfence
    - 375ns
  - mfence
    - 427ns (+14%)

![Image of Correct Peterson's solution](spcl.inf.ethz.ch)

```c
volatile int flag[];
volatile int victim;

void lock() {
    int j = tid;
    flag[j] = 1; // I'm interested
    victim = tid; // other goes first
    if (flag[j] && victim == tid) {}; // wait
}

void unlock() {
    mfence();
    flag[tid] = 0; // I'm not interested
}
```

Relative Power of Synchronization

- Design-Problem I: Multi-core Processor
  - Which atomic operations are useful?
- Design-Problem II: Complex Application
  - What atomic should I use?
  - Concept of "consensus number" $C$: If a primitive can be used to solve the "consensus problem" in a finite number of steps (even if threads stop)
    - atomic registers have $C = 1$ (thus locks have $C = 1$)
    - CAS, LL/SC, TM have $C = 2$
    - TASval, Swap, Fetch&Op have $C = \infty$

Hardware Support?

- Hardware atomic operations:
  - Test&Set
    - Write const to memory while returning the old value
  - Atomic swap
    - Atomically exchange memory and register
  - Fetch&Op
    - Get value and apply operation to memory location
  - Compare&Swap
    - Compare two values and swap memory with register if equal
  - Load-linked/Store-Conditional LL/SC
    - Loads value from memory, allows operations, commits only if no other updates committed \rightarrow \text{no-TM}
  - Intel TSX (transactional synchronization extensions)
  - Hardware-TM (roll your own atomic operations)

Test-and-Set Locks

- Test-and-Set semantics
  - Memorize old value
  - Set fixed value TASval (true)
  - Return old value
- After execution:
  - Pre-condition is a fixed (constant) value!
Cacheline contention (or: why I told you about MESI and friends)

- On x86, the XCHG instruction is used to implement TAS
- x86 lock is implicit in xchg!
- Cacheline is read and written
- Ends up in exclusive state, invalidates other copies
- Cacheline is "thrown" around uselessly
- High load on memory subsystem
- xchg lock is essentially a full memory barrier

Exponential backoff eliminates contention statistically

Do TATAS locks still have contention?

- When lock is released, k threads fight for
- All threads at the same time, no cache coherency/memory traffic

Danger!

- Efficient but use with great care!
- Generalizations are very dangerous

Test-and-Test-and-Set (TATAS) Locks

- Spinning in TAS is not a good idea
- Spin on cache line in shared state

Problem: Memory ordering leads to race-conditions!

Warning: Even Experts get it wrong!

- Example: Double-Checked Locking

TAS Lock with Exponential Backoff

- Exponential backoff eliminates contention statistically
- Locks granted in unpredictable order
- Starvation possible but unlikely
- How can we make it even less likely?

Volatile int lck = 0;
void lock() {
do {
while (TestAndSet(&lck) == 1);
} while (TestAndSet(&lck) == 1);
}
void unlock() {
lck = 0;
}

TAS Lock with Exponential Backoff

- Exponential backoff eliminates contention statistically
- Locks granted in unpredictable order
- Starvation possible but unlikely
- Maximum waiting time makes it less likely

Volatile int lck = 0;
const int maxtime=1000;
void lock() {
do {
while (TestAndSet(&lck) == 1) {
wait(time);
if (time > maxtime) {
orlck = 0;
}
}
void unlock() {
lck = 0;
}
Comparison of TAS Locks

<table>
<thead>
<tr>
<th>Lock</th>
<th>Backoff</th>
<th>Peterson</th>
<th>Peterson_1F</th>
<th>Peterson_2F</th>
<th>TAS</th>
<th>TQS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time per increment [ns]</td>
<td>20</td>
<td>150</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td># of threads</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>20</td>
<td>50</td>
<td>100</td>
</tr>
</tbody>
</table>

Array Queue Lock

- Array to implement queue
- Tail pointer shows next free queue position
- Each thread spins on own location
- C2 padding
- Index array can be put in TLS

Queue locks

- Each thread spins at a different location
- Spinning position is fixed!
- Releasing lock spins
- Spin location changes
- Can be hidden!

Array to implement queue

- List-based (same queue principle)
- Discovered twice by Craig, Landin, Hagersten 1993/94
- 2N+M words
- N threads, M locks
- Requires thread-local qnode pointer
- Can be hidden!

CLH Lock (1993)

- Make queue explicit
- Acquire lock by appending to queue
- Spin on own node until locked is reset
- Similar advantages as CLH but
- Only 2N + M words
- Spinning position is fixed!
- Benefits cache less NUMA

What are the issues?

- Relieving lock spins
- More atomics!

MCS Lock (1991)

- Make queue explicit
- Acquire lock by appending to queue
- Spin on own node until locked is reset
- Similar advantages as CLH but
- Only 2N + M words
- Spinning position is fixed!
- Benefits cache less NUMA

What are the issues?

- Relieving lock spins
- More atomics!

Improvements?

- Are TAS locks perfect?
- What are the two biggest issues?
- Cache coherence traffic (contention on same location with expensive atomics)
- Critical section underutilization (waiting for backoff times will delay entry to CPU)
- What would be a fix for that?
- How is this solved at airports and shops (often at least)?
- Queue locks – Threads enqueue
- Learn from predecessor if it's their turn
- Each threads spins at a different location
- FIFO fairness

Qnode objects represent thread state!

- Succ_blocked == 0 if released
- Can be hidden!

What would be a fix for that?

- Array to implement queue
- Queue locks
- Each thread spins on own
- Similar advantages
- New
- Spinning position is fixed!
- Releasing lock spins
- Spin location changes
- Can be hidden!

Can we do better?

- How is this solved at airports and shops (often at least)?
- One node per thread
- Spin location changes
- NUMA issues (cacheless)
- Can we do new?
Lessons Learned!

- Key Lesson:
  - Reducing memory (coherency) traffic is most important!
  - Not always straightforward (need to reason about CL states)
- MCS: 2006 Dijkstra Prize in distributed computing
  - "an outstanding paper on the principles of distributed computing, whose significance and impact on the theory and/or practice of distributed computing has been evident for at least a decade"
  - "probably the most influential practical mutual exclusion algorithm ever"
  - "vastly superior to all previous mutual exclusion algorithms"
  - Fast, fair, scalable → widely used, always compared against!

Time to Declare Victory?

- Down to memory complexity of 2N+M
  - Probably close to optimal
  - Only local spinning
  - Several variants with low expected contention
  - But: we assumed sequential consistency
  - Reality causes trouble sometimes
  - Sprinkling memory fences may harm performance
  - Open research on minimally-synching algorithms!
  - Come and talk to me if you’re interested

Fighting CPU waste: Condition Variables

- Allow threads to yield CPU and leave the OS run queue
  - Other threads can get them back on the queue!
  - `cond_wait(cond, lock) → yield and go to sleep`
  - `cond_signal(cond) → wake up sleeping threads`
- Wait and signal are OS calls
  - Often expensive, which one is more expensive?
  - Wait, because it has to perform a full context switch

When to Spin and When to Block?

- Optimal time depends on the future
  - When will the active thread leave the CR?
  - Can compute optimal offline schedule
  - What is the optimal offline schedule (assuming we know the future, i.e., when the lock will become available)?
  - Actual problem is an online problem
- Competitive algorithms
  - An algorithm is c-competitive if for a sequence of actions x and a constant c holds:
    \[ T(x) \leq c \cdot T^\ast(x) + a \]
  - What would a good spinning algorithm look like and what is the competitiveness?

Competitive Spinning

- If T is the overhead to process a wait, then a locking algorithm that spins for time T before it blocks is 2-competitive!
- If randomized algorithms are used, then \( e/(e-1) \)-competitiveness (~1.58) can be achieved
  - See paper above!
A lock of objects of class C and Threads write to different registers (order doesn’t matter).

Proof technique borrowed from: Each level of the hierarchy has a “consensus number” assigned.

Binary consensus with two threads (A, B)!

Final state = state after all threads finished

Synchronization instructions are not equally powerful!

Assume arbitrary consensus protocol, thread A, B

Protocol state = state of threads + state of shared objects

A wait method call finishes in a finite number of steps (implies lock-

Theorem [Herlihy’91]: Atomic registers have consensus number one

Each thread calls it at most once, the function returns a value that meets two conditions:

number: the value is some thread’s output

Simplification: binary consensus (inputs in {0, 1})

Remember: lock-free vs. wait-free

• A lock-free method guarantees that infinitely often some method call finishes in a finite number of steps

• A wait-free method guarantees that each method call finishes in a finite number of steps (implies lock-free)

• Synchronization instructions are not equally powerful!

• Indeed, they form an infinite hierarchy; no instruction (primitive) in level x can be used for lock-/wait-free implementations of primitives in level 2x.

Understanding Consensus

Can a particular class solve n-thread consensus wait-free?

A class C solves n-thread consensus if there exists a consensus protocol using any number of objects of class C and any number of atomic registers.

The protocol has to be wait-free (bounded number of steps per thread)

The consensus number of a class C is the largest n for which that class solves n-thread consensus (may be infinite)

Assume we have a class C whose objects can be constructed from objects out of class C. If class C has consensus number n, what does class D have?

Concept: Consensus Number

• Each level of the hierarchy has a “consensus number” assigned.

• Is the maximum number of threads for which primitives in level x can solve the consensus problem

• The consensus problem:

– Has single function: decide(s)

– Each thread calls it at most once, the function returns a value that meets two conditions: consistency: all threads get the same value

– validity: the value is some thread’s output

– Simplification: binary consensus (inputs in {0, 1})

Atomic Registers

• Theorem [Herlihy’91]: Atomic registers have consensus number one

– I.e., they cannot be used to solve even two-thread consensus! Really?

– Proof outline:

– Assume arbitrary consensus protocol, thread A, B

– Run until it reaches critical state where next action determines outcome (show that it must have a critical state first)

– Show all options using atomic registers and show that they cannot be used to determine one outcome for all possible executions!

  1) Any thread reads (other thread runs solo until end)

  2) Threads write to different registers (order doesn’t matter)

  3) Threads write to same register (solo thread can start after each write)

Starting simple ...

• Binary consensus with two threads (A, B)!

– Each thread moves until it decides on a value

– May update shared objects

– Protocol state = state of threads + state of shared objects

– Initial state = state before any thread moved

– Final state = state after all threads finished

States form a tree, wait-free property guarantees a finite tree

Example with two threads and two moves each!

Atomic Registers

• Theorem [Herlihy’91]: Atomic registers have consensus number one

– Corollary: It is impossible to construct a wait-free implementation of any object with consensus number of >1 using atomic registers

– “perhaps one of the most striking impossibility results in Computer Science" (Herlihy, Shavit)

– We need hardware atomics or Transactional Memory!

– Proof technique borrowed from:

• Very influential paper, always worth a read!

– Nicely shows proof techniques that are central to parallel and distributed computing!
Other Atomic Operations

- Simple RMW operations (Test&Set, Fetch&Op, Swap, basically all functions where the op commutes or overwrites) have consensus number 2!
  - Similar proof technique (bivalence argument)
- CAS and TM have consensus number $\infty$
- Constructive proof!

CAS and Swap Consensus

```c
const int first = -1;
volatile int thread = -1;
int proposed[n];

int decide(v) {
    proposed[tid] = v;
    if (CAS(thread, first, tid))
        return v; // I won!
    else
        return proposed[thread]; // thread won
}
```

- CAS provides an infinite consensus number
- Machines providing CAS are asynchronous computation equivalents of the Turing Machine
- i.e., any concurrent object can be implemented in a wait-free manner (not necessarily fast!)

Now you know everything 😊

- Not really ... ;-)
- We’ll argue more about performance now!
- But you have all the tools for:
  - Efficient locks
  - Efficient lock-based algorithms
  - Efficient lock-free algorithms (or even wait-free)
  - Reasoning about parallelism!
- What now?
  - A different class of problems
    - Impact on wait-free/lock-free on actual performance is not well understood
  - Relevant to HPC, applies to shared and distributed memory
  - Group communications