# Lecture 24: Progress and Locks

COSC 273: Parallel and Distributed Computing

Spring 2023

# Announcements

1. Homework 03 Draft Posted
   - due Friday, April 14th
2. Short quiz on Friday, April 7th
   - Given two implementations, which is faster?
   - Reason about parallelism/locality of reference

# Today

1. Progress
2. Lock Implementations

# UnboundedQueue Enqueue

```
public void enq (T value) {
    enqLock.lock();
    try {
        Node nd = new Node(value);
        tail.next = nd;
        tail = nd;
    } finally {
        enqLock.unlock();
    }
}
```

# LockFreeQueue Enqueue

```
public void enq(T item) {
    if (item == null) throw new NullPointerException();
    Node node = new Node(item);
    while (true) {
        Node last = tail.get();
        Node next = last.next.get();
        if (last == tail.get()) {
            if (next == null) {
                if (last.next.compareAndSet(next, node))
                    tail.compareAndSet(last, node); return;
            } else {
                tail.compareAndSet(last, next);}}}}
```

# UnboundedQueue Progress

Guarantee: **Starvation Freedom** (assuming lock is starvation-free)

eg. enq, deq

- if *all* pending method calls continue to take steps, then *every* pending method call completes in a finite number of steps
- this is **blocking progress**: if even one thread stops taking steps, then all other threads can be impeded

**Question.** When is this "good?"

→ we "know" that all threads scheduled fairly

# LockFreeQueue Progress

Guarantee: **Lock Freedom**

- if *some* pending method call makes progress, then *some* pending method call completes in a finite number of steps
- this is **nonblocking progress**: if some threads stall, others are still guaranteed to make progress

if scheduling is not known to be fair of if a thread could crash, non-blocking progress is preferable to blocking.

[∀ threads .. ] ⇒ [ ]

# Progress, 4 Ways

Blocking Progress:

- **deadlock freedom** if *all* threads take steps, *some* completes in finite time
- **starvation freedom** if *all* threads take steps, *all* complete in finite time

Nonblocking Progress:

- **lock freedom** if *some* threads take steps, *some* completes in finite time
- **wait freedom** *all* threads taking steps complete in finite time

[∃ thread] ⇒ [ ]

# What About Performance?

Demo: `concurrent-queues.zip`

# Lock Implementations

1. Peterson Lock ←
2. Test-and-set Lock
3. Test-and-test-and-set Lock
4. CLH Lock

# The Peterson Lock

```
class Peterson implements Lock {
    private boolean[] flag = new boolean[2];
    private int victim;
    public void lock () {
      int i = ThreadID.get(); int j = 1 - i;
      flag[i] = true;          // set my flag
      victim = i;              // set myself to be victim
      while (flag[j] && victim == i) {}; }
    public void unlock () { int i = ThreadID.get();
       flag[i] = false; }
}
```

Download: `peterson-lock.zip`

# A Challenge

Peterson lock assumes 2 threads, with IDs `0` and `1`

- How do we accomplish this?

# Make a Thread Subclass

We'll use this `thread` to increment a counter

```
public class PetersonThread extends Thread {
    private int id;
    private LockedCounter ctr;
    private int numIncrements;
    public PetersonThread (id, ctr, numIncrements) {
        super(); this.id = id; this.ctr = ctr;
        this.numIncrements = numIncrements; }
    public int getPetersonId() { return id; }
    @Override
    public void run () {
        for (int i=0; i<numIncrements; ++i) { ctr.increment(); }}
}
```

Next week: A better way

# Making a PetersonLock

```
class PetersonLock {
    private boolean[] flag = new boolean[2];
    private int victim;
    public void lock () {
        int i = ((PetersonThread)Thread.currentThread())
                 .getPetersonId();
        int j = 1 - i; flag[i] = true; victim = i;
        while (flag[j] && victim == i) {};
    }
    public void unlock () {...}
}
```

# And Now: A Locked Counter

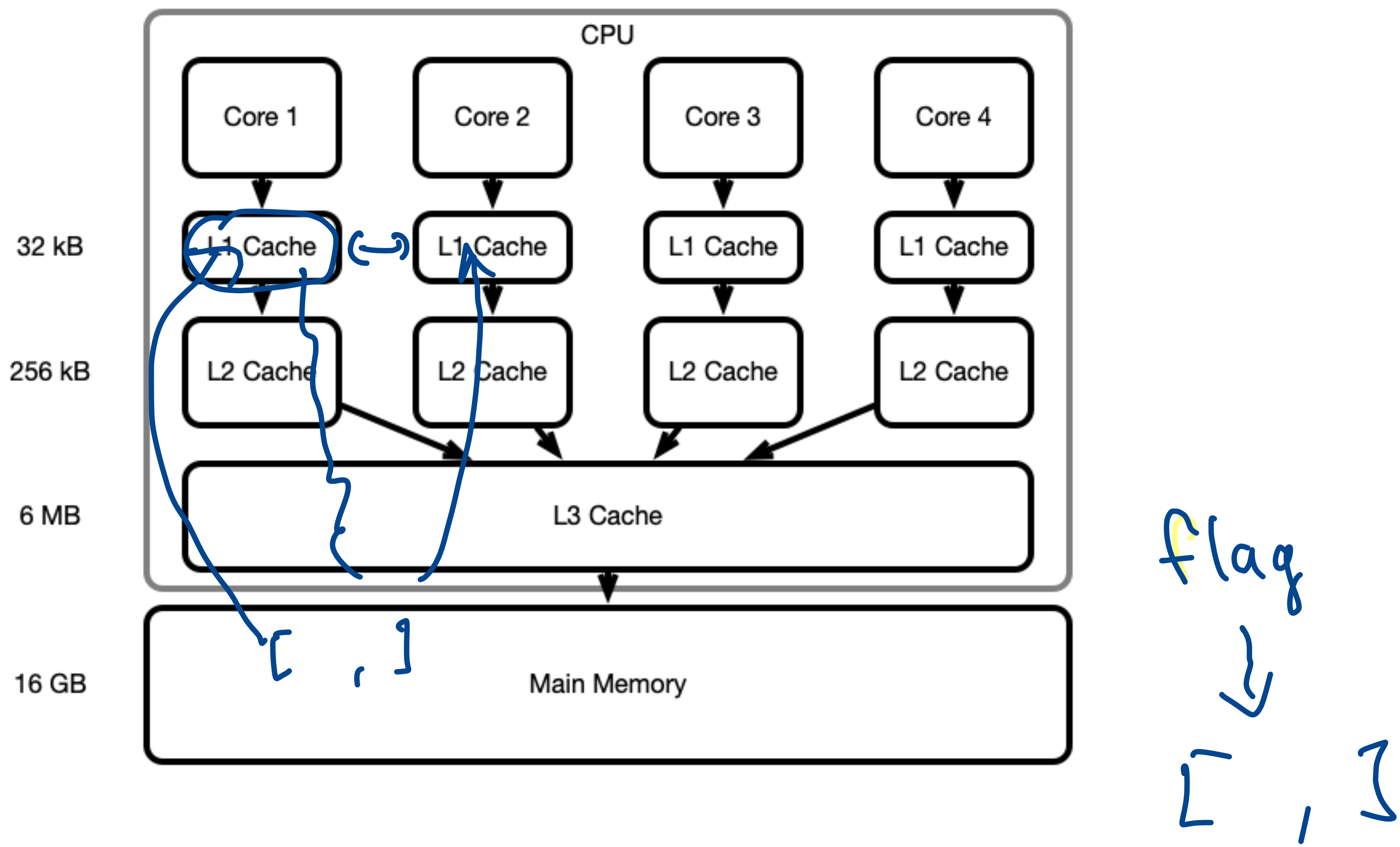
```
public class LockedCounter {
    private int count = 0;
    PetersonLock lock = new PetersonLock();
    public void increment () {
        lock.lock();
        try { ++count; }
        finally { lock.unlock(); }
    }
    public int read () {
        return count;
    }
}
```

# Finally, We’re Ready to Test It!

# D’oh!

What happened?

# Memory Consistency!



# `volatile` Variables

Java can make variables visible between threads:

- use `volatile` keyword
- individual read/write operations to `volatile` are atomic

Drawbacks:

- `volatile` variables are less efficient
- *only* single read/write operations are atomic
  - e.g. `count++` not atomic
- only primitive datatypes are visible
  - if `volatile SomeClass...`, only the *reference* is treated as volatile

# What Variables Should be `volatile`?

- In `PetersonLock`?
  - `flag`?
  - `victim`?

- In `LockedCounter`?
  - `count`?

# A Problem

Only primitive datatypes can be `volatile`

- `volatile boolean[] flag` makes the *reference* `volatile`, not the data itself

How to fix this?

# A Fix

Just make 2 `boolean` variables, `flag0` and `flag1`

- Yes, I know this is ugly

# Fixing Implementation

- `peteson-lock.zip`

# Testing Our Counter Again

# Finally!!!

What have we done?

1. *Proven* correctness of a lock
    - idealized model of computation
    - atomic read/write operations
2. Implemented lock
    - used Java to resemble idealized model
3. Used lock
    - saw expected behavior

Theory and practice converge!

# Peterson: Good and Bad

The Good:

1. It works!
2. It only uses read/write operations!

The Bad:

1. It only works with two threads!
2. Ugly implementation
    - need a separate `PetersonThread` to assign IDs

**Question.** How could we lock more simply?

# Better Tech!

Use more advanced `Atomic` Objects!

**Introducing** the `AtomicBoolean` class:

- `var ab = new AtomicBoolean(boolean value)` make an `AtomicBoolean` with initial value `value`
- `ab.get()` return the current value
- `ab.getAndSet(boolean newValue)` atomically set the value to `newValue` and return the old value
- `ab.compareAndSet(boolean expected, boolean new)` atomically update to new if previous value was `expected` and return whether or not the value was updated

# A Simpler Lock?

**Question.** How could we use `AtomicBooleans` to design a simpler lock?

- no Java gymnastics to deal with thread IDs
- no complicated data structures