Synchronization without Locks

Charles E. Leiserson

November 2, 2014
SEQUENTIAL CONSISTENCY

© 2010–2014 by Charles E. Leiserson and I–Ting Angelina Lee
Memory Models

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

**Processor 0**
- `mov 1, a ; Store`
- `mov b, %ebx ; Load`

**Processor 1**
- `mov 1, b ; Store`
- `mov a, %eax ; Load`

**Q.** Is it possible that Processor 0’s `%eax` and Processor 1’s `%ebx` both contain the value 0 after the processors have both executed their code?

**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 LOAD’s and STORE’s obey some global linear order.
Who Builds a Skyscraper without Drawing Blueprints?

Leslie Lamport
Microsoft Research

Wednesday, November 12, 2014
4:00 P.M. to 5:30 P.M.

Refreshments: 3:45 P.M.
32–123 (Kirsch Auditorium)
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.
Sequential consistency implies that no execution ends with \( %eax = %ebx = 0 \).
Reasoning about Seq. Consistency

- An execution induces a “happens before” relation, which we shall denote as $\rightarrow$.
- The $\rightarrow$ relation is linear, meaning that for any two distinct instructions $x$ and $y$, either $x \rightarrow y$ or $y \rightarrow x$.
- The $\rightarrow$ relation respects processor order, the order of instructions in each processor.
- A LOAD from a location in memory reads the value written by the most recent STORE to that location according to $\rightarrow$.
- For the memory resulting from an execution to be sequentially consistent, there must exist such a linear order $\rightarrow$ which yields that memory state.
Mutual Exclusion without Locks
Mutual-Exclusion Problem

Recall
A *critical section* is a piece of code that accesses a shared data structure that must not be accessed by two or more threads at the same time (*mutual exclusion*).

Most implementations of mutual exclusion employ an *atomic read-modify-write* instruction or the equivalent, usually to implement a lock:
- e.g., `xchg`, test-and-set, compare-and-swap, load-linked-store-conditional.
Mutual-Exclusion Problem

Q. Can mutual exclusion be implemented with LOAD’s and STORE’s as the only memory operations?

A. Yes, Theodorus J. Dekker and Edsger Dijkstra showed that it can, as long as the computer system is sequentially consistent.
Peterson’s Algorithm

### Algorithm Code

```c
widget x; //protected variable
bool A_wants(false);
bool B_wants(false);
enum theirs {A, B} turn;

A_wants = true;
turn = B;
while (B_wants && turn==B);
frob(x); //critical section
A_wants = false;

B_wants = true;
turn = A;
while (A_wants && turn==A);
borf(x); //critical section
B_wants = false;
```
Peterson’s Algorithm

Intuition

- If Alice and Bob both try to enter the critical section, then whoever writes last to turn spins and the other progresses.
- If only Alice tries to enter the critical section, then she progresses, since B_wants is false.
- If only Bob tries to enter the critical section, then he progresses, since A_wants is false.

But we can do better!

Alice

A_wants = true;
turn = B;
while (B_wants && turn==B);
frob(x); //critical section
A_wants = false;

Bob

B_wants = true;
turn = A;
while (A_wants && turn==A);
borf(x); //critical section
B_wants = false;
Proof of Mutual Exclusion

**Theorem.** Peterson’s algorithm achieves mutual exclusion on the critical section.

**Proof.**

- Assume for the purpose of contradiction that both Alice and Bob find themselves in the critical section together.
- Consider *the most–recent time* that each of them executed the code before entering the critical section.
- We shall derive a contradiction.
Proof of Mutual Exclusion

Alice

A_wants = true;
turn = B;
while (B_wants && turn==B);
frob(x);  //critical section
A_wants = false;

Bob

B_wants = true;
turn = A;
while (A_wants && turn==A);
borf(x);  //critical section
B_wants = false;
Proof of Mutual Exclusion

Alice

A_wants = true;
turn = B;
while (B_wants && turn==B);
frob(x);  //critical section
A_wants = false;

Bob

B_wants = true;
turn = A;
while (A_wants && turn==A);
borf(x);  //critical section
B_wants = false;

• WLOG, assume that Bob was the last to write to turn:
  write_A(turn = B) \rightarrow write_B(turn = A) \, .
Proof of Mutual Exclusion

Alice

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

Bob

\[
\begin{align*}
B\_wants &= \text{true}; \\
turn &= A; \\
\text{while (A\_wants && turn==A);} \\
borf(x); &//\text{critical section} \\
B\_wants &= \text{false};
\end{align*}
\]

- WLOG, assume that Bob was the last to write to turn: write\(_A(\text{turn} = B) \rightarrow write\(_B(\text{turn} = A)\).

- Alice’s program order:
  write\(_A(\text{A\_wants = true}) \rightarrow write\(_A(\text{turn} = B)).
### Proof of Mutual Exclusion

**Alice**

- `A_wants = true;`
- `turn = B;`
- `while (B_wants && turn==B);`
- `frob(x); //critical section`
- `A_wants = false;`

- WLOG, assume that **Bob** was the last to write to `turn`:
  - `write_A(\text{turn} = B) \rightarrow write_B(\text{turn} = A)`.  
- **Alice**’s program order:
  - `write_A(\text{A_wants} = \text{true}) \rightarrow write_A(\text{turn} = B)`.  
- **Bob**’s program order:
  - `write_B(\text{turn} = A) \rightarrow read_B(\text{A_wants}) \rightarrow read_B(\text{turn})`. 

**Bob**

- `B_wants = true;`
- `turn = A;`
- `while (A_wants && turn==A);`
- `borf(x); //critical section`
- `B_wants = false;`
Proof of Mutual Exclusion

- WLOG, assume that Bob was the last to write to turn: 
  \[ \text{write}_A(\text{turn} = B) \Rightarrow \text{write}_B(\text{turn} = A). \]
- Alice’s program order:
  \[ \text{write}_A(A_{\text{wants}} = \text{true}) \Rightarrow \text{write}_A(\text{turn} = B). \]
- Bob’s program order:
  \[ \text{write}_B(\text{turn} = A) \Rightarrow \text{read}_B(A_{\text{wants}}) \Rightarrow \text{read}_B(\text{turn}). \]
## Proof of Mutual Exclusion

### Alice

```plaintext
A_wants = true;
turn = B;
while (B_wants && turn==B);
frob(x); //critical section
A_wants = false;
```

### Bob

```plaintext
B_wants = true;
turn = A;
while (A_wants && turn==A);
borf(x); //critical section
B_wants = false;
```

- **WLOG**, assume that **Bob** was the last to write to `turn`:
  
  \[ \text{write}_A(\text{turn} = B) \Rightarrow \text{write}_B(\text{turn} = A). \]

- **Alice**’s program order:
  
  \[ \text{write}_A(\text{A_wants} = true) \Rightarrow \text{write}_A(\text{turn} = B). \]

- **Bob**’s program order:
  
  \[ \text{write}_B(\text{turn} = A) \Rightarrow \text{read}_B(\text{A_wants}) \Rightarrow \text{read}_B(\text{turn}). \]

- What did **Bob** read?
  
  \[
  \begin{align*}
  \text{A_wants: } & \text{true} \\
  \text{turn: } & A
  \end{align*}
  \]

  \[ \text{Bob should spin. Contradiction. } \square \]
Starvation Freedom

**Theorem:** Peterson’s algorithm guarantees *starvation freedom*: While Alice wants to execute her critical section, Bob cannot execute his critical section twice in a row, and vice versa.

*Proof.* Exercise. ■
Relaxed Memory Consistency
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 might the hardware or compiler decide to reorder these instructions?

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

Program Order

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

Execution Order

mov b, %ebx ;Load
mov 1, a ;Store
Instruction Reordering

Q. When is it safe for the hardware or compiler to perform this reordering?

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

A’. And there’s no concurrency.

Program Order

Execution Order

\begin{verbatim}
mov 1, a   ;Store
mov b, %ebx ;Load
mov b, %ebx ;Load
mov 1, a   ;Store
\end{verbatim}
Hardware Reordering

- The processor can issue `STORE`'s faster than the network can handle them ⇒ store buffer.
- Since a `LOAD` can 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.
X86–64 Total Store Order

House rules:

1. LOAD’s are *not* reordered with LOAD’s.
2. STORE’s are *not* reordered with STORE’s.
3. STORE’s are *not* reordered with prior LOAD’s.
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. LOAD’s and STORE’s are *not* reordered with LOCK instructions.
6. STORE’s to the same location respect a *global total order*.
7. LOCK instructions respect a *global total order*.
8. Memory ordering preserves *transitive visibility* (“causality”).
X86–64 Total Store Order

House rules:

1. LOAD’s are *not* reordered with LOAD’s.
2. STORE’s are *not* reordered with STORE’s.
3. STORE’s are *not* reordered with prior LOAD’s.
4. A LOAD or STORE is not reordered with prior store(s) to a different location.
5. LOAD’s may be reordered with prior STORE’s to a different location but *not* with a prior STORE to the same location.
6. STORE’s to the same location respect a *global total order*.
7. LOCK instructions respect a *global total order*.
8. Memory ordering preserves *transitive visibility* ("causality").

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

Processor 0

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

Processor 1

3. mov 1, b ;Store
4. mov a, %eax ;Load
Impact of Reordering

The ordering \( \langle 2, 4, 1, 3 \rangle \) produces \( \%\text{eax} = \%\text{ebx} = 0 \).

Instruction reordering violates sequential consistency!
Further Impact of Reordering

Peterson’s algorithm revisited

Alice

A_wants = true;
turn = B;
while (B_wants && turn==B);
frob(x); //critical section
A_wants = false;

Bob

B_wants = true;
turn = A;
while (A_wants && turn==A);
borf(x); //critical section
B_wants = false;

- The LOAD’s of B_wants and A_wants can be reordered before the STORE’s of A_wants and B_wants, respectively.
- Both Alice and Bob 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.

Restoring Consistency

Memory fences can restore sequential consistency.

Well, sort of. You also need to make sure that the compiler doesn’t screw you over.

© 2010–2014 by Charles E. Leiserson and I–Ting Angelina Lee
**Restoring Consistency**

<table>
<thead>
<tr>
<th>Alice</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A_wants = true;</code>&lt;br&gt;<code>turn = B;</code>&lt;br&gt;<code>__mm_mfence();</code>&lt;br&gt;<code>while (B_wants &amp;&amp; turn==B);</code>&lt;br&gt;<code>asm volatile(&quot;&quot;:&quot;memory&quot;);</code>&lt;br&gt;<code>frob(x); //critical section</code>&lt;br&gt;<code>asm volatile(&quot;&quot;:&quot;memory&quot;);</code>&lt;br&gt;<code>A_wants = false;</code></td>
<td><code>B_wants = true;</code>&lt;br&gt;<code>turn = A;</code>&lt;br&gt;<code>__mm_mfence();</code>&lt;br&gt;<code>while (A_wants &amp;&amp; turn==A);</code>&lt;br&gt;<code>asm volatile(&quot;&quot;:&quot;memory&quot;);</code>&lt;br&gt;<code>borf(x); //critical section</code>&lt;br&gt;<code>asm volatile(&quot;&quot;:&quot;memory&quot;);</code>&lt;br&gt;<code>B_wants = false;</code></td>
</tr>
</tbody>
</table>

In addition to the hardware fence:
- you must declare variables as `volatile` to prevent the compiler from optimizing away memory references;
- you need *compiler fences* around `frob()` and `borf()` to prevent compiler reordering.
Implementing General Mutexes

**Theorem [Burns–Lynch].** Any $n$–thread deadlock–free mutual–exclusion algorithm using only \texttt{LOAD} and \texttt{STORE} memory operations requires $\Omega(n)$ space.

**Theorem [Attiya et al.]:** Any $n$–thread deadlock–free mutual–exclusion algorithm on a modern machine must use an expensive operation such as a memory fence or an atomic compare–and–swap operation.

Thus, hardware designers are justified when they implement special operations to support atomicity.
COMPARE-AND-SWAP
The Lock–Free Toolbox

Memory operations

- LOAD
- STORE
- CAS (compare-and-swap)
Compare-and-Swap

The *compare-and-swap* operation is provided by the `cmpxchg` instruction on x86-64. The GCC compiler implements *CAS* via the built-in function

\[
\text{\_sync\_bool\_compare\_and\_swap()}
\]

which operates on values of type `int`, `long`, `long long`, and their unsigned counterparts.*

**Specification**

```c
bool CAS (T *x, T old, T new) {
    if (*x == old) {
        *x = new;
        return true;
    }
    return false;
}
```

- Executes atomically.
- Implicit fence.

Mutex Using CAS

Theorem. An n-thread deadlock-free mutual-exclusion algorithm using CAS can be implemented using $\Theta(1)$ space.

Proof.

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

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

Just the space for the mutex itself. ■
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;
}
```
int compute(const X& v);
int main() {
    const int n = 1000000;
    extern X myArray[n];
    mutex L;
    // ...

    int result = 0;
    cilk_for (int i = 0; i < n; ++i) {
        int temp = compute(myArray[i]);
        L.lock();
        result += temp;
        L.unlock();
    }
    printf( "The result is: %f\n", result );
    return 0;
}
int compute(const X& v);

int main()
{
  const int n = 1000000;
  X myArray[n];

  mutex L;
  // ...

  int result = 0;
  cilk_for (int i = 0; i < n; ++i) {
    int temp = compute(myArray[i]);
    L.lock();
    result += temp;
    L.unlock();
  }
  printf( "The result is: %f\n", result );
  return 0;
}
CAS Solution

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

Q. Now what happens if the operating system swaps out a loop iteration?

A. No other loop iteration needs to wait. The algorithm is nonblocking.
LOCK–FREE ALGORITHMS
Lock-Free Stack

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

struct Stack {
    Node* head;
    ...
}

head: [Diagram showing a linked list 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 fails!

void push(Node* node) {
  do {
    node->next = head;
  } while (!__sync_bool_compare_and_swap(&head, node->next, node));
}
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 are known for a variety of classical data structures (e.g., linked lists, queues, skip lists, hash tables).
- In theory, a thread might starve. Because of contention, its operation might never complete. In practice, starvation rarely happens.
- *Transactional memory* is revolutionizing this area.

**Practical Issues**
- Memory management.
- Contention.
- The ABA problem.
THE ABA PROBLEM
1. Thread 1 begins to pop the node containing 15, but stalls after reading current->next.
1. Thread 1 begins to pop the node containing 15, but stalls after reading current->next.
2. Thread 2 pops the node containing 15.
ABA Example

1. Thread 1 begins to pop the node containing 15, but stalls after reading \texttt{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.
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.
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.