# Concurrent programming

## Introduction

Companion slides for

The Art of Multiprocessor Programming

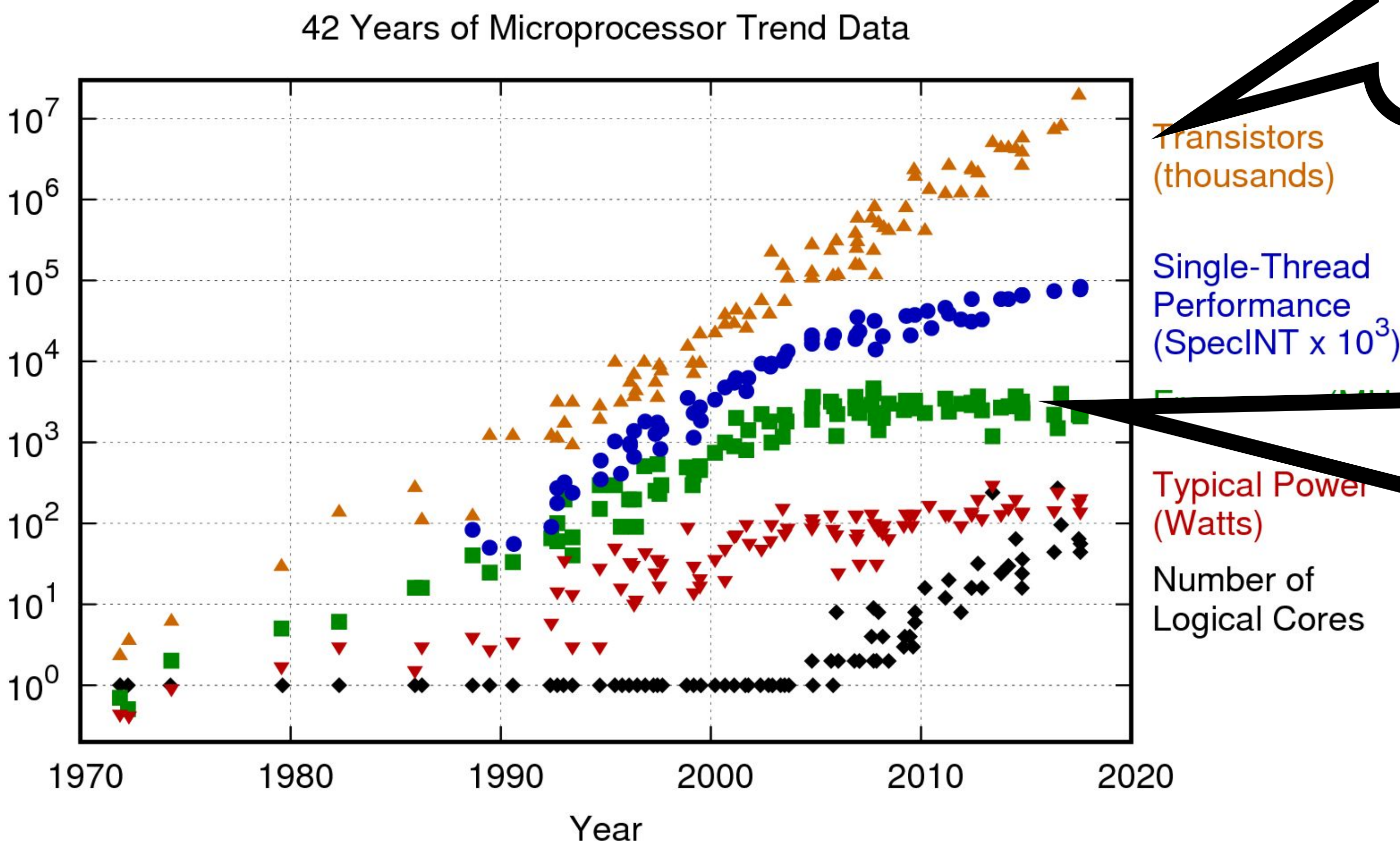
by Maurice Herlihy, Nir Shavit, Victor Luchangco,
and Michael Spear

Modified by Piotr Witkowski

# Moore’s Law

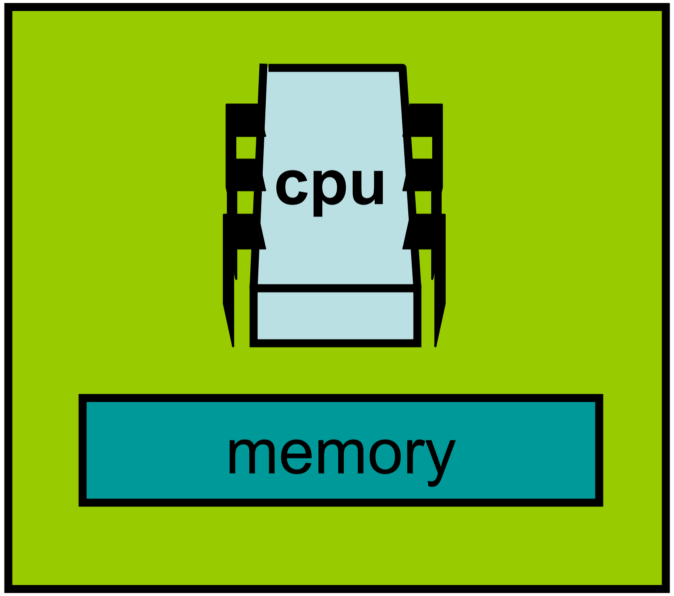
Transistor count still rising

Clock speed flattening sharply

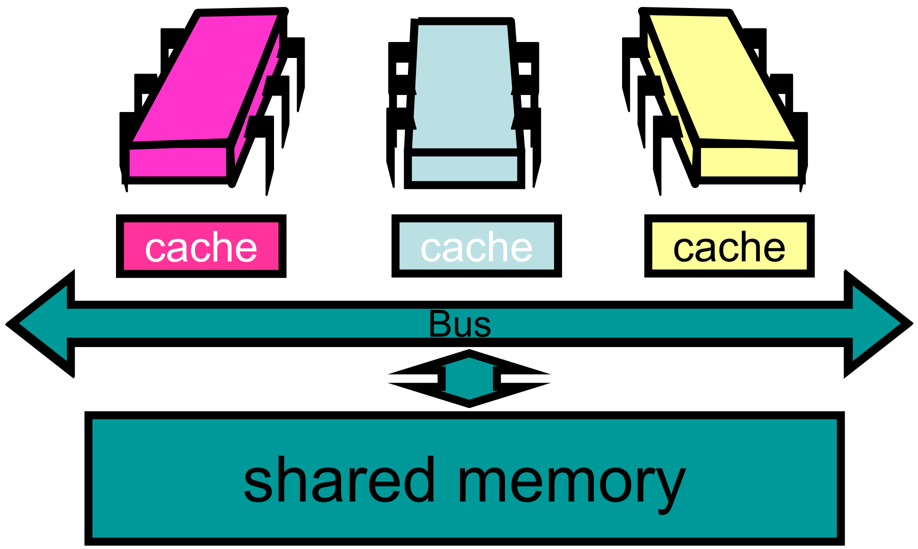



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

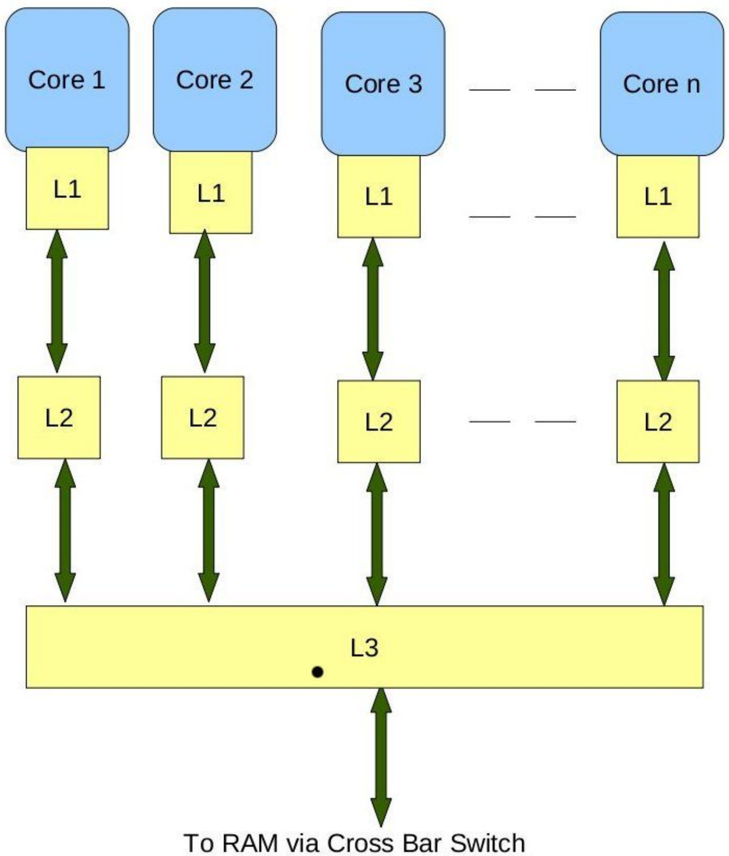
# Extinct: the Uniprocesor




# Extinct: The Shared Memory Multiprocessor (SMP)




# The New Boss: The Multicore Processor




**All on the same chip / chiplet**

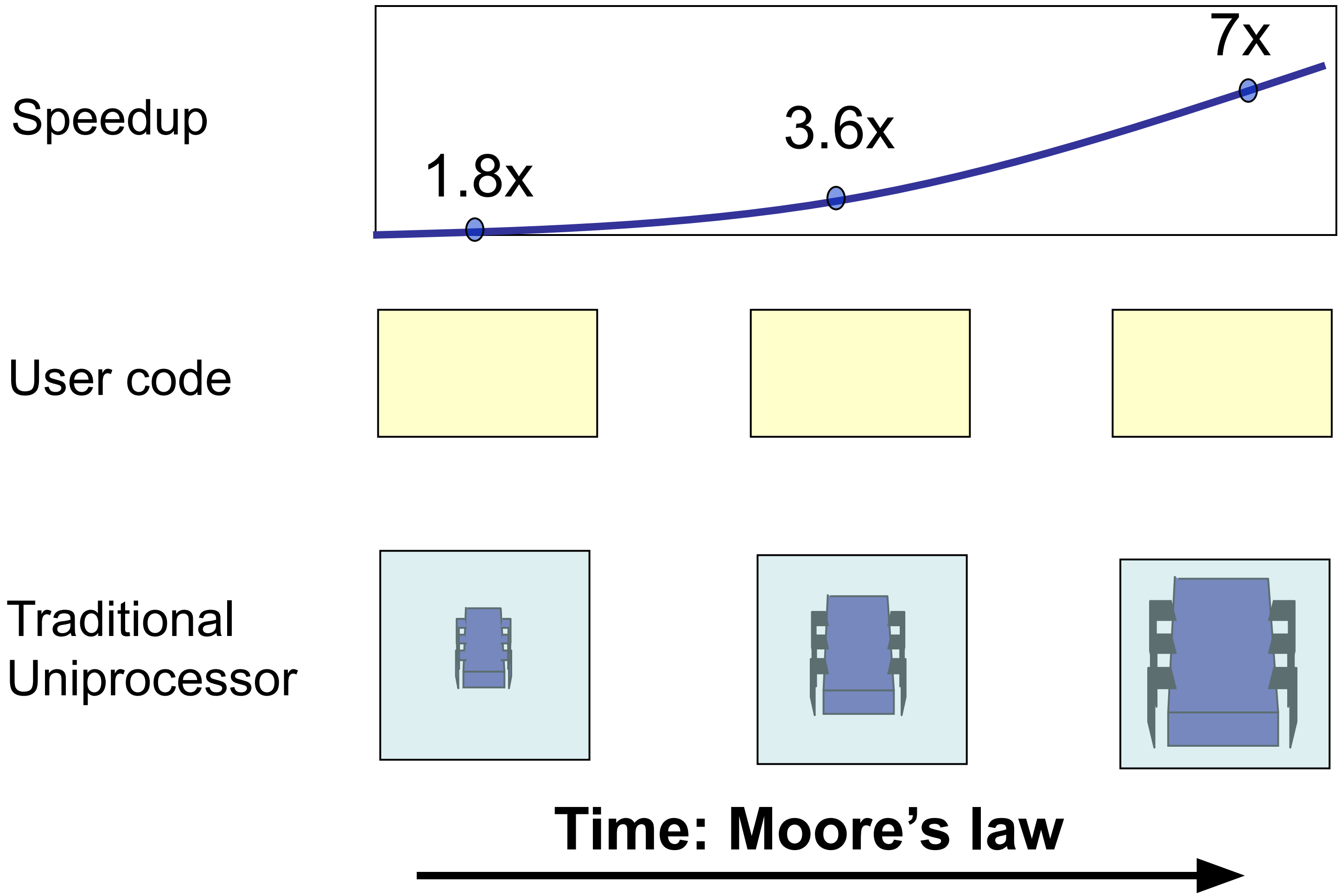
Intel Core i9 upto 18 cores

AMD Ryzen upto 64 cores

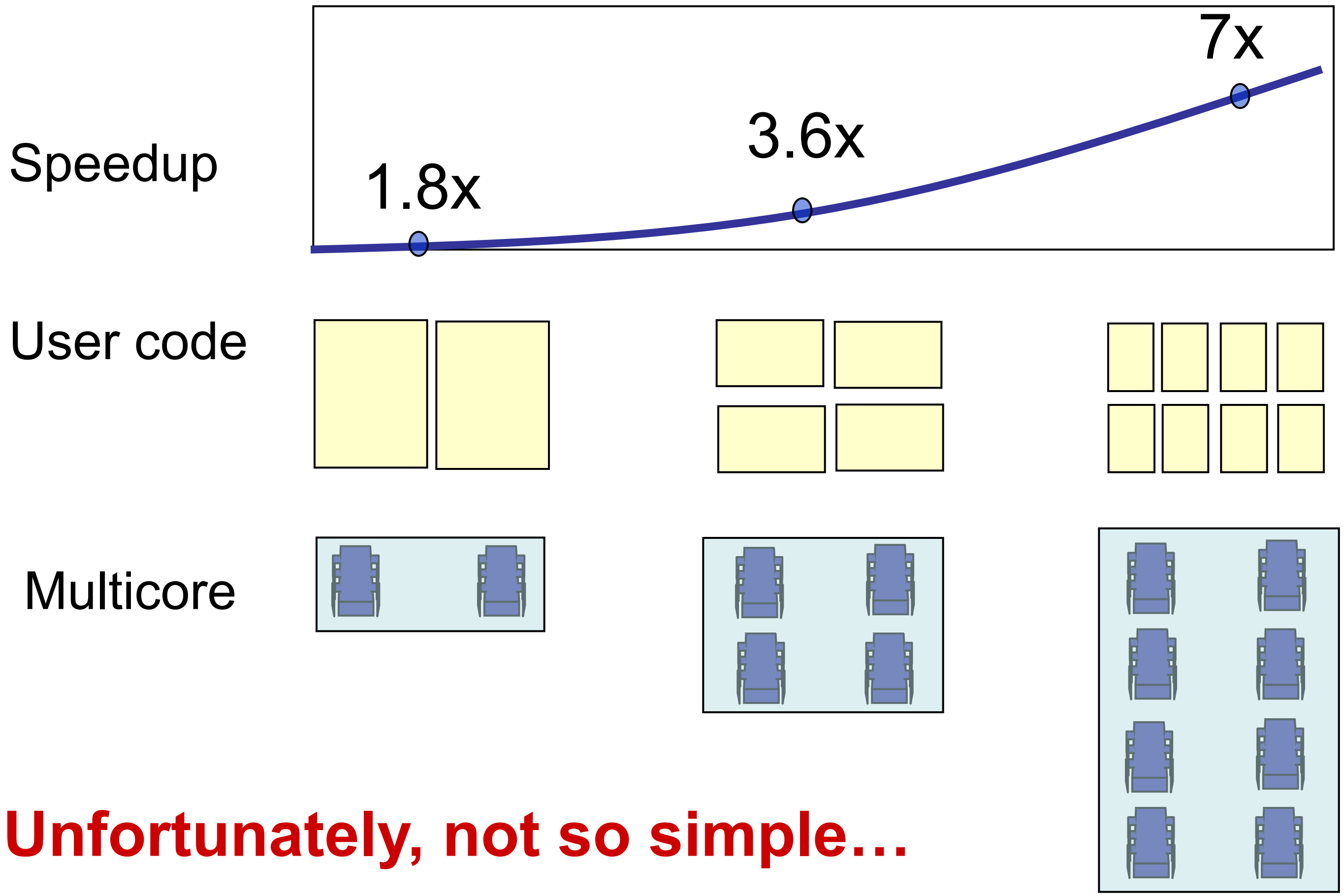
# Why do we care?

- Time no longer cures software bloat
  - The “free ride” is over
- When you double your program’s path length
  - You can’t just wait 6 months
  - Your software must somehow exploit twice as much concurrency

# Traditional Scaling Process

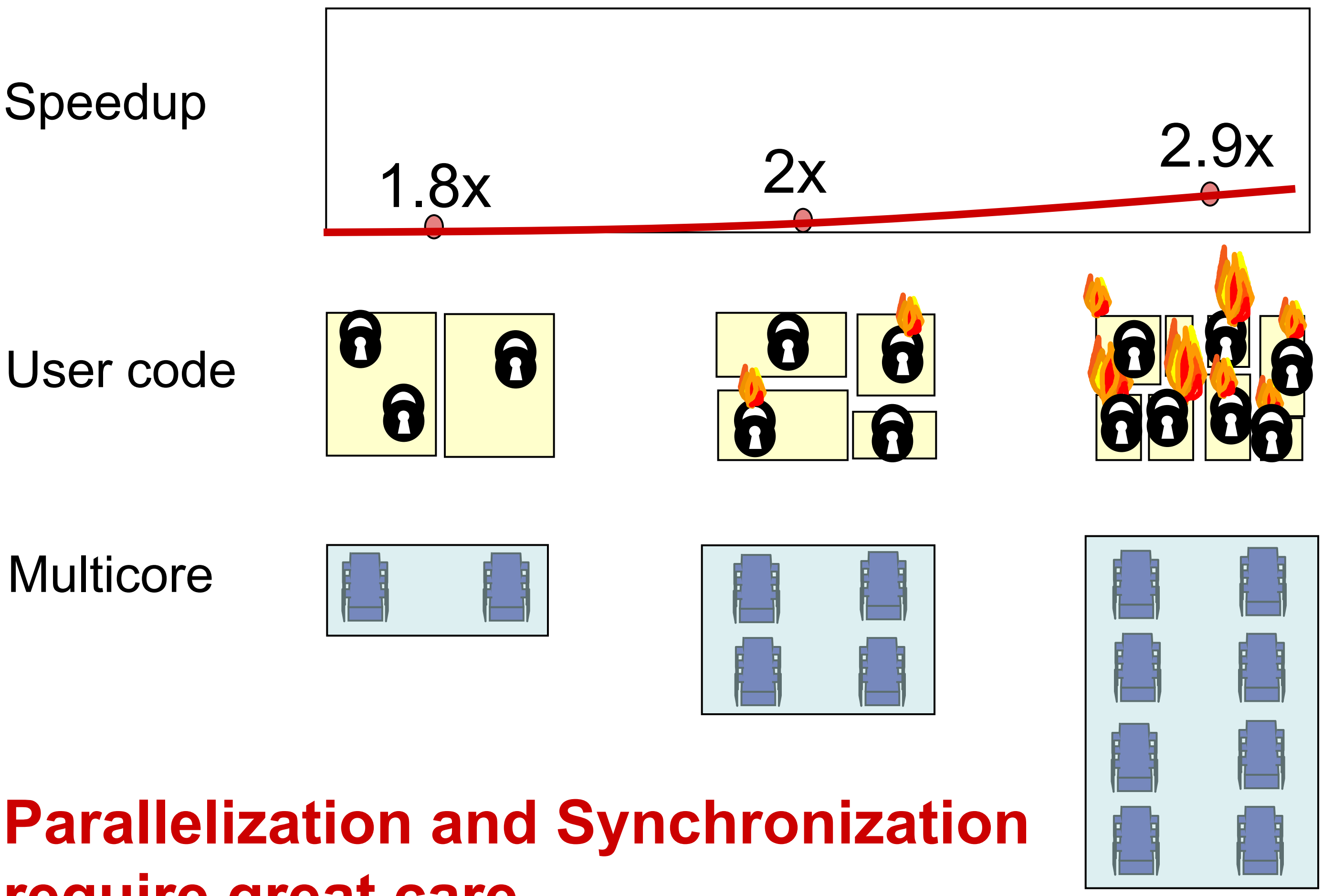


# Ideal Scaling Process




**Unfortunately, not so simple…**

# Actual Scaling Process

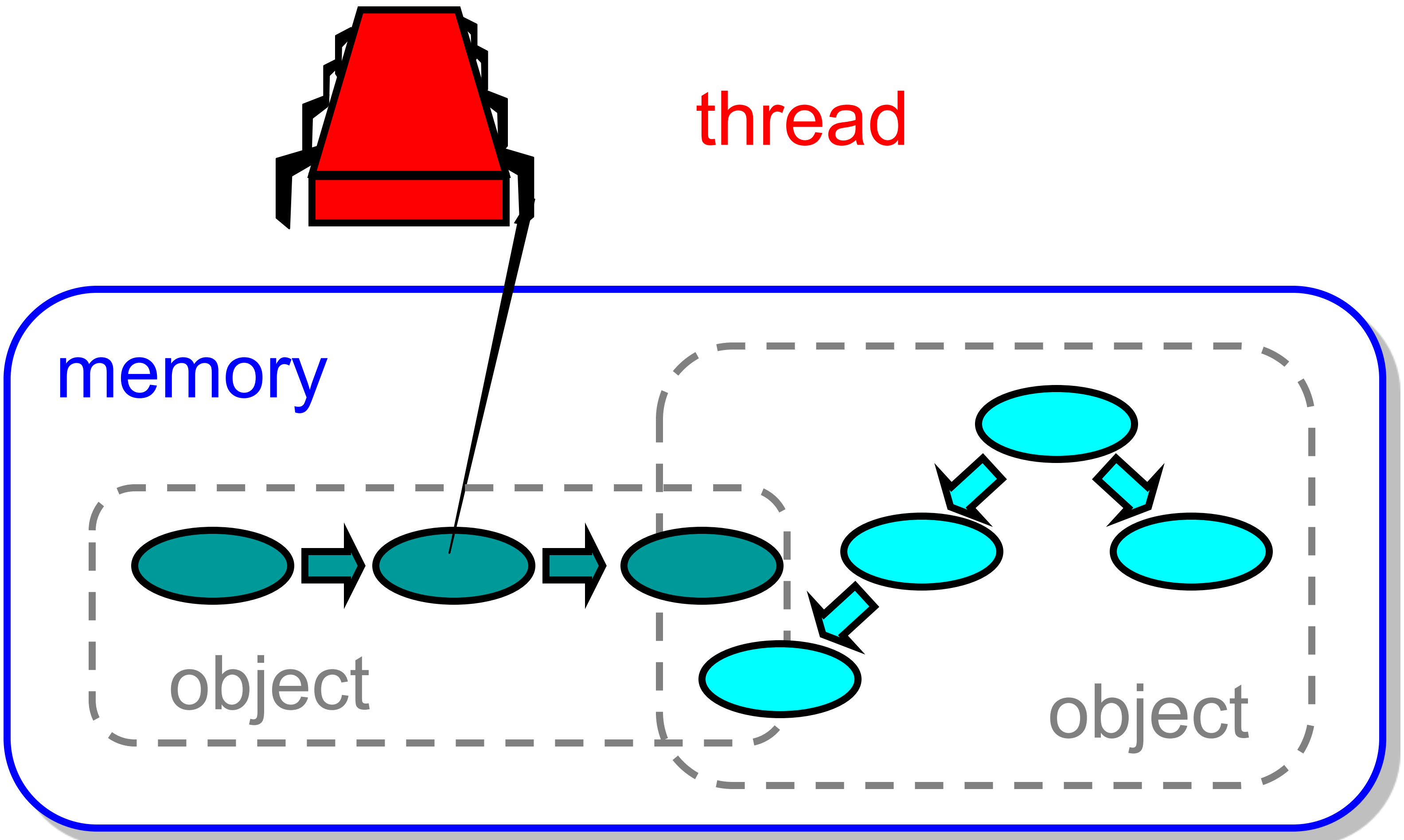



**Parallelization and Synchronization require great care…**

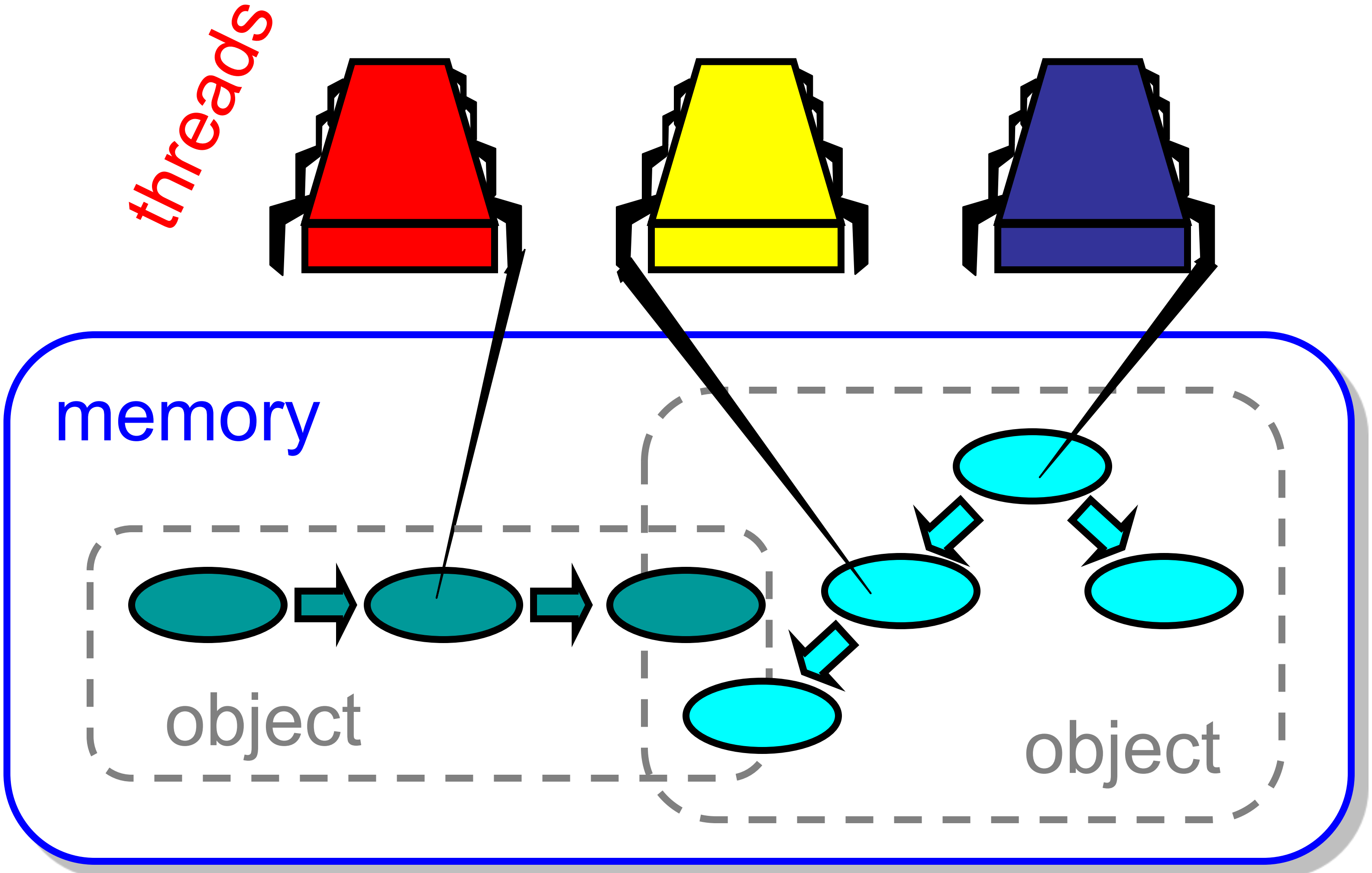
# Concurrent Programming: Course Overview

- Fundamentals
  - Models, algorithms, impossibility
- Real-World programming
  - Architectures
  - Techniques

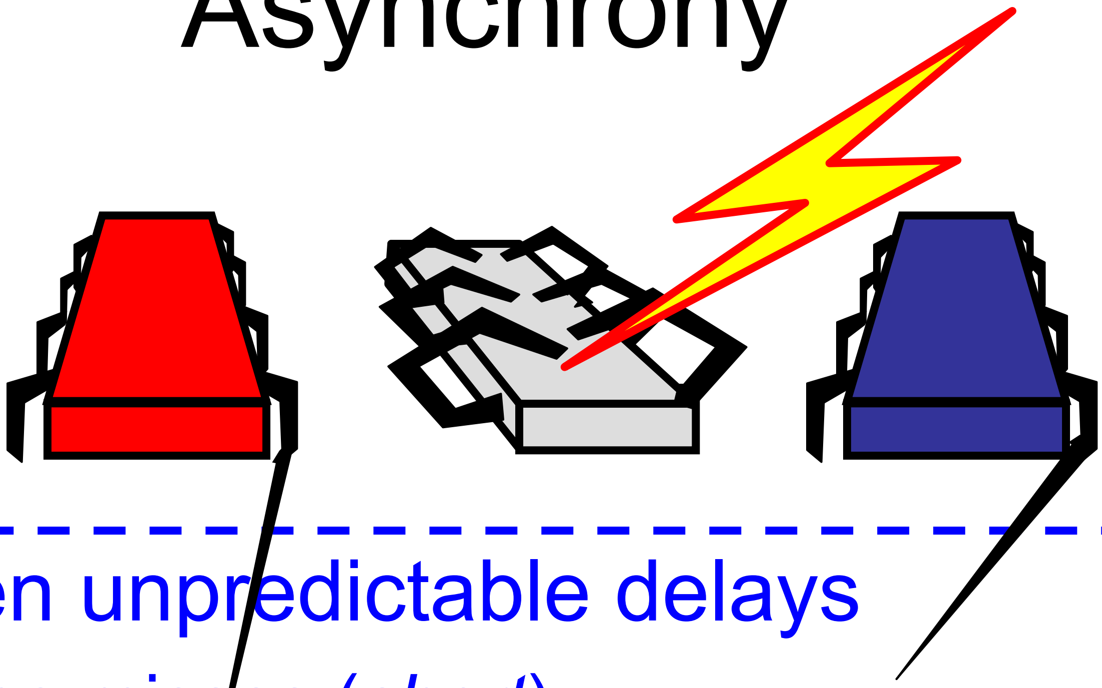
# Sequential Computation



# Concurrent Computation



# Asynchrony



- Sudden unpredictable delays
  – Cache misses (*short*)
  – Page faults (*long*)
  – Scheduling quantum used up (*really long*)

# Model Summary

- Multiple *threads*
  - Sometimes called *processes*
- Single shared *memory*
- *Objects* live in memory
- Unpredictable asynchronous delays

# Road Map

- We are going to focus on principles first, then practice
  - Start with idealized models
  - Look at simplistic problems
  - Emphasize correctness over pragmatism
  - "Correctness may be theoretical, but incorrectness has practical impact"

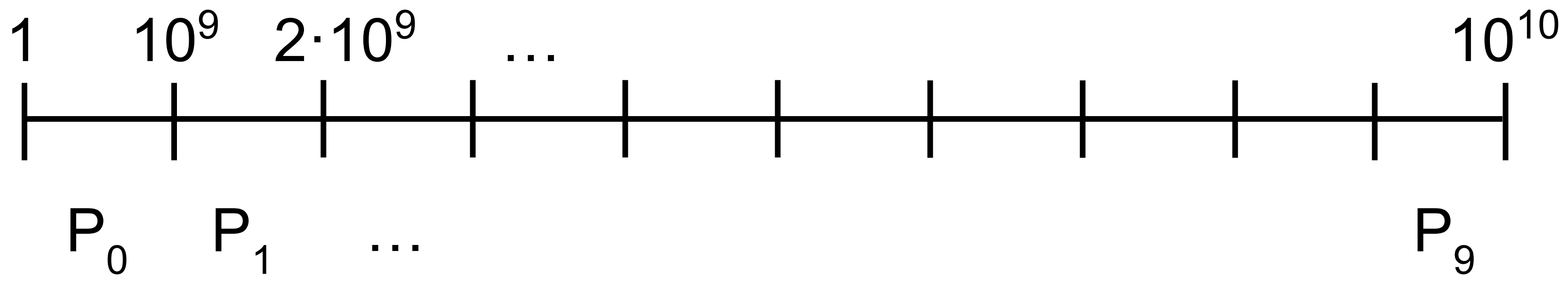
# Concurrency Jargon

- Hardware
  - processors ≈ cores
- Software
  - threads ≈ processes
- Programing
  - concurrent ≈ parallel ≈ multiprocessor ≈ multicore
- Sometimes OK to confuse them, sometimes not.

# Parallel Primality Testing

- Challenge
  – Print primes from 1 to $10^{10}$
- Given
  – Ten-processor multiprocessor
  – One thread per processor
- Goal
  – Get ten-fold speedup (or close)

# Load Balancing




- Split the work evenly
- Each thread tests range of $10^9$

# Procedure for Thread *i*

```
void primePrint {
  int i = ThreadID.get(); // IDs in {0..9}
  for (j = i*10^9+1, j<(i+1)*10^9; j++) {
    if (isPrime(j))
      print(j);
  }
}
```

# Issues

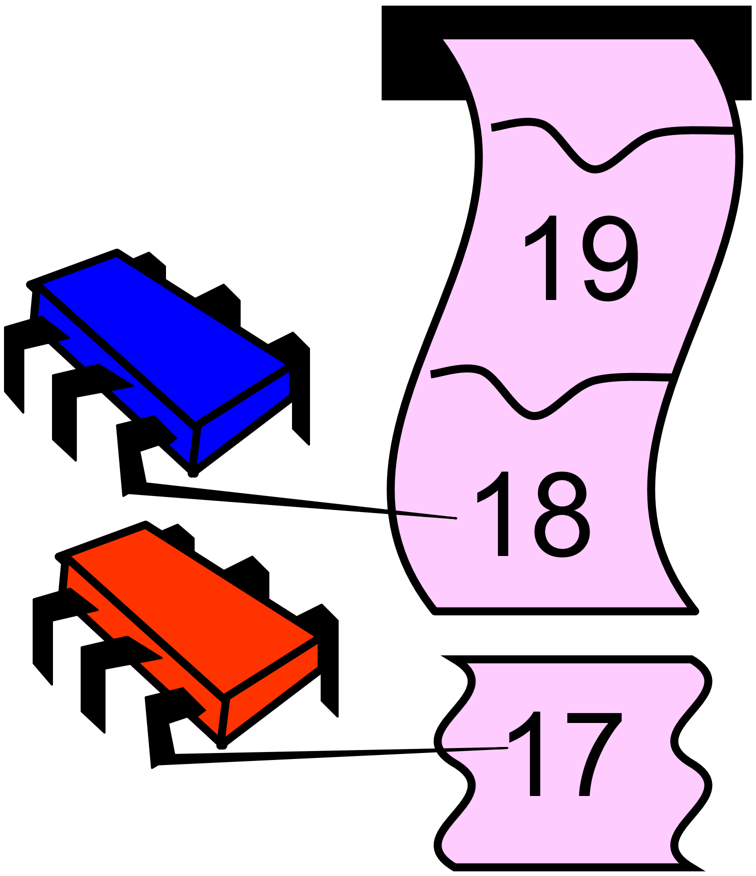
- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict

# Issues

- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict
- Need *dynamic* load balancing

rejected

# Shared Counter




each thread
takes a number

# Procedure for Thread *i*

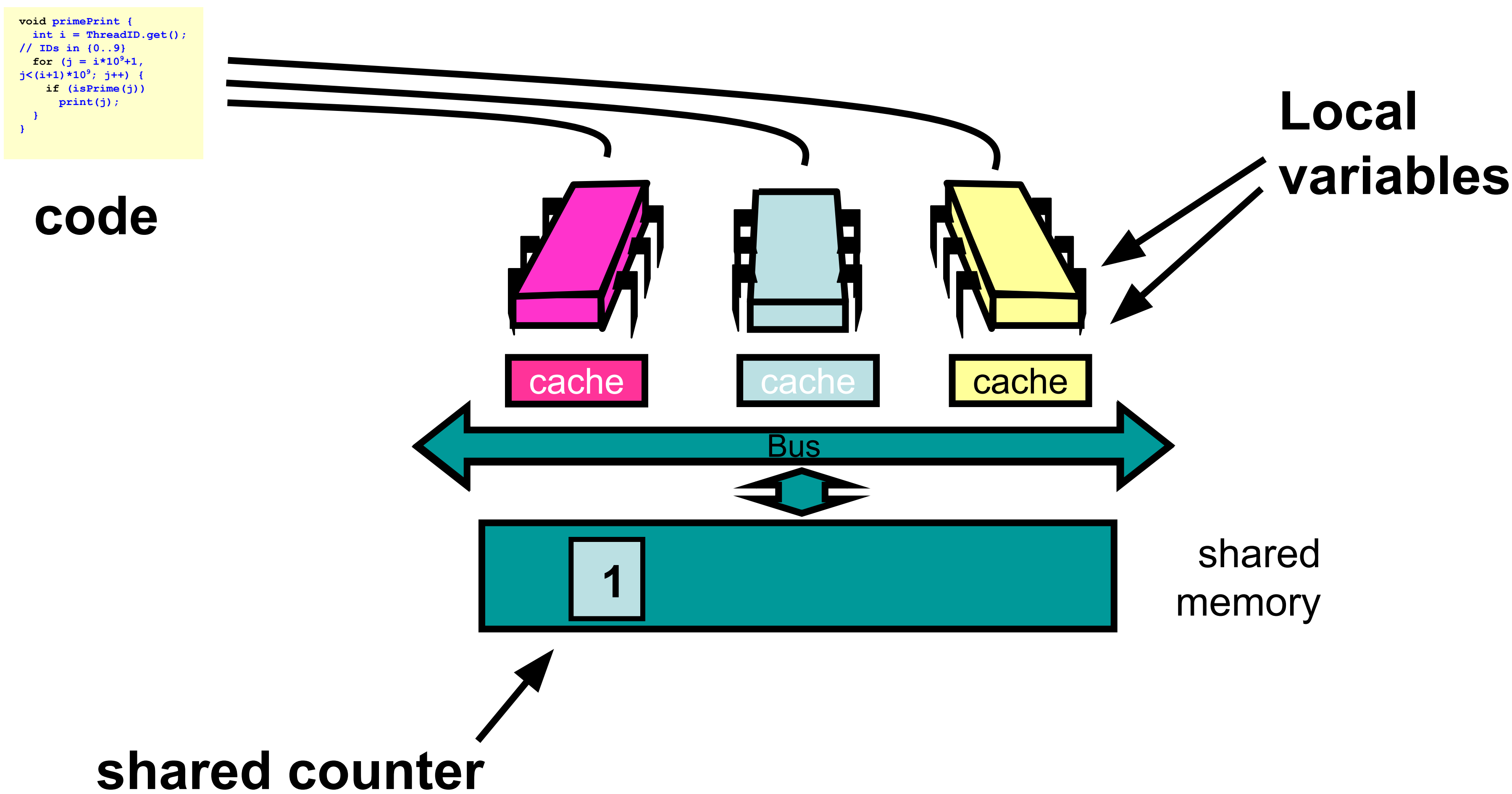
```
int counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

# Procedure for Thread *i*

```
Counter counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

**Shared counter object**

# Where Things Reside



# Procedure for Thread *i*

```
Counter counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

**Stop when every value taken**

# Procedure for Thread *i*

```
Counter counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

**Increment & return each new value**

# Counter Implementation

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

# Counter Implementation

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

OK for single thread,
not for concurrent threads

# What It Means

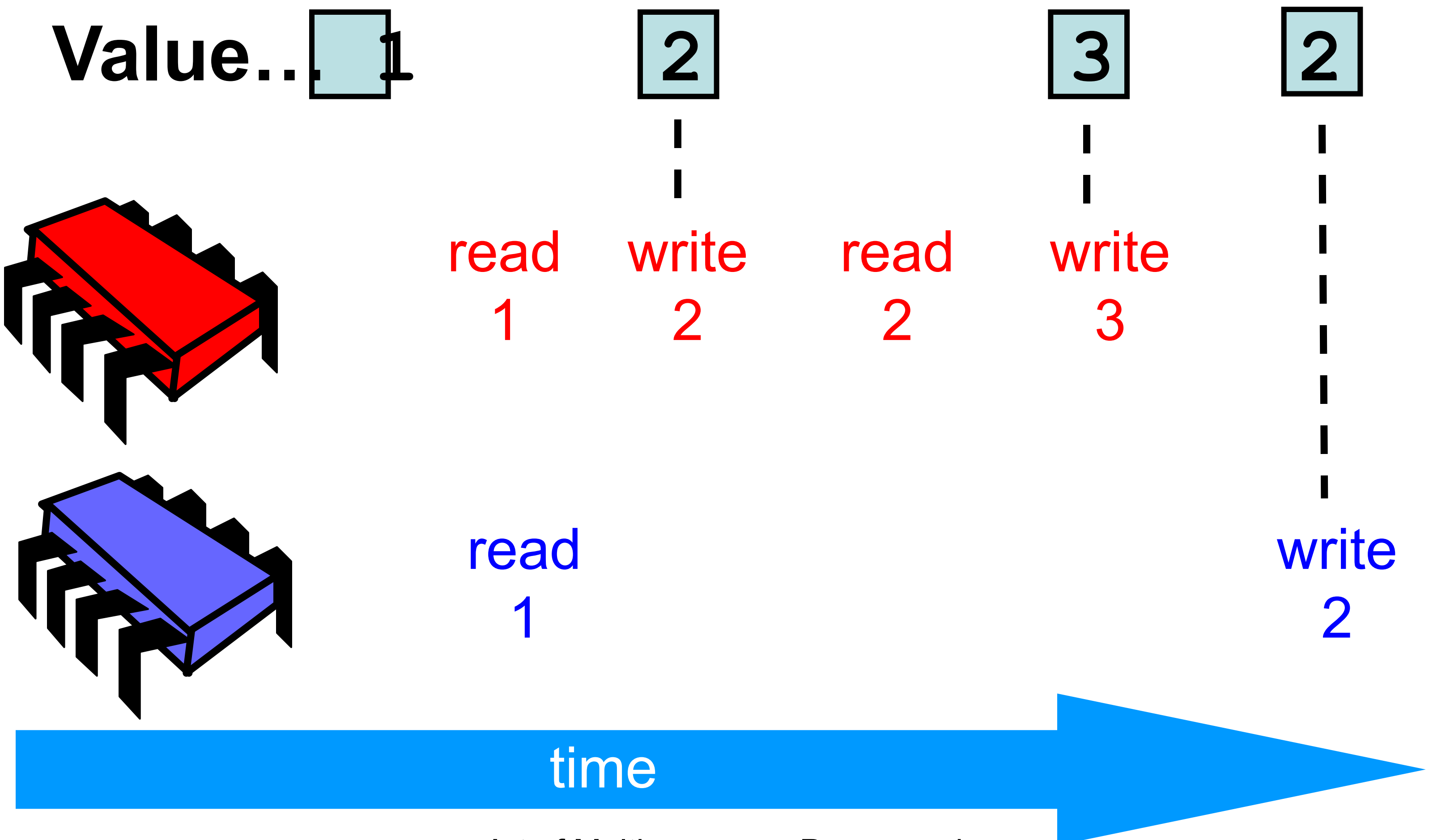
```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

# What It Means

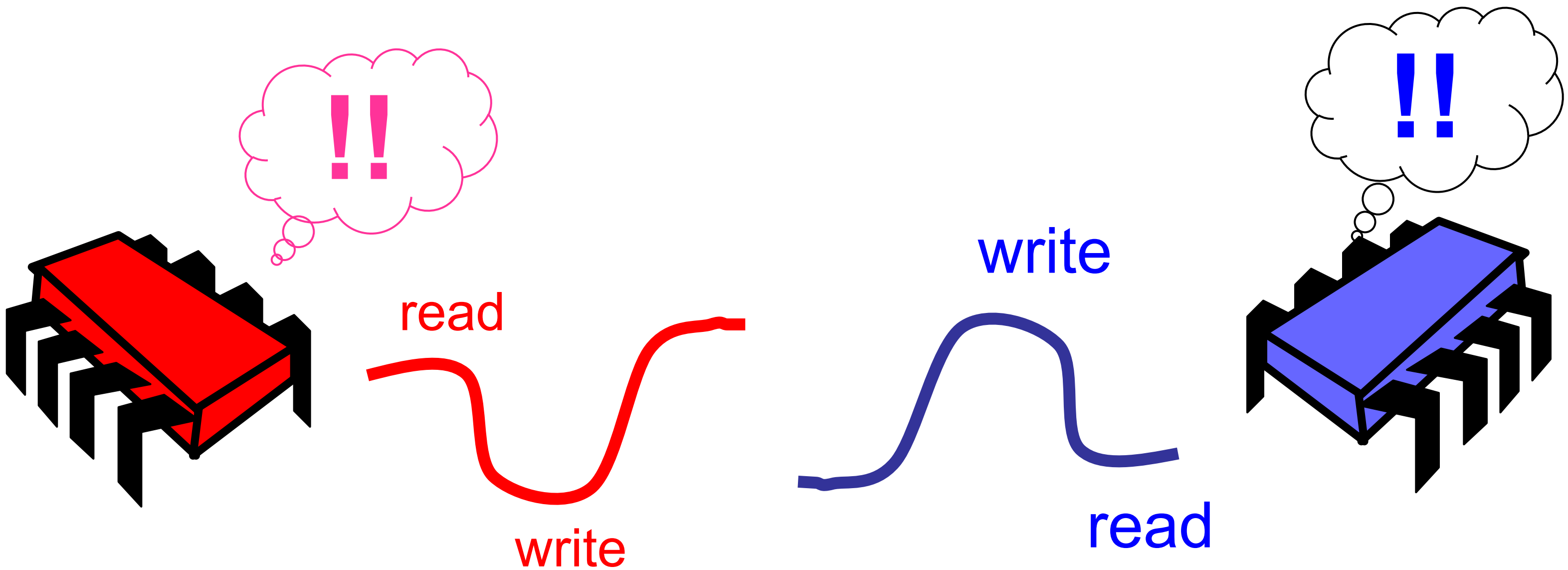
```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

```
temp  = value;
value = temp + 1;
return temp;
```

# Not so good…



# Is this problem inherent?




If we could only glue reads and writes together…

# Challenge

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    value = temp + 1;
    return temp;
  }
}
```

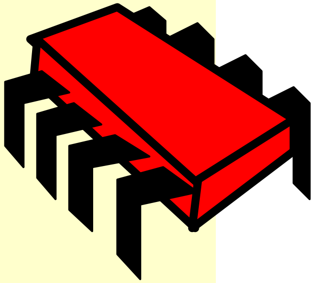
# Challenge

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    value = temp + 1;
    return temp;
  }
}
```

**Make these steps *atomic* (indivisible)**

# Hardware Solution

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    value = temp + 1;
    return temp;
  }
}
```



**ReadModifyWrite() instruction**

# An Aside: Java™

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    synchronized {
      temp  = value;
      value = temp + 1;
      }
    return temp;
  }
}
```

# An Aside: Java™

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    synchronized {
      temp  = value;
      value = temp + 1;
      }
    return temp;
  }
}
```

Synchronized block

# An Aside: Java™

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    synchronized {
      temp  = value;
      value = temp + 1;
    }
    return temp;
  }
}
```

Mutual Exclusion