# Lecture 25: Atomic Locks

COSC 273: Parallel and Distributed Computing

Spring 2023

# Announcements

Homework 03 is finalized

- no new questions
- due next Friday

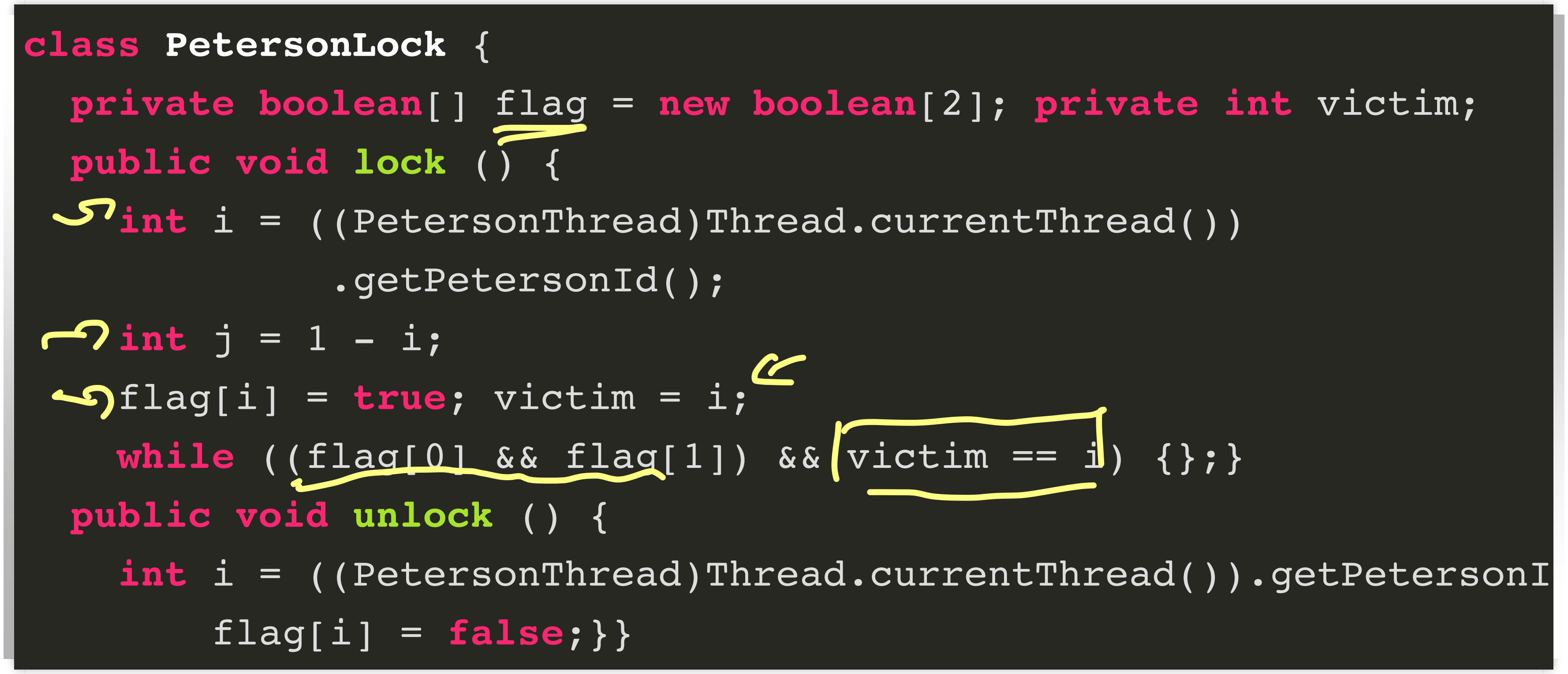
# Today

- More Lock Implementations

# Last Time:

- Peterson lock implementation
  - `peterson-lock.zip`
- disappointment
  - it didn’t achieve mutual exclusion!

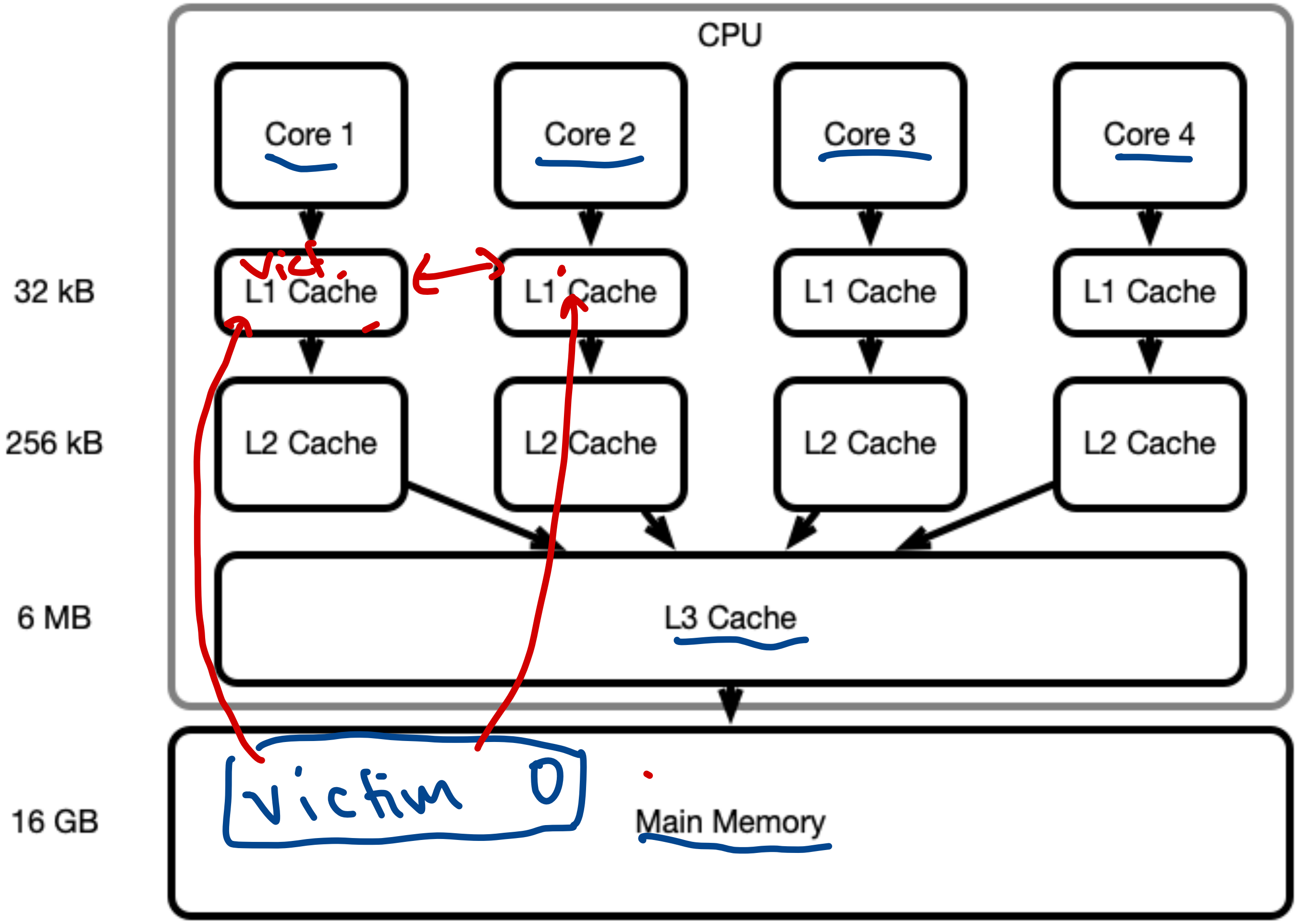
# Peterson Lock Code



```
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[0] && flag[1]) && victim == i) {};}
  public void unlock () {
    int i = ((PetersonThread)Thread.currentThread()).getPetersonI
        flag[i] = false;}}
```

# Memory Consistency!

" Cache Coherence "



# `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

# Making Variables Volatile

- In `PetersonLock`
  - `flag`: an *array* (object) can't be volatile
    - replace with boolean `flag0`, `flag1`
  - `victim` int
- In `LockedCounter`
  - `count`

# Fixing Implementation

- `peteson-lock.zip`

# 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

```
if (value == expected)
        value = new
        return true
else
        return false
```

# A Simpler Lock?

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

Idea: use array of atomic b.
for flags

Another idea: have one A. B.
to store "state" of lock

A.B. locked:

→ to obtain set locked to true
only obtain lock if
- locked was false,
and
- I set it to true

# Test and Set Lock

**Idea.** An `AtomicBoolean locked` stores state of the lock:

- `locked.get() == true` indicates the lock is in use
- `locked.get() == false` indicates the lock is free

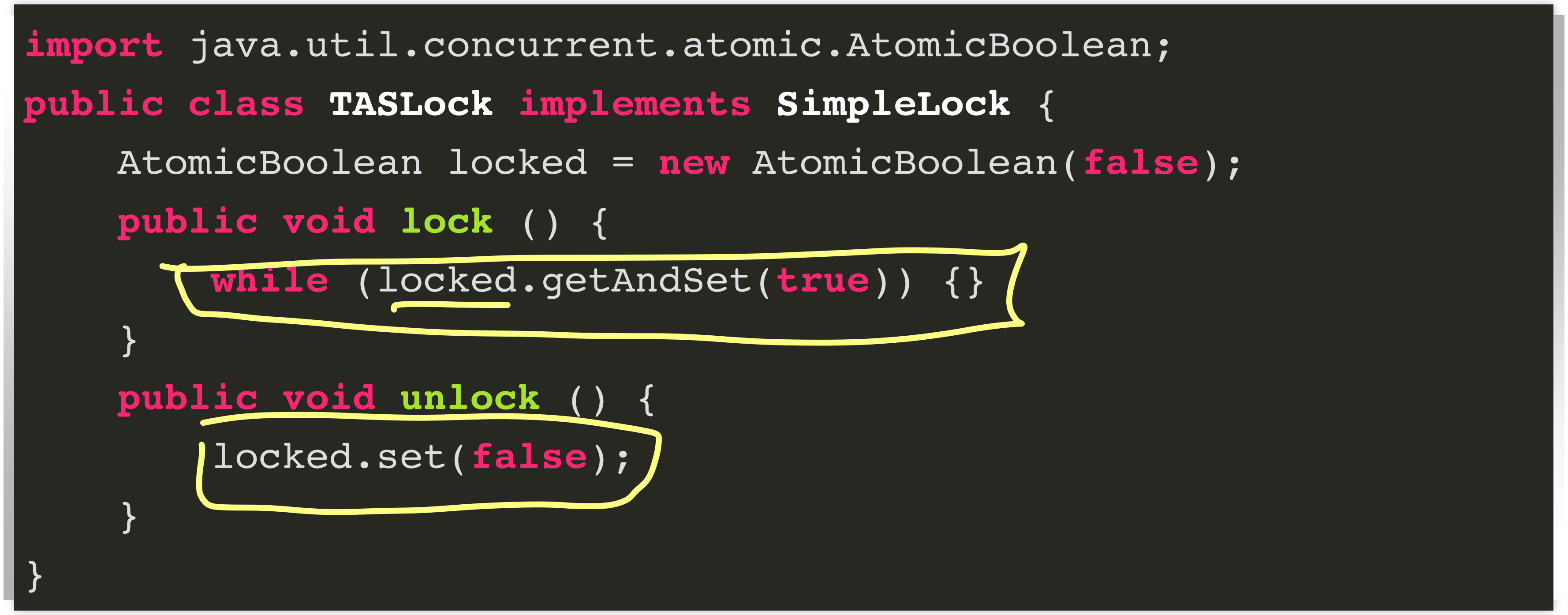
Obtaining the lock:

- wait until `locked` is `false`, and set it to `true`

Releasing the lock:

- set `locked` to false

# TASLock in Code



```java
import java.util.concurrent.atomic.AtomicBoolean;
public class TASLock implements SimpleLock {
    AtomicBoolean locked = new AtomicBoolean(false);
    public void lock () {
        while (locked.getAndSet(true)) {}
    }
    public void unlock () {
        locked.set(false);
    }
}
```

- download `tas-locks.zip`

# Progress Guarantees

**Question.** Is `TASLock` deadlock-free? Starvation-free?

# Alternative Implementation

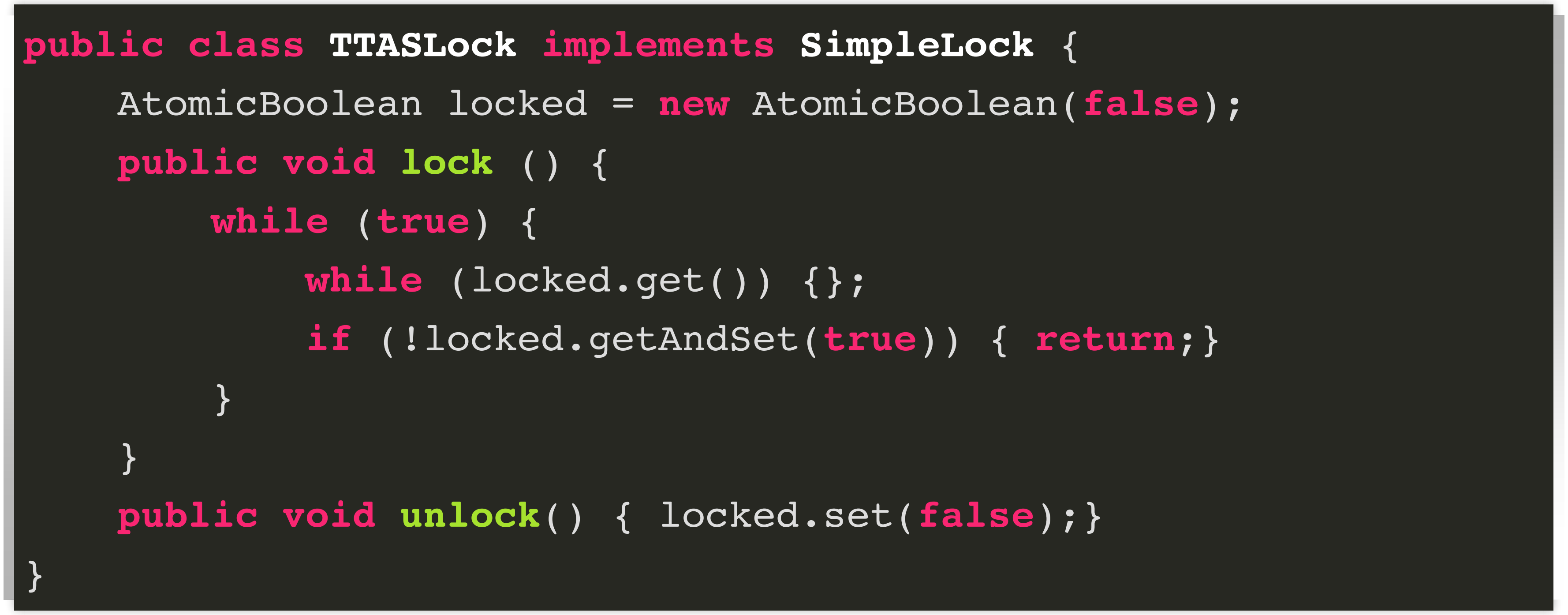
Potential Issue:

- `getAndSet` operation is somewhat inefficient
    - slower than just `get`

Test and Test and Set Lock:

- check if locked
    - if not, attempt `getAndSet`
    - return if successful

# TTASLock Implementation



```
public class TTASLock implements SimpleLock {
    AtomicBoolean locked = new AtomicBoolean(false);
    public void lock () {
        while (true) {
            while (locked.get()) {};
            if (!locked.getAndSet(true)) { return;}
        }
    }
    public void unlock() { locked.set(false);}
}
```

## Comparing Efficiency

- `tas-locks.zip`