6.172
Performance
Engineering
of Software
Systems

LECTURE 17

SYNCHRONIZATION WITHOUT LOCKS

Charles E. Leiserson

Sequential Consistency

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 %ebx and Processor 1’s %eax 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.

Example

Initially, `a = b = 0`.

Processor 0

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

Processor 1

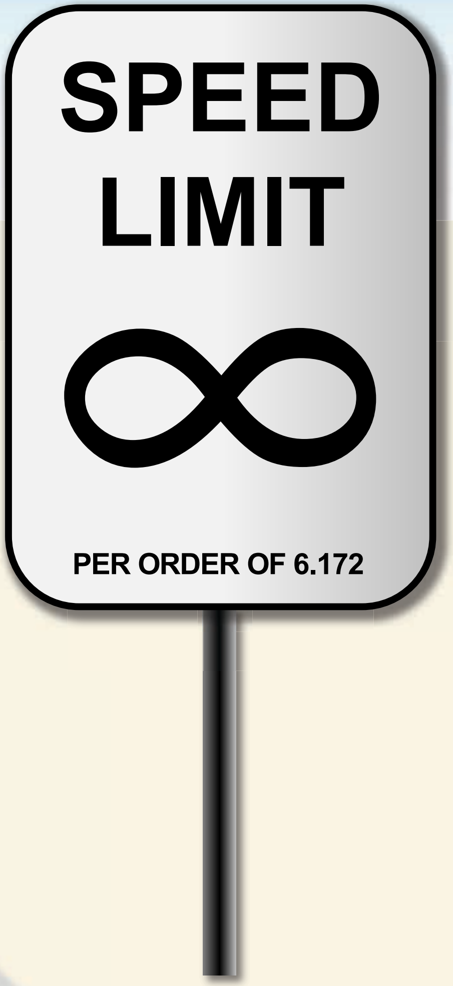
```
3 mov 1, b      ;Store
4 mov a, %eax   ;Load
```

	Interleavings					
	1	1	1	3	3	3
	2	3	3	1	1	4
	3	2	4	2	4	1
	4	4	2	4	2	2
%eax	1	1	1	1	1	0
%ebx	0	1	1	1	1	1

Sequential consistency implies that no execution ends with `%eax = %ebx = 0`.

Reasoning about Sequential Consistency

- An execution induces a "*happens before*" relation, which we shall denote as →.
- The → relation is *linear*, meaning that for any two distinct instructions x and y, either x → y or y → x.
- The → 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 →.
- For the memory resulting from an execution to be sequentially consistent, there must exist such a linear order → that 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:

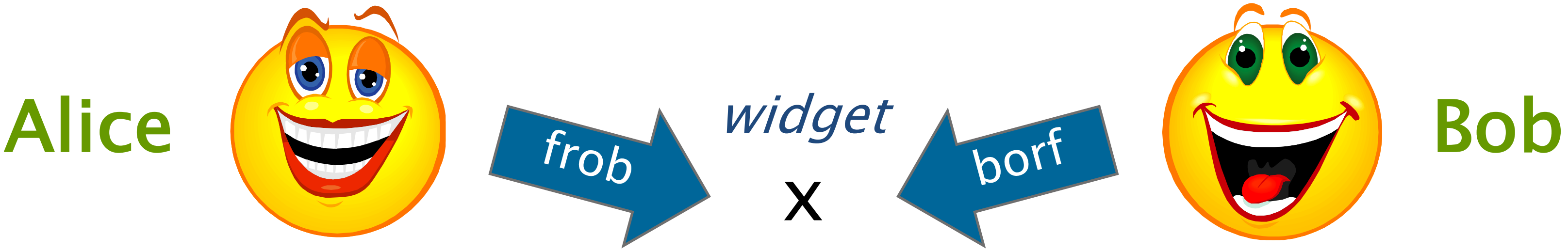
- e.g., xchg, test-and-set, compare-and-swap, load-linked-store-conditional.

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 Edsgar Dijkstra showed that it can, as long as the computer system is sequentially consistent.

Peterson's Algorithm



```
widget x; //protected variable
bool A_wants = false;
bool B_wants = false;
enum {A, B} turn;
```

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;
```

Peterson’s Algorithm

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;
```

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!

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`) → write$_B$(`turn = A`) .

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:

$$\text{write}_A(\texttt{turn = B}) \rightarrow \text{write}_B(\texttt{turn = A}) .$$

- Alice's program order:

$$\text{write}_A(\texttt{A\_wants = true}) \rightarrow \text{write}_A(\texttt{turn = B}) .$$

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) → write$_B$(turn = A) .

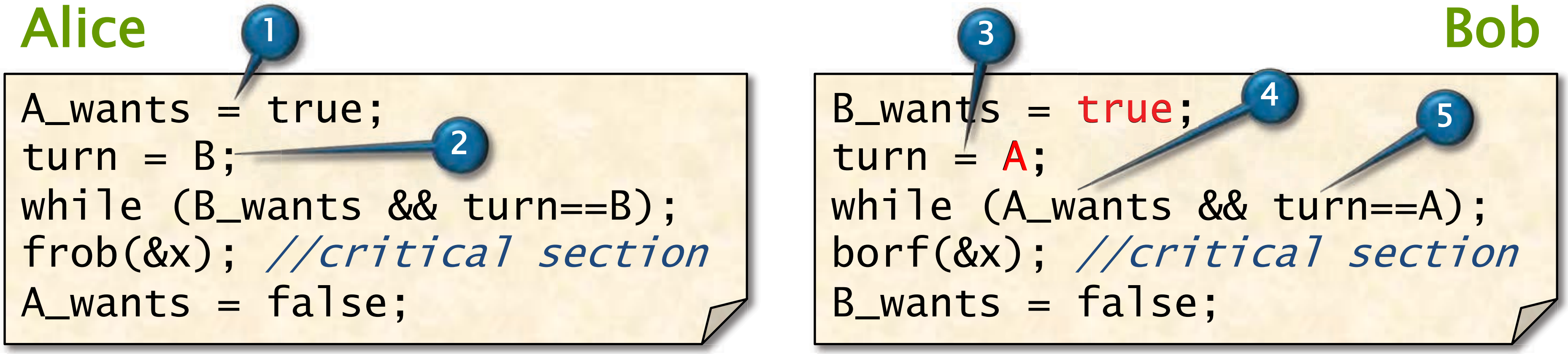
- Alice's program order:

  write$_A$(A_wants = true) → write$_A$(turn = B) .

- Bob's program order:

  write$_B$(turn = A) → read$_B$(A_wants) → read$_B$(turn) .

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) → write_B(turn = A) .

- Alice's program order:

  write_A(A_wants = true) → write_A(turn = B) .

- Bob's program order:

  write_B(turn = A) → read_B(A_wants) → read_B(turn) .

- What did Bob read?

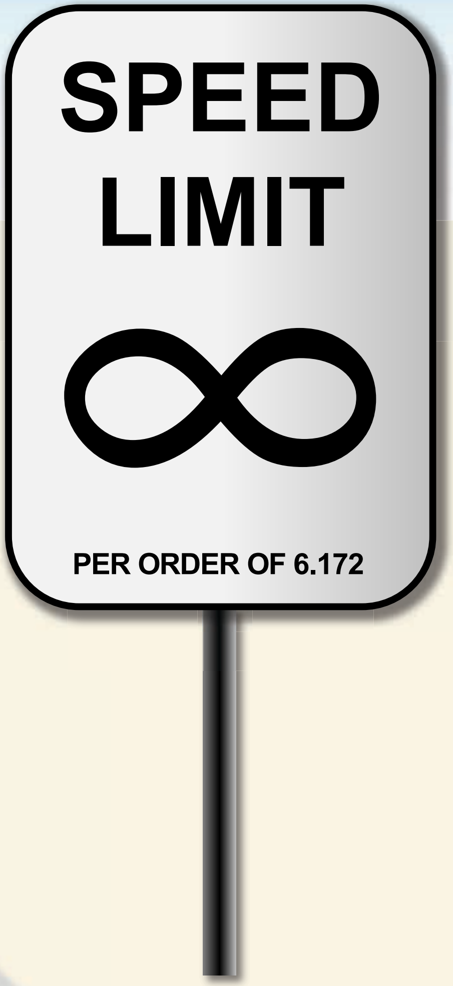
  A_wants: true
  turn: A

  } Bob should spin. Contradiction. ■

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. ■

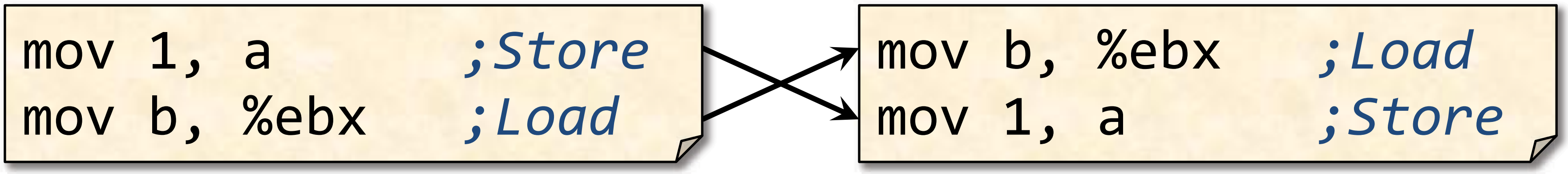


Relaxed Memory Consistency

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.

Instruction Reordering



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

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

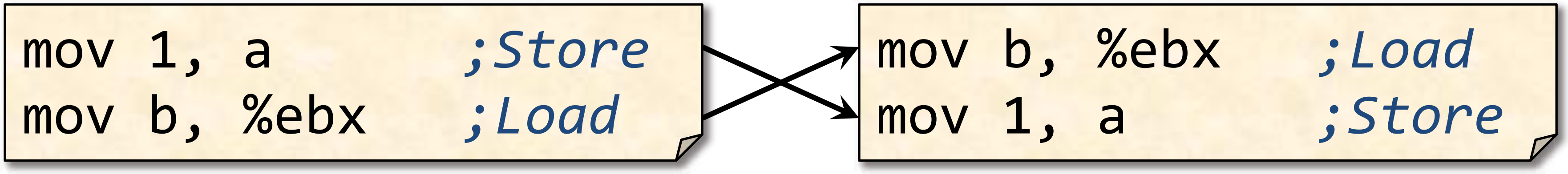
Program Order

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



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

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

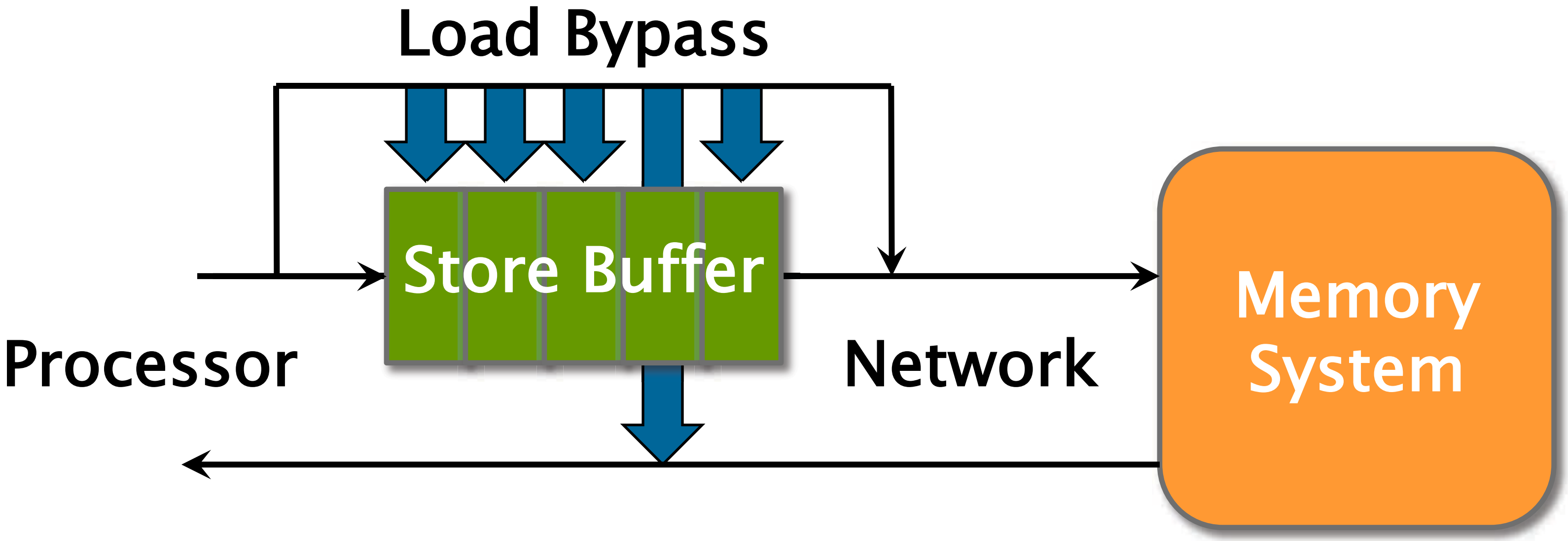
Program Order **Execution Order**

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

A. When a ≠ b.

A′. And there’s no concurrency.

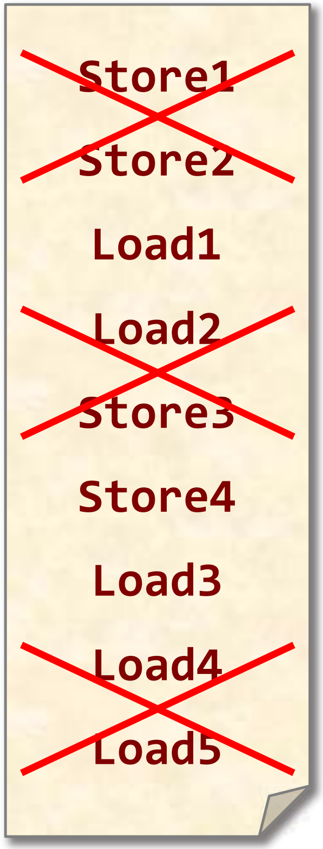
Hardware Reordering



- 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

Instruction Trace



~~Store1~~

~~Store2~~

Load1

~~Load2~~

~~Store3~~

Store4

Load3

~~Load4~~

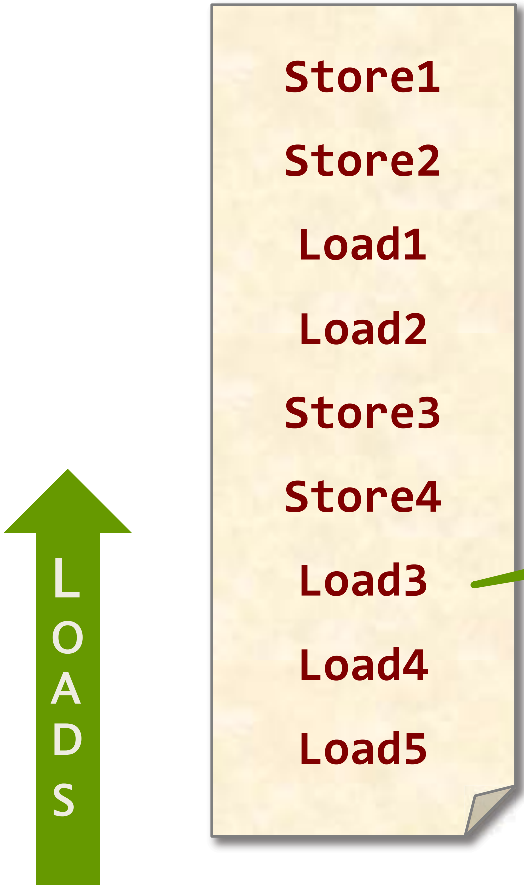
~~Load5~~

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

Instruction Trace



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 rior S ot w ation.
5. L ed wit uctions.
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

Processor 0

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

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

Processor 1

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

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

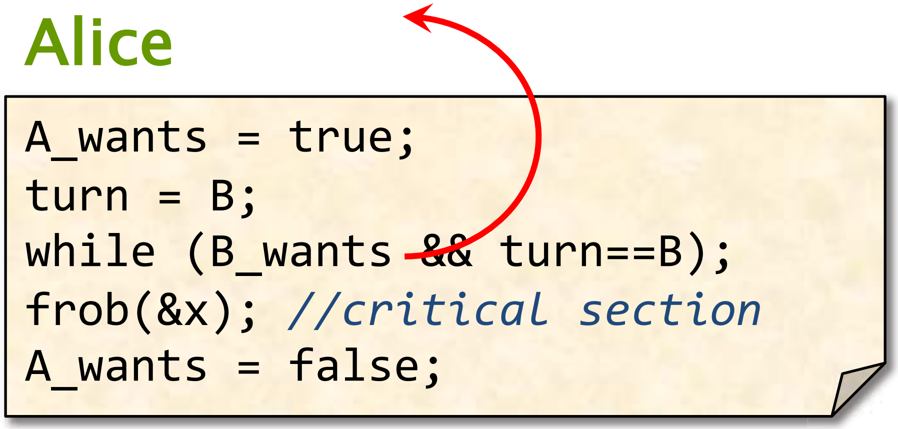
The ordering ⟨2, 4, 1, 3⟩ produces `%eax = %ebx = 0`.

Instruction reordering violates sequential consistency!

Further Impact of Reordering

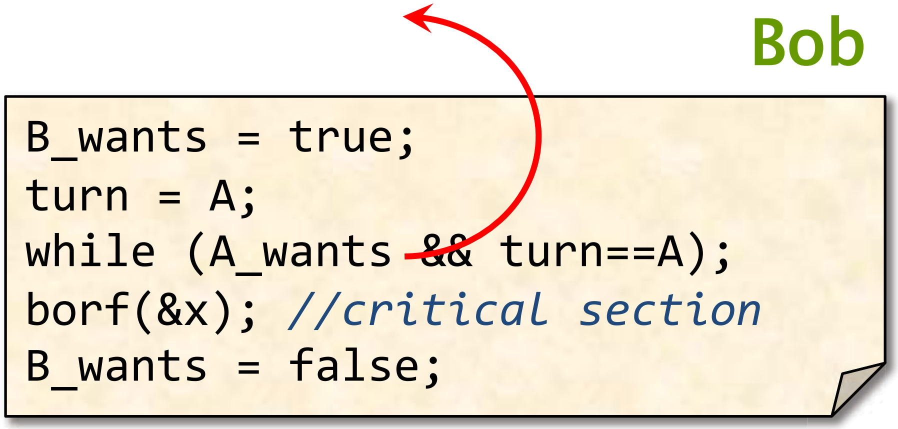
Peterson’s algorithm revisited

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 Tapir/LLVM compiler implements a memory fence via the function `atomic_thread_fence()` defined in the C header file `stdatomic.h`.*
- The typical cost of a memory fence is comparable to that of an L2-cache access.

*See http://en.cppreference.com/w/c/atomic .

Restoring Consistency

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;
```

Memory fences can restore sequential consistency.

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

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

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

Restoring Consistency

Alice

```
A_wants = true;
turn = B;
atomic_thread_fence();
while (B_wants && turn==B);
asm volatile("":::"memory");
frob(&x); //critical section
asm volatile("":::"memory");
A_wants = false;
```

Bob

```
B_wants = true;
turn = A;
atomic_thread_fence();
while (A_wants && turn==A);
asm volatile("":::"memory");
borf(&x); //critical section
asm volatile("":::"memory");
B_wants = false;
```

Back in the day, in addition to the memory 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.

Restoring Consistency with C11

Alice

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

Bob

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

The C11 language standard defines its own weak memory model, in which you can control hardware and compiler reordering of memory operations by:

- Declaring variables as `_Atomic`; and
- Using the functions `atomic_load()`, `atomic_store()`, etc. as needed.

See http://en.cppreference.com/w/c/atomic.

Implementing General Mutexes

Theorem [Burns-Lynch]. Any n-thread deadlock-free mutual-exclusion algorithm using only `LOAD` and `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*)

This image is in the public domain.

Compare-and-Swap

The *compare-and-swap* operation is provided by the cmpxchg instruction on x86-64. The C header file stdatomic.h provides CAS via the built-in function

atomic_compare_exchange_strong()

which can operate on various integer types.*

Specification

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

- Executes atomically.
- Implicit fence.

* See http://en.cppreference.com/w/cpp/atomic/atomic_compare_exchange .

Mutex Using CAS

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

Proof.

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

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

Just the space for the mutex itself. ■

Summing Problem

```
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;
}
```

Race!

Mutex Solution

```
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;
}
```

Mutex Solution

```
          ute(const X& v);
          ) {
          nt n = 1000000;
          X myArray[n];
          ;


    esult = 0;
  c   _for (int i = 0; i < n; ++i) {
    I   t temp = compute(myArray[i]);
    L. ock();
    result += temp;
    L.unlock();
  }
  printf( "The result is: %f\n”, result );
  return 0;
}
```

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?

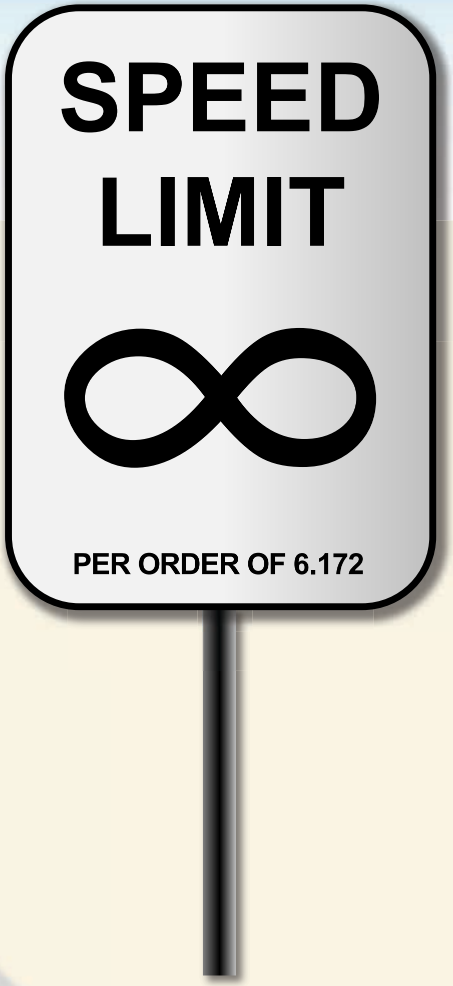
A. All other loop iterations must wait.

CAS Solution

```
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 (!CAS(&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*.

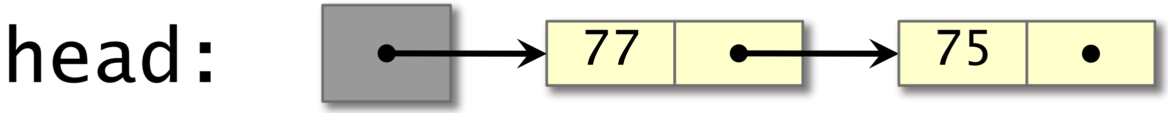


Lock-Free Algorithms

Lock-Free Stack

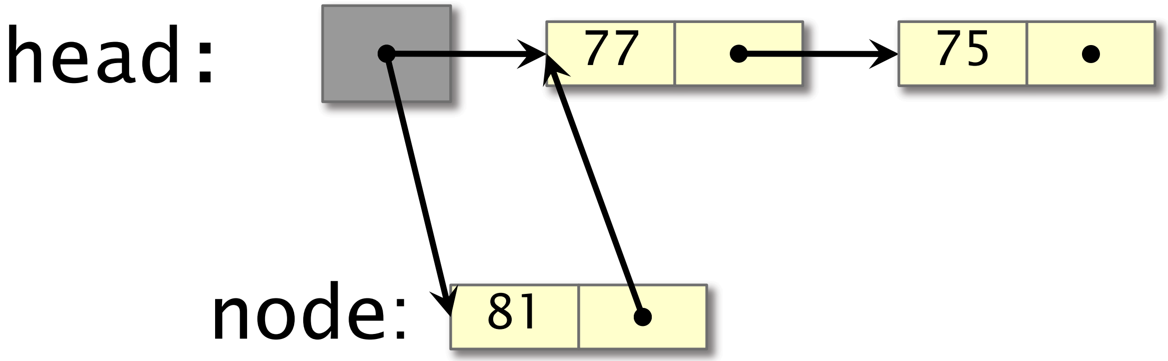
```
struct Node {
  Node* next;
  int data;
};

struct Stack {
    Node* head;
         ⋮
```



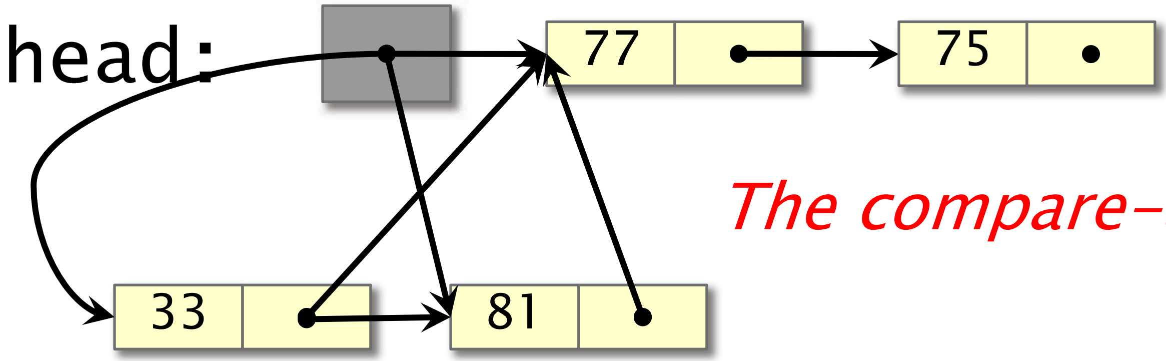
Lock-Free Push

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



Lock-Free Push with Contention

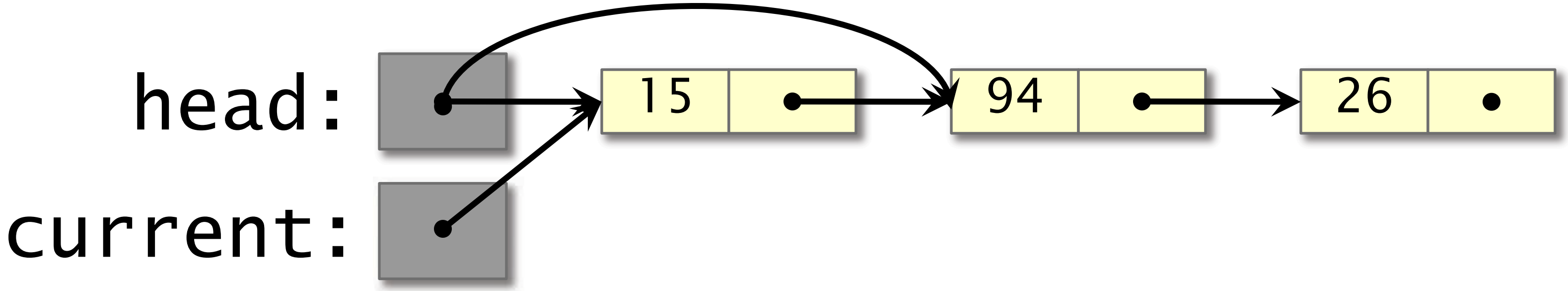
```
void push(Node* node) {
  do {
      node->next = head;
    } while (!CAS(&head, node->next, node));
}
```



The compare-and-swap fails!

Lock-Free Pop

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



Compare and compare-and-swap

Compare-and-swap acquires a cache line in exclusive mode, invalidating the cache line in other caches.

Result: High contention if all processors are doing CAS's to same cache line.

Better way: First read if value at memory location changed before doing CAS, and only do CAS if value didn't change.

Lock-Free Push and Pop

```
void push(Node* node) {
  do {
       node->next = head;
     } while (head != node->next ||
             !CAS(&head, node->next, node));
}
```

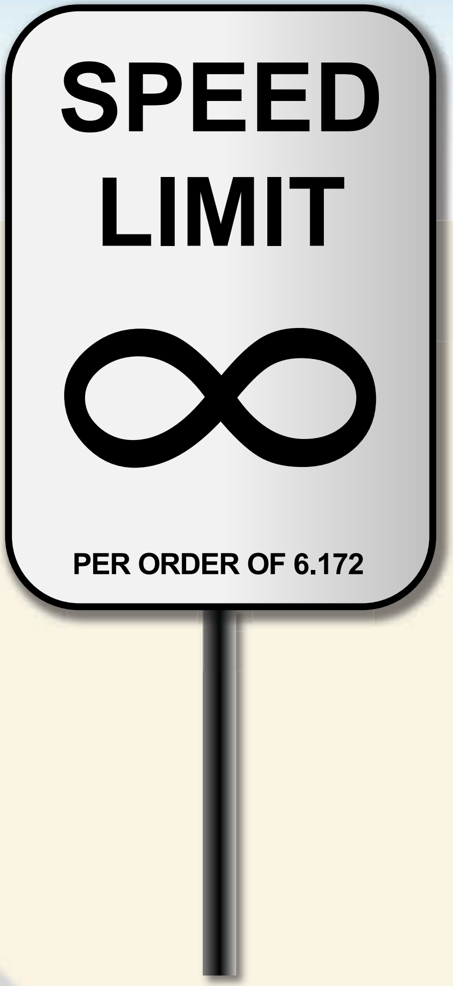
```
Node* pop() {
  Node* current = head;
  while (current) {
    if (head == current &&
        CAS(&head, current, current->next)) break;
    current = head;
  }
  return current;
}
```

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.
  - Allows executing a block of code atomically.

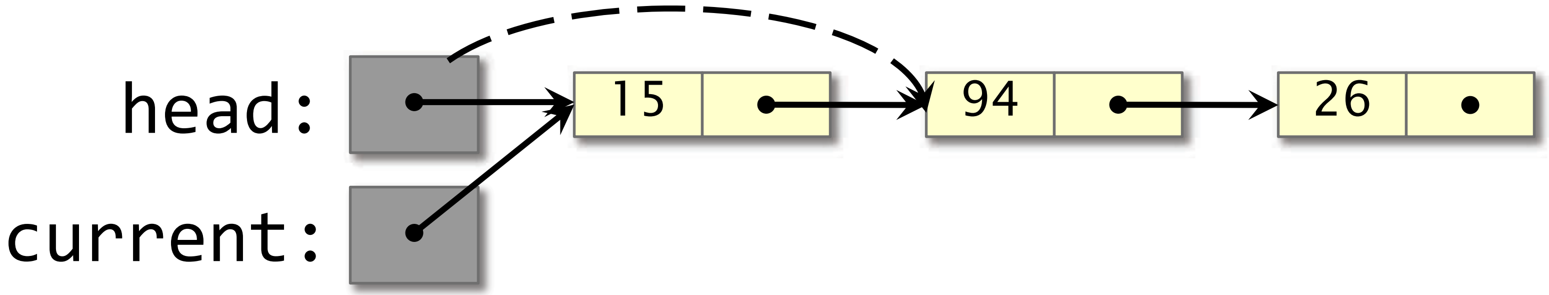
Practical Issues

- Memory management.
- Contention.
- The ABA problem.



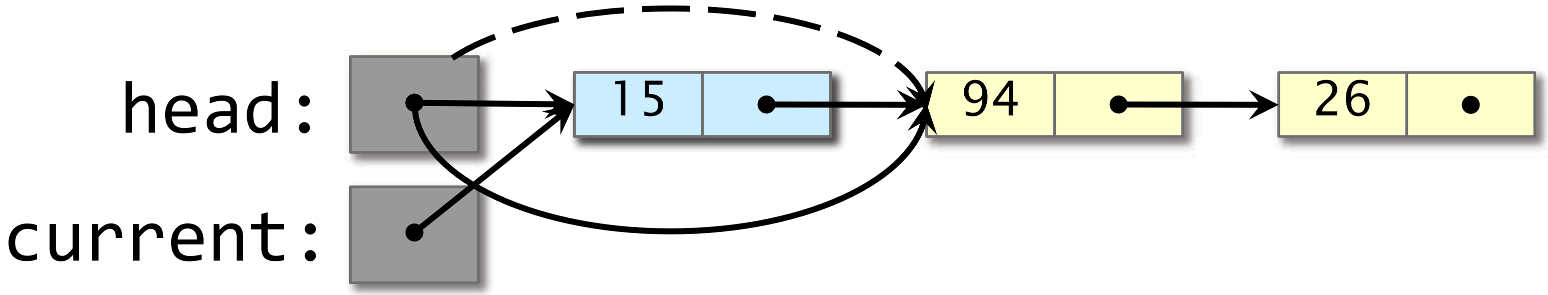
The ABA Problem

ABA Example



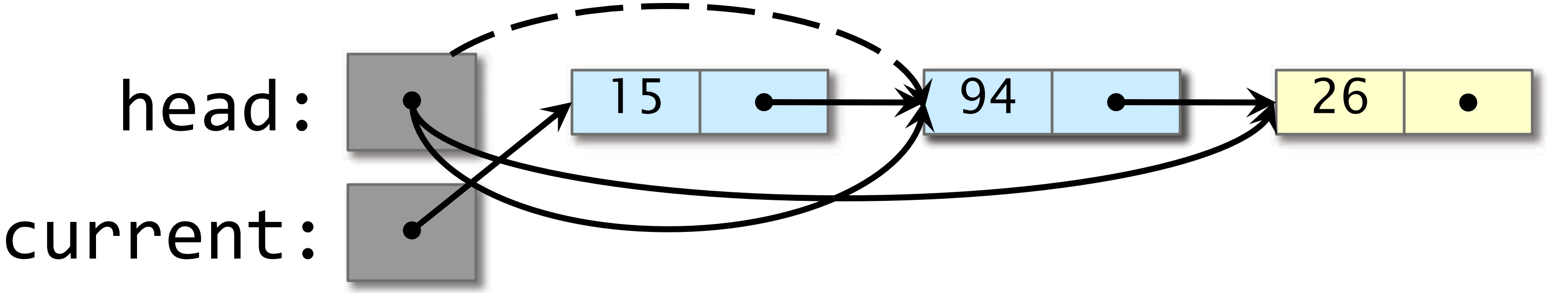
1. Thread 1 begins to pop the node containing 15, but stalls after reading `current->next`.

ABA Example



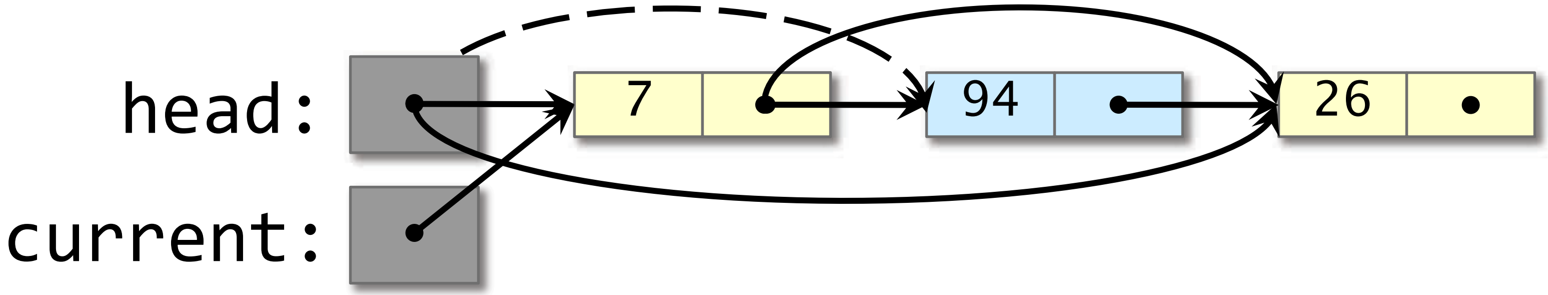
1. Thread 1 begins to pop the node containing 15, but stalls after reading `current->next`.
2. Thread 2 pops the node containing 15.

ABA Example



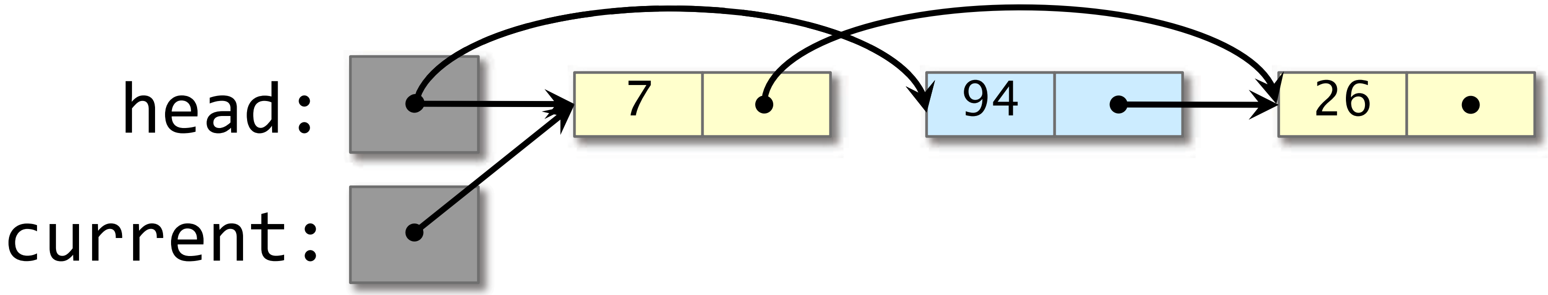
1. Thread 1 begins to pop the node containing 15, but stalls after reading current->next.
2. Thread 2 pops the node containing 15.
3. Thread 2 pops the node containing 94.

ABA 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 pushes the node 7, reusing 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 pushes the node 7, reusing 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.