Principles of Robust Concurrent Computing

Professor Rachid Guerraoui

Distributed Programming Laboratory

EPFL

*This course introduces the **principles** of **robust** and **concurrent** computing…*

Principles

Certain things are **incorrect** and it is important to understand why

(at least what correctness means)

Certain things are **impossible** and its important to understand why

(at least to not try)

Major chip manufacturers have recently announced a major paradigm shift:

New York Times, 8 May 2004:

Intel … [has] decided to focus its development efforts on «dual core» processors … with two engines instead of one, allowing for greater efficiency because the processor workload is essentially shared.

Multiple processors

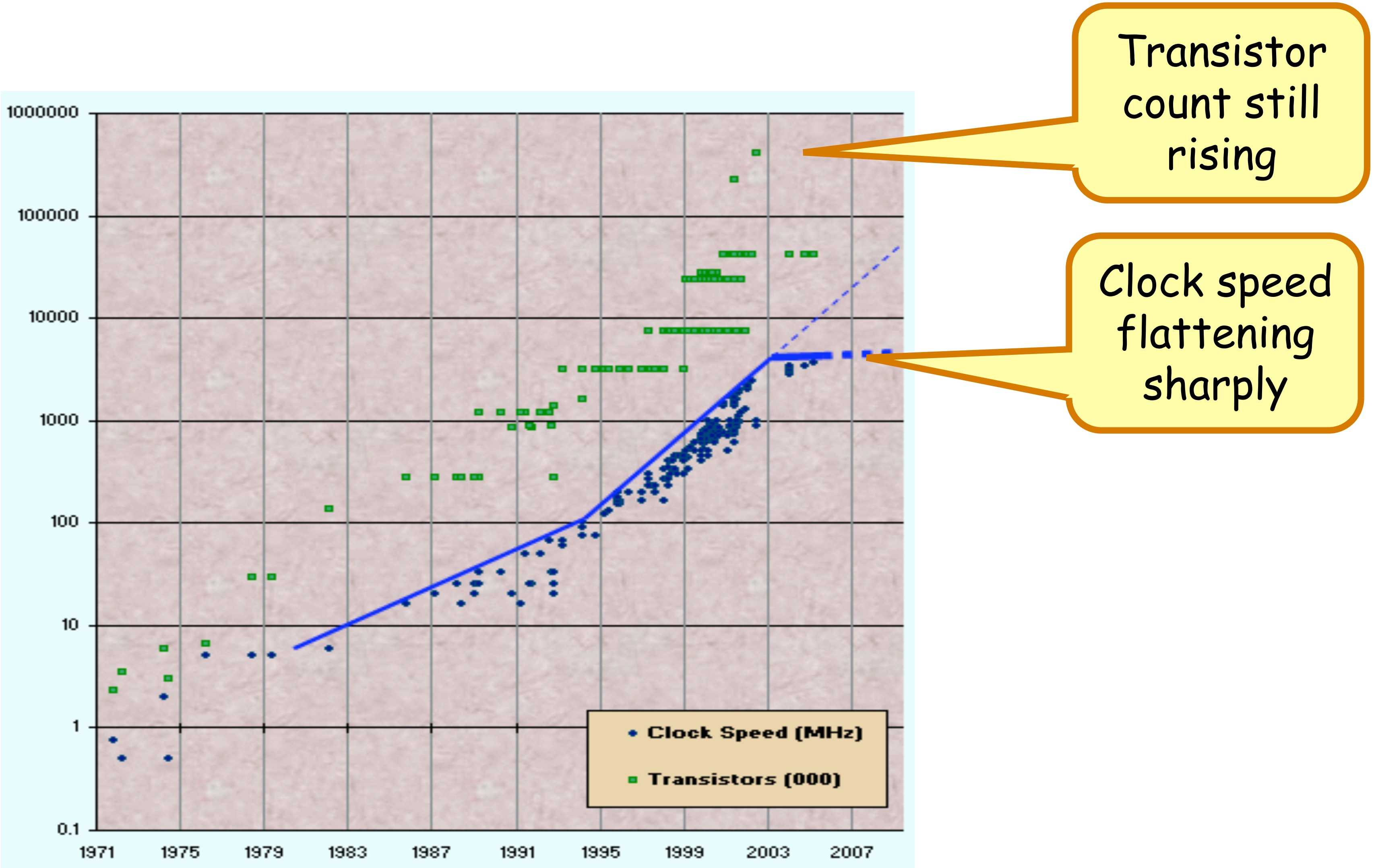
vs

Faster processors

The clock speed of a processor cannot be increased without overheating

But

More and more processors can fit in the same space

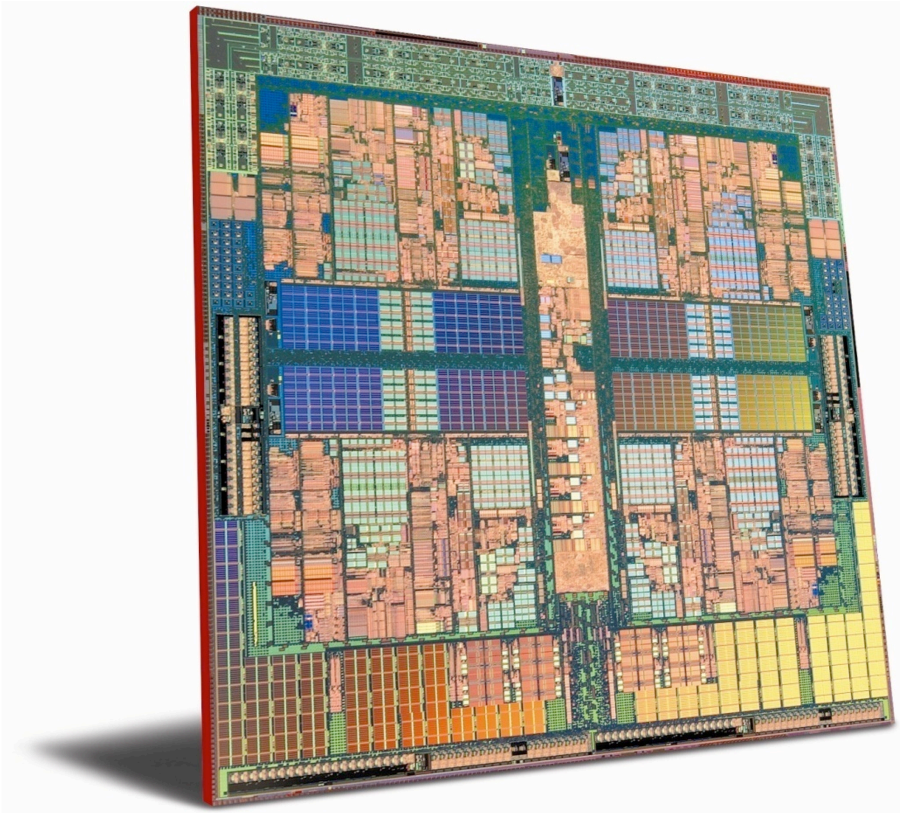


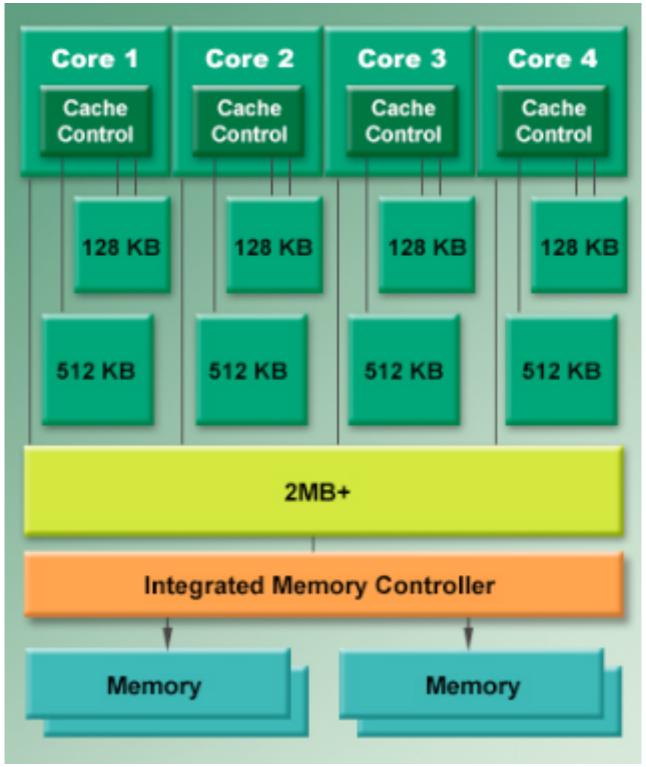
1000000
100000
10000
1000
100
10
1
0.1
1971
1975
1979
1983
1987
1991
1995
1999
2003
2007
Clock Speed (MHz)
Transistors (000)
Transistor count still rising
Clock speed flattening sharply

Multicores are almost everywhere

- **Dual-core commonplace in laptops**
- **Quad-core in desktops**
- **Dual quad-core in servers**
- **All major chip manufacturers produce multicore CPUs**
  - **SUN Niagara (8 cores, 32 threads)**
  - **Intel Xeon (4 cores)**
  - **AMD Opteron (4 cores)**

AMD Opteron (4 cores)



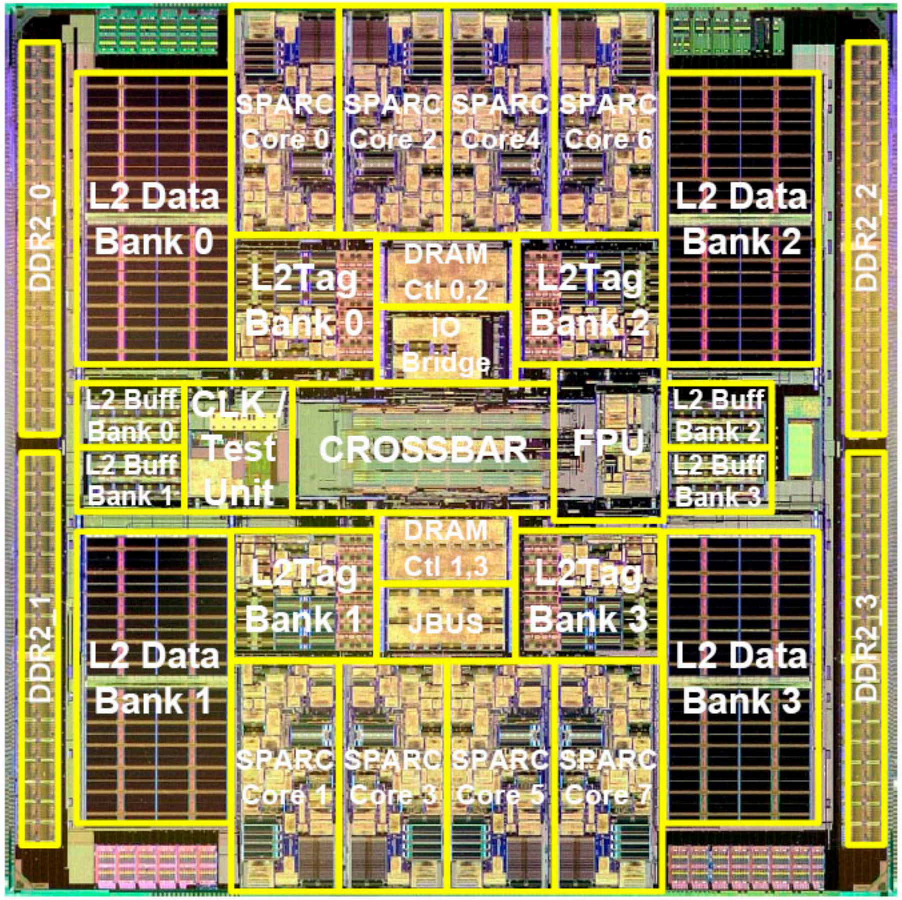


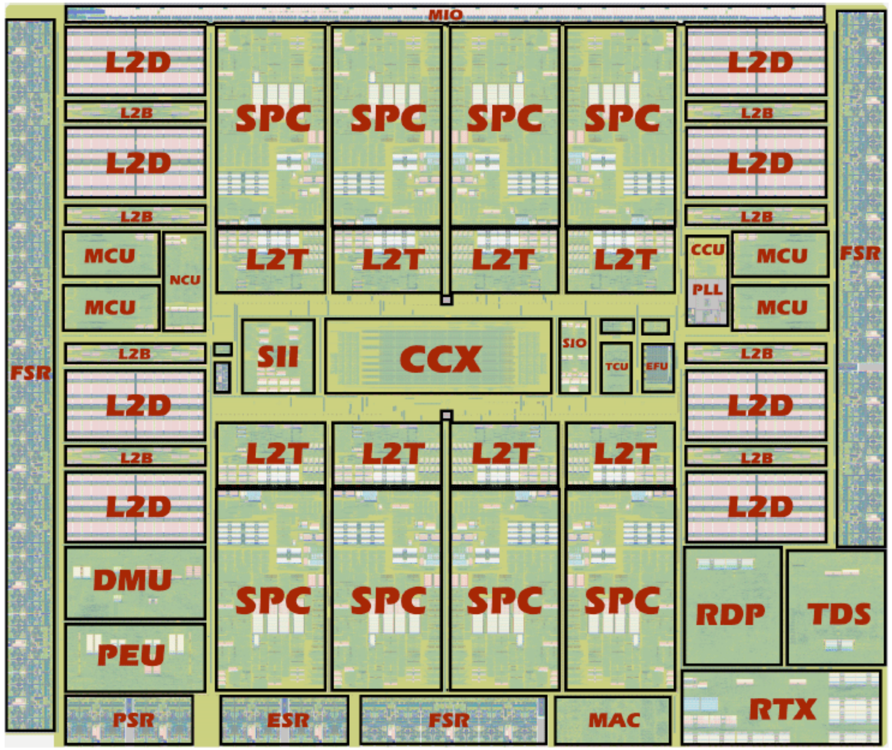
L1 cache

L2 cache

L3 cache
(shared)

SUN's Niagara CPU2 (8 cores)





CCX – Crosssbar
CCU – Clock control
DMU/PEU – PCI Express
EFU – Efuse for redundancy
ESR – Ethernet SERDES
FSR – FBD SERDES
L2B – L2 write-back buffers
L2D – L2 data arrays
L2T – L2 tag arrays
MCU – Memory controller
MIO – Miscellaneous I/O
PSR – PCI Express SERDES
RDP/TDS/RTX/MAC – Ethernet
SII/SIO – I/O data path to and from memory
SPC – SPARC cores
TCU – Test and control unit

Multiprocessors

- Multiple hardware processors, each executes a series of processes (software constructs) modeling sequential programs

- Multicore architecture: multiple processors are placed on the same chip

Principles of an architecture

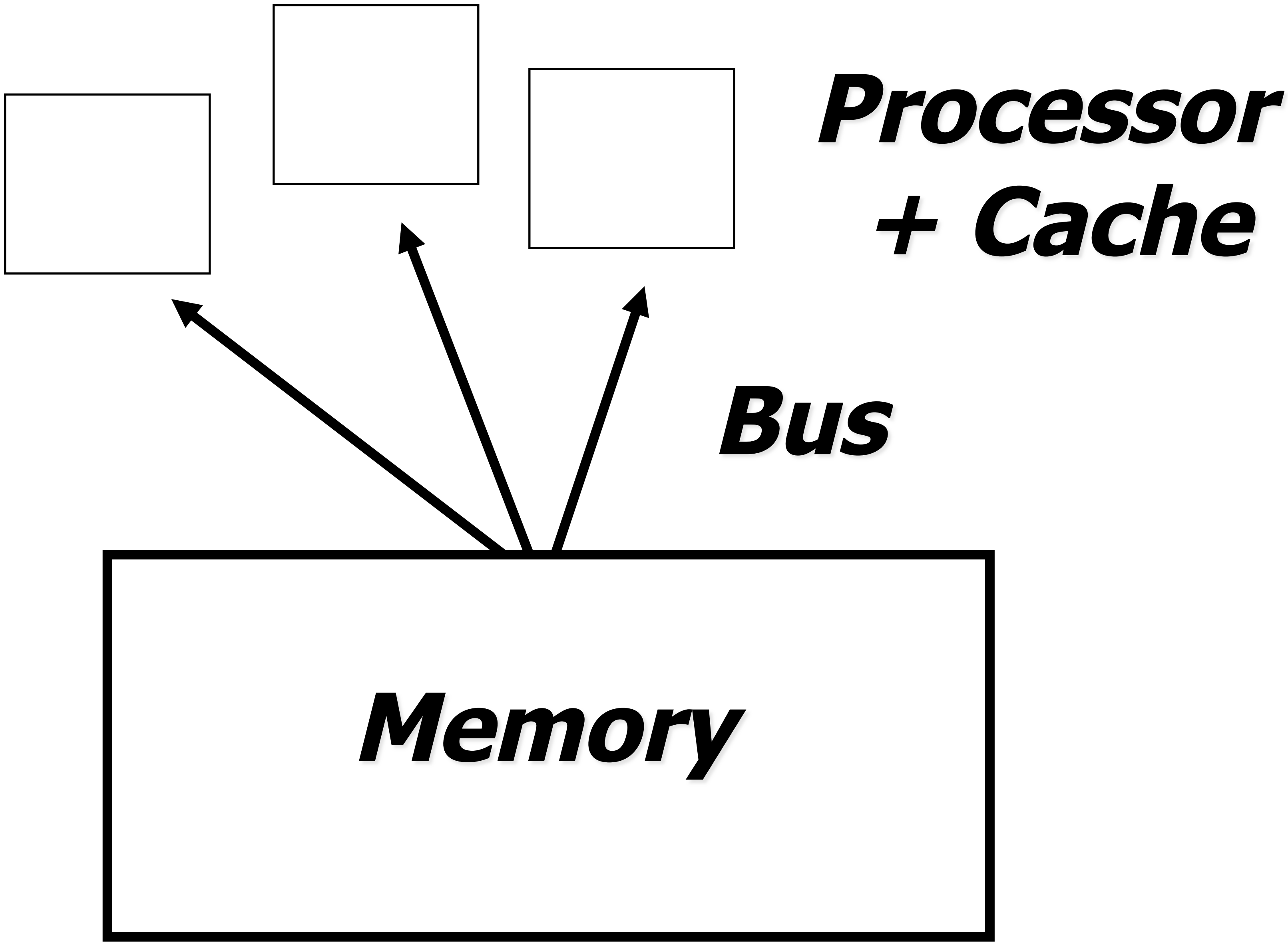
- Two fundamental components that ***fall apart***: ***processors*** and ***memory***

- The Interconnect links the processors with the memory:
- \- ***SMP*** (symmetric): bus (a tiny Ethernet)
- \- ***NUMA*** (network): point-to-point network

Cycles

- The basic unit of time is the cycle: time to execute an instruction
- This changes with technology but the relative cost of instructions (local vs memory) does not

Simple view



Hardware synchronization objects

- The basic unit of communication is the read and write to the memory (through the cache)
- More sophisticated objects are sometimes provides and, as we will see, necessary: C&S, T&S, LL/SC

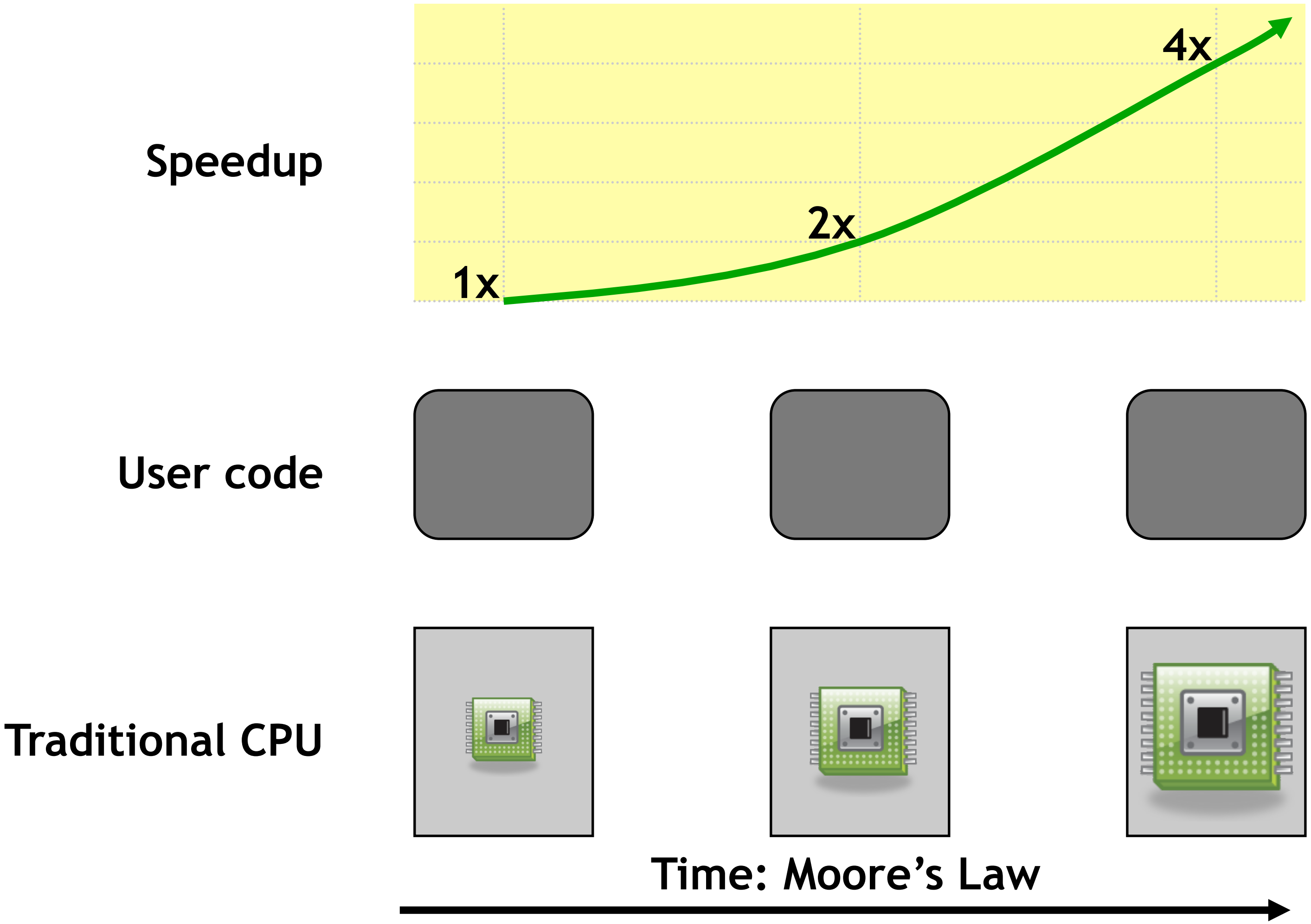
The free ride is over

- **Cannot rely on CPUs getting faster in every generation**
- **Utilizing more than one CPU core requires concurrency**

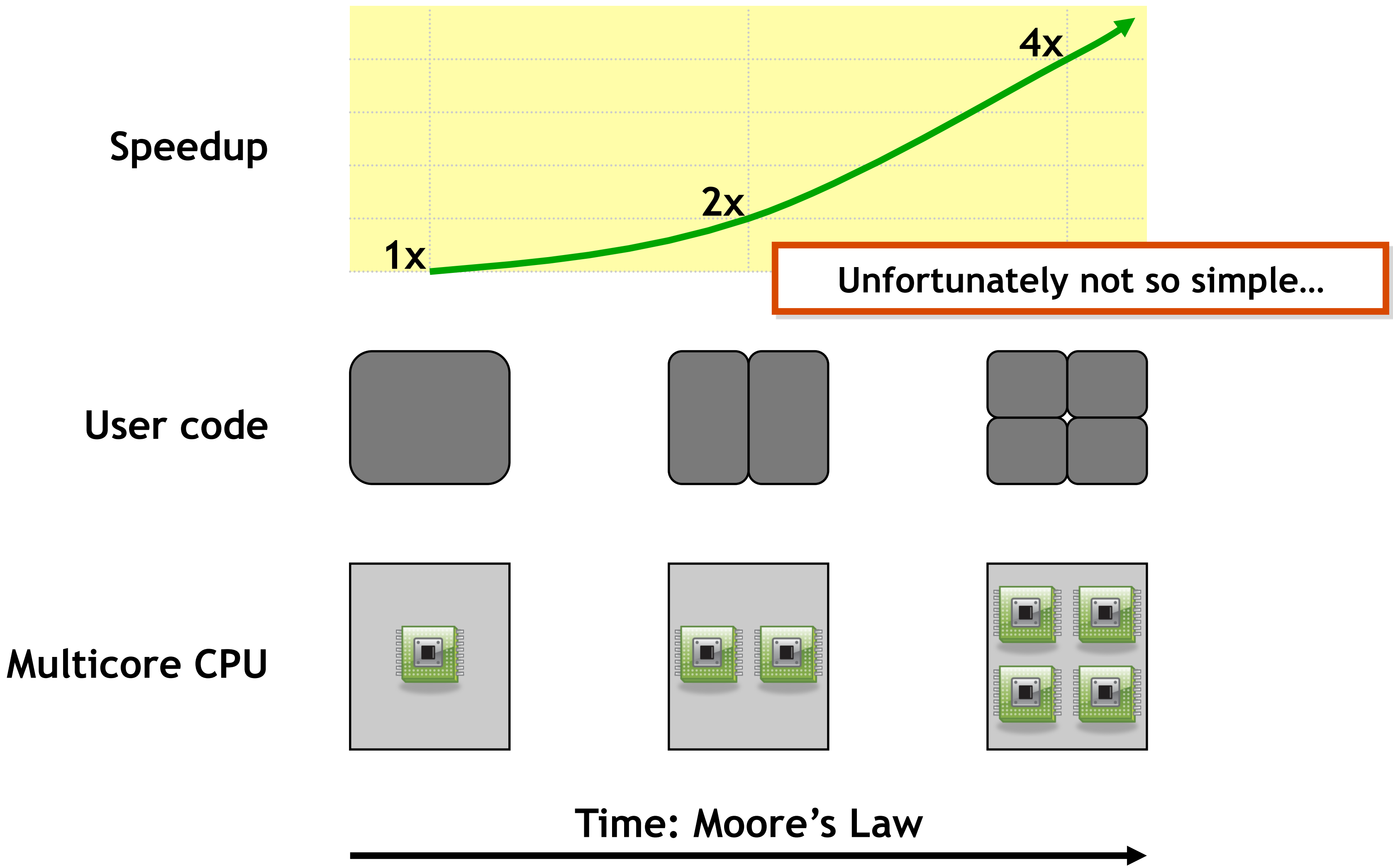
The free ride is over

- One of the biggest future software challenges: exploiting concurrency
  - Every programmer will have to deal with it
  - Concurrent programming is hard to get right

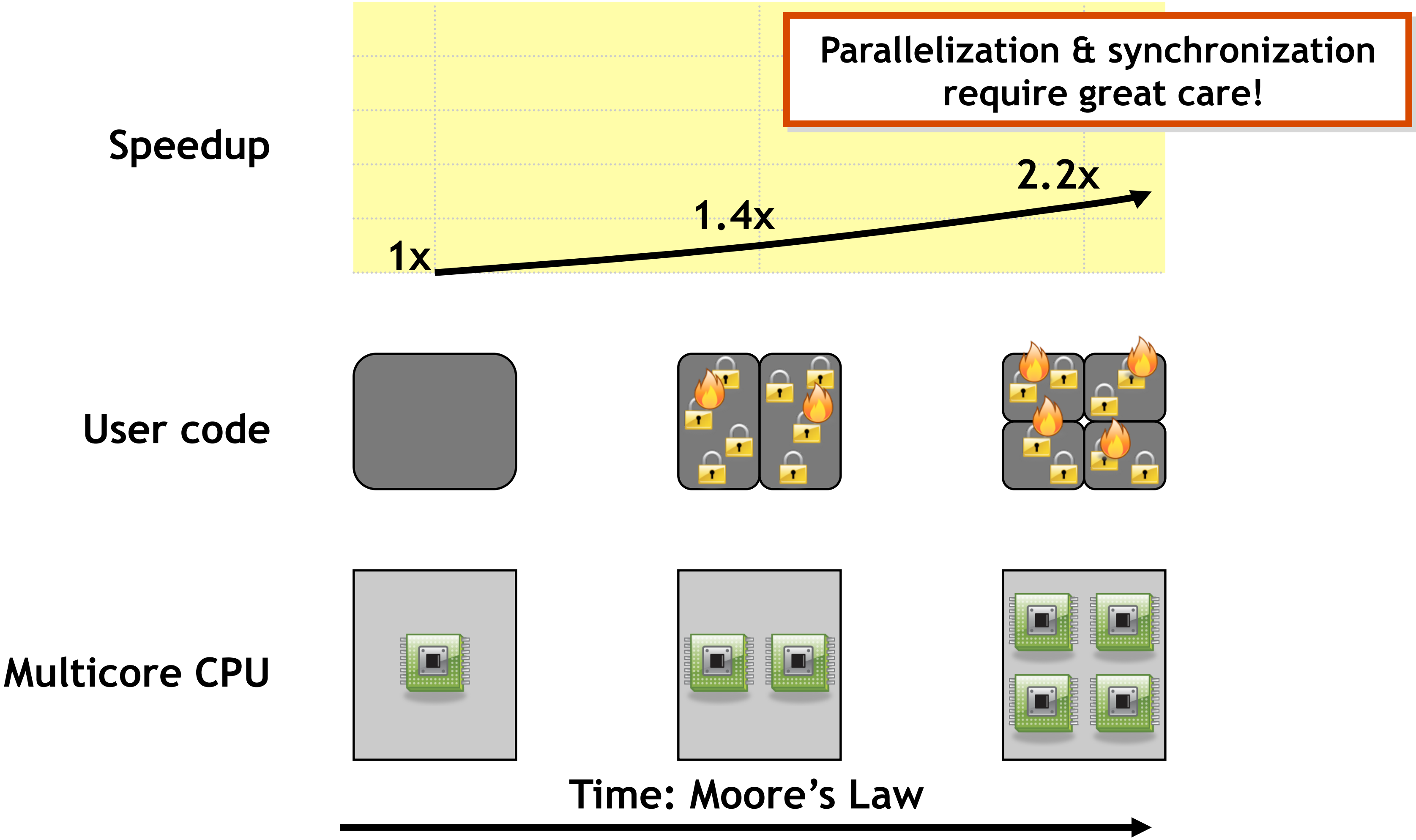
Traditional scaling



Ideal multicore scaling



Real-world scaling



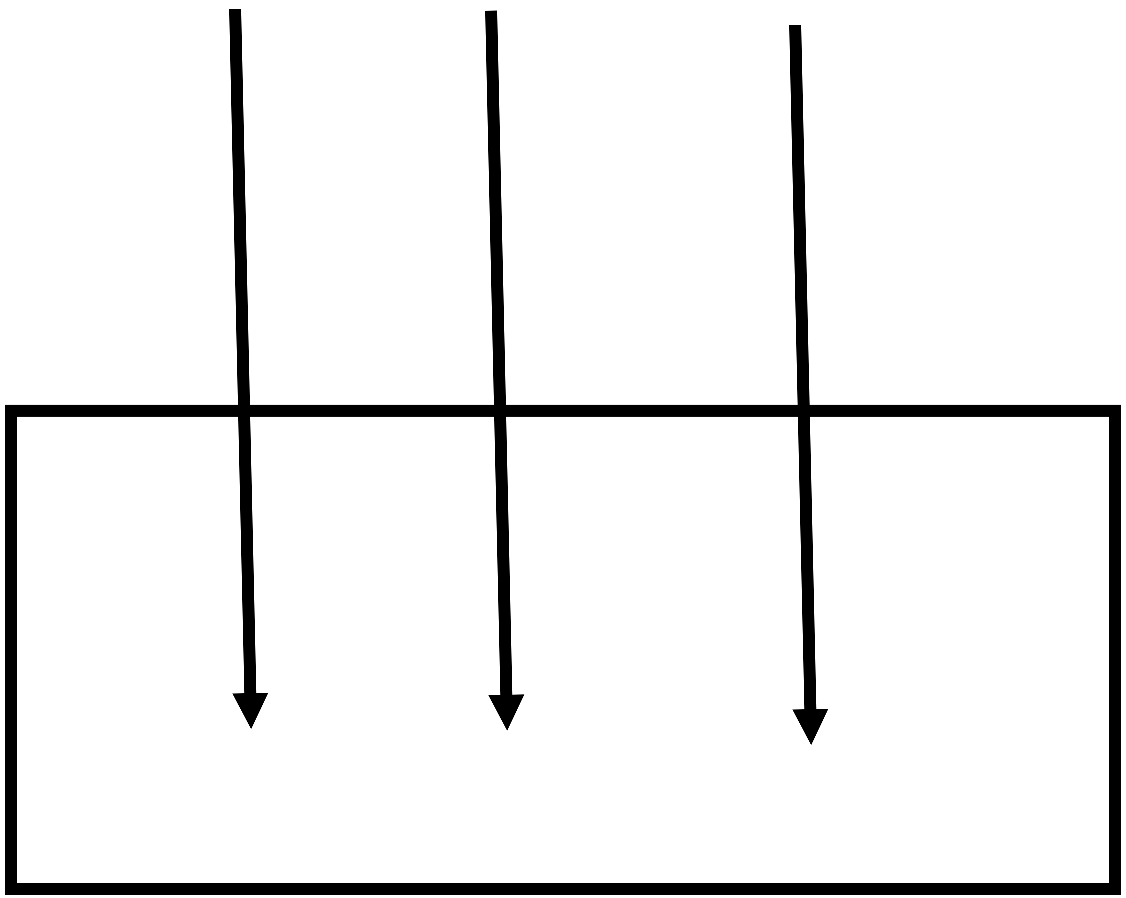
Real-world scaling

- *Forking processes is easy*

But…

- *Synchronizing accesses to shared objects is hard*

The key: shared objects

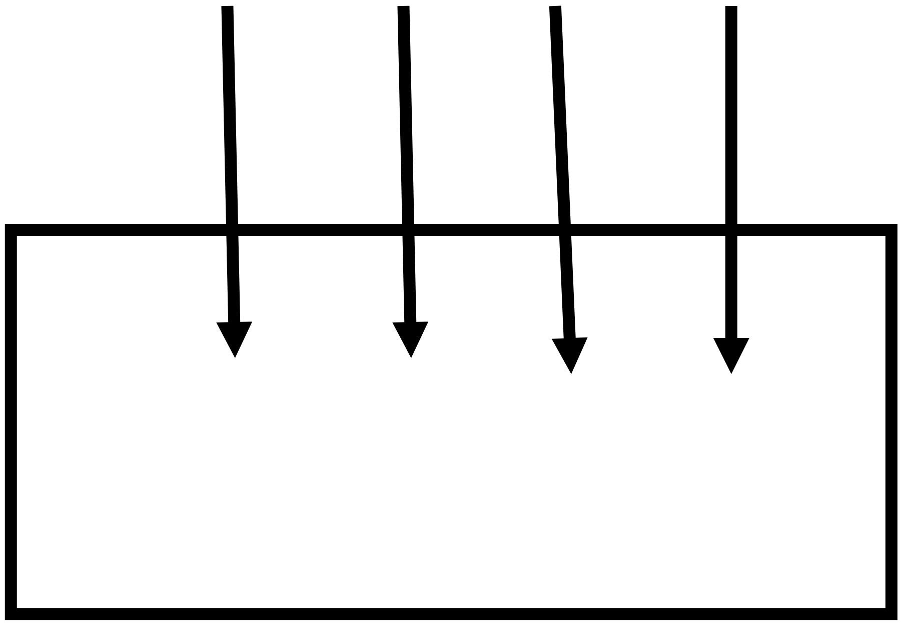


Counter

```
public class Counter

private long value;

public Counter(int i) { value = i;}

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

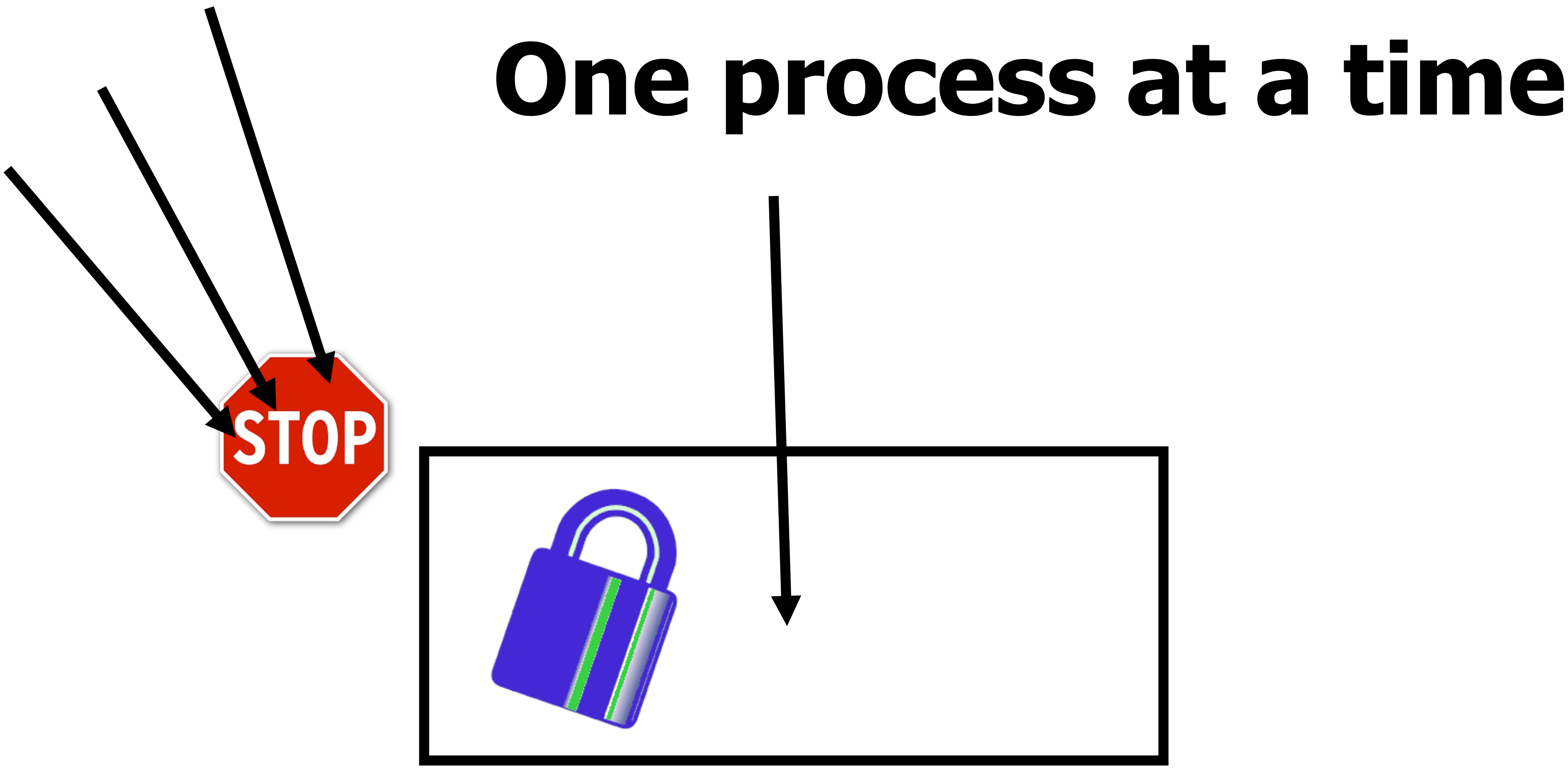
How to synchronize?

Concurrent processes



Shared object

Locking (mutual exclusion)



Locked object

Locking with compare&swap()

- A ***Compare&Swap*** object maintains a value x, init to ⊥, and y;

- It provides one operation: ***c&s(v,w);***

  ✓Sequential spec:

  - c&s(old,new)

    {y := x; if x = old then x := new; return(y)}

Locking with compare&swap()

```
lock() {
repeat until
unlocked = this.c&s(unlocked,locked)
}

unlock() {
         this.c&s(locked,unlocked)
     }
```

Locking with test&set()

- A ***test&set*** object maintains binary values x, init to 0, and y;

- It provides one operation: ***t&s()***

  - ✓Sequential spec:
  - ✓ t&s() {y := x; x: = 1; return(y);}

Locking with test&set()

```
lock() {
repeat until (0 = this.t&s());
}

unlock() {
         this.setState(0);
     }
```

Locking with test&set()

```
lock() {
while (true)
 {
 repeat until (0 = this.getState());
 if 0 = (this.t&s()) return(true);
 }
}

unlock() {
        this.setState(0);
     }
```

Explicit use of a lock

```
Lock l = ...;
    l.lock();
    try {
// access the resource protected by this lock
    } finally {
        l.unlock();
    }
```

Implicit use of a lock

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void getAndincrement()
 {
        c++; return c;
    }
    public synchronized int value() {
        return c;
    }
}
```

Problems with locks

- 50% of the bugs reported in Java come from the mis-use of « synchronized »

Concurrency conflicts are time -sensitive

They might never be detected before application deployment

Locks are fragile

- ***Blocking***
- ***Non-composable***

Locks are blocking

- A process holding a lock prevents all others from progressing: deadlock, livelock, priority inversion, etc.

Processes are asynchronous

- *Page faults*
- *Pre-emptions*
- *Failures*
- *Cache misses, …*

Processes are asynchronous

- A cache miss can delay a process by ten instructions
- A page fault by few millions
- An os preemption by hundreds of millions…

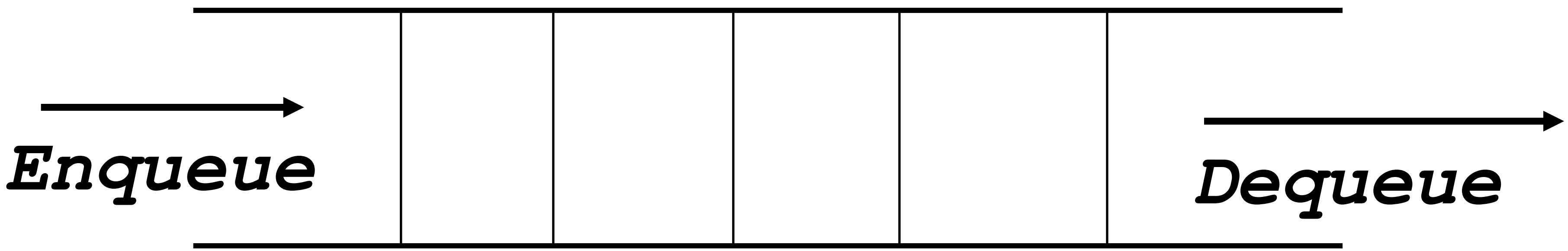
From the Linux kernel

```
/* * When a locked buffer is visible
  to the I/O layer * BH_Launder is
  set. This means before unlocking *
  we must clear BH_Launder,mb on alpha
  and then * clear BH_Lock, so no
  reader can see BH_Launder set * on
  an unlocked buffer and then risk to
 deadlock. */
```

Coarse grained locks => slow

Fine grained locks => errors

Double-ended queue



Fine-grained locking

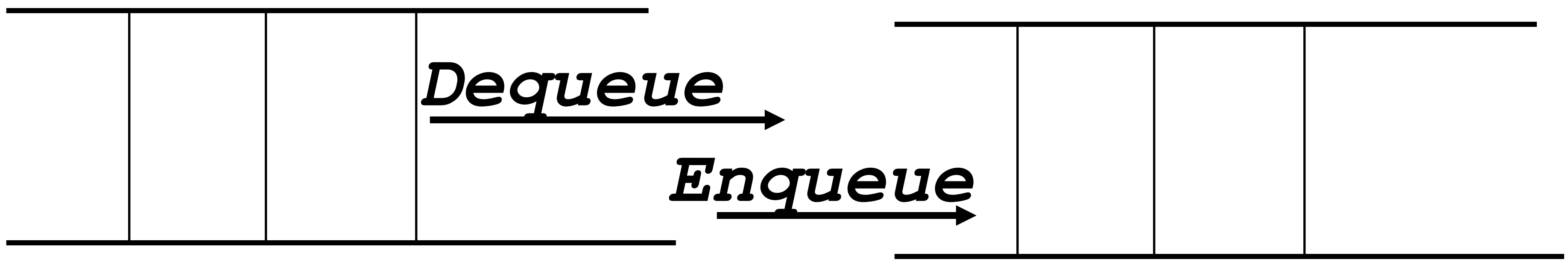
- It took two years for the Java Standards Committee to approve for inclusion in Java 5 class libraries a fine-grained locking-based implementation of a hash-table

- The implementation was devised by The Java concurrency expert

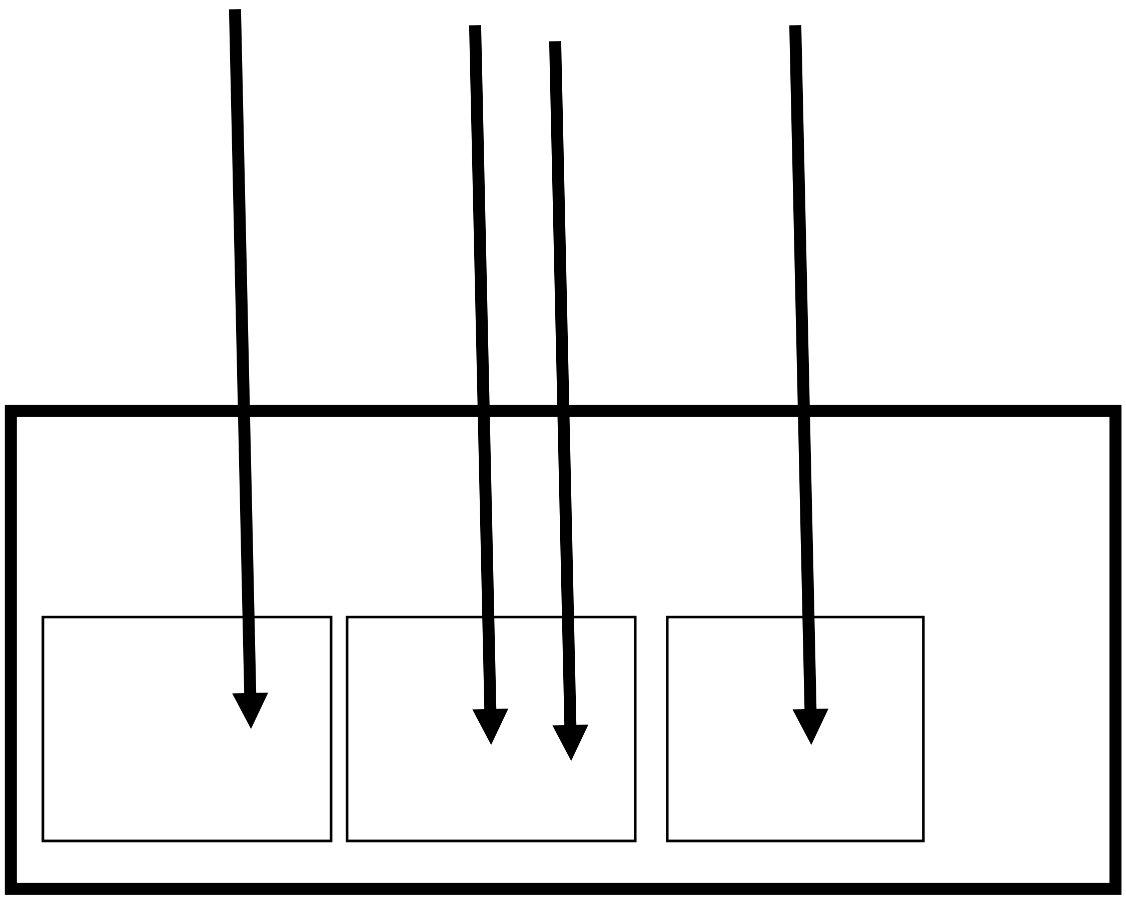
Locks are fragile

- ***Blocking***
- ***Non-composable***

Locks do not compose



Alternative to locking?



Wait-free computing

- ***Wait-freedom:*** every process that invokes an operation eventually returns from the invocation … unlike locking.

- ***Atomicity:*** every operation appears to execute instantaneously … as if the shared object was locked.

*This course presents the **principles** of **wait-free** computing…*