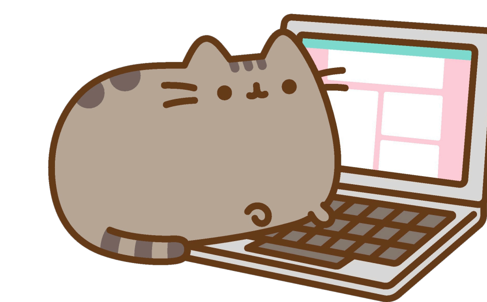
Announcements

- Today's guest lecture is postponed to next Tuesday, April 16. The corresponding report is postponed to one week after that, Tuesday April 23 (the usual time, 5pm).
- Slight change of plans for next lecture - it will be on filters and in person.
- The deadline for project slides has been postponed to **1:30pm on Wednesday, April 17**, the new start of the presentations. Like for the proposal, the slides are due at the same time for everyone for fairness.
- Please sign up for a project presentation slot if you have not yet done so.
- Extra credit is due May 1 @ 5pm, final project writeup is due May 2 @ 5pm.

+ 

Lecture 23: Synchronization Without Locks

Helen Xu
hxu615@gatech.edu

Georgia Tech College of Computing
School of Computational Science and Engineering

Slides from MIT 6.172

Sequential Consistency

Memory Models

- A memory model defines the rules under which **writes to stored object data become visible to later reads** to that data.

- Compiler optimizations (e.g., loop fusion) move statements in the program, which can affect the **order of read/write operations** to potentially shared variables.

- Without a memory model, a compiler **cannot apply such optimizations** to multithreaded programs in general.

```
for (i = 0; i < 300; i++)
  a[i] = a[i] + 3;

for (i = 0; i < 300; i++)
  b[i] = b[i] + 4;
```

Loop fusion →

```
for (i = 0; i < 300; i++)
  {
    a[i] = a[i] + 3;
    b[i] = b[i] + 4;
  }
```

https://en.wikipedia.org/wiki/Memory_model_(programming)

https://compileroptimizations.com/category/loop_fusion.htm

Motivating Example

Initially, `a = b = 0`

Processor 0

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

Processor 1

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

Motivating Example

Initially, `a = b = 0`

Processor 0

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

Processor 1

```
mov b, 1  ;Store
mov %eax, a ;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?

From MIT OCW 6.172

Motivating Example

Initially, `a = b = 0`

Processor 0

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

Processor 1

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

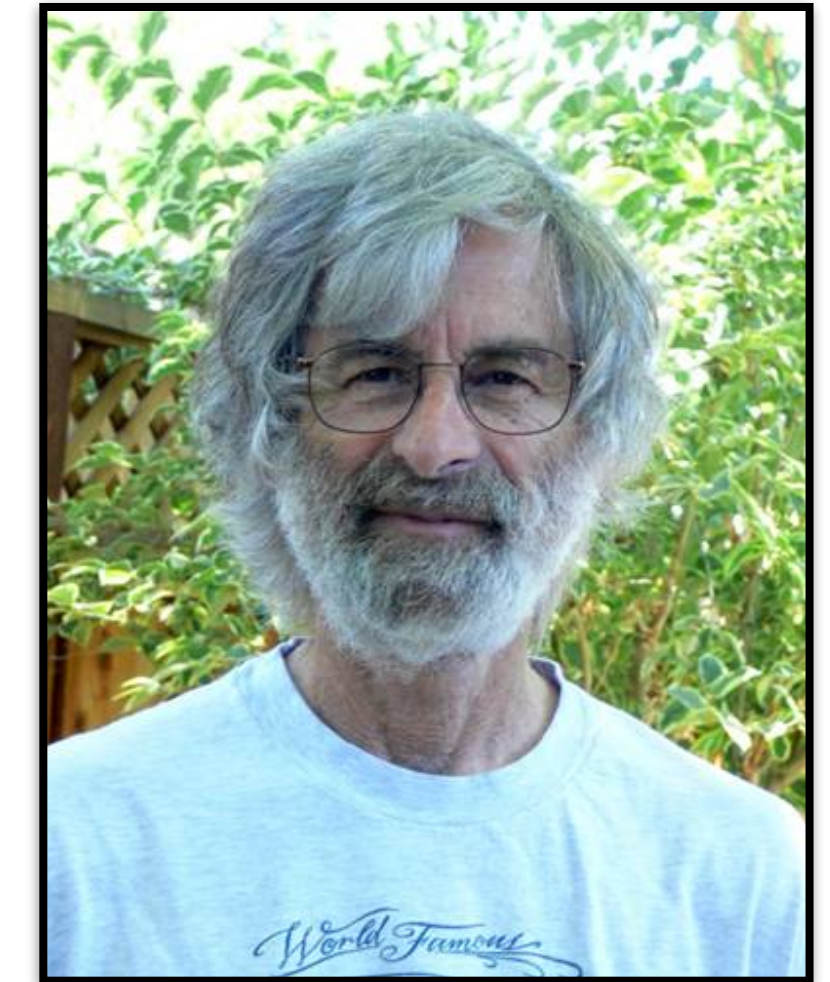
Q: Is it possible that Processor 0's %ebx and Processor 1's %eax both contain the value 0 after the processors have both executed their code?

A: It depends on the **memory model**: how memory operations perform in the parallel computer system.

From MIT OCW 6.172

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

Processor 1

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

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

Example

Initially, `a = b = 0`

Processor 0

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

Processor 1

```
3 mov b, 1   ;Store
4 mov %eax, a ;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					
%ebx	0					

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

Example

Initially, `a = b = 0`

Processor 0

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

Processor 1

```
3 mov b, 1   ;Store
4 mov %eax, a ;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				
%ebx	0	1				

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

From MIT OCW 6.172

Example

Initially, `a = b = 0`

Processor 0

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

Processor 1

```
3 mov b, 1   ;Store
4 mov %eax, a ;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			
%ebx	0	1	1			

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

Example

Initially, `a = b = 0`

Processor 0

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

Processor 1

```
3 mov b, 1   ;Store
4 mov %eax, a ;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		
%ebx	0	1	1	1		

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

Example

Initially, `a = b = 0`

Processor 0

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

Processor 1

```
3 mov b, 1   ;Store
4 mov %eax, a ;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	
%ebx	0	1	1	1	1	

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

Example

Initially, `a = b = 0`

Processor 0

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

Processor 1

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

Modern machines do not implement sequential consistency

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

From MIT OCW 6.172

Reasoning about Sequential 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 $\rightarrow$ that yields that memory state.

Mutual Exclusion Without Locks

Recall: 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, etc.

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;
```

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);
foo(&x); // critical section
A_wants = false;
```

Bob

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

From MIT OCW 6.172

Intuition behind Peterson's Algorithm

Alice

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

Bob

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

- 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.

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);
foo(&x); // critical section
A_wants = false;
```

Bob

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

Proof of Mutual Exclusion

Alice

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

Bob

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

- WLOG, assume that Bob was the last to write to turn:

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

Proof of Mutual Exclusion

Alice

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

Bob

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

Bob

```
B_wants = true;
turn = A;
while (A_wants && turn == A);
bar(&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:

$$\text{write}_A(\texttt{A_wants = true}) \rightarrow \text{write}_A(\texttt{turn = B})$$

- Bob's program order:

$$\text{write}_B(\texttt{turn = A}) \rightarrow \text{read}_B(\texttt{A_wants}) \rightarrow \text{read}_B(\texttt{turn})$$

Proof of Mutual Exclusion

Alice

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

Callouts: 1 (A_wants = true;), 2 (turn = B;)

Bob

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

Callouts: 3 (turn = A;), 4 (A_wants), 5 (turn == A)

- 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:

$$\text{write}_A(\texttt{A_wants = true}) \rightarrow \text{write}_A(\texttt{turn = B})$$

- Bob's program order:

$$\text{write}_B(\texttt{turn = A}) \rightarrow \text{read}_B(\texttt{A_wants}) \rightarrow \text{read}_B(\texttt{turn})$$

- What did Bob read?

```
A_wants = ?
turn = ?
```

Proof of Mutual Exclusion

Alice

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

Bob

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

1 2 3 4 5

- 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:

$$\text{write}_A(\texttt{A_wants = true}) \rightarrow \text{write}_A(\texttt{turn = B})$$

- Bob's program order:

$$\text{write}_B(\texttt{turn = A}) \rightarrow \text{read}_B(\texttt{A_wants}) \rightarrow \text{read}_B(\texttt{turn})$$

- What did Bob read?

  `A_wants = true`

  `turn = A`

Contradiction: Bob should spin

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

Program Order

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

Execution Order

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

Why might the hardware decide to reorder these instructions?

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

Instruction Reordering

Program Order

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

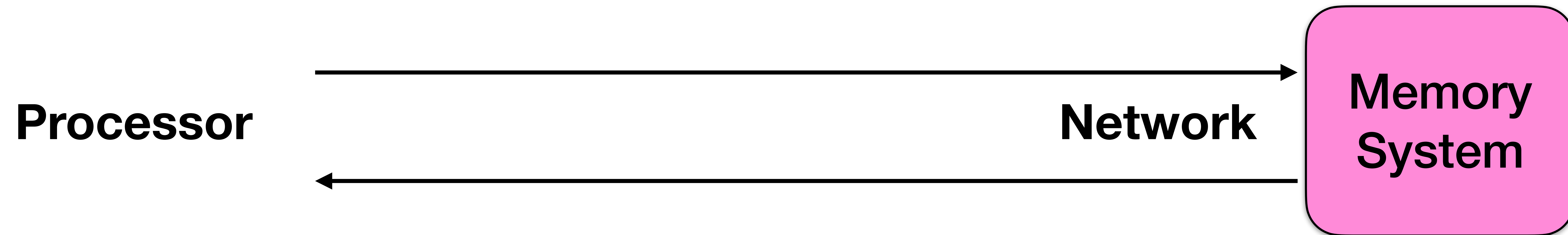
Execution Order

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

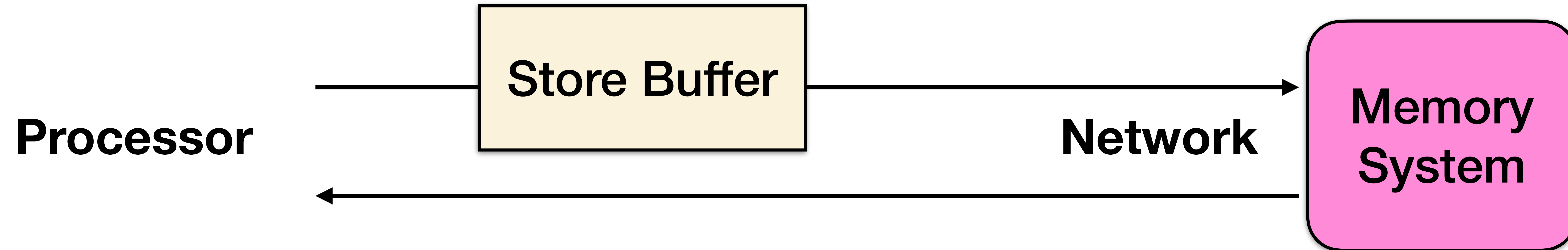
When is it safe for the hardware or compiler to perform this reordering?

When a =/= b, and there is no concurrency.

Hardware Reordering

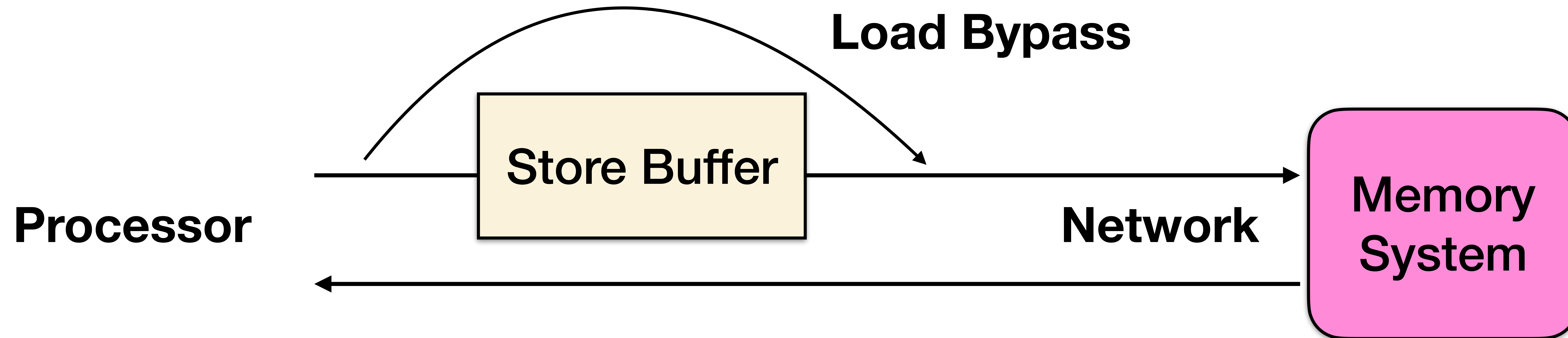


Hardware Reordering



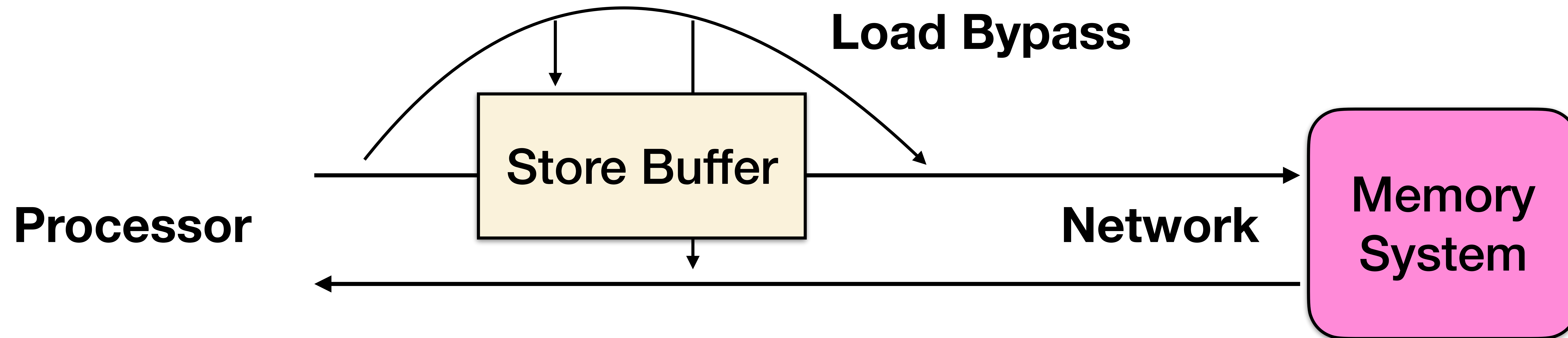
- The processor can issue stores faster than the network can handle them: put them in a **store buffer**

Hardware Reordering



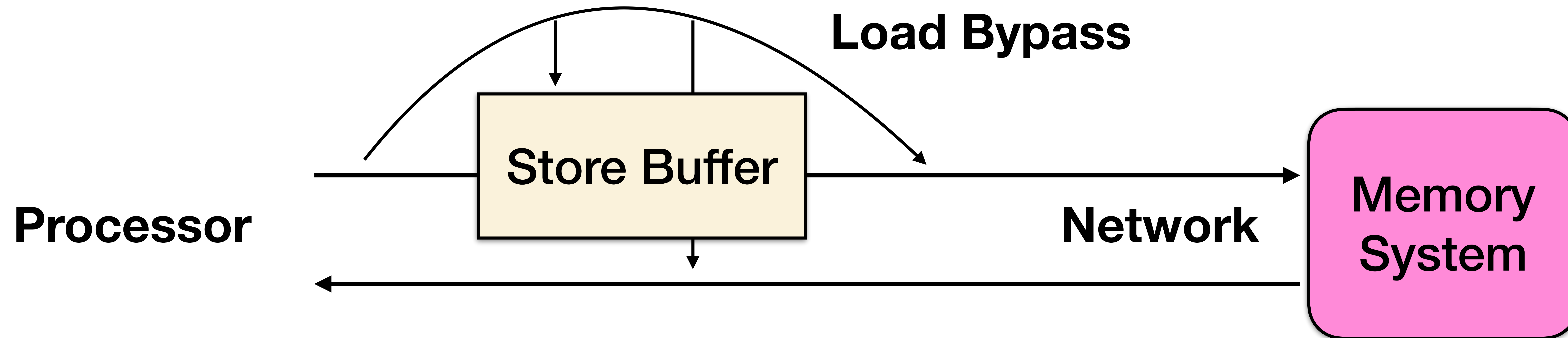
- The processor can issue stores faster than the network can handle them: put them in a **store buffer**
- Since a load can stall the processor until it is satisfied, **loads take priority**, bypassing the store buffer.

Hardware Reordering



- The processor can issue stores faster than the network can handle them: put them in a **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.

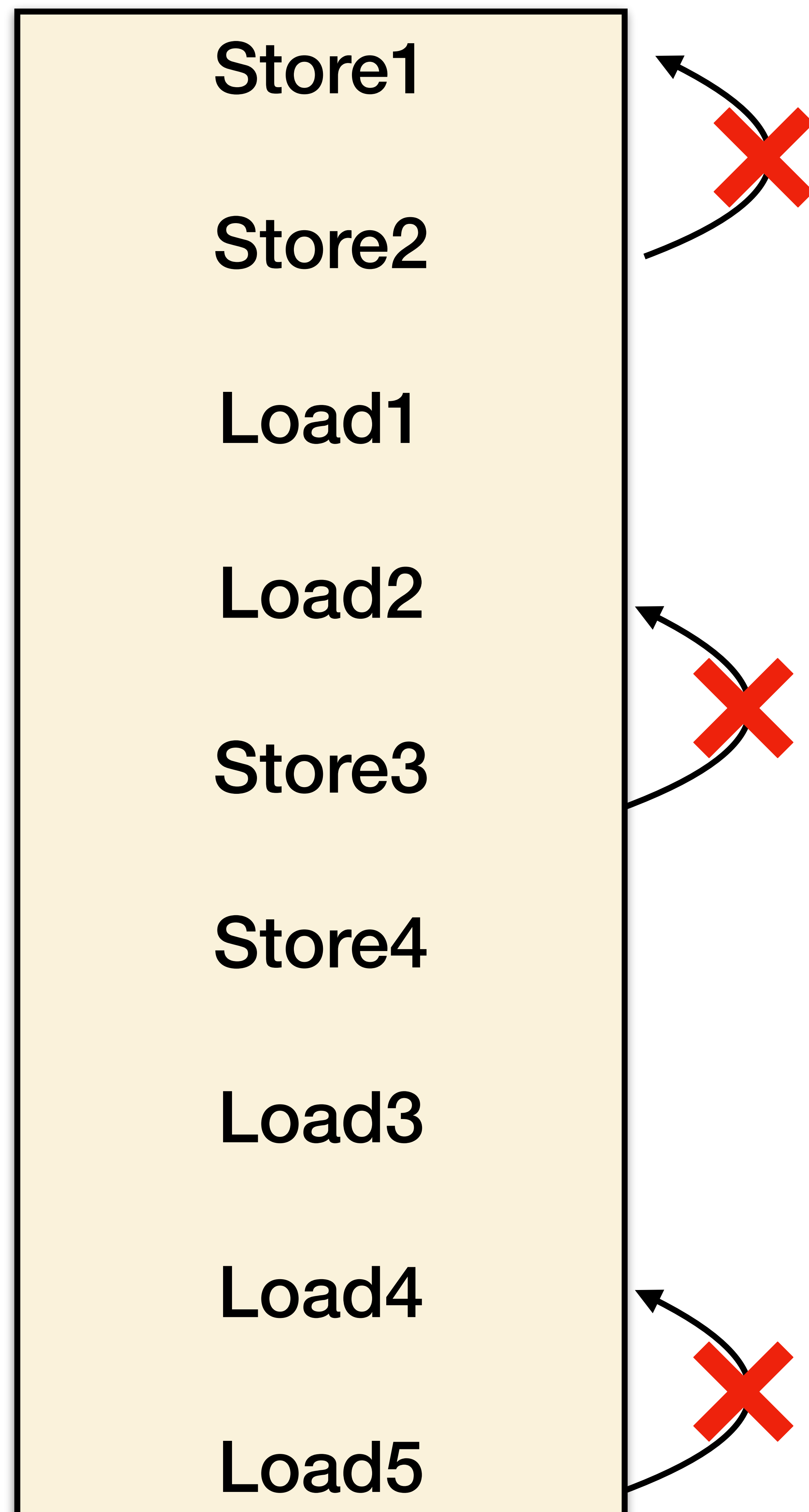
Hardware Reordering



- The processor can issue stores faster than the network can handle them: put them in a **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

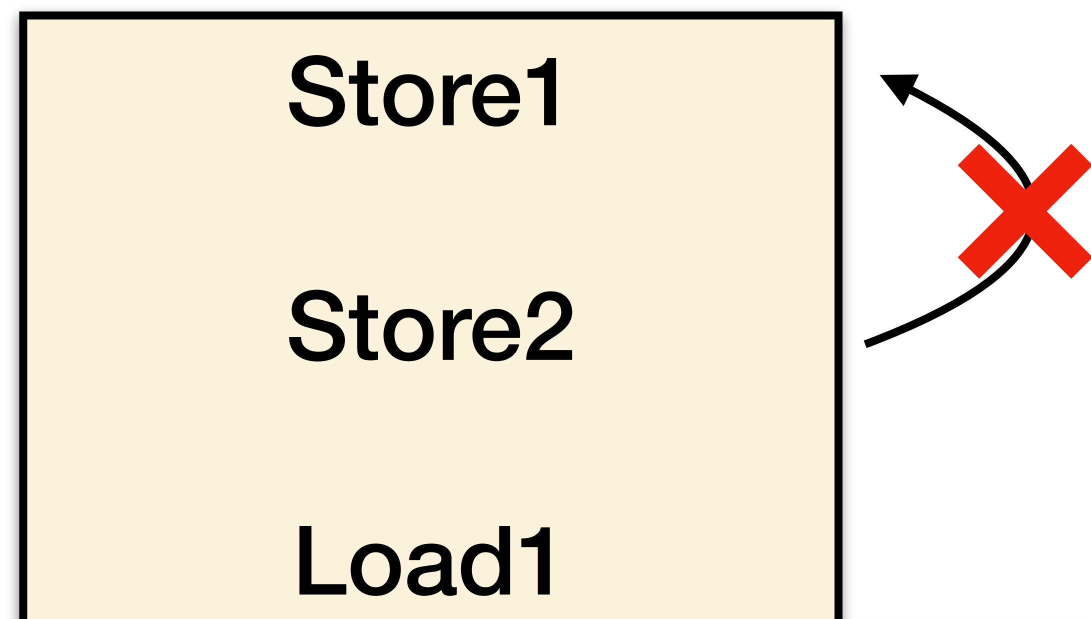


House rules:

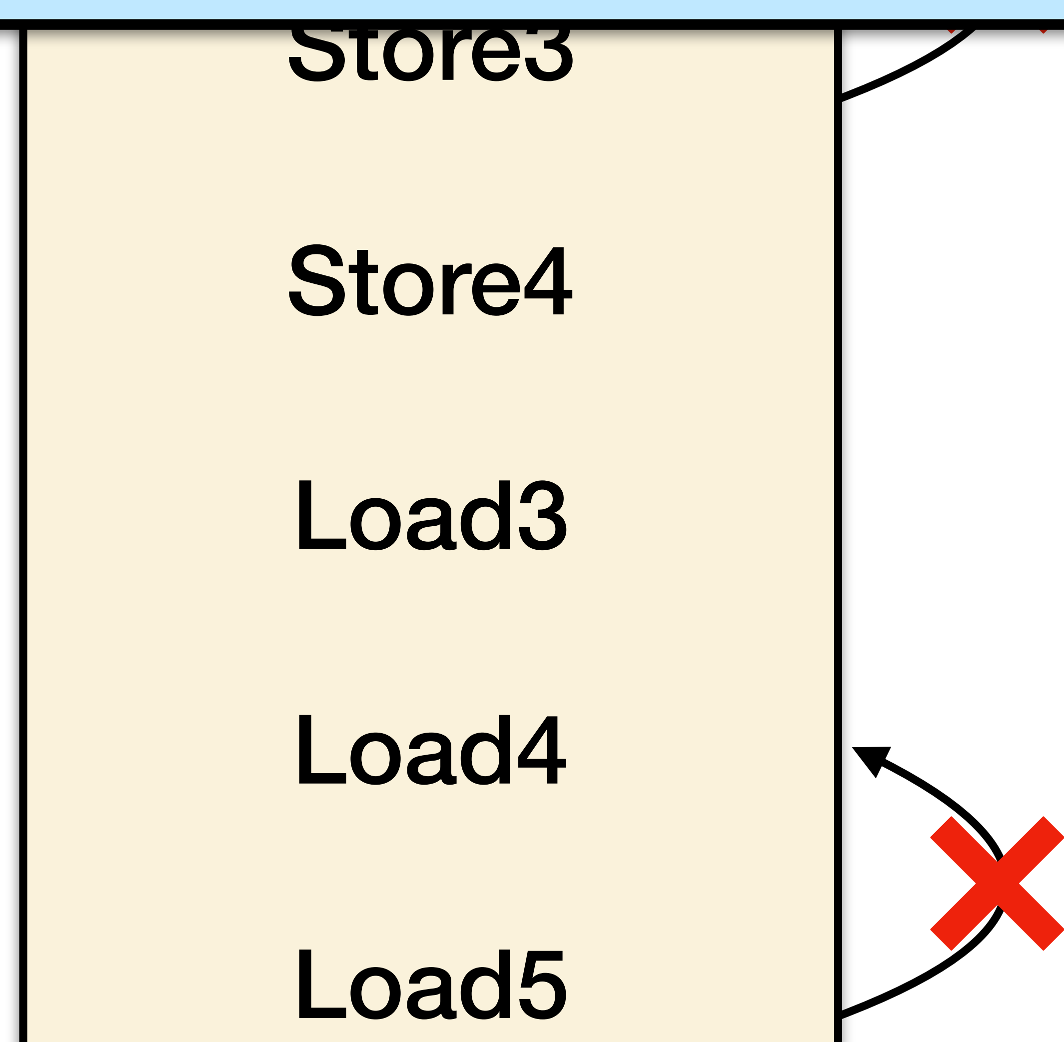
- Loads are not reordered with Loads.
- Stores are not reordered with Stores.
- Stores are not reordered with prior Loads.
- A Load may be reordered with a prior Store to a **different location** but not with a prior Store to the same location.
- Loads and Stores are not reordered with Lock instructions.
- Stores to the same location respect a global total order.
- Lock instructions respect a global total order.
- Memory ordering preserves transitive visibility (“causality”).

x86-64 Total Store Order

Instruction Trace



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



Loads

House rules:

- Loads are not reordered with Loads.
- Stores are not reordered with Stores.
- Stores are not reordered with prior Loads.
- A Load may be reordered with a prior ... with
- ... with Lock instructions.
- Stores to the same location respect a global total order.
- Lock instructions respect a global total order.
- Memory ordering preserves transitive visibility (“causality”).

Impact of Reordering

Initially, `a = b = 0`

Processor 0

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

Processor 1

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

Impact of Reordering

Initially, a = b = 0

Processor 0

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

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

Processor 1

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

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

The ordering <2, 4, 1, 3> produces `%eax = %ebx = 0`.

Instruction reordering violates sequential consistency!

Peterson's Algorithm Revisited

Alice

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

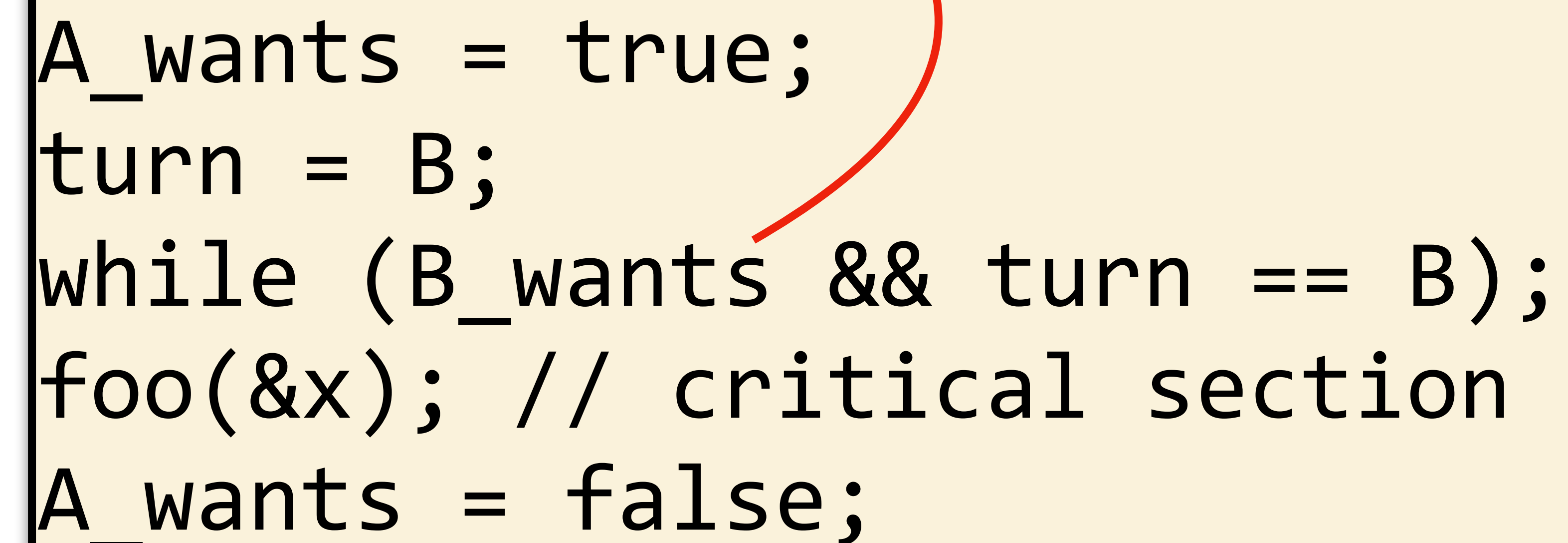
Bob

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

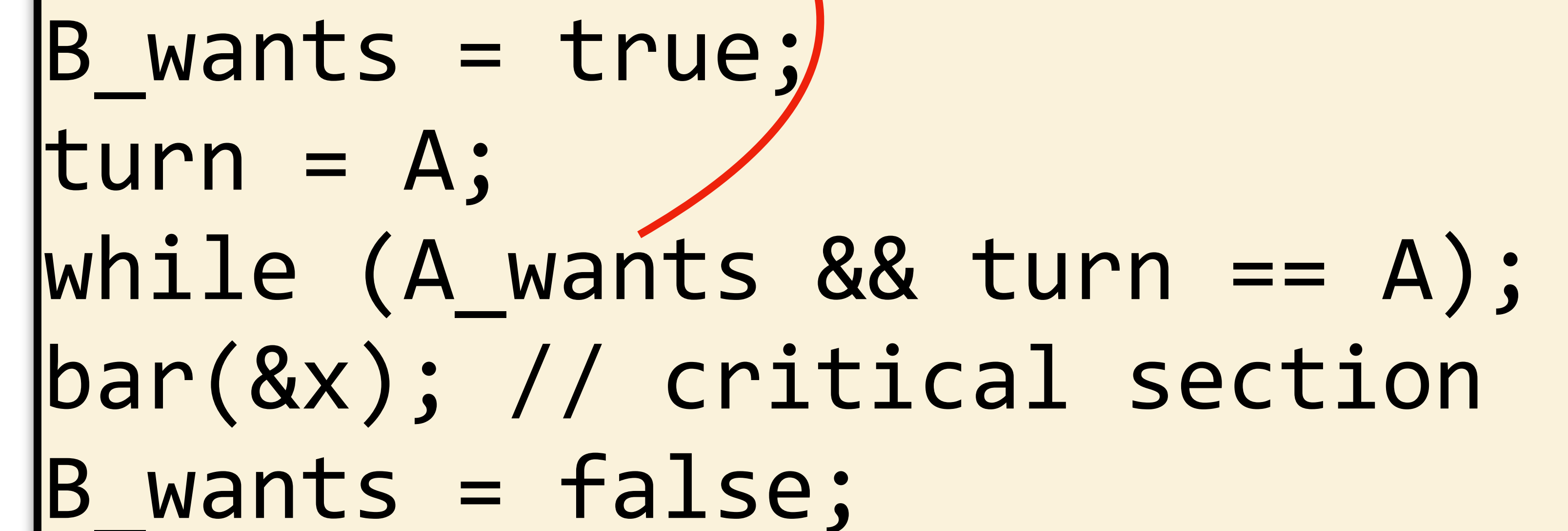
Can reordering cause issues in the proof of correctness for Peterson's algorithm?

Peterson's Algorithm Revisited

Alice



Bob



- The loads of `B_wants` and `A_wants` can be reordered before the stores 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 std library for C++ implements a memory fence via the function atomic_thread_fence()*
- The typical cost of a memory fence is comparable to that of an **L2-cache access**.

* https://en.cppreference.com/w/cpp/atomic/atomic_thread_fence

Restoring Consistency

Alice

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

Bob

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

Memory fences can restore sequential consistency.

Alice

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

Bob

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

Just a memory fence is not enough - you also need to account for the **compiler**.

Restoring Consistency

Alice

```
A_wants = true;
turn = B;
atomic_thread_fence();
while (B_wants && turn == B);
asm volatile(“”:::”memory”);
foo(&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”);
bar(&x); // critical section
asm volatile(“”:::”memory”);
B_wants = false;
```

In addition to the memory fence, we would need to declare variables as `volatile` to prevent the compiler from optimizing away memory references.

The `volatile` keyword signals to the compiler that the variable should be **re-read from memory each time it is used** (instead of being optimized out or cached by the compiler in a register).

From MIT OCW 6.172

Restoring Consistency with C11

Alice

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

→

Alice

```
atomic_store(&A_wants, true);
atomic_store(&turn, B);
while (atomic_load(&B_wants) &&
atomic_load(&turn) == B);
foo(&x); // critical section
atomic_store(&A_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.

https://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)

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.

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

- Executes atomically
- Implicit fence

https://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.

Just the space for the mutex

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

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

Summing Problem

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

  int result = 0;
  #pragma omp parallel for
  for (size_t i = 0; i < n; ++i) {
    result += compute(myArray[i]);
  }
  printf("The result is: %d\n”, result);
  return 0;
}
```

Race

Mutex Solution

```
int compute(const X& v);
int main() {
  const size_t n = 1000000;
  extern X myArray[n];
  mutex L;
  // …

  int result = 0;
  #pragma omp parallel for
  for (size_t i = 0; i < n; ++i) {
    int temp = compute(myArray[i]);
    L.lock();
    result += temp;
    L.unlock();
  }
  printf("The result is: %d\n”, result);
  return 0;
}
```

Why is compute() outside the lock?

Mutex Solution

```
int compute(const X& v);
int main() {
  const size_t n = 1000000;
  extern X myArray[n];
  mutex L;
  // …

  int result = 0;
  #pragma omp parallel for
  for (size_t i = 0; i < n; ++i) {
    int temp = compute(myArray[i]);
    L.lock();
    result += temp;
    L.unlock();
  }
  printf("The result is: %d\n”, result);
  return 0;
}
```

What happens if the operating system swaps out a loop iteration just after it acquires the mutex?

Mutex Solution

```
int compute(const X& v);
int main() {
  const size_t n = 1000000;
  extern X myArray[n];
  mutex L;
  // …

  int result = 0;
  #pragma omp parallel for
  for (size_t i = 0; i < n; ++i) {
    int temp = compute(myArray[i]);
    L.lock();
    result += temp;
    L.unlock();
  }
  printf("The result is: %d\n", result);
  return 0;
}
```

What happens if the operating system swaps out a loop iteration just after it acquires the mutex?

All other loop iterations must wait.

Mutex Solution

```
int compute(const X& v);
int main() {
  const size_t n = 1000000;
  extern X myArray[n];
  mutex L;
  // …

  int result = 0;
  #pragma omp parallel for
  for (size_t i = 0; i < n; ++i) {
    int temp = compute(myArray[i]);
    L.lock();
    result += temp;
    L.unlock();
  }
  printf("The result is: %d\n”, result);
  return 0;
}
```

All we want is to atomically load x, add temp, and then store x

What happens if the operating system swaps out a loop iteration just after it acquires the mutex?

All other loop iterations must wait.

CAS Solution

```
int compute(const X& v);
int main() {
  …

  int result = 0;
  #pragma omp parallel for
  for (size_t i = 0; i < n; ++i) {
    int temp = compute(myArray[i]);
    int old, new
    do {
      old = result;
      new = old + temp;
    } while (!CAS(&result, old, new));
  }
  …
}
```

CAS Solution

```
int compute(const X& v);
int main() {
  …

  int result = 0;
  #pragma omp parallel for
  for (size_t i = 0; i < n; ++i) {
    int temp = compute(myArray[i]);
    int old, new
    do {
      old = result;
      new = old + temp;
    } while (!CAS(&result, old, new));
  }
  …
}
```

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

CAS Solution

```
int compute(const X& v);
int main() {
  …

  int result = 0;
  #pragma omp parallel for
  for (size_t i = 0; i < n; ++i) {
    int temp = compute(myArray[i]);
    int old, new
    do {
      old = result;
      new = old + temp;
    } while (!CAS(&result, old, new));
  }
  …
}
```

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

No other loop iteration needs to wait. The algorithm is **nonblocking**.

Lock-Free Algorithms

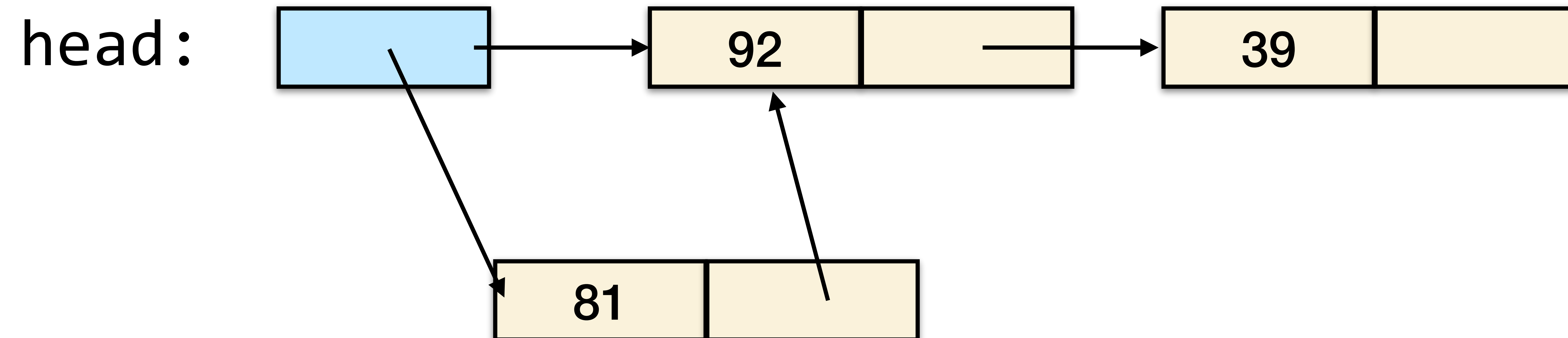
Lock-Free Stack

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

struct Stack {
  Node* head;
  …
```

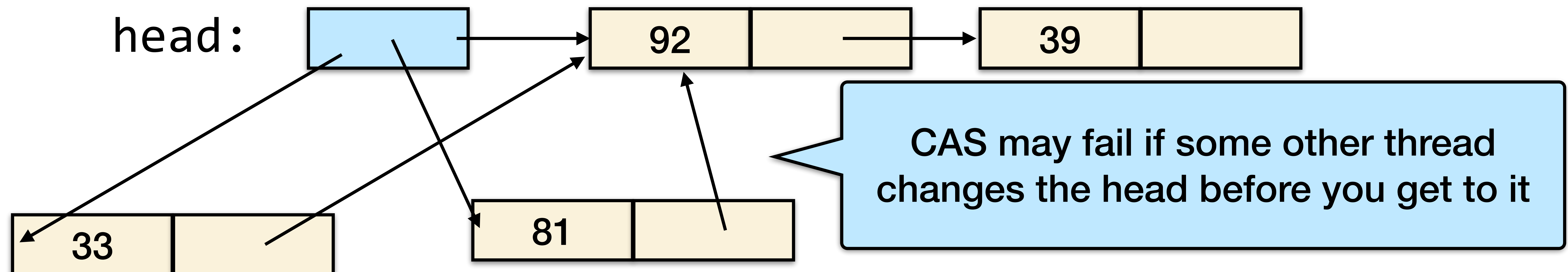
Lock-Free Push

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



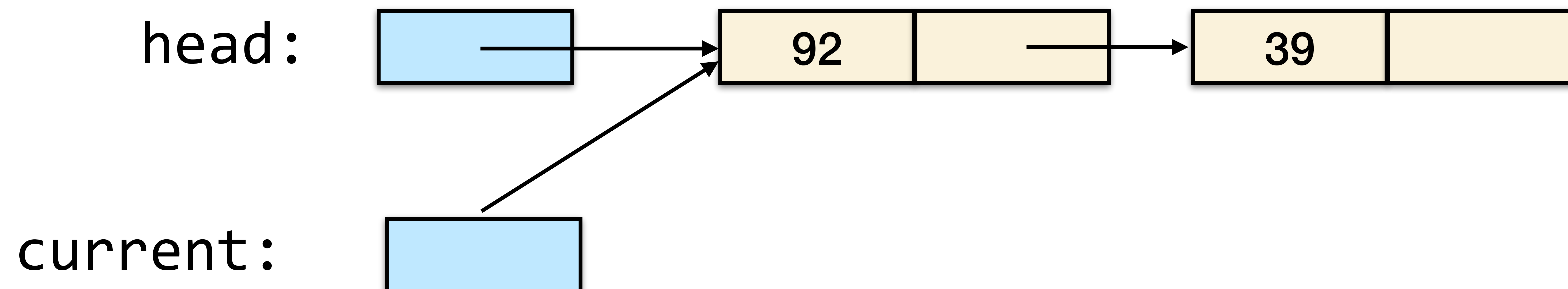
Lock-Free Push with Contention

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



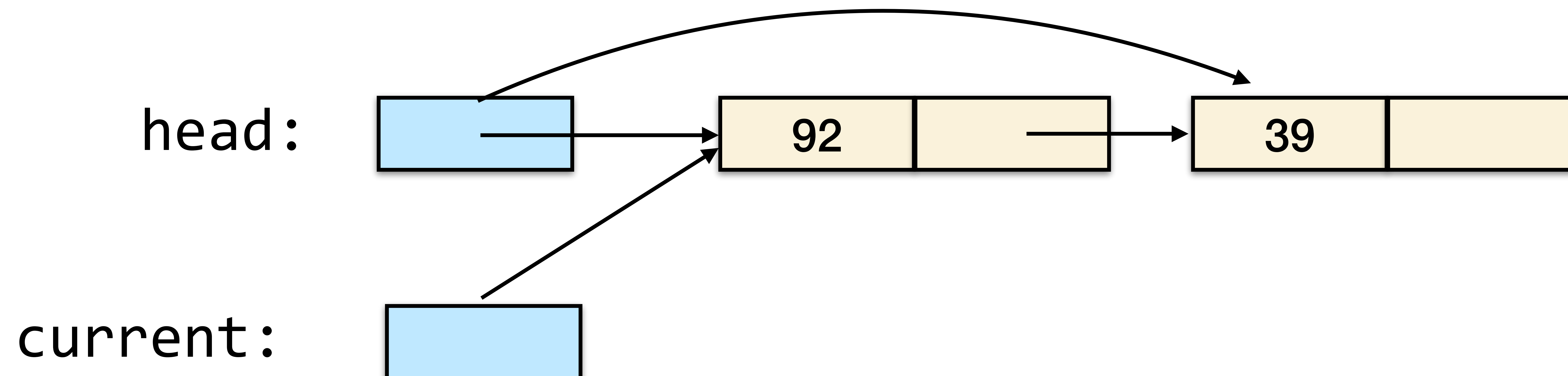
Lock-Free Pop

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



Lock-Free Pop

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



Optimization: Compare and CAS

- 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;
}
```

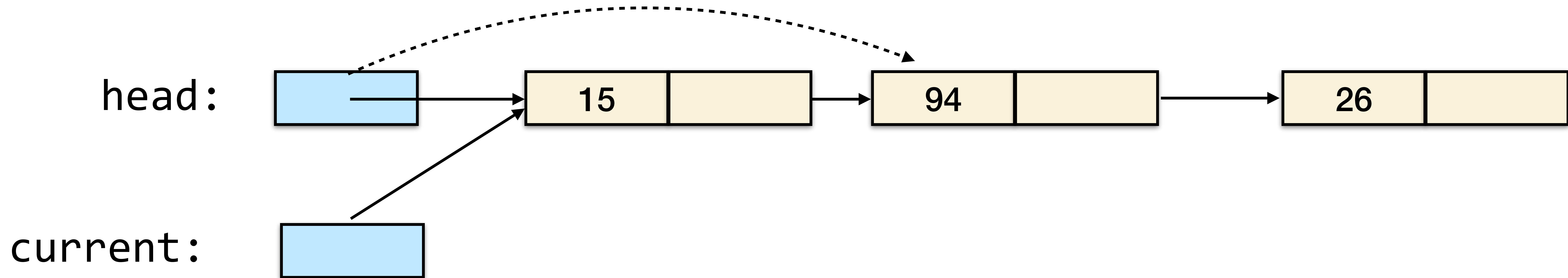
Lock-Free Data Structures

- 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.

Practical issues:
- Memory management
- Contention
- ABA problem

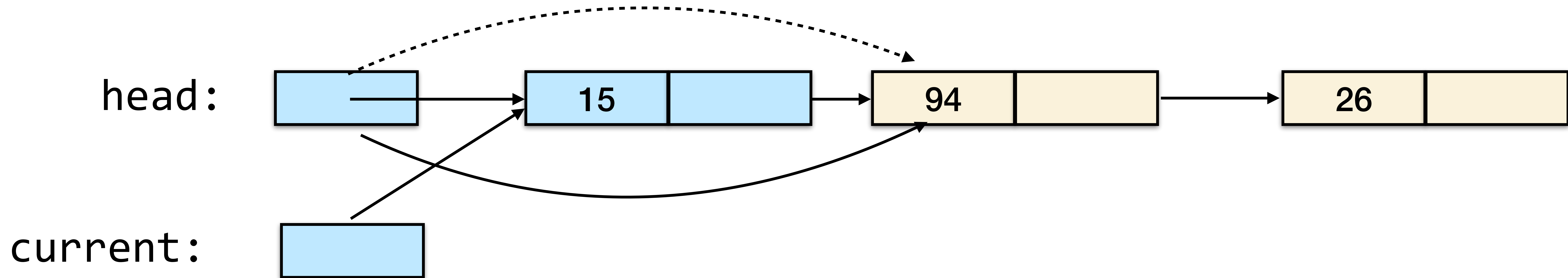
The ABA Problem

ABA Example



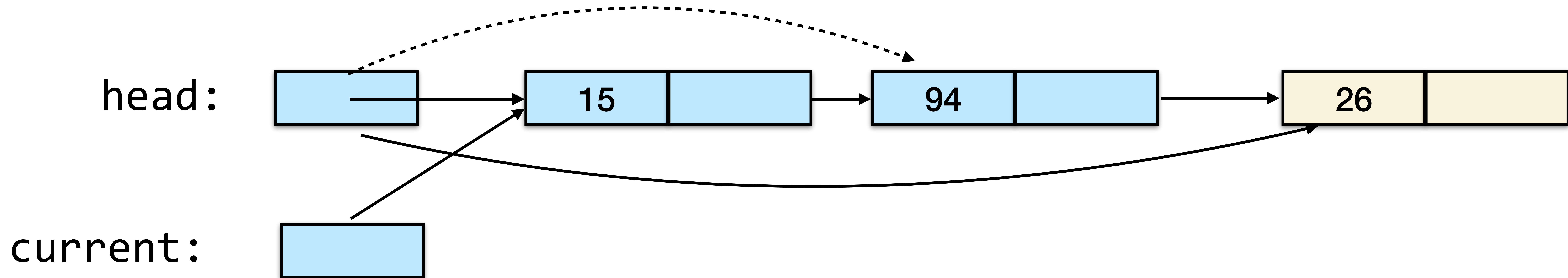
1. Thread 1 begins to pop the node containing 15, but stalls after reading current->next.

ABA Example



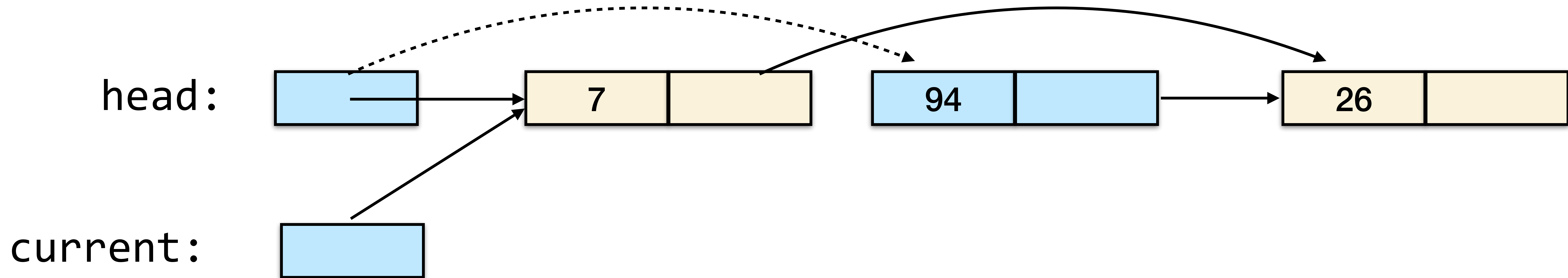
1. Thread 1 begins to pop the node containing 15, but stalls after reading current->next.
2. Thread 2 pops the node containing 15.

ABA Example



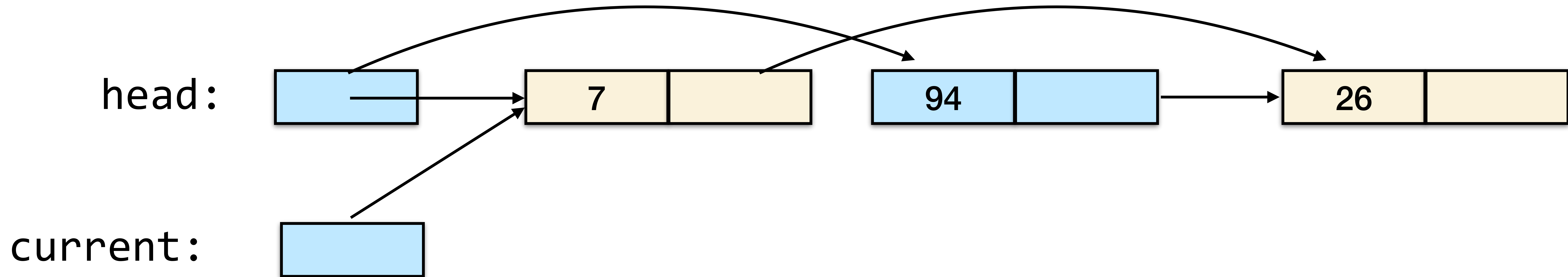
1. Thread 1 begins to pop the node containing 15, but stalls after reading current->next.
2. Thread 2 pops the node containing 15.
3. Thread 2 pops the node containing 94.

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.