Synchronization without Locks

Dan Alistarh

November 07, 2013
Last Time: Summing Problem

```c
int compute(const X& v);
int main() {
    const int n = 1000000;
    extern X myArray[n];
    // ...

    int result = 0;
    cilk_for (int i = 0; i < n; ++i) {
        result += compute(myArray[i]);
    }
    printf( "The result is: %.1f\n", result);
    return 0;
}
```
Last Time: Summing Problem

```c
int compute(const X& v);
int main() {
    const int n = 1000000;
    extern X myArray[n];
    // ...

    int result = 0;
    cilk_for (int i = 0; i < n; ++i) {
        result += compute(myArray[i]);
    }
    printf( "The result is: %f\n", result );
    return 0;
}
```

Load of result followed by a store of result
What is the problem?

**Correctness:** We would like a total order on the operations, so that each load is followed by its corresponding store.

**Performance:** We would like the solution to be *fast*.

Q. Can we get both?
A Solution: Use Mutex

```c
int result = 0;
mutex L;
cilk_for (int i = 0; i < n; ++i) {
  int temp = compute(myArray[i]);
  L.lock();
  result += temp;
  L.unlock();
}
```

Make the load and store “atomic” by using a mutual exclusion lock.

Q. What’s the problem?
A. Performance. This solution is blocking.
Mutual exclusion: no two threads can execute the critical section concurrently.

Starvation Freedom: every call to lock() will eventually return.

Q. Can mutual exclusion be implemented with only atomic loads and stores?
Peterson’s Algorithm
Peterson's 2-Thread Algorithm

```c
widget x; //protected variable
bool she_wants = false;
bool he_wants = false;
enum {hers, his} turn;

Alice
she_wants = true;
turn = his;
while(he_wants && turn==his);
frob(x); //critical section
she_wants = false;

Bob
he_wants = true;
turn = hers;
while(she_wants && turn==hers);
borf(x); //critical section
he_wants = false;
```
Peterson’s 2–Thread Algorithm

```c
widget x; //=protected variable
bool she_wants = false;
bool he_wants = false;
enum {hers, his} turn;
```

Intuition:
1. If both “participate,” then the who writes last to
   `turn` will spin
2. If only one “participates,” then it progresses
   since the other’s “wants” bit is `false`

Alice

```
she_wants = true;
turn = his;
while(he_wants && turn==his);
frob(x); //=critical section
she_wants = false;
```

Bob

```
he_wants = true;
turn = hers;
while(she_wants && turn==hers);
borf(x); //=critical section
he_wants = false;
```
Theorem: Peterson’s 2–thread algorithm achieves mutual exclusion on the critical section.

Assume by way of contradiction that both threads are in the critical section together.

We analyze the last time each of the threads went through the code before entering the critical section.

We will derive a contradiction.
Proof of Mutual-Exclusion

“Freeze time” when both Alice and Bob are in the critical section.

W.L.O.G., assume Bob is the last thread to write to turn.

(1) write_A(turn=his) \rightarrow write_B(turn=hers)

“Happened Before” Relation

© 2012 Charles E. Leiserson and I–Ting Angelina Lee
Before Bob’s write to turn:

\((0)\) \(\text{write}_A(\text{she\_wants}=\text{true}) \implies \text{write}_A(\text{turn}=\text{his})\)

\[
\begin{align*}
\text{she\_wants} &= \text{true}; \\
\text{turn} &= \text{his}; \\
\text{while}(\text{he\_wants} \&\& \text{turn}==\text{his}); \\
\text{frob}(x); &\text{ //critical section} \\
\text{she\_wants} &= \text{false};
\end{align*}
\]

From Alice’s Code
Proof of Mutual-Exclusion

After Bob’s write to turn

\[(2) \text{ write}_B(\text{turn}=\text{hers}) \Rightarrow \text{read}_B(\text{she\_wants}) \Rightarrow \text{read}_B(\text{turn})\]

he\_wants = true;
turn = hers;
while(she\_wants && turn==hers);
borf(x); //critical section
he\_wants = false;

From Bob’s Code
Proof of Mutual-Exclusion

(0) \(\text{write}_A(\text{she\_wants=\text{true}}) \Rightarrow \text{write}_A(\text{turn=his})\)

(1) \(\text{write}_A(\text{turn=his}) \Rightarrow \text{write}_B(\text{turn=hers})\)

(2) \(\text{write}_B(\text{turn=hers}) \Rightarrow \text{read}_B(\text{she\_wants}) \Rightarrow \text{read}_B(\text{turn})\)
Proof of Mutual-Exclusion

(0) write_\text{A} (\text{she\_wants} = \text{true}) \rightarrow

(1) write_\text{A} (\text{turn} = \text{his}) \rightarrow

(2) write_\text{B} (\text{turn} = \text{hers}) \rightarrow \text{read}_\text{B} (\text{she\_wants}) \rightarrow \text{read}_\text{B} (\text{turn})

Q: What must Bob read?
Proof of Mutual-Exclusion

(0) \(write_A (\text{she\_wants} = \text{true}) \rightarrow\)

(1) \(write_A (\text{turn} = \text{his}) \rightarrow\)

(2) \(write_B (\text{turn} = \text{hers}) \rightarrow read_B (\text{she\_wants}) \rightarrow read_B (\text{turn})\)

Q: What must Bob read?

Bob reads \(\text{she\_wants} = \text{true}\) and \(\text{turn} = \text{hers}\), so it could not have entered the critical section.

There’s a symmetric argument for the opposite case of (1). This completes the proof.

Q.E.D.
**Theorem:** Peterson’s 2–thread algorithm guarantees starvation freedom.

Try it out!
The Status

So we can implement things using only loads and stores.

Should we?
Memory Consistency
Observation: our proof relied on the fact that if a location is stored and later loaded, the *later load will return the latest stored value*.
Initially, \( a = b = 0 \).

Q. Is it possible that both \%eax and \%ebx are \( 0 \) after both processors have executed?

A. It depends on the memory model: how memory operations behave in the parallel computer system.
Sequential Consistency

“[T]he result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.”

— Leslie Lamport [1979]

- The sequence of instructions as defined by a processor’s program are *interleaved* with the corresponding sequences defined by the other processors’ programs to produce a global *linear order* of all instructions.
- A load instruction receives the value stored to that address by the most recent store instruction that precedes the load, according to the linear order.
- The hardware can do whatever it wants, but for the execution to be sequentially consistent, it must *appear* as if loads and stores obey some global linear order.
Example

Initially, \( a = b = 0 \).

Processor 0

1. \texttt{mov 1, a ;Store}
2. \texttt{mov b, %ebx ;Load}

Processor 1

3. \texttt{mov 1, b ;Store}
4. \texttt{mov a, %eax ;Load}

Sequential consistency implies that no execution ends with \( %\text{eax} = %\text{ebx} = 0 \).
Memory Models Today

- No modern-day processor implements sequential consistency.
- All implement some form of relaxed consistency.
- Hardware actively reorders instructions.
- Compilers may reorder instructions, too.

Q. Why?

A. Because most of the performance is derived from a single thread’s unsynchronized execution of code.
Instruction Reordering

**Processor 0:**

```
mov 1, a ;Store  
mov b, %ebx ;Load  
mov b, %ebx ;Load  
mov 1, a ;Store
```

**Program Order**

**Execution Order**

**Q.** Why might the hardware or compiler decide to reorder these instructions?

**A.** To obtain higher performance by covering load latency — instruction-level parallelism.
Instruction Reordering

Processor 0:

Program Order

1. mov 1, a ;Store
2. mov b, %ebx ;Load
3. mov b, %ebx ;Load
4. mov 1, a ;Store

Execution Order

Q. Assume no concurrency. When is it safe for the hardware or compiler to perform this reordering?

A. When a \( \neq \) b.
Hardware Reordering

- The processor can issue stores faster than the network can handle them ⇒ *store buffer*.
- Since a load may stall the processor until it is satisfied, *loads take priority*, bypassing the store buffer.
- If a load address matches an address in the store buffer, the store buffer returns the result.
- Thus, a load can *bypass* a store to a different address.
**House rules:**

1. **Loads** are *not* reordered with **loads**.
2. **Stores** are *not* reordered with **stores**.
3. **Stores** are *not* reordered with prior **loads**.
4. A **load** *may* be reordered with a prior **store** to a **different** location but *not* with a prior **store** to the **same** location.
5. **Loads** and **stores** are *not* reordered with **lock** instructions.
6. **Stores** to the same location respect a **global total order**.
7. **Lock** instructions respect a **global total order**.
8. Memory ordering preserves **transitive visibility** (“causality”).
X86: Memory Consistency

House rules:

1. **Loads** are *not* reordered with **loads**.
2. **Stores** are *not* reordered with **stores**.
3. **Stores** are *not* reordered with prior **loads**.
4. A load is reordered with a prior **store** to a different location but *not* with a prior **store** to the same location.
5. **Loads** and **stores** are *not* reordered with **lock** instructions.
6. **Stores** to the same location respect a **global total order**.
7. **Lock** instructions respect a **global total order**.
8. Memory ordering preserves **transitive visibility** (“causality”).

Thread’s Code

- Store1
- Store2
- Load1
- Load2
- Store3
- Store4
- Load3
- Load4
- Load5

Total Store Ordering (TSO)…weaker than sequential consistency
Impact of TSO Reordering

Peterson’s algorithm revisited

```c
she_wants = true;
turn = his;
while(he_wants && turn==his);
frob(x); //critical section
she_wants = false;
```

```
he_wants = true;
turn = hers;
while(she_wants && turn==hers);
borf(x); //critical section
he_wants = false;
```

- If the **loads** of `he_wants/she_wants` are reordered before the **stores** of `she_wants/he_wants` our proof no longer holds and both threads might enter their critical sections simultaneously!
Memory Fences

- A *memory fence* (or *memory barrier*) is a hardware action that enforces an **ordering** constraint between the instructions before and after the fence.

- A memory fence can be issued explicitly as an instruction (**x86: mfence**) or be performed **implicitly** by locking, exchanging, and other synchronizing instructions.

- The Intel compilers implement a memory fence via the built-in function `__mm_mfence()`.*

- The typical cost of a memory fence is comparable to that of an **L2-cache access**.

Peterson’s in the Real World

Memory fences can be used to restore sequential consistency.

```c
she_wants = true;
turn = his;
__mm_mfence();
while(he_wants && turn==his);
frob(x); //critical section
she_wants = false;

he_wants = true;
turn = hers;
__mm_mfence();
while(she_wants && turn==hers);
borf(x); //critical section
he_wants = false;
```

Note #1: must declare variables as volatile to prevent compiler from optimizing away memory references.

Note #2: need compiler fences around frob() and borf() to prevent compiler reordering.
Memory fences can be used to restore sequential consistency*.

* Sort of. Also need to make sure that the compiler doesn’t screw you over.
Loads/stores to implement locks?

**Thm [Burns–Lynch]:** any \( n \) thread deadlock-free mutex algorithm using loads and stores requires \( \Omega(n) \) space.

**Thm [Attiya et al., POPL11]:** Any \( n \) thread deadlock-free mutex algorithm needs to use either a fence or an atomic compare-and-swap operation.

So loads/stores are **not great** for mutex.

They do make sense for read-only operations though.
The Lock-Free Toolbox

- Loads/Stores
- Compare-and-Swap (CAS)

**Sequential Specification:**

```c
bool __sync_bool_compare_and_swap (T *x, T old, T new) {
    if (*x == old) { *x = new; return 1; }
    return 0;
}
```
CAS is provided by the `cmpxchg` instruction on x86. The gcc and Intel compilers implement `compare-and-swap` via the built-in function
`__sync_bool_compare_and_swap()` which operates on values of type `int`, `long`, `long long`, and their unsigned counterparts.*

**Sequential Specification:**

```c
bool
__sync_bool_compare_and_swap (T *x, T old, T new) {
    if (*x == old) { *x = new; return 1; }
    return 0;
}
```

Mutex Using CAS

```c
void lock() {
    do {} while (!__sync_bool_compare_and_swap(&lock_var, false, true));
}

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

An n thread deadlock-free mutex algorithm using CAS requires $O(1)$ space.
Lock–Free using CAS
Mutex for the Summing Problem

```c
int result = 0;
mutex L;
cilk_for (int i = 0; i < n; ++i)
  {
    int temp = compute(myArray[i]);
    L.lock();
    result += temp;
    L.unlock();
  }
```

Yet all we want is to atomically execute a Load of x followed by a store of x

Q. What happens if the operating system swaps out a loop iteration just after it acquires the mutex?
A. All other loop iterations must wait.
CAS for Summing

```c
int result = 0;
cilk_for (int i = 0; i < n; ++i) {
    int temp = compute(myArray[i]);
    int old, new;
    do {
        old = result;
        new = result + temp;
    } while ( !_sync_bool_compare_and_swap (&result, old, new) );
}
```

Q. What happens now if the operating system swaps out a loop iteration?
A. No other loop iterations need wait. The algorithm is *non-blocking*. 
Lock-Free Stack

```c
struct Node {
    Node* next;
    int data;
};

struct Stack {
    Node* head;
};
```

head:

![Diagram of a lock-free stack with elements 77 and 75]
```c
void push(Node* node) {
    do {
        node->next = head;
    } while (!__sync_bool_compare_and_swap(&head, node->next, node));
}
```
The compare-and-swap by thread pushing 33 fails!
The compare-and-swap on new pointer by thread pushing 33 succeeds!
Node* pop() {
    Node* current = head;
    while (current) {
        if (__sync_bool_compare_and_swap(&head, current, current->next)) break;
        current = head;
    }
    return current;
}
Lock-Free Data Structures

Efficient Lock-Free algorithms known for several classic sequential data structures (linked lists, queues, skip lists, hash tables) Still an active research topic.

In theory, a thread might starve: because of contention, its operation might not complete. In practice, this happens very rarely.

Real Issues:
1. Memory Management (MM) is hard
2. The ABA Problem
3. Still suffer from contention
Lock-Free Stack* [Treiber]

Contestation is a problem.

* Unoptimized, take with a grain of salt. Much better lock-free stack implementations exist.
Lock-Free Linked List* [Harris]

Memory Management is also a problem.

* Unoptimized, take with a grain of salt
The ABA Problem
1. Thread 1 begins to pop node that contains 15, but stalls after reading current->next.
ABA Example

1. Thread 1 begins to pop node that contains 15, but stalls after reading current->next.
2. Thread 2 pops node containing 15.
ABA Example

1. Thread 1 begins to pop the node containing 15, but stalls after reading current->next.
2. Thread 2 pops the node containing 15.
3. Thread 2 pops the node containing 94.
ABA Problem

1. Thread 1 begins to pop the node containing 15, but stalls after reading current->next.
2. Thread 2 pops the node containing 15.
3. Thread 2 pops the node containing 94.
4. Thread 2 reuses the node that contained 15.
1. Thread 1 begins to pop the node containing 15, but stalls after reading `current->next`.
2. Thread 2 pops the node containing 15.
3. Thread 2 pops the node containing 94.
4. Thread 2 reuses the node that contained 15.
5. Thread 1 resumes, and its CAS succeeds, removing 7, but putting garbage back on the list.
Solutions to ABA

Versioning
- Pack a version number with each pointer in the same atomically updatable word.
- Increment the version number every time the pointer is changed.
- Compare-and-swap both the pointer and the version number as a single atomic operation.

Issue: Version numbers may need to be very large.

Reclamation
- Prevent node reuse while pending requests exist
- For Example: prevent node 15 from being reused as node 7 while Thread 1 still executing
The Lock–Free Toolbox

- Loads/Stores
- Compare–and–Swap (CAS)