Synchronization without Locks

I–Ting Angelina Lee

November 01, 2012
Recall: 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]);
    }
    std::cout << "The result is: " << result << std::endl;
    return 0;
}
```
Race!
Recall: 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]);
    }
    std::cout << "The result is: " << result << std::endl;
    return 0;
}
```

Load of result followed by a store of result
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.
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

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);  //critical section
frob(x);
she_wants = false;

Bob

he_wants = true;
turn = hers;
while(she_wants && turn==hers);
borf(x);  //critical section
he_wants = false;
Proof of Mutual-Exclusion

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

Let's derive a contradiction.
(1) write$_A$(turn=his) \rightarrow write$_B$(turn=her)

W.L.O.G. assume Bob is the last thread to write turn.
(2) \( \text{write}_A(\text{she}_{-}\text{wants}=\text{true}) \rightarrow \text{write}_A(\text{turn}=\text{his}) \)

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

From Alice’s Code
Proof of Mutual-Exclusion

(3) write$_B$(turn=hers) $\rightarrow$ read$_B$(she_wants) $\rightarrow$ read$_B$(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

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

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

(3) $\text{write}_B(\text{turn} = \text{hers}) \Rightarrow \text{read}_B(\text{she\_wants})$ \Rightarrow \text{read}_B(\text{turn})$

Combine these observations.
Proof of Mutual-Exclusion

(2) \( \text{write}_A(\text{she}_\text{wants}=\text{true}) \rightarrow \)

(1) \( \text{write}_A(\text{turn}=\text{his}) \rightarrow \)

(3) \( \text{write}_B(\text{turn}=\text{hers}) \rightarrow \text{read}_B(\text{she}_\text{wants}) \rightarrow \text{read}_B(\text{turn}) \)

Combine these observations.
Proof of Mutual-Exclusion

(2) write$_A$(she_wants = true) $\rightarrow$

(1) write$_A$(turn = his) $\rightarrow$

(3) write$_B$(turn = hers) $\rightarrow$ read$_B$(she_wants) $\rightarrow$ read$_B$(turn)

Bob reads she_wants == true and turn == hers, so it could not have entered the critical section. The symmetric argument for the opposite case of (1) completes the proof.
Proof of Starvation Freedom

**Theorem:** Peterson’s 2-thread algorithm guarantees *starvation freedom*.

WLOG, assume Alice is blocked. This is possible only if he\_wants==true and turn==his.

Two cases:

- Bob has not entered critical section \(\rightarrow\) he sets turn to hers.
- Bob is in critical section \(\rightarrow\) once he exists, he sets he\_wants to false;

```java
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;
```
Memory Consistency
Memory Models

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.

(1) \text{write}_A(\text{turn}=\text{his}) \rightarrow

(2) \text{write}_A(\text{she}_\text{wants}=\text{true}) \rightarrow

(3) \text{write}_B(\text{turn}=\text{hers}) \rightarrow \text{read}_B(\text{she}_\text{wants}): \text{true} \rightarrow \text{read}_B(\text{turn})
Initially, $a = b = 0$.

Processor 0

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

Processor 1

```plaintext
mov 1, b  ;Store
mov a, %eax ;Load
```

Q. What are the final possible values of \%eax and \%ebx 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. `mov 1, a ;Store`
2. `mov b, %ebx ;Load`

Processor 1

3. `mov 1, b ;Store`
4. `mov a, %eax ;Load`

<table>
<thead>
<tr>
<th>Interleavings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 3 3 3</td>
</tr>
<tr>
<td>2 3 3 1 1 4</td>
</tr>
<tr>
<td>3 2 4 2 4 1</td>
</tr>
<tr>
<td>4 4 2 4 2 2</td>
</tr>
<tr>
<td>%eax 1 1 1 1 1 0</td>
</tr>
<tr>
<td>%ebx 0 1 1 1 1 1</td>
</tr>
</tbody>
</table>

Sequential consistency implies that no execution ends with $%eax = %ebx = 0$. 
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 performance is derived from a single thread’s unsynchronized execution of code.
Instruction Reordering

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

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

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

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.
• The processor can issue stores faster than the network can handle them ⇒ \textit{store buffer}.
• Since a load may stall the processor until it is satisfied, \textit{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 \textit{bypass} a store to a \textit{different} address.
X86: Memory Consistency

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").
Thread’s Code

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. **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").

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

Peterson’s algorithm revisited

- 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!

```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;
```
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

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*. 

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

* Sort of. Also need to make sure that the compiler doesn’t screw you over.
Compare-and-Swap

Compare-and-swap 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.*

**Implementation logic**

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

Compare-and-Swap

Compare-and-swap 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` on values of type `int`, long, and their unsigned counterparts.*

Implementation logic

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

Atomically execute a Load of $x$ followed by a store of $x$

Includes a memory fence

An n thread deadlock-free mutex algorithm using CAS requires $O(1)$ space.
Lock-Free Protocols
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]);
  do {
    int old = result;
    int 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.
Lock-Free Stack

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

class Stack {
    private:
        Node* head;
    ...
}
```

head: [Diagram showing a linked list with nodes 77 and 75]
public:
    void push(Node* node) {
        do {
            node->next = head;
        } while (!__sync_bool_compare_and_swap (&head, node->next, node));
    }

head:

node:
public:
    void push(Node* node) {
        do {
            node->next = head;
        } while (!__sync_bool_compare_and_swap(&head, node->next, node));
    }

head:  [77] → 75
node:  81
The compare-and-swap by thread pushing 81 succeeds.
public:
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!
public:
    void push(Node* node) {
        do {
            node->next = head;
        } while (!__sync_bool_compare_and_swap(&head, node->next, node));
    }

head:

The compare-and-swap on new pointer by thread pushing 33 succeeds!
public:
void push(Node* node) {
  do {
    node->next = head;
  } while (!__sync_bool_compare_and_swap(&head, node->next, node));
}

head:

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 Pop

Node* pop() {
    Node* current = head;
    while (current) {
        if (__sync_bool_compare_and_swap (&head, current, current->next)) break;
        current = head;
    }
    return current;
}

current: [ ]
The ABA Problem
1. Thread 1 begins to pop node that contains 15, but stalls after reading current->next.
ABA Problem

1. Thread 1 begins to pop node that contains 15, but stalls after reading current->next.
2. Thread 2 pops node containing 15.
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.
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 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.
5. Thread 1 resumes, and the compare-and-swap completes, removing 7, but putting the garbage 94 back on the list.
Solution 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.

As an alternative to compare-and-swap, some architectures feature a *load-linked, store conditional* instruction.