Lecture 8: introduction to atomics

read-modify-write, get-and-add, compare-and-swap, spin lock, lock-free stack, ABA problem

Alexander Filatov
filatovaur@gmail.com

https://github.com/Svazars/parallel-programming/blob/main/slides/pdf/l8.pdf

In previous episodes

Formalization of concurrent execution

- Timeline, Event, Interval, Precedence

Concurrent objects

- Linearizability, linearization points, Sequential consistency, Quiescent consistency

Progress conditions

- Dependent progress: Deadlock-freedom, Starvation-freedom
- Non-blocking progress: Lock-freedom, Wait-freedom
- Dependent non-blocking progress: Obstruction-freedom

Register design space

- Bool/Int, SRSW/MRSW/MRMW, Safe/Regular/Atomic
- Atomic snapshot

Consensus number as a tool to formalize relative synchronization power

In previous episodes

Formalization of concurrent execution

- Timeline, Event, Interval, Precedence

Concurrent objects

- Linearizability, linearization points, Sequential consistency, Quiescent consistency

Progress conditions

- Dependent progress: Deadlock-freedom, Starvation-freedom
- Non-blocking progress: Lock-freedom, Wait-freedom
- Dependent non-blocking progress: Obstruction-freedom

Register design space

- Bool/Int, SRSW/MRSW/MRMW, Safe/Regular/Atomic
- Atomic snapshot

Consensus number as a tool to formalize relative synchronization power

It was quite intensive material about **theory and foundations** of concurrency.

In previous episodes

Formalization of concurrent execution

- Timeline, Event, Interval, Precedence

Concurrent objects

- Linearizability, linearization points, Sequential consistency, Quiescent consistency

Progress conditions

- Dependent progress: Deadlock-freedom, Starvation-freedom
- Non-blocking progress: Lock-freedom, Wait-freedom
- Dependent non-blocking progress: Obstruction-freedom

Register design space

- Bool/Int, SRSW/MRSW/MRMW, Safe/Regular/Atomic
- Atomic snapshot

Consensus number as a tool to formalize relative synchronization power

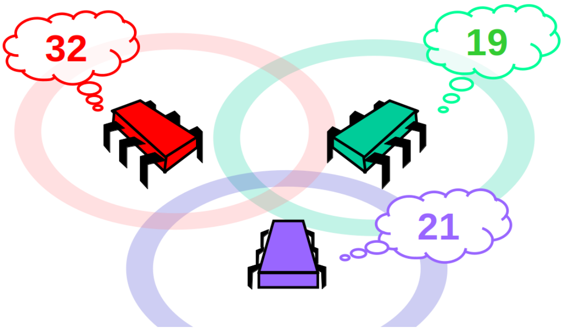
It was quite intensive material about **theory and foundations** of concurrency.

Let's switch to practical aspects.

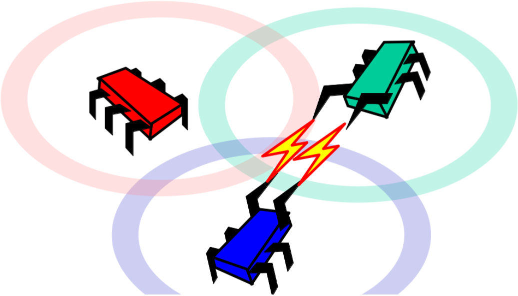
Lecture plan

1. Reminder: consensus
2. Read-Modify-Write
3. Concurrent Counter: puzzlers
4. Basic spin locks
5. Lock-free stack and ABA
6. Summary

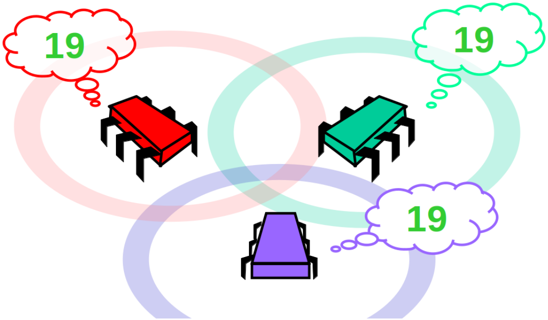
Each thread has private input



They communicate



They agree on some input



Consensus

- **Consistent**: all threads decide the same value
- **Valid**: the common decision value is some thread's input

Consensus

- **Consistent**: all threads decide the same value
- **Valid**: the common decision value is some thread's input

Theorem

There is no wait-free implementation of n-thread consensus from read-write registers

Consensus

- **Consistent**: all threads decide the same value
- **Valid**: the common decision value is some thread's input

Theorem

There is no wait-free implementation of n-thread consensus from read-write registers

Implication:

- Asynchronous computability different from Turing computability

Consensus

- **Consistent**: all threads decide the same value
- **Valid**: the common decision value is some thread's input

Theorem

There is no wait-free implementation of n-thread consensus from read-write registers

Implication:

- Asynchronous computability different from Turing computability

Read-write registers formalized in terms of safe/regular/atomic concurrent objects.

Consensus

- **Consistent**: all threads decide the same value
- **Valid**: the common decision value is some thread's input

Theorem

There is no wait-free implementation of n-thread consensus from read-write registers

Implication:

- Asynchronous computability different from Turing computability

Read-write registers formalized in terms of safe/regular/atomic concurrent objects.

Theorem could be adapted to:

- Registers
- Message-passing
- Carrier pigeons
- Any kind of asynchronous computation

Consensus number

An object **X** has *consensus number* **n**

- If it can be used to solve **n**-thread consensus
  - Take any number of instances of **X**
  - together with atomic read/write registers
  - and implement **n**-thread consensus
- But not **(n+1)**-thread consensus

Consensus number

An object **X** has *consensus number* **n**

- If it can be used to solve **n**-thread consensus
  - Take any number of instances of **X**
  - together with atomic read/write registers
  - and implement **n**-thread consensus
- But not **(n+1)**-thread consensus

Theorem

Atomic read/write registers have consensus number 1

Atomic registers cannot implement multiple assignment

- Single write/multi read OK
- Multi write/multi read impossible

Consensus number

An object **X** has *consensus number* **n**

- If it can be used to solve **n**-thread consensus
  - Take any number of instances of **X**
  - together with atomic read/write registers
  - and implement **n**-thread consensus
- But not **(n+1)**-thread consensus

Theorem

Atomic read/write registers have consensus number 1

Atomic registers cannot implement **wait-free** multiple assignment

- Single write/multi read OK
- Multi write/multi read impossible

Consensus numbers measure synchronization power

Theorem

- *If you can implement* X *from* Y
- *And* X *has consensus number* c
- *Then* Y *has consensus number at least* c

Consensus numbers measure synchronization power

Theorem

- *If you can implement X from Y*
- *And X has consensus number c*
- *Then Y has consensus number at least c*

Registers have consensus number 1

Consensus numbers measure synchronization power

Theorem

- *If you can implement* X *from* Y
- *And* X *has consensus number* c
- *Then* Y *has consensus number at least* c

Registers have consensus number 1

There are practically interesting problems that require higher consensus number

Consensus numbers measure synchronization power

Theorem

- *If you can implement* X *from* Y
- *And* X *has consensus number* c
- *Then* Y *has consensus number at least* c

Registers have consensus number 1

There are practically interesting problems that require higher consensus number

What should we do next?

Lecture plan

Read-Modify-Write Objects

Method call

- Returns previous value x
- Replaces x with `mumble(x)`

Read-Modify-Write Objects

Method call

- Returns previous value x
- Replaces x with `mumble(x)`

```
public abstract class RMWRegister {
  private int value;
  public int synchronized getAndMumble() {
    int prior  = value;
    value = mumble(value);
    return prior;
  }
}
```

Read-Modify-Write Objects

Method call

- Returns previous value x
- Replaces x with mumble(x)

```
public abstract class RMWRegister {
  private int value;
  public int synchronized getAndMumble() {
    int prior  = value;
    value = mumble(value);
    return prior;
  }
}
```

RMW everywhere!

- Most synchronization instructions are RMW methods
- The rest can be trivially transformed into RMW methods

Read-Modify-Write: read

```
public abstract class RMWRegister {
  private int value;
  public int synchronized read() {
    int prior  = value;
    value = value; // mumble == identity
    return prior;
  }
}
```

Read-Modify-Write: getAndSet

```
public abstract class RMWRegister {
  private int value;
  public int synchronized getAndSet(int v) {
    int prior  = value;
    value = v; // mumble(x) = v, constant
    return prior;
  }
}
```

Read-Modify-Write: getAndIncrement

```
public abstract class RMWRegister {
  private int value;
  public int synchronized getAndIncrement() {
    int prior  = value;
    value = value + 1; // mumble(x) = x + 1
    return prior;
  }
}
```

Read-Modify-Write: getAndAdd

```
public abstract class RMWRegister {
  private int value;
  public int synchronized getAndAdd(int a) {
    int prior  = value;
    value = value + a; // mumble(x) = x + a
    return prior;
  }
}
```

Read-Modify-Write: compareAndSet

```
public abstract class RMWRegister {
  private int value;
  public boolean synchronized compareAndSet(int expected, int update) {
    int prior = value;        // load witness value
    if (value == expected) { // if current value equals to expected
      value = update;         // then replace it with update
      return true;            // and return with success
    }
    return false;             // else return with failure, change nothing
  }
}
```

Read-Modify-Write: compareAndExchange

```
public abstract class RMWRegister {
  private int value;
  public int synchronized compareAndExchange(int expected, int update) {
    int prior = value;        // load witness value
    if (value == expected) { // if current value equals to expected
      value = update;         // then replace it with update
    }
    return prior;             // return witness value
  }
}
```

Read-Modify-Write: now you know it

AtomicInteger[1]:

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/AtomicInteger.html

Read-Modify-Write: now you know it

AtomicInteger[1]:

- `get()`, `getAndSet(int)`

1 https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/AtomicInteger.html

Read-Modify-Write: now you know it

AtomicInteger[1]:

- `get()`, `getAndSet(int)`
- `incrementAndGet()`, `addAndGet(int)`

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/AtomicInteger.html

Read-Modify-Write: now you know it

AtomicInteger[1]:

- `get()`, `getAndSet(int)`
- `incrementAndGet()`, `addAndGet(int)`
- `compareAndSet(int,int)`, `compareAndExchange(int,int)`

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/AtomicInteger.html

Read-Modify-Write: now you know it

AtomicInteger[1]:

- `get()`, `getAndSet(int)`
- `incrementAndGet()`, `addAndGet(int)`
- `compareAndSet(int,int)`, `compareAndExchange(int,int)`
- `updateAndGet(java.util.function.IntUnaryOperator)`
- `accumulateAndGet(int,java.util.function.IntBinaryOperator)`

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/AtomicInteger.html

Read-Modify-Write: now you know it

AtomicInteger[1]:

- `get()`, `getAndSet(int)`
- `incrementAndGet()`, `addAndGet(int)`
- `compareAndSet(int,int)`, `compareAndExchange(int,int)`
- `updateAndGet(java.util.function.IntUnaryOperator)`
- `accumulateAndGet(int,java.util.function.IntBinaryOperator)`

Atomic*[2]

- AtomicBoolean
- AtomicInteger, AtomicIntegerArray
- AtomicLong, AtomicLongArray
- AtomicReference, AtomicReferenceArray

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/AtomicInteger.html

[2] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/package-summary.html

Read-Modify-Write: now you know it

AtomicInteger[1]:

- `get()`, `getAndSet(int)`
- `incrementAndGet()`, `addAndGet(int)`
- `compareAndSet(int,int)`, `compareAndExchange(int,int)`
- `updateAndGet(java.util.function.IntUnaryOperator)`
- `accumulateAndGet(int,java.util.function.IntBinaryOperator)`

Atomic*[2]

- AtomicBoolean
- AtomicInteger, AtomicIntegerArray
- AtomicLong, AtomicLongArray
- AtomicReference, AtomicReferenceArray
- AtomicMarkableReference, AtomicStampedReference

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/AtomicInteger.html

[2] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/package-summary.html

Atomic RMW: what's the point?

General form of atomic RMW:

```
public int synchronized getAndMumble() {
  int prior  = value;
  value = mumble(value);
  return prior;
}
```

Atomic RMW: what's the point?

General form of atomic RMW:

```
public int synchronized getAndMumble() {
  int prior  = value;
  value = mumble(value);
  return prior;
}
```

Interesting if and only if it is a non-blocking wait-free operation

Atomic RMW: what's the point?

General form of atomic RMW:

```
public int synchronized getAndMumble() {
  int prior  = value;
  value = mumble(value);
  return prior;
}
```

Interesting if and only if it is a non-blocking wait-free operation

- Finishes in constant time
- Even in case of contention
- Independently of thread scheduling

Atomic RMW: what's the point?

General form of atomic RMW:

```
public int synchronized getAndMumble() {
  int prior  = value;
  value = mumble(value);
  return prior;
}
```

Interesting if and only if it is a non-blocking wait-free operation

- Finishes in constant time
- Even in case of contention
- Independently of thread scheduling
- Looks like a "mutex on steroids"

Atomic RMW: what's the point?

General form of atomic RMW:

```
public int synchronized getAndMumble() {
  int prior  = value;
  value = mumble(value);
  return prior;
}
```

Interesting if and only if it is a non-blocking wait-free operation

- Finishes in constant time
- Even in case of contention
- Independently of thread scheduling
- Looks like a "mutex on steroids"
- Supported on hardware level

Atomic RMW: what's the point?

```
public int synchronized getAndMumble() {
  int prior  = value;
  value = mumble(value);
  return prior;
}
```

Non-blocking wait-free operation

- Finishes in constant time
- Even in case of contention
- Independently of thread scheduling

Atomic RMW: what's the point?

```
public int synchronized getAndMumble() {
  int prior  = value;
  value = mumble(value);
  return prior;
}
```

Non-blocking wait-free operation

- Finishes in constant time
- Even in case of contention
- Independently of thread scheduling

Theorem

Any non-trivial RMW object has consensus number at least 2

Atomic RMW: what's the point?

```
public int synchronized getAndMumble() {
  int prior  = value;
  value = mumble(value);
  return prior;
}
```

Non-blocking wait-free operation

- Finishes in constant time
- Even in case of contention
- Independently of thread scheduling

Theorem

Any non-trivial RMW object has consensus number at least 2

- No wait-free implementation of RMW registers from atomic registers
- Hardware RMW instructions not just a convenience

Foundations: here we go again

Atomic registers

- From safe SRSW Boolean register

Foundations: here we go again

Atomic registers

- From safe SRSW Boolean register up to atomic MRMW Integer register

Foundations: here we go again

Atomic registers

- From safe SRSW Boolean register up to atomic MRMW Integer register and atomic snapshot of N registers

Foundations: here we go again

Atomic registers

- From safe SRSW Boolean register up to atomic MRMW Integer register and atomic snapshot of N registers

This journey required a lot of complicated wait-free constructions!

Foundations: here we go again

Atomic registers

- From safe SRSW Boolean register up to atomic MRMW Integer register and atomic snapshot of N registers

This journey required a lot of complicated wait-free constructions! And in the very end ...

Foundations: here we go again

Atomic registers

- From safe SRSW Boolean register up to atomic MRMW Integer register and atomic snapshot of N registers

This journey required a lot of complicated wait-free constructions! And in the very end ...

Theorem

Atomic read/write registers have consensus number 1

Foundations: here we go again

Atomic registers

- From safe SRSW Boolean register up to atomic MRMW Integer register and atomic snapshot of N registers

This journey required a lot of complicated wait-free constructions! And in the very end ...

Theorem

Atomic read/write registers have consensus number 1

Atomic RMW registers

- Generalization of any "local" synchronized operation

Foundations: here we go again

Atomic registers

- From safe SRSW Boolean register up to atomic MRMW Integer register and atomic snapshot of N registers

This journey required a lot of complicated wait-free constructions! And in the very end ...

Theorem

Atomic read/write registers have consensus number 1

Atomic RMW registers

- Generalization of any "local" synchronized operation

Could implement a lot of even more complicated wait-free algorithms!

Foundations: here we go again

Atomic registers

- From safe SRSW Boolean register up to atomic MRMW Integer register and atomic snapshot of N registers

This journey required a lot of complicated wait-free constructions! And in the very end ...

Theorem

Atomic read/write registers have consensus number 1

Atomic RMW registers

- Generalization of any "local" synchronized operation

Could implement a lot of even more complicated wait-free algorithms! And in the very end ...

Foundations: here we go again

Atomic registers

- From safe SRSW Boolean register up to atomic MRMW Integer register and atomic snapshot of N registers

This journey required a lot of complicated wait-free constructions! And in the very end ...

Theorem

Atomic read/write registers have consensus number 1

Atomic RMW registers

- Generalization of any "local" synchronized operation

Could implement a lot of even more complicated wait-free algorithms! And in the very end ...

Theorem

Any non-trivial RMW object has consensus number at least 2

Foundations: here we go again

Atomic registers

- From safe SRSW Boolean register up to atomic MRMW Integer register and atomic snapshot of N registers

This journey required a lot of complicated wait-free constructions! And in the very end ...

Theorem

Atomic read/write registers have consensus number 1

Atomic RMW registers

- Generalization of any "local" synchronized operation

Could implement a lot of even more complicated wait-free algorithms! And in the very end ...

Theorem

Any non-trivial RMW object has consensus number at least 2

When will we stop?

Consensus: final point

Theorem

Atomic read/write registers have consensus number 1

Theorem

Any non-trivial RMW object has consensus number at least 2

Consensus: final point

Theorem

Atomic read/write registers have consensus number 1

Theorem

Any non-trivial RMW object has consensus number at least 2

Theorem

`getAndSet`, `getAndIncrement`, `getAndAdd` *have consensus number exactly 2*

Consensus: final point

Theorem

Atomic read/write registers have consensus number 1

Theorem

Any non-trivial RMW object has consensus number at least 2

Theorem

$getAndSet$, $getAndIncrement$, $getAndAdd$ *have consensus number exactly 2*

Theorem

$compareAndSet$ *has* ∞ *consensus number*

Consensus: final point

Theorem

$\mathit{compareAndSet}$ has ∞ consensus number

Consensus: final point

Theorem

compareAndSet has ∞ consensus number

```
public class RMWConsensus {
  private AtomicInteger r = new AtomicInteger(-1); // init with -1
  private int[] proposed = new int[N]; // values proposed by threads
  public int decide(int value) {
    int i = ThreadID.get();    // I am thread i
    proposed[i] = value;       // I propose value
    r.compareAndSet(-1, i);    // try to set my id as `winner`
    return proposed[r.get()]; // return value proposed by winner
  }
}
```

The consensus hierarchy

- 1. Read/Write Registers, Snapshots ...
- 2. `getAndSet`, `getAndIncrement` ...
- ...
- ∞. `compareAndSet` ...

The consensus hierarchy

- **1**. Read/Write Registers, Snapshots ...
- **2**. `getAndSet`, `getAndIncrement` ...
- ...
- ∞. `compareAndSet` ...

Multiprocessor concurrent environment is **very** friendly to wait-free algorithms. Hardware engineers know how to efficiently implement `compareAndSet`.

The consensus hierarchy

- 1. Read/Write Registers, Snapshots ...
- 2. `getAndSet`, `getAndIncrement` ...
- ...
- ∞. `compareAndSet` ...

Multiprocessor concurrent environment is **very** friendly to wait-free algorithms. Hardware engineers know how to efficiently implement `compareAndSet`. We will discuss it in the very next Lecture 9 (cache coherency).

The consensus hierarchy

- **1**. Read/Write Registers, Snapshots ...
- **2**. `getAndSet`, `getAndIncrement` ...
- ...
- ∞. `compareAndSet` ...

Multiprocessor concurrent environment is **very** friendly to wait-free algorithms. Hardware engineers know how to efficiently implement `compareAndSet`. We will discuss it in the very next Lecture 9 (cache coherency).

Some other environments – distributed systems, supercomputers, telegraph, pigeon posts etc. – could be **different**. Now you know how to do the analysis of "computability" in such environments.

The consensus hierarchy

- **1**. Read/Write Registers, Snapshots ...
- **2**. `getAndSet`, `getAndIncrement` ...
- ...
- ∞. `compareAndSet` ...

Multiprocessor concurrent environment is **very** friendly to wait-free algorithms. Hardware engineers know how to efficiently implement `compareAndSet`. We will discuss it in the very next Lecture 9 (cache coherency).

Some other environments – distributed systems, supercomputers, telegraph, pigeon posts etc. – could be **different**. Now you know how to do the analysis of "computability" in such environments.

Challenges of the new era:

- Highly distributed systems (internet nodes, geographically distributed data centers) with non-guaranteed message delivery (UDP)
- Heterogeneous computing (CPU, GPU, FPGA, GPGPU ...)

Lecture plan

SynchronizedCounter

```java
public class SynchronizedCounter {
    private long cnt;
    public SynchronizedCounter(long initial) { this.cnt = initial; }
    // assume delta > 0
    public synchronized void increment(int delta) { cnt += delta; }
    public synchronized long get() { return cnt; }
}
```

- Wait-free, Lock-free, Obstruction-free?
- Starvation-free, Deadlock-free?

FairLockedCounter

```
public class FairLockedCounter {
    private long cnt;
    private final Lock lock = new ReentrantLock(/* fair = */ true);
    public FairLockedCounter(long initial) { this.cnt = initial; }
    public void increment(int delta) {
      lock.lock() try { cnt += delta; } finally { lock.unlock(); }
    }
    public long get() {
      lock.lock(); try { return cnt; } finally { lock.unlock(); }
    }
}
```

- Wait-free, Lock-free, Obstruction-free?
- Starvation-free, Deadlock-free?

AtomicLoopCounter

```
public class AtomicLoopCounter {
    private long cnt;
    private final AtomicBoolean status = new AtomicBoolean(false);
    public AtomicLoopCounter(long initial) { this.cnt = initial; }
    public void increment(int delta) {
      while (status.getAndSet(true) == true) {}
      try { cnt += delta; } finally { status.set(false); }
    }
    public long get() {
      while (status.getAndSet(true) == true) {}
      try { return cnt; } finally { status.set(false); }
    }}
```

- Wait-free, Lock-free, Obstruction-free?
- Starvation-free, Deadlock-free?

AtomicIncCounter

```
public class AtomicIncCounter {
    private final AtomicLong cnt = new AtomicLong(0);
    public AtomicIncCounter(long initial) { cnt.set(initial); }
    public void increment(int delta) {
      cnt.getAndAdd(delta);
    }
    public long get() {
      return cnt.get();
    }}
```

- Wait-free, Lock-free, Obstruction-free?
- Starvation-free, Deadlock-free?

AtomicCASCounter

```
public class AtomicCASCounter {
    private final AtomicLong cnt = new AtomicLong(0);
    public AtomicCASCounter(long initial) { cnt.set(initial); }
    public void increment(int delta) {
      long expected = cnt.get();
      while (true) {
        long witness = cnt.compareAndExchange(expected, expected + delta);
        if (witness == expected) { return; } else { expected = witness; }
      }
    }
    public long get() { return cnt.get(); }}
```

- Wait-free, Lock-free, Obstruction-free?
- Starvation-free, Deadlock-free?

ReplicatedCounter

```java
static final ThreadLocal<Long> privateCounter = new ThreadLocal<Long>();
public void increment(int delta) {
  privateCounter.set(privateCounter.get() + delta);
}
public long get() {
  long result = 0;
  for (Thread t: ALL_THREADS) result += t.privateCounter.get(); // PSEUDOCODE
  return result;
}
```

- Wait-free, Lock-free, Obstruction-free?
- Starvation-free, Deadlock-free?

ReplicatedCounter

```
static final ThreadLocal<Long> privateCounter = new ThreadLocal<Long>();
public void increment(int delta) {
  privateCounter.set(privateCounter.get() + delta);
}
public long get() {
  long result = 0;
  for (Thread t: ALL_THREADS) result += t.privateCounter.get(); // PSEUDOCODE
  return result;
}
```

- Wait-free, Lock-free, Obstruction-free?
- Starvation-free, Deadlock-free?
- Linearizable?

ReplicatedCounter

```
static final ThreadLocal<Long> privateCounter = new ThreadLocal<Long>();
public void increment(int delta) {
  privateCounter.set(privateCounter.get() + delta);
}
public long get() {
  long result = 0;
  for (Thread t: ALL_THREADS) result += t.privateCounter.get(); // PSEUDOCODE
  return result;
}
```

- Wait-free, Lock-free, Obstruction-free?
- Starvation-free, Deadlock-free?
- Linearizable? What if `ThreadLocal` are Atomic MRMW registers?

Concurrent counters: summary

- Progress conditions describe the whole method/algorithm/object

Concurrent counters: summary

- Progress conditions describe the whole method/algorithm/object
- Fixed number lines of code – does not mean wait-free

Concurrent counters: summary

- Progress conditions describe the whole method/algorithm/object
- Fixed number lines of code – does not mean wait-free
- No explicit locks – does not mean lock-free

Concurrent counters: summary

- Progress conditions describe the whole method/algorithm/object
- Fixed number lines of code – does not mean wait-free
- No explicit locks – does not mean lock-free
- `synchronized` does not guarantee starvation freedom

Concurrent counters: summary

- Progress conditions describe the whole method/algorithm/object
- Fixed number lines of code – does not mean wait-free
- No explicit locks – does not mean lock-free
- `synchronized` does not guarantee starvation freedom
- `ReentrantLock` could be configured as fair (starvation-free)

Concurrent counters: summary

- Progress conditions describe the whole method/algorithm/object
- Fixed number lines of code – does not mean wait-free
- No explicit locks – does not mean lock-free
- `synchronized` does not guarantee starvation freedom
- `ReentrantLock` could be configured as fair (starvation-free)
- `CAS` is not always better than `getAndAdd`

Concurrent counters: summary

- Progress conditions describe the whole method/algorithm/object
- Fixed number lines of code – does not mean wait-free
- No explicit locks – does not mean lock-free
- `synchronized` does not guarantee starvation freedom
- `ReentrantLock` could be configured as fair (starvation-free)
- `CAS` is not always better than `getAndAdd`
- Wait-free and smart does not mean correct (consistent or linearizable)

Lecture plan

Test-And-Set: non-reentrant boolean spin lock

```
class TASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  void lock() {
    while (true) {
      boolean before = state.getAndSet(LOCKED); // <-----------|
      if (before == UNLOCKED) { return; } // I win            |
      else {}                             // I lose, repeat ---+
  }}
  void unlock() { state.set(UNLOCKED); }
}
```

Test-And-Set: non-reentrant boolean spin lock

```
class TASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  void lock() {
    while (true) {
      boolean before = state.getAndSet(LOCKED); // <-----------/
      if (before == UNLOCKED) { return; } // I win            /
      else {}                             // I lose, repeat ---+
  }}
  void unlock() { state.set(UNLOCKED); }
}
```

- Pros: easy to implement, easy to prove correctness, ultra-fast ownership handoff
- Cons: burn CPU, contention on memory bus

Test-And-Set: reentrant spin lock

Homework, mail

Task 8.1 Replace `AtomicBoolean` with `AtomicReference(Thread.cuurentThread)` and make TAS lock reentrant. Provide at least 3 tests written in JCStress.

Test-And-Set: non-reentrant boolean spin lock

```
class TASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  void lock() {
    while (true) {
      boolean before = state.getAndSet(LOCKED); // <-----------/
      if (before == UNLOCKED) { return; } // I win            /
      else {}                             // I lose, repeat ---+
  }}
  void unlock() { state.set(UNLOCKED); }
}
```

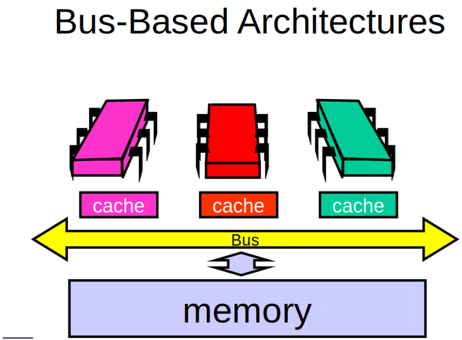
- Pros: easy to implement, easy to prove correctness, ultra-fast ownership handoff.
- Cons: burn CPU, **contention on memory bus**.

Test-And-Set: non-reentrant boolean spin lock

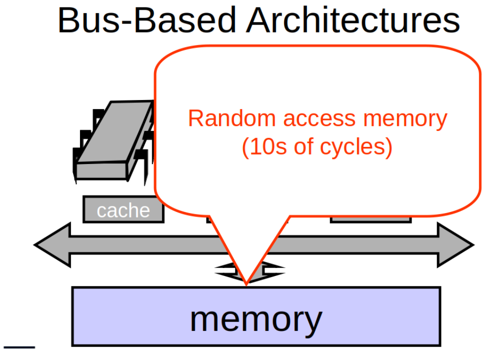
```
class TASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  void lock() {
    while (true) {
      boolean before = state.getAndSet(LOCKED); // <-----------|
      if (before == UNLOCKED) { return; } // I win             |
      else {}                             // I lose, repeat ---+
  }}
  void unlock() { state.set(UNLOCKED); }
}
```

- Pros: easy to implement, easy to prove correctness, ultra-fast ownership handoff.
- Cons: burn CPU, **contention on memory bus**. **WHAT?**

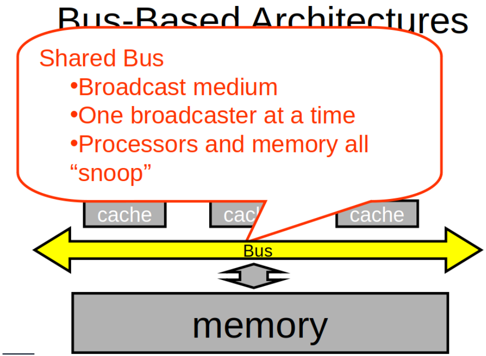
Memory bus



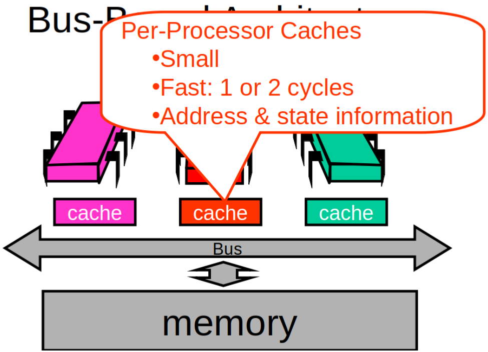
Memory bus


Bus-Based Architectures
Random access memory
(10s of cycles)
cache
memory

Memory bus


Bus-Based Architectures
Shared Bus
•Broadcast medium
•One broadcaster at a time
•Processors and memory all "snoop"
cache
cac
cache
Bus
memory

Memory bus


Bus-B
Per-Processor Caches
•Small
•Fast: 1 or 2 cycles
•Address & state information
cache
cache
cache
Bus
memory

Cache

- **Cache hit** – data found in local cache of current processor
- **Cache miss** – up-to-date data must be found elsewhere

Cache

- **Cache hit** – data found in local cache of current processor
- **Cache miss** – up-to-date data must be found elsewhere

There are many questions related to cache-based design:

- How many level of caches should be used?
- What is optimal size for each level?
- How to properly synchronize caches?
- How to move out stale data from cache?
- ...

Cache

- **Cache hit** – data found in local cache of current processor
- **Cache miss** – up-to-date data must be found elsewhere

There are many questions related to cache-based design:

- How many level of caches should be used?
- What is optimal size for each level?
- How to properly synchronize caches?
- How to move out stale data from cache?
- ...

We will go deeper in next Lecture.

Cache

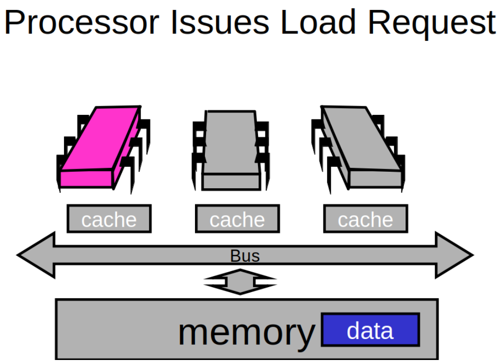
- **Cache hit** – data found in local cache of current processor
- **Cache miss** – up-to-date data must be found elsewhere

There are many questions related to cache-based design:

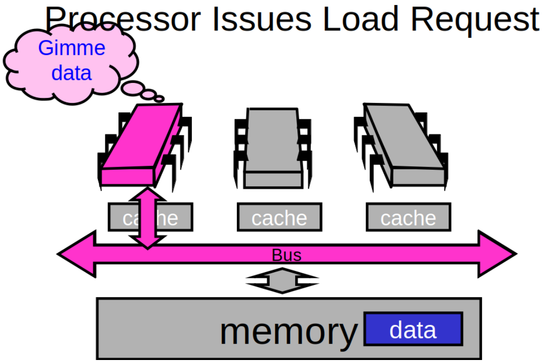
- How many level of caches should be used?
- What is optimal size for each level?
- How to properly synchronize caches?
- How to move out stale data from cache?
- ...

We will go deeper in next Lecture.
Today we will try to illustrate that even basic low-level knowledge helps to speed-up concurrent algorithms.

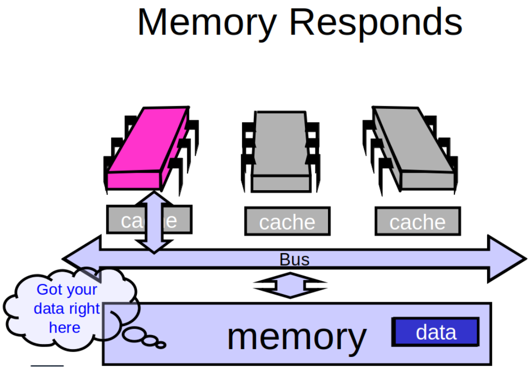
Read data


Processor Issues Load Request
cache
cache
cache
Bus
memory
data

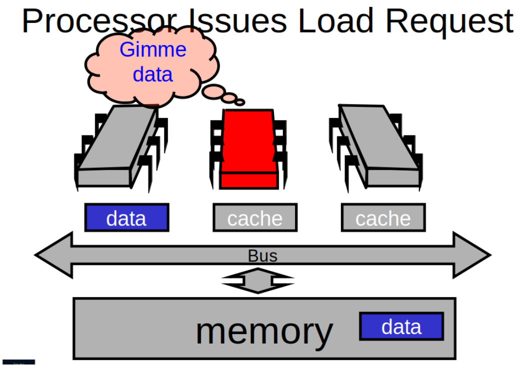
Read data


Processor Issues Load Request
Gimme
data
cache
cache
Bus
memory
data

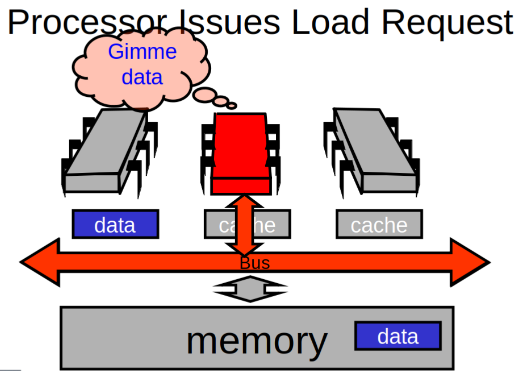
Read data


Memory Responds
ca e
cache
cache
Bus
Got your data right here
memory
data

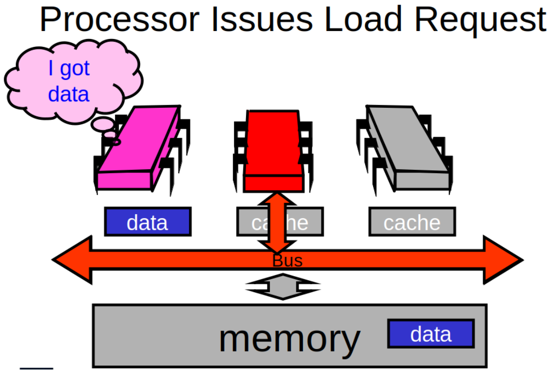
Read data


Processor Issues Load Request
Gimme data
data
cache
cache
Bus
memory
data

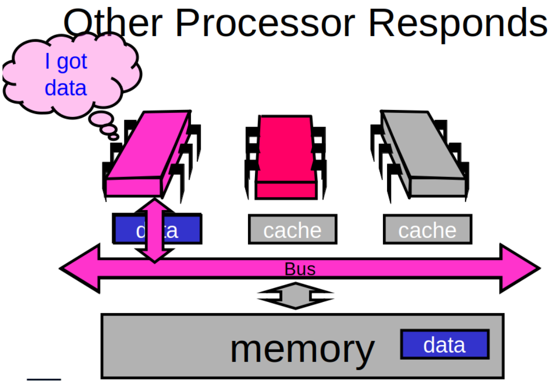
Read data


Processor Issues Load Request
Gimme data
data
ca he
cache
Bus
memory
data

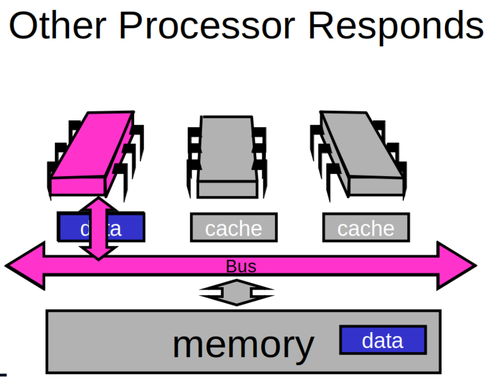
Read data


Processor Issues Load Request
I got data
data
ca he
cache
Bus
memory
data

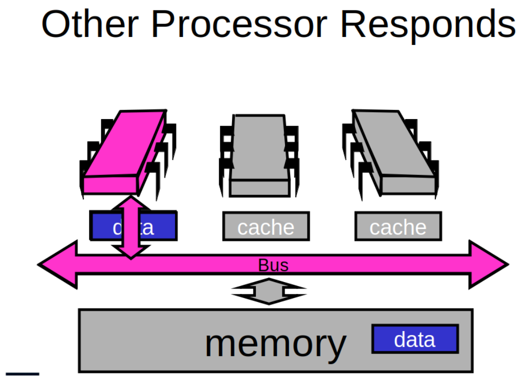
Read data


Other Processor Responds
I got
data
cache
cache
Bus
memory
data

Read data



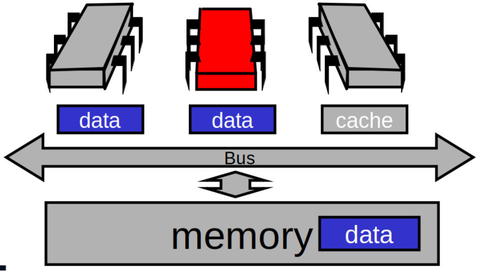
Read data



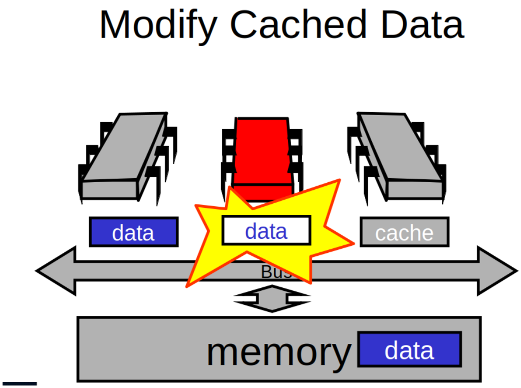
And now we have several copies of the same data in different places

Modify data

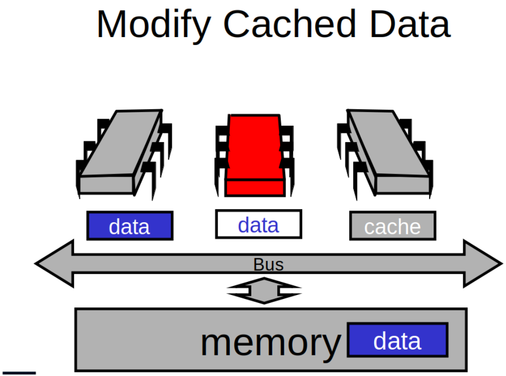
Modify Cached Data



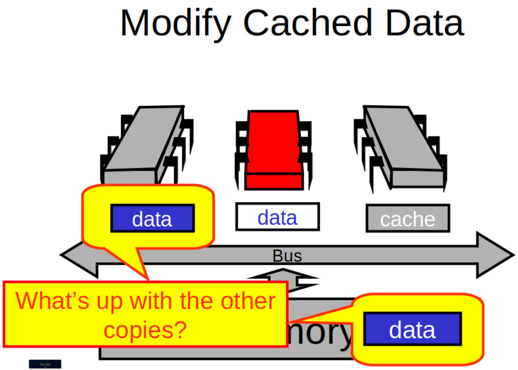
Modify data



Modify data



Modify data


Modify Cached Data
data
data
cache
Bus
What’s up with the other copies?
mory
data

Cache coherence

We have lots of copies of data

- Original copy in memory
- Cached copies at processors

Cache coherence

We have lots of copies of data

- Original copy in memory
- Cached copies at processors

Some processor modifies its own copy

- What do we do with the others?
- How to avoid confusion?

Cache coherence

We have lots of copies of data

- Original copy in memory
- Cached copies at processors

Some processor modifies its own copy

- What do we do with the others?
- How to avoid confusion?

Cache coherence protocol – algorithm that handles such problems.

Cache coherence

We have lots of copies of data

- Original copy in memory
- Cached copies at processors

Some processor modifies its own copy

- What do we do with the others?
- How to avoid confusion?

Cache coherence protocol – algorithm that handles such problems.

Low-level concurrent algorithm that is implemented in hardware

Cache coherence

We have lots of copies of data

- Original copy in memory
- Cached copies at processors

Some processor modifies its own copy

- What do we do with the others?
- How to avoid confusion?

Cache coherence protocol – algorithm that handles such problems.

Low-level concurrent algorithm that is implemented in hardware

- No ReentrantLocks, Semaphores, ReadWriteLocks
- Critical for speed of the whole system (hope it is non-blocking)

Cache coherence

We have lots of copies of data

- Original copy in memory
- Cached copies at processors

Some processor modifies its own copy

- What do we do with the others?
- How to avoid confusion?

Cache coherence protocol – algorithm that handles such problems.

Low-level concurrent algorithm that is implemented in hardware

- No ReentrantLocks, Semaphores, ReadWriteLocks
- Critical for speed of the whole system (hope it is non-blocking)
- Provides some consistency guarantees on "our view of" memory cells

Cache coherence

We have lots of copies of data

- Original copy in memory
- Cached copies at processors

Some processor modifies its own copy

- What do we do with the others?
- How to avoid confusion?

Cache coherence protocol – algorithm that handles such problems.

Low-level concurrent algorithm that is implemented in hardware

- No ReentrantLocks, Semaphores, ReadWriteLocks
- Critical for speed of the whole system (hope it is non-blocking)
- Provides some consistency guarantees on "our view of" memory cells

What a joy that you have already encountered such tasks before!

Cache coherence

We have lots of copies of data

- Original copy in memory
- Cached copies at processors

Some processor modifies its own copy

- What do we do with the others?
- How to avoid confusion?

Cache coherence protocol – algorithm that handles such problems.

Low-level concurrent algorithm that is implemented in hardware

- No ReentrantLocks, Semaphores, ReadWriteLocks
- Critical for speed of the whole system (hope it is non-blocking)
- Provides some consistency guarantees on "our view of" memory cells

What a joy that you have already encountered such tasks before!

Next Lecture will be s-o-o-o interesting.

Cache coherence

We have lots of copies of data

- Original copy in memory
- Cached copies at processors

Some processor modifies its own copy

- What do we do with the others?
- How to avoid confusion?

Cache coherence protocol – algorithm that handles such problems.

Low-level concurrent algorithm that is implemented in hardware

- No ReentrantLocks, Semaphores, ReadWriteLocks
- Critical for speed of the whole system (hope it is non-blocking)
- Provides some consistency guarantees on "our view of" memory cells

What a joy that you have already encountered such tasks before!

Next Lecture will be s-o-o-o interesting.

Let's go back to the naive multiprocessor model.

Cache coherence: invalidation

We have lots of copies of data

- Original copy in memory
- Cached copies at processors

Some processor modifies its own copy

Cache coherence: invalidation

We have lots of copies of data

- Original copy in memory
- Cached copies at processors

Some processor modifies its own copy

- Broadcast a special message "invalidate data"

Cache coherence: invalidation

We have lots of copies of data

- Original copy in memory
- Cached copies at processors

Some processor modifies its own copy

- Broadcast a special message "invalidate data"
- Ensure other processors replied to this as "Yes, sir!"

Cache coherence: invalidation

We have lots of copies of data

- Original copy in memory
- Cached copies at processors

Some processor modifies its own copy

- Broadcast a special message "invalidate data"
- Ensure other processors replied to this as "Yes, sir!"
- Now current processor is the unique owner of data, others will consult with him

Cache coherence: invalidation

We have lots of copies of data

- Original copy in memory
- Cached copies at processors

Some processor modifies its own copy

- Broadcast a special message "invalidate data"
- Ensure other processors replied to this as "Yes, sir!"
- Now current processor is the unique owner of data, others will consult with him
- Modify data

Cache coherence: message passing

Messages

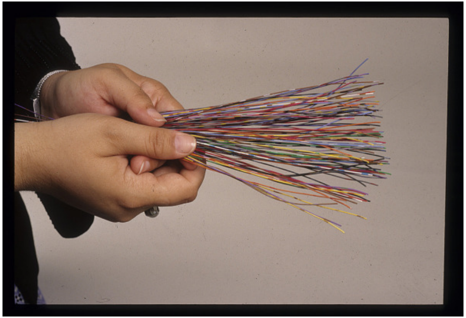
- Request for read, Respond with data
- Request for invalidate, Acknowledge invalidation
- Write-back modified data to main memory

Cache coherence: message passing

Messages

- Request for read, Respond with data
- Request for invalidate, Acknowledge invalidation
- Write-back modified data to main memory

Nanoseconds[3]:



[3] https://americanhistory.si.edu/collections/object/nmah_692464

Memory bus, cache coherence, multiprocessor: takeaways

Multiprocessor systems have complicated memory subsystem

Memory bus, cache coherence, multiprocessor: takeaways

Multiprocessor systems have complicated memory subsystem

- Set of caches

Memory bus, cache coherence, multiprocessor: takeaways

Multiprocessor systems have complicated memory subsystem

- Set of caches
- synchronized with each other on low hardware level

Memory bus, cache coherence, multiprocessor: takeaways

Multiprocessor systems have complicated memory subsystem

- Set of caches
- synchronized with each other on low hardware level
- using replication pattern (copy to read, invalidate for write)

Memory bus, cache coherence, multiprocessor: takeaways

Multiprocessor systems have complicated memory subsystem

- Set of caches
- synchronized with each other on low hardware level
- using replication pattern (copy to read, invalidate for write)
- to guarantee cache coherence (consistency of memory cell view)

Memory bus, cache coherence, multiprocessor: takeaways

Multiprocessor systems have complicated memory subsystem

- Set of caches
- synchronized with each other on low hardware level
- using replication pattern (copy to read, invalidate for write)
- to guarantee cache coherence (consistency of memory cell view)

Simplified cost model:

- Read from private non-shared data is ultra-fast
- Write to private non-shared data is ultra-fast
- Read from private shared data is ultra-fast
- Write to shared data requires communication (invalidate and await)
- Read/write of non-cached location takes significant time

TASLock: reminder

```
class TASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  void lock() {
    while (true) {
      boolean before = state.getAndSet(LOCKED); // <------------/
      if (before == UNLOCKED) { return; } // I win             /
      else {}                             // I lose, repeat ---+
  }}
  void unlock() { state.set(UNLOCKED); }
}
```

- Pros: easy to implement, easy to prove correctness, ultra-fast ownership handoff.
- Cons: burn CPU, **contention on memory bus**.

Test-and-Test-and-Set: non-reentrant boolean spin lock

```
class TATASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  void lock() {
    while (true) {
      while (state.get() == LOCKED) {} // do not write if mutex is busy
      if (state.getAndSet(LOCKED) == UNLOCKED) { // try acquire
        return;
}}}}
```

Test-and-Test-and-Set: non-reentrant boolean spin lock

```
class TATASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  void lock() {
    while (true) {
      while (state.get() == LOCKED) {} // do not write if mutex is busy
      if (state.getAndSet(LOCKED) == UNLOCKED) { // try acquire
        return;
}}}}
```

Lock acquisition is OK, no excessive cache coherence bus traffic.

Test-and-Test-and-Set: non-reentrant boolean spin lock

```
class TATASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  void lock() {
    while (true) {
      while (state.get() == LOCKED) {} // do not write if mutex is busy
      if (state.getAndSet(LOCKED) == UNLOCKED) { // try acquire
        return;
}}}}
```

Lock acquisition is OK, no excessive cache coherence bus traffic.

What happens on release?

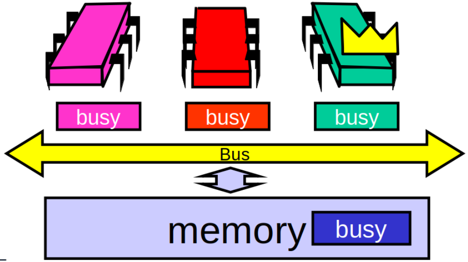
Test-and-Test-and-Set: non-reentrant boolean spin lock

```
class TATASlock {
  private static final boolean LOCKED = true, UNLOCKED = false;
  private final AtomicBoolean state = new AtomicBoolean(UNLOCKED);
  void lock() {
    while (true) {
      while (state.get() == LOCKED) {} // do not write if mutex is busy
      if (state.getAndSet(LOCKED) == UNLOCKED) { // try acquire
        return;
}}}}
```

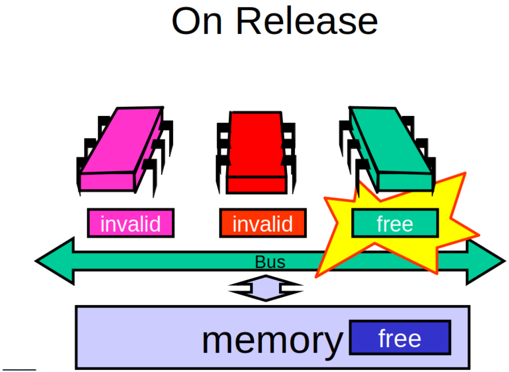
Lock acquisition is OK, no excessive cache coherence bus traffic.
What happens on release?
Invalidation storm.

Invalidation storm

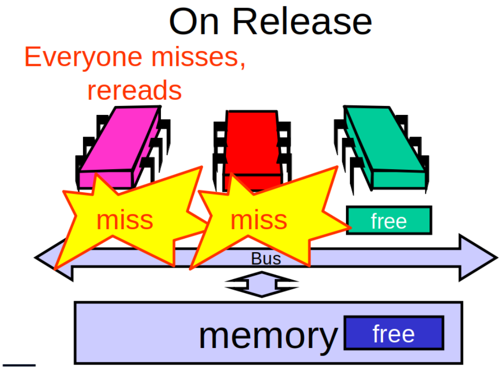
Local Spinning while Lock is Busy



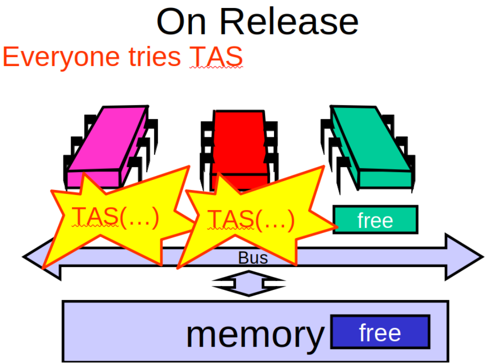
Invalidation storm



Invalidation storm



Invalidation storm



TATAS: invalidation storm

- Everyone misses

TATAS: invalidation storm

- Everyone misses
- Everyone does TAS

TATAS: invalidation storm

- Everyone misses
- Everyone does TAS
- Invalidates others' caches

TATAS: invalidation storm

- Everyone misses
- Everyone does TAS
- Invalidates others' caches
- Eventually *quiesces* after lock acquired

TATAS: invalidation storm

- Everyone misses
- Everyone does TAS
- Invalidates others' caches
- Eventually *quiesces* after lock acquired

How long does this take?

TATAS: invalidation storm

- Everyone misses
- Everyone does TAS
- Invalidates others' caches
- Eventually *quiesces* after lock acquired

How long does this take?

General pattern for TATAS:

TATAS: invalidation storm

- Everyone misses
- Everyone does TAS
- Invalidates others' caches
- Eventually *quiesces* after lock acquired

How long does this take?

General pattern for TATAS:

- Acquire lock

TATAS: invalidation storm

- Everyone misses
- Everyone does TAS
- Invalidates others' caches
- Eventually *quiesces* after lock acquired

How long does this take?
General pattern for TATAS:

- Acquire lock
- Pause without using bus

TATAS: invalidation storm

- Everyone misses
- Everyone does TAS
- Invalidates others' caches
- Eventually *quiesces* after lock acquired

How long does this take?
General pattern for TATAS:

- Acquire lock
- Pause without using bus
- Use bus heavily

TATAS: invalidation storm

- Everyone misses
- Everyone does TAS
- Invalidates others' caches
- Eventually *quiesces* after lock acquired

How long does this take?
General pattern for TATAS:

- Acquire lock
- Pause without using bus
- Use bus heavily
- If pause > quiescence time
  - critical section duration independent of number of threads

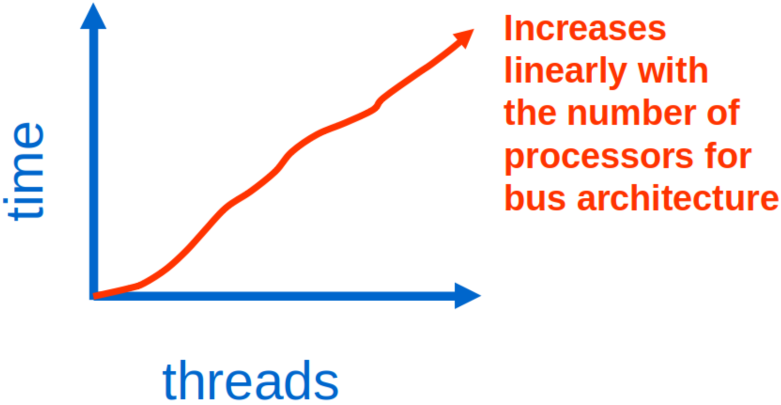
TATAS: invalidation storm

- Everyone misses
- Everyone does TAS
- Invalidates others' caches
- Eventually *quiesces* after lock acquired

How long does this take?
General pattern for TATAS:

- Acquire lock
- Pause without using bus
- Use bus heavily
- If pause $>$ quiescence time
  - critical section duration independent of number of threads
- If pause $<$ quiescence time
  - critical section duration slower with more threads

Quiescence Time



Spin locks and hardware, you know so far

- Easy to implement and prove correctness
- Burn CPU for better responsiveness
- Read before trying to acquire lock, writes to actually shared memory are expensive
- Suffer from invalidation storm in contended environment when lock is released

Spin locks and hardware, you know so far

- Easy to implement and prove correctness
- Burn CPU for better responsiveness
- Read before trying to acquire lock, writes to actually shared memory are expensive
- Suffer from invalidation storm in contended environment when lock is released

Problems arise when

Spin locks and hardware, you know so far

- Easy to implement and prove correctness
- Burn CPU for better responsiveness
- Read before trying to acquire lock, writes to actually shared memory are expensive
- Suffer from invalidation storm in contended environment when lock is released

Problems arise when

- Many threads compete for the single lock

Spin locks and hardware, you know so far

- Easy to implement and prove correctness
- Burn CPU for better responsiveness
- Read before trying to acquire lock, writes to actually shared memory are expensive
- Suffer from invalidation storm in contended environment when lock is released

Problems arise when

- Many threads compete for the single lock
- Critical section is quite small ($\approx$ quiescence time of h/w)

Spin locks and hardware, you know so far

- Easy to implement and prove correctness
- Burn CPU for better responsiveness
- Read before trying to acquire lock, writes to actually shared memory are expensive
- Suffer from invalidation storm in contended environment when lock is released

Problems arise when

- Many threads compete for the single lock
- Critical section is quite small ($\approx$ quiescence time of h/w)
- Maybe application is misdesigned?

Spin locks and hardware, you know so far

- Easy to implement and prove correctness
- Burn CPU for better responsiveness
- Read before trying to acquire lock, writes to actually shared memory are expensive
- Suffer from invalidation storm in contended environment when lock is released

Problems arise when

- Many threads compete for the single lock
- Critical section is quite small ($\approx$ quiescence time of h/w)
- Maybe application is misdesigned?

Any kind of "storm" (trap storm, invalidation storm, scheduler storm etc.) is a strong sign of misdesign.

Spin locks and hardware, you know so far

- Easy to implement and prove correctness
- Burn CPU for better responsiveness
- Read before trying to acquire lock, writes to actually shared memory are expensive
- Suffer from invalidation storm in contended environment when lock is released

Problems arise when

- Many threads compete for the single lock
- Critical section is quite small ($\approx$ quiescence time of h/w)
- Maybe application is misdesigned?

Any kind of "storm" (trap storm, invalidation storm, scheduler storm etc.) is a strong sign of misdesign. You could workaround it to some extent, but it will manifest itself later (more threads, faster h/w, better OS).

Spin locks and hardware, you know so far

- Easy to implement and prove correctness
- Burn CPU for better responsiveness
- Read before trying to acquire lock, writes to actually shared memory are expensive
- Suffer from invalidation storm in contended environment when lock is released

Problems arise when

- Many threads compete for the single lock
- Critical section is quite small ($\approx$ quiescence time of h/w)
- Maybe application is misdesigned?

Any kind of "storm" (trap storm, invalidation storm, scheduler storm etc.) is a strong sign of misdesign. You could workaround it to some extent, but it will manifest itself later (more threads, faster h/w, better OS).
Anyway, let's try to avoid the invalidation storm and (maybe) improve our skills with "designing it right"

Exponential Backoff

```
public void lock() {
  int delay = MIN_DELAY;
  while (true) {
    while (state.get() == LOCKED) {} // read-only polling
    if (lock.getAndSet(LOCKED) == UNLOCKED) return; // return on success
    sleep(random() % delay); // delay next write request on failure
    delay = Math.min(delay * 2, MAX_DELAY); // keep delay within bounds
  }
}
```

Exponential Backoff

```java
public void lock() {
  int delay = MIN_DELAY;
  while (true) {
    while (state.get() == LOCKED) {} // read-only polling
    if (lock.getAndSet(LOCKED) == UNLOCKED) return; // return on success
    sleep(random() % delay); // delay next write request on failure
    delay = Math.min(delay * 2, MAX_DELAY); // keep delay within bounds
  }
}
```

- Trade-off: CPU overuse vs. latency

Exponential Backoff

```
public void lock() {
  int delay = MIN_DELAY;
  while (true) {
    while (state.get() == LOCKED) {} // read-only polling
    if (lock.getAndSet(LOCKED) == UNLOCKED) return; // return on success
    sleep(random() % delay); // delay next write request on failure
    delay = Math.min(delay * 2, MAX_DELAY); // keep delay within bounds
  }
}
```

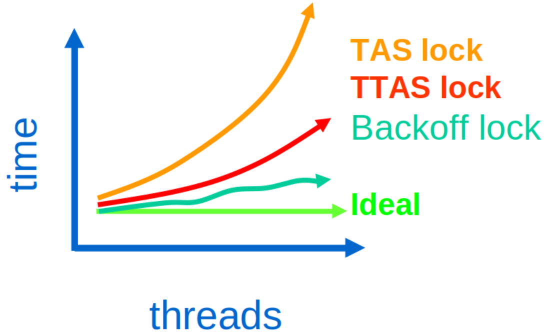
- Trade-off: CPU overuse vs. latency
- Still very naive, we will continue in Lecture 11 (queue locks)

Spin locks: summary

- Easy to implement and prove correctness
- Burn CPU vs. better responsiveness
- Actual performance depend on h/w (cache coherency) and application (contention)

Spin locks: summary

- Easy to implement and prove correctness
- Burn CPU vs. better responsiveness
- Actual performance depend on h/w (cache coherency) and application (contention)



Spin locks: summary

- Easy to implement and prove correctness
- Burn CPU vs. better responsiveness
- Actual performance depend on h/w (cache coherency) and application (contention)

Critically important:

Spin locks: summary

- Easy to implement and prove correctness
- Burn CPU vs. better responsiveness
- Actual performance depend on h/w (cache coherency) and application (contention)

Critically important:

- Spin locking is a design pattern, building block for "normal" locking algorithms

Spin locks: summary

- Easy to implement and prove correctness
- Burn CPU vs. better responsiveness
- Actual performance depend on h/w (cache coherency) and application (contention)

Critically important:

- Spin locking is a design pattern, building block for "normal" locking algorithms
- **NEVER** use plain spin locks anywhere in production system[4]

[4] https://www.realworldtech.com/forum/?threadid=189711&curpostid=189723

Spin locks: summary

- Easy to implement and prove correctness
- Burn CPU vs. better responsiveness
- Actual performance depend on h/w (cache coherency) and application (contention)

Critically important:

- Spin locking is a design pattern, building block for "normal" locking algorithms
- **NEVER** use plain spin locks anywhere in production system[4] unless you know what you are doing.

[4] https://www.realworldtech.com/forum/?threadid=189711&curpostid=189723

Spin locks: summary

- Easy to implement and prove correctness
- Burn CPU vs. better responsiveness
- Actual performance depend on h/w (cache coherency) and application (contention)

Critically important:

- Spin locking is a design pattern, building block for "normal" locking algorithms
- **NEVER** use plain spin locks anywhere in production system[4] unless you know what you are doing. After finishing this introductory course on concurrency, you definitely **don't**.

[4] https://www.realworldtech.com/forum/?threadid=189711&curpostid=189723

Spin locks: homework

Homework, mail

Task 8.2 Measure performance of `TASLock`, `TATASLock`, `ExpBackoffLock` *on*

- *1, 2, 3, 4, 5, 6, 7, 8, 16, 32 concurrent threads*
- *With small critical section (increment)*
- *With moderate critical section (compute n-th fibonacci number recursively)*

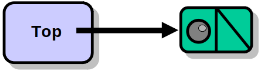
using JMH or JCStress, plot resulting figures (with error bars!), explain gathered data.

Lecture plan

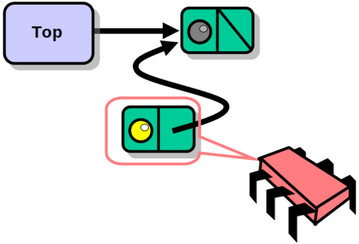
Lock-free stack

- `push(x)`
- `pop()`
- Last-in, First-out (LIFO) order

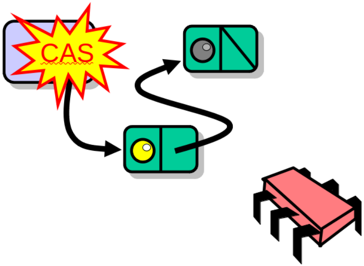
Lock-free stack: push



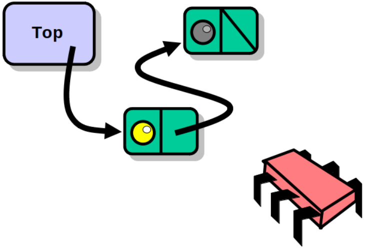
Lock-free stack: push



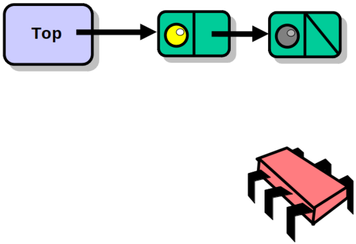
Lock-free stack: push



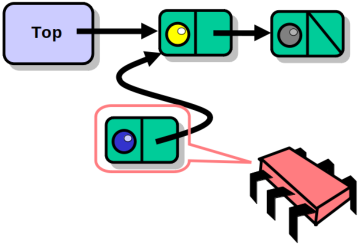
Lock-free stack: push



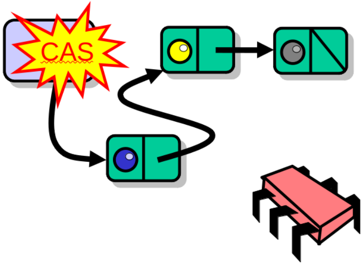
Lock-free stack: push



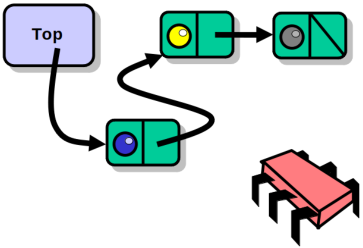
Lock-free stack: push



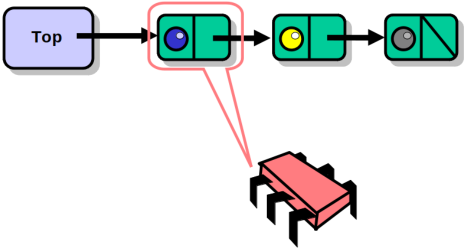
Lock-free stack: push



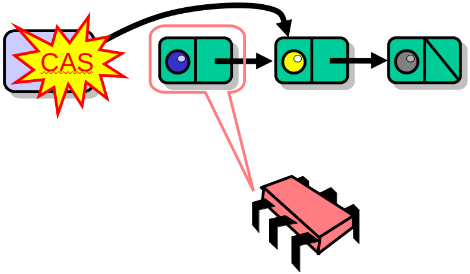
Lock-free stack: push



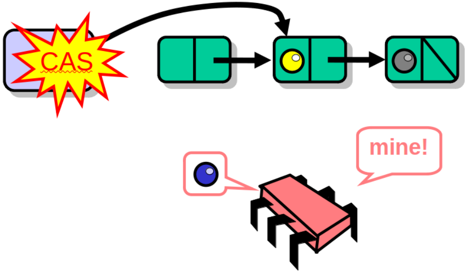
Lock-free stack: pop



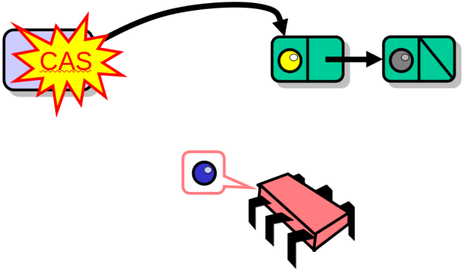
Lock-free stack: pop



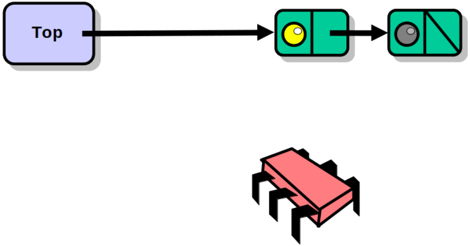
Lock-free stack: pop



Lock-free stack: pop



Lock-free stack: pop



Lock-free stack

```
public class LockFreeStack {
  private AtomicReference top = new AtomicReference(null);
  public boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return top.compareAndSet(oldTop, node);
  }
  public void push(T value) {
    Node node = new Node(value);
    while (true) {
      if (tryPush(node)) return;
      backoff.backoff();
  }}
```

Lock-free stack

```
public class LockFreeStack {
  private AtomicReference top = new AtomicReference(null);
  public boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return top.compareAndSet(oldTop, node);
  }
  public void push(T value) {
    Node node = new Node(value);
    while (true) {
      if (tryPush(node)) return;
      backoff.backoff();
  }}
```

- Wait-free, lock-free or obstruction-free?

Lock-free stack: summary

- Your first non-trivial and useful lock-free algorithm based on RMW
- Elegant
- Easy to reason about
- Easy to define linearization points

Lock-free stack: summary

- Your first non-trivial and useful lock-free algorithm based on RMW
- Elegant
- Easy to reason about
- Easy to define linearization points

But not perfect?

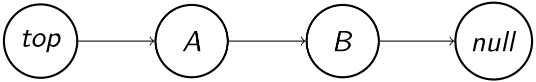
Node pooling

```
public class LockFreeStack {
  private AtomicReference top = new AtomicReference(null);
  public boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return top.compareAndSet(oldTop, node);
  }
  public void push(T value) {
    Node node = new Node(value); // Yikes!
    while (true) {
      if (tryPush(node)) return;
      backoff.backoff();
  }}
```

Node pooling

```
public class LockFreeStack {
  private AtomicReference top = new AtomicReference(null);
  public boolean tryPush(Node node){
    Node oldTop = top.get();
    node.next = oldTop;
    return top.compareAndSet(oldTop, node);
  }
  public void push(T value) {
    Node node = ThreadLocalNodePool.get(); // successful `pop`
    //                                        will populate this cache
    node.value = value;
    while (true) {
      if (tryPush(node)) return;
      backoff.backoff();
  }}
```

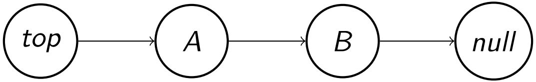
ABA problem



Thread 1

Thread 2

ABA problem

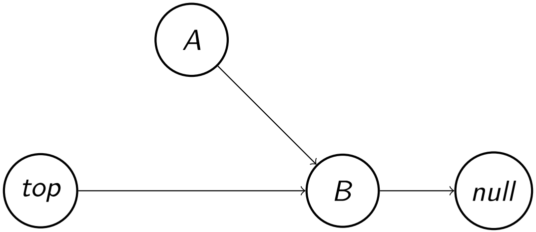


Thread 1

Start `pop()` (`top.CAS(A, B)`)

Thread 2

ABA problem



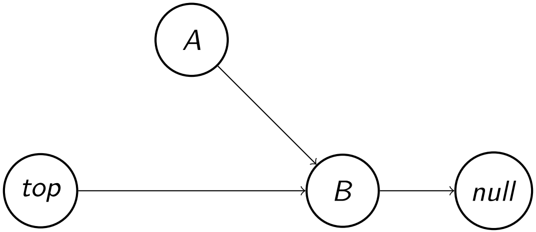
Thread 1

Start `pop()` (`top.CAS(A, B)`)

Thread 2

`A = pop()`

ABA problem



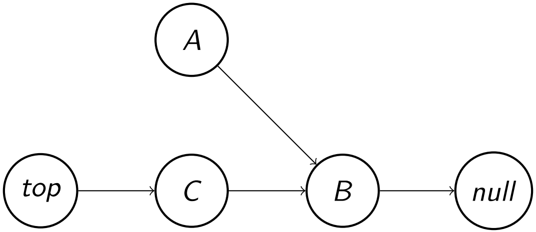
Thread 1

Start `pop()` (`top.CAS(A, B)`)

Thread 2

```
A = pop()
reclaim(A)
```

ABA problem



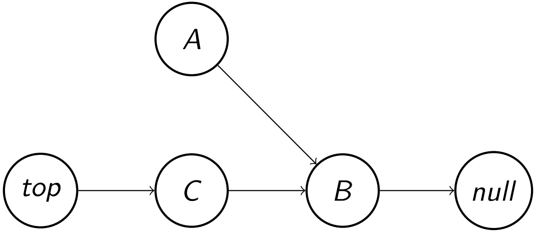
Thread 1

Start `pop()` (`top.CAS(A, B)`)

Thread 2

```
A = pop()
reclaim(A)
push(C)
```

ABA problem



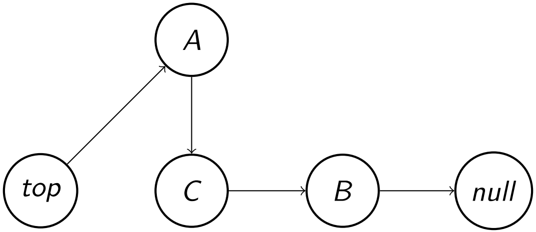
Thread 1

Start `pop()` (`top.CAS(A, B)`)

Thread 2

```
A = pop()
reclaim(A)
push(C)
push(A')
```

ABA problem



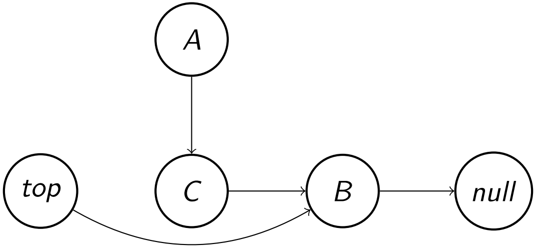
Thread 1

Start `pop()` (`top.CAS(A, B)`)

Thread 2

```
A = pop()
reclaim(A)
push(C)
push(A')
```

ABA problem



Thread 1	Thread 2
Start `pop()` (`top.CAS(A, B)`)	
Finish `pop()`	`A = pop()`
	`reclaim(A)`
	`push(C)`
	`push(A')`

ABA problem

ABA problem is always a headache in non-GC languages.
It still happens in managed languages, be aware.
Advanced algorithms you could or could not use:

- `AtomicStampedReference`[5]
- immortal memory
- RCU
- reference counting
- other variation of GC
- hazard pointers
- ...

[5] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/AtomicStampedReference.html

Summary

Read-Modify-Write atomic operations

- `getAndSet`, `getAndAdd` – consensus number **2**
- `compareAndSet`, `compareAndExchange` – consensus number ∞

Spin locks

- Test-and-set
- Test-and-test-and-set
- Exponential Backoff

Memory hierarchy

- Multi-level caching
- Cache coherence via replication and message passing
- Invalidation storm

Lock-free stack and ABA for pooling.

Summary: homework

Homework, mail

Task 8.1 Replace `AtomicBoolean` *with* `AtomicReference(Thread.cuurentThread)` *and make TAS lock reentrant. Provide at least 3 tests written in JCStress.*

Homework, mail

Task 8.2 Measure performance of `TASLock`, `TATASLock`, `ExpBackoffLock` *on*

- *1, 2, 3, 4, 5, 6, 7, 8, 16, 32 concurrent threads*
- *With small critical section (increment)*
- *With moderate critical section (compute n-th fibonacci number recursively)*

using JMH or JCStress, plot resulting figures (with error bars!), explain gathered data.