Multi-threaded programming

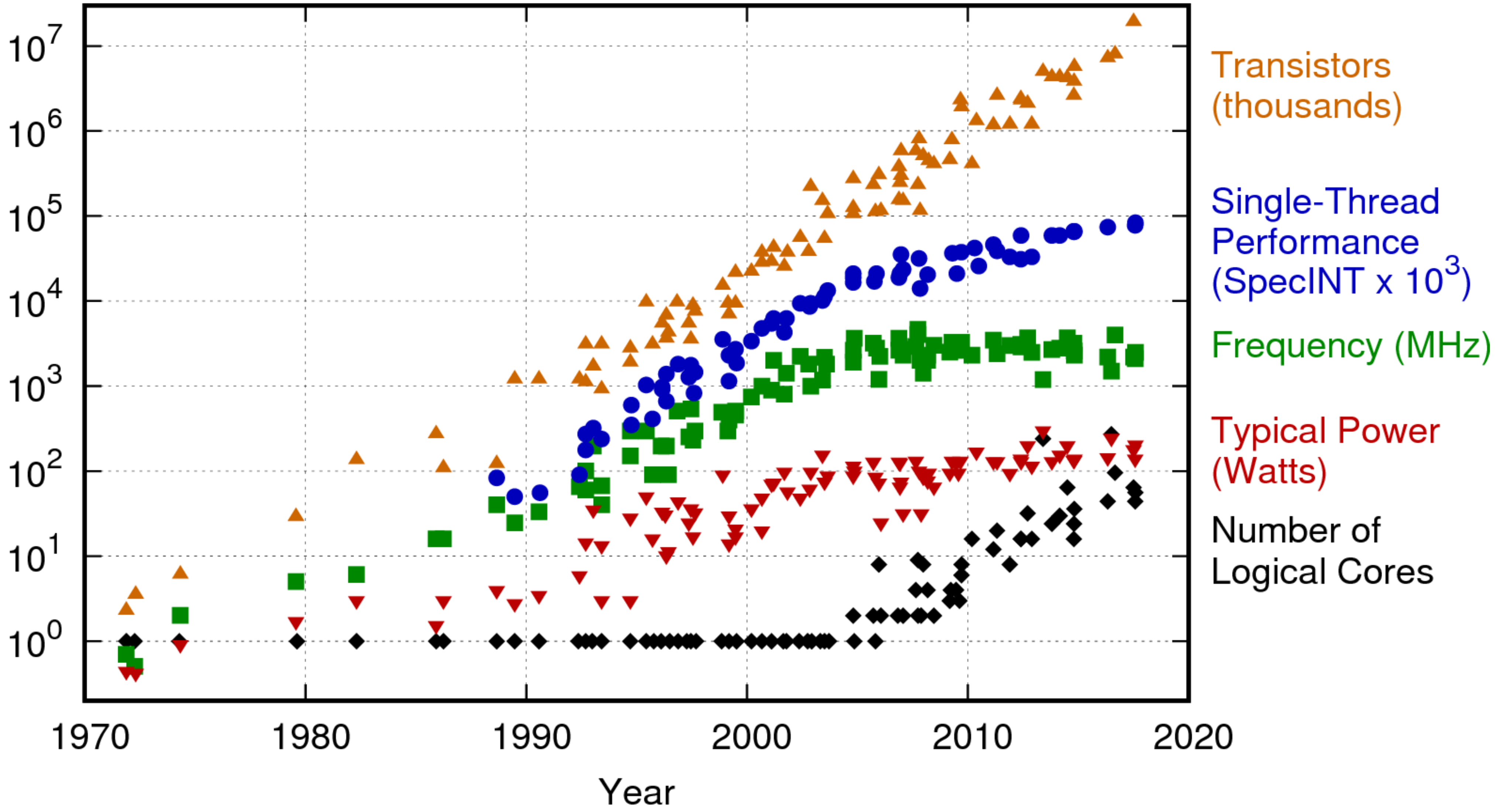
Thomas Schwarz, SJ

Preliminaries

- There are many different types of parallel execution
  - Very large instruction words etc.:
    - Execute many instructions at once
  - Pipelining / Arithmetic Sub-Units:
    - Execute many instructions at once
  - Single Instruction Multiple Data:
    - e.g. GUI
  - Multiple Instruction Multiple Data:
    - Multiple cores
    - Parallel / distributed programming using messages

Preliminaries

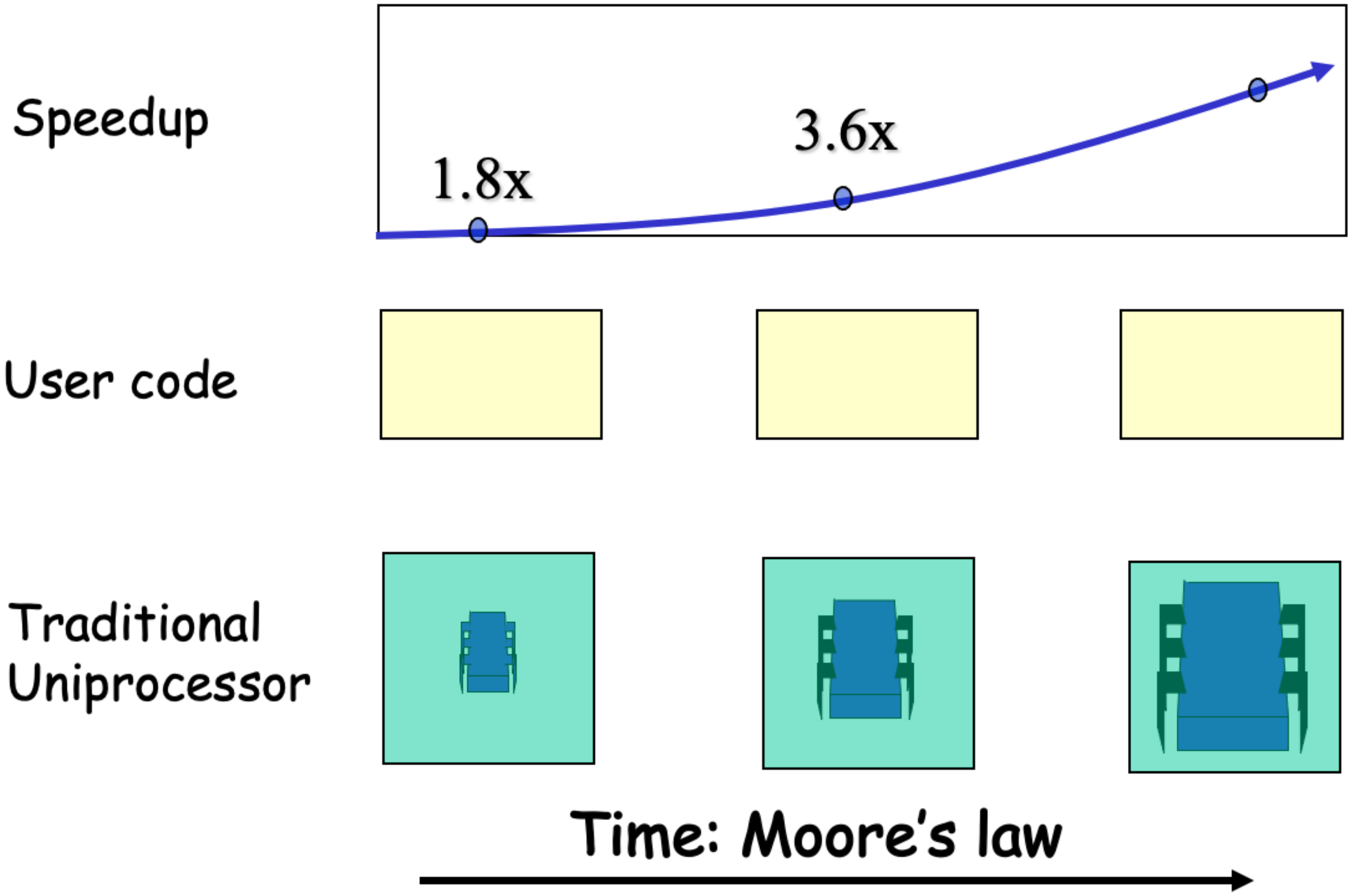
- Moore's Law



42 Years of Microprocessor Trend Data

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

The Way it Used to Be



Speedup

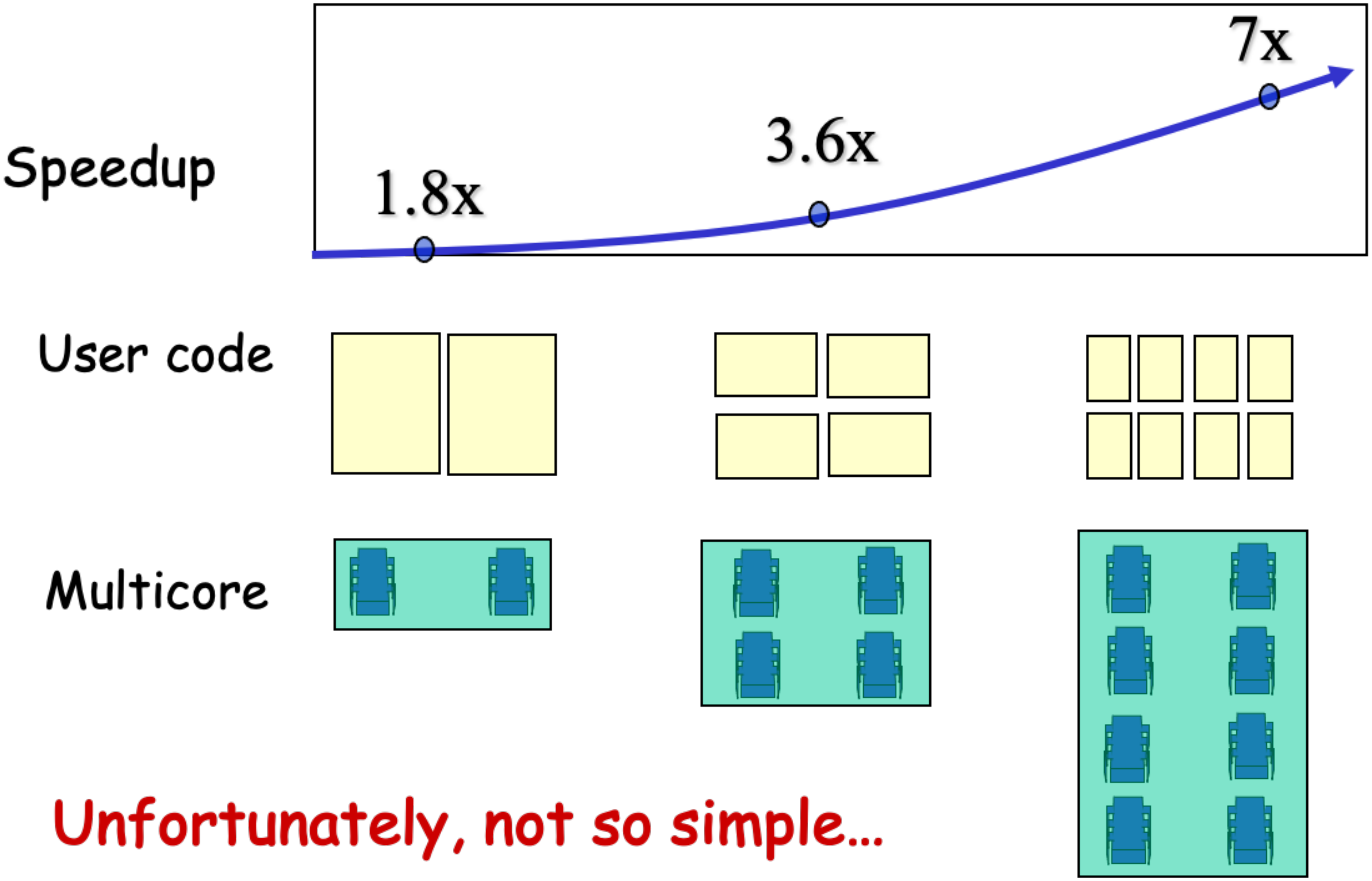
User code

Traditional
Uniprocessor

Time: Moore's law

Herlihy, Shavit, Luchangco, Spear: The Art of Multiprocessor Programming

Preliminaries



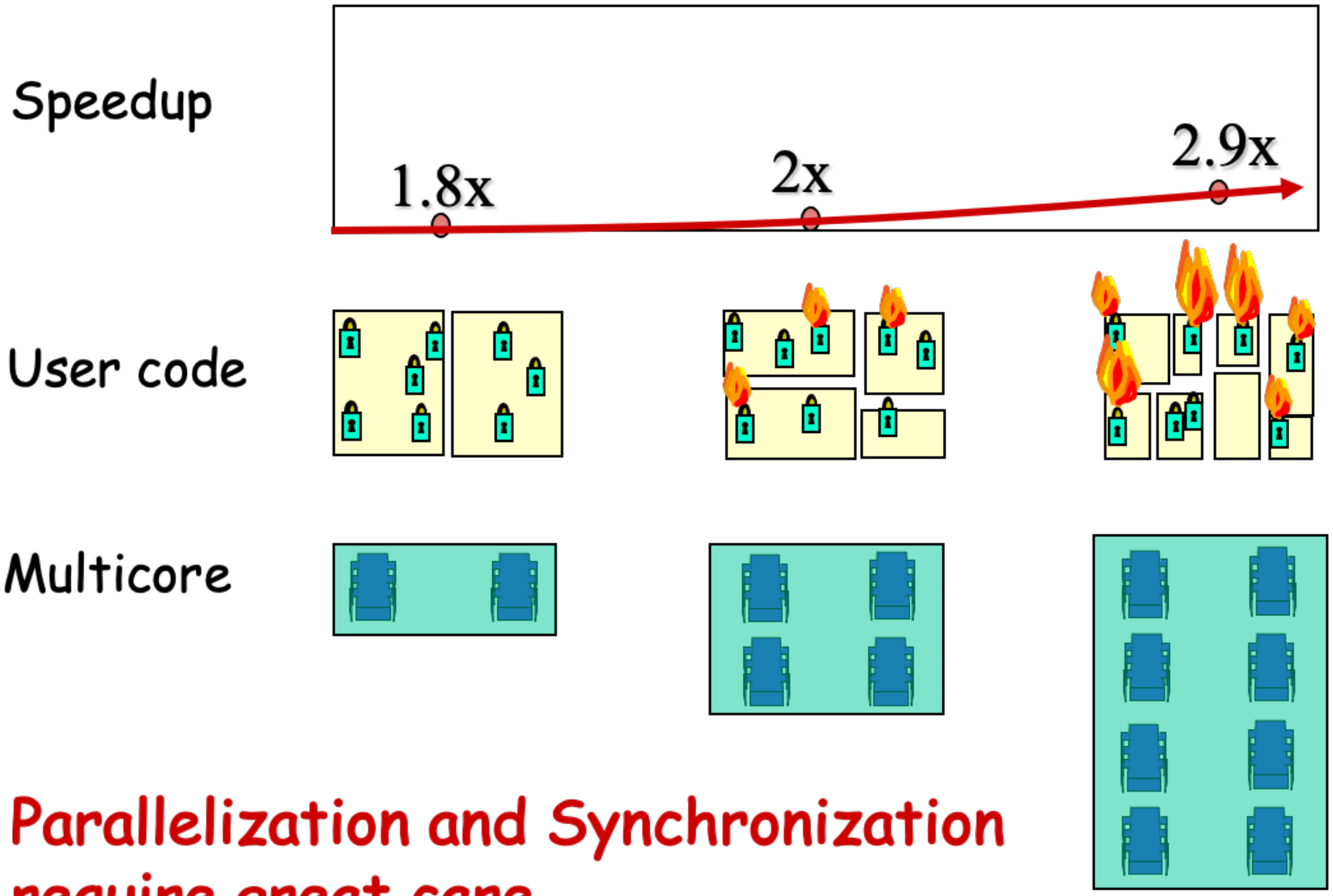
Speedup

User code

Multicore

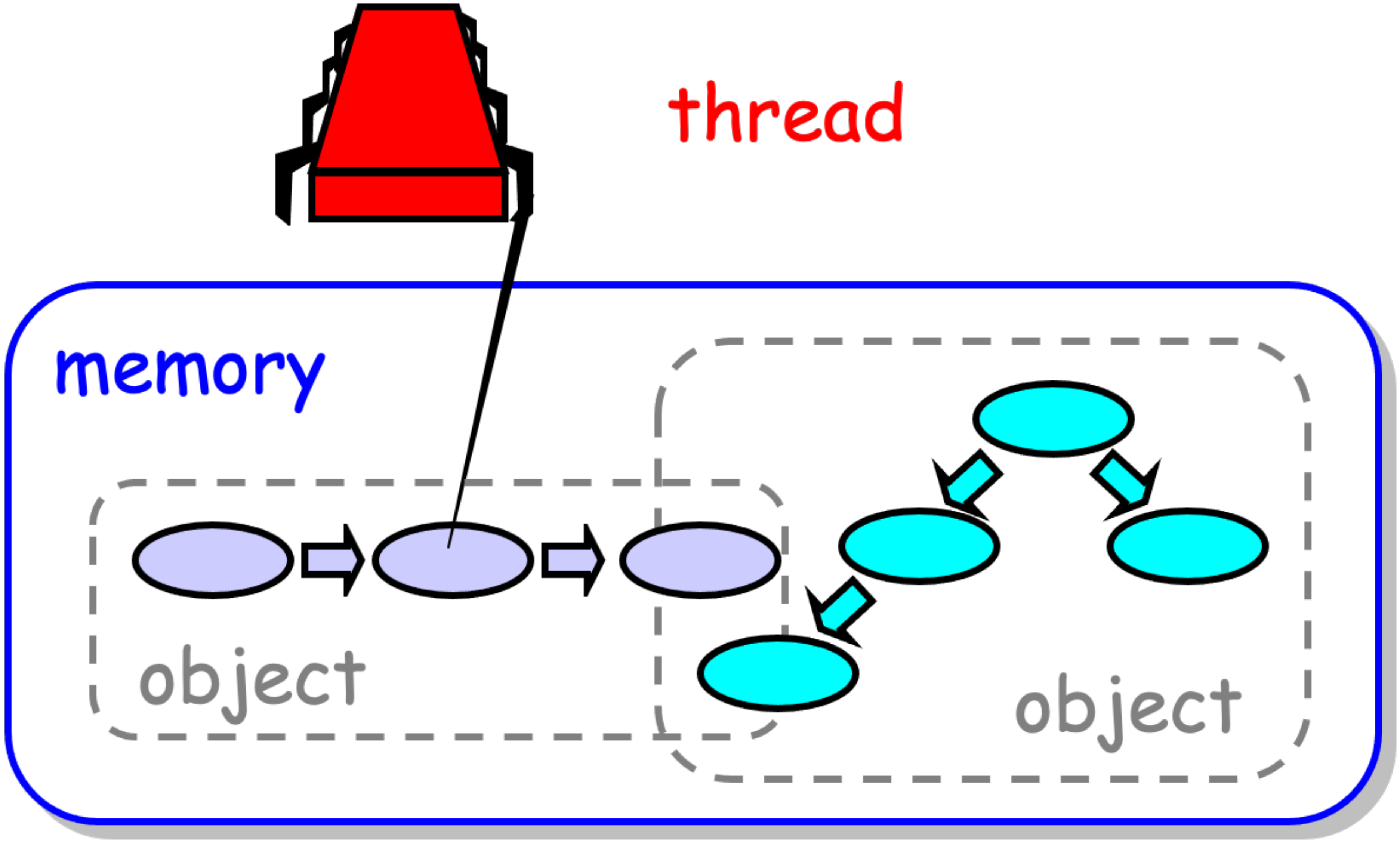
Unfortunately, not so simple…

Preliminaries

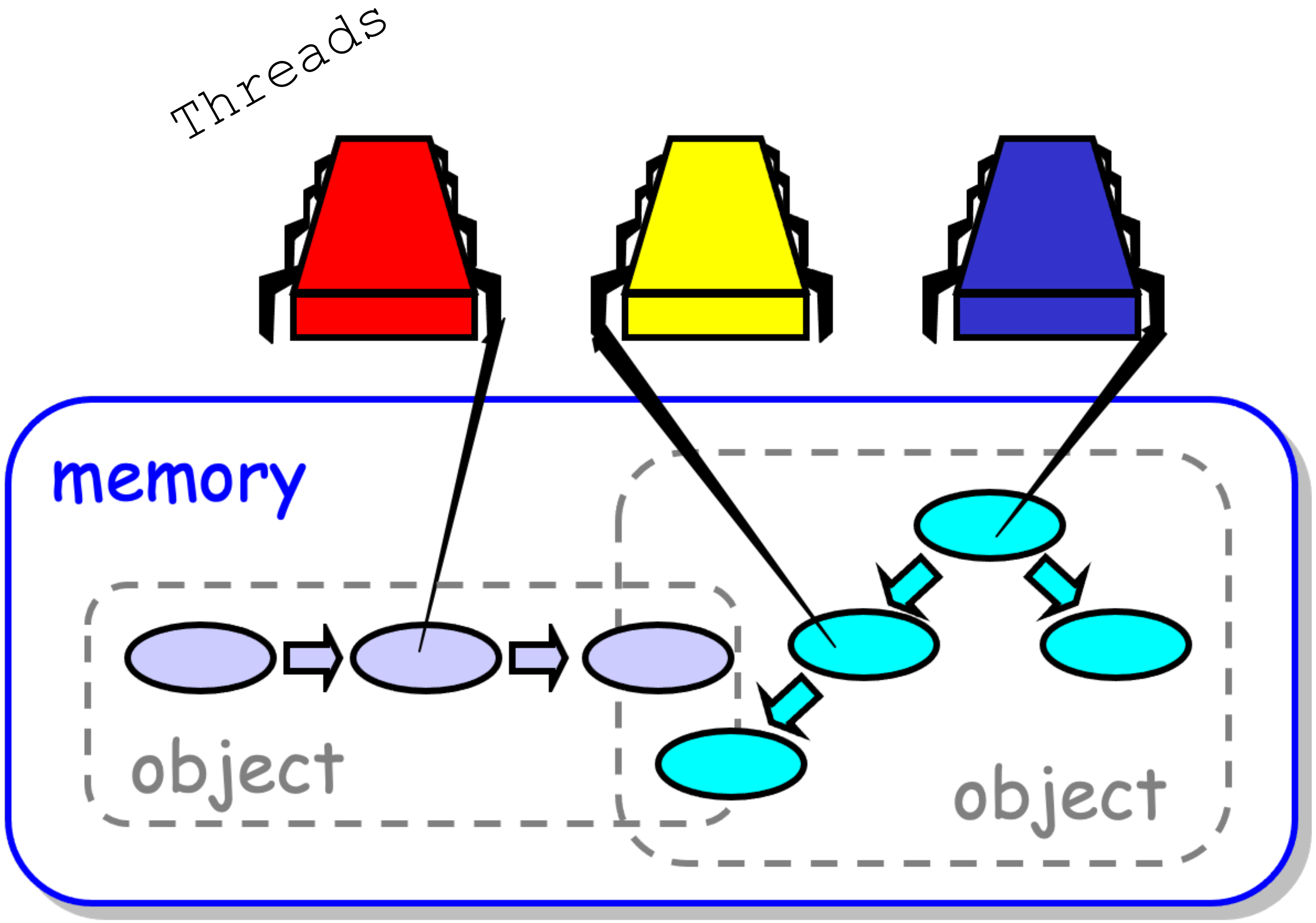


Parallelization and Synchronization
require great care...

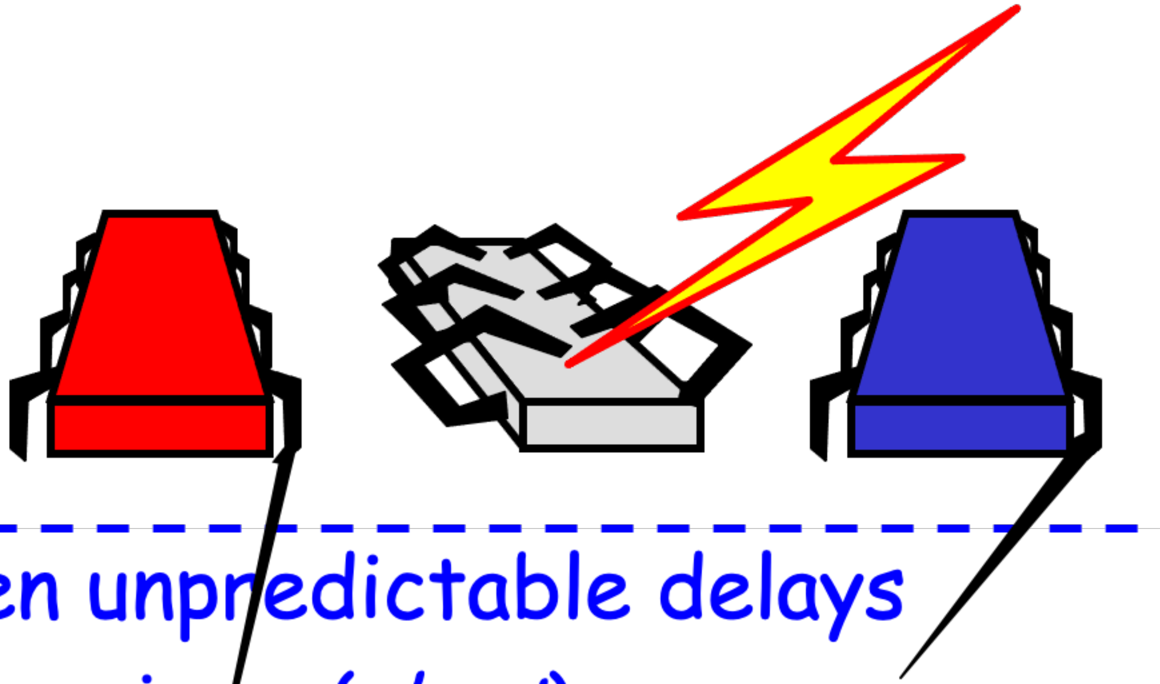
Single Thread Programming



Multiple Thread Programming



Asynchrony



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

Execution Model

- Multiple threads
  - Sometimes called processes
- Single shared memory
  - (This might change in the future)
- Objects live in memory
  - Unpredictable asynchronous delays

Preliminaries

- Simple example:
  - Print out all prime numbers between 10 000 000 000 and 10 100 000 000 on a 10-processor architecture
  - Load Balancing:
    - Assume a probabilistic primality test
    - Assign number n to processor $n \pmod{10}$

Preliminaries

- Problem:
  - We can predict that even processors will not print out anything!

Preliminaries

- Simple example:
  - Print out all prime numbers between 10 000 000 000 and 10 100 000 000 on a 10-processor architecture
  - Load Balancing:
    - Assign each processor a range

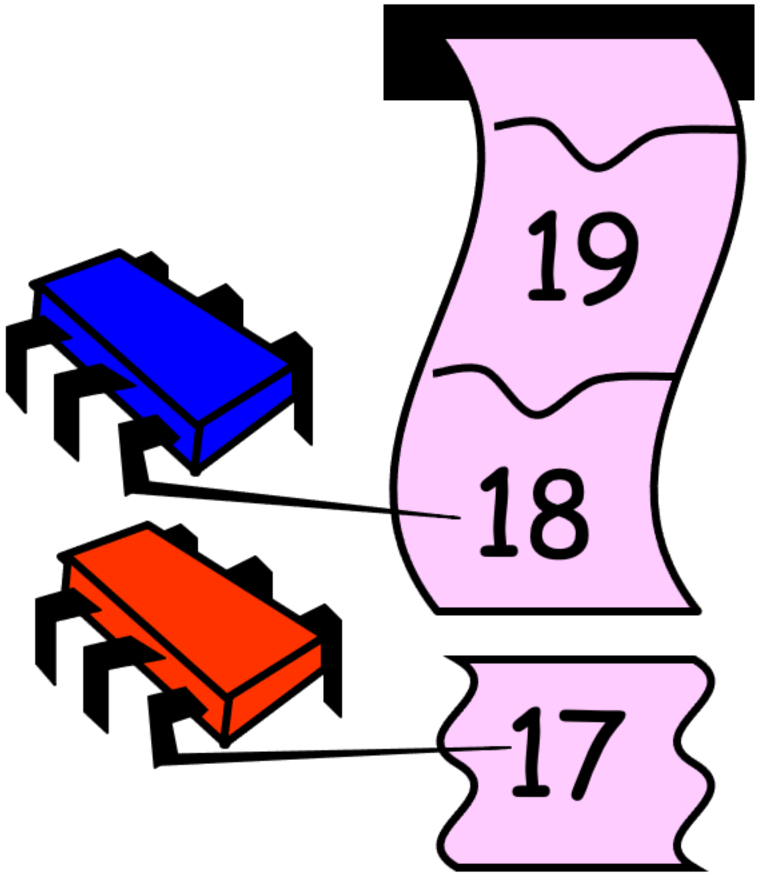
```
void primePrint {
  int i = ThreadID.get(); // IDs in {0..9}
  for (j = base + i*spread/10; j<base +(i+1)*spread/10;j++)
    if (isPrime(j))
      print(j);
  }
}
```

Preliminaries

- Problem:
    - Ranges have different numbers of prime numbers
    - Workload is not evenly spread
    - Need dynamic *load balancing*

Preliminaries

- Shared Counter
  - Each thread works on a number
  - After finishing, gets a new number



Preliminaries

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

void primePrint {
  long j = 1000000000;
  while (j < 1001000000) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

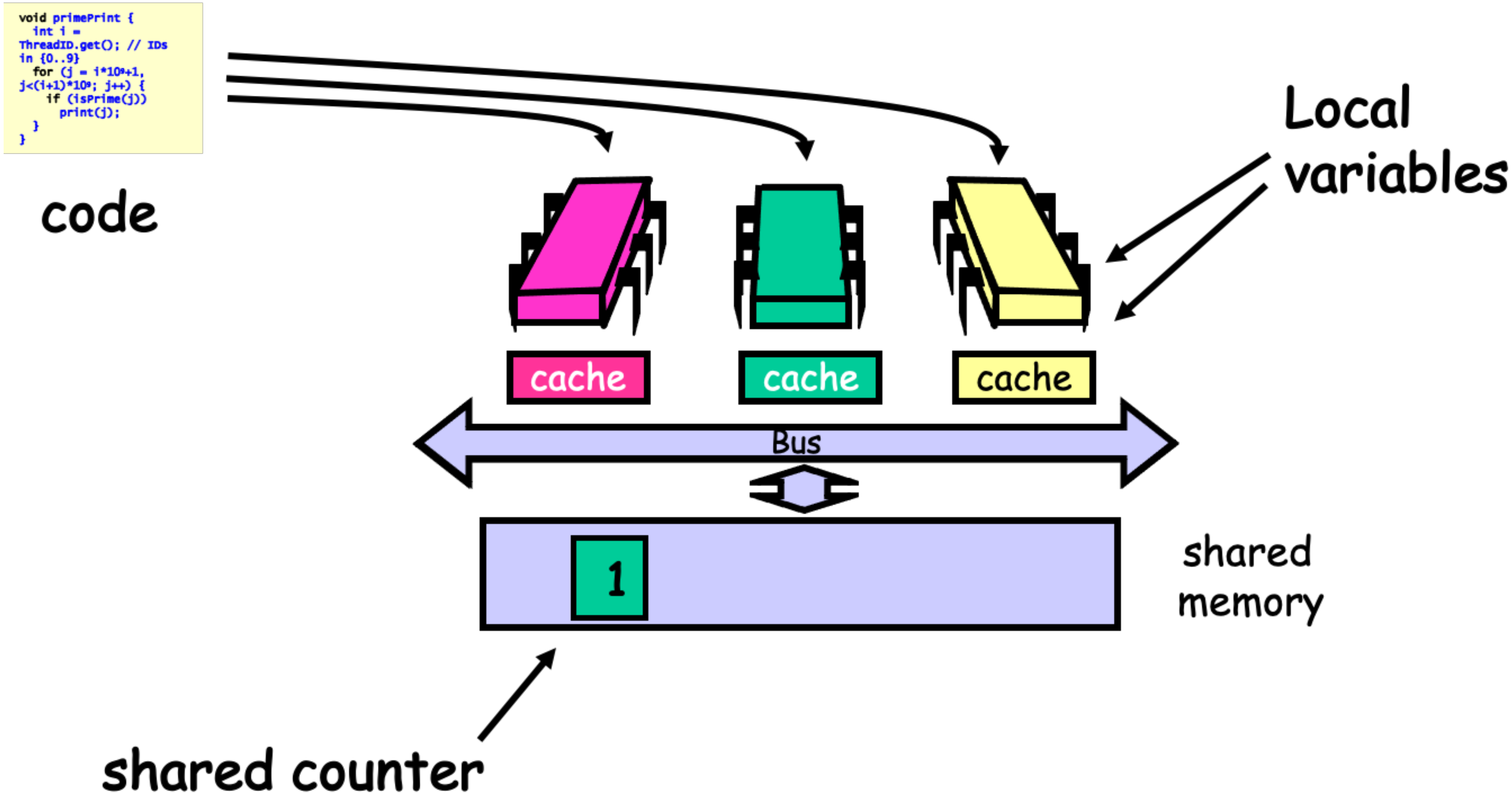
Preliminaries

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

void primePrint {
  long j = 1000000000;
  while (j < 1001000000) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Shared Counter Object

Where Things Reside



```
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);
  }
}
```

Implementing Counter

- What is wrong here?

```
public class Counter {
  private long value;

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

Implementing Counter

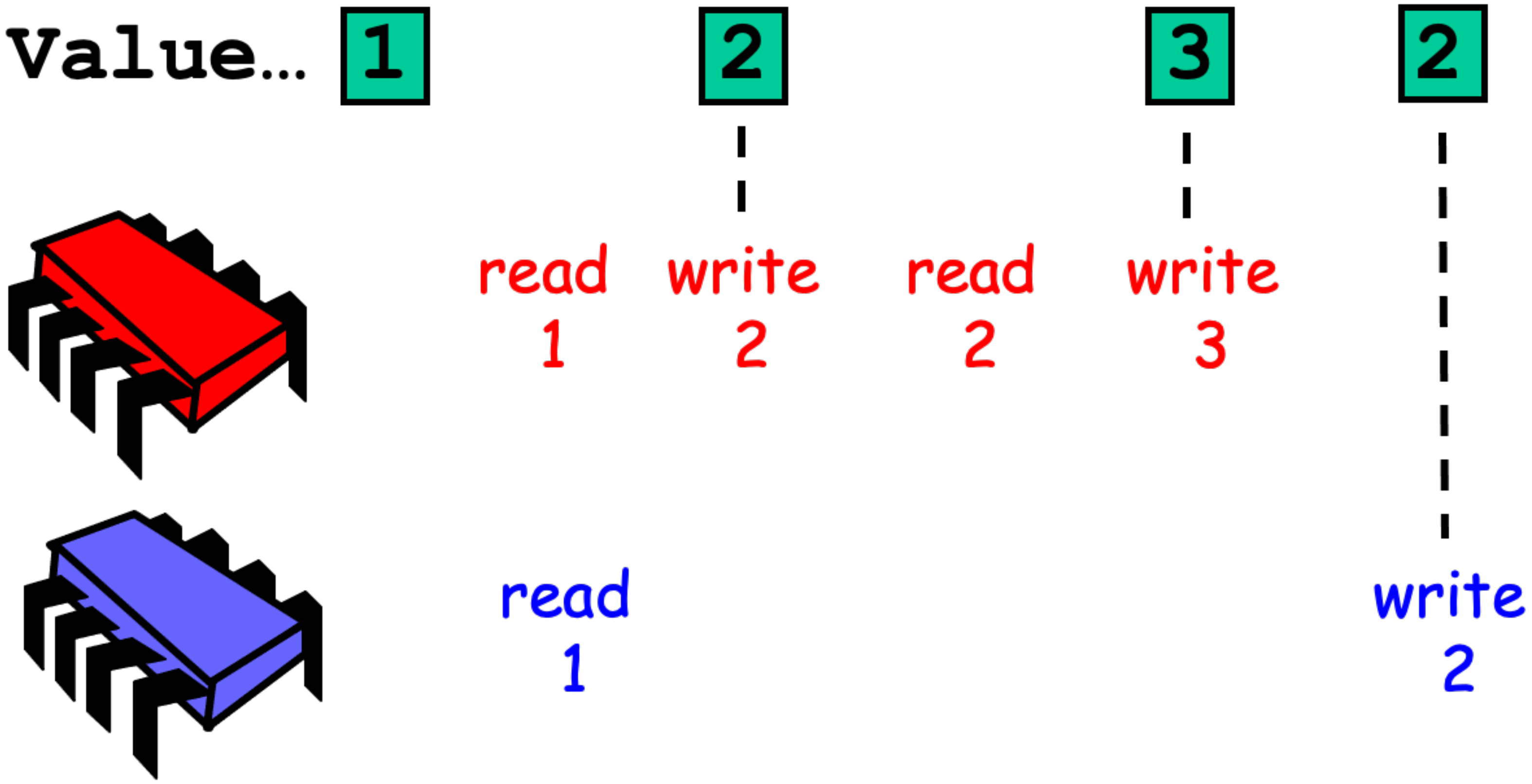
```
public class Counter {
  private long value;

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

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

Implementing Counter

- So this can happen



Value... 1 2 3 2

read 1

write 2

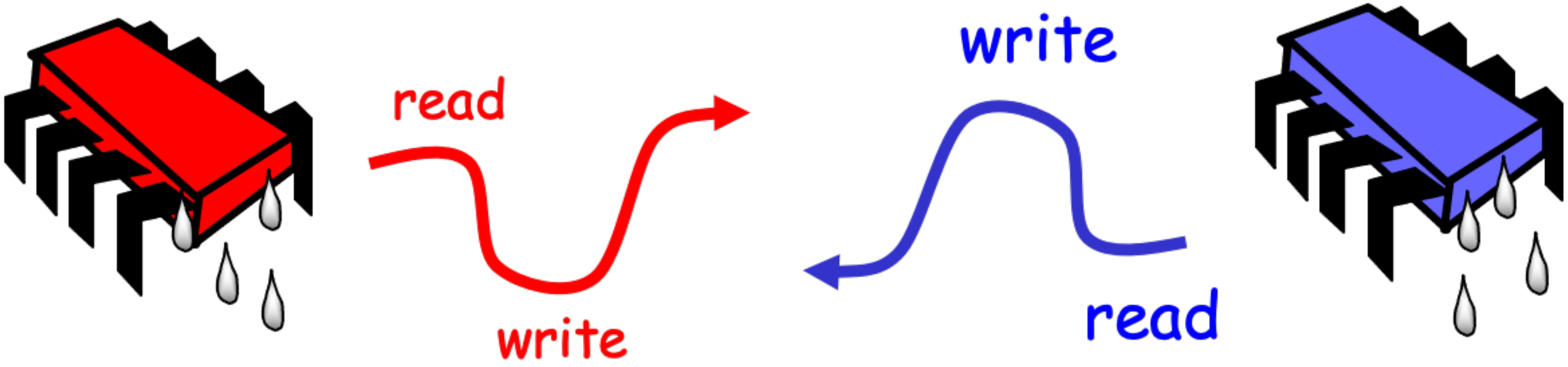
read 2

write 3

read 1

write 2

Implementing Counter



If we could only glue reads and writes…

Implementing Counter

```
public class Counter {
  private long value;

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

ReadModifyWrite instruction

Implementing Counter

- C++

```
std::atomic<>::fetch_add()
std::atomic<>::fetch_sub()
std::atomic<>::fetch_and()
std::atomic<>::fetch_or()
std::atomic<>::fetch_xor()
std::atomic<>::exchange()
std::atomic<>::compare_exchange_strong()
std::atomic<>::compare_exchange_weak()
```

Implementing Counter

- In Java:

```
public class Counter {
  private long value;

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

Implementing Counter

- Mutual Exclusion
- Only one thread can enter

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

Formalizing the Problem

- Safety properties:
    - Nothing bad ever happens
- Liveness properties:
    - Something good eventually happens

Formalizing the Problem

- Mutual Exclusion:
  - Now to threads are even in the critical region
  - Safety Property
- No deadlock:
  - If only one thread wants it, it gets in
  - If more threads want it, one gets in
  - Liveness Property
- No starvation:
  - "Your turn"

Communication about Shared Object

- Threads cannot see what other threads are doing
- Attempt 1: Receive permission from other threads before entering critical space
  - Threads might be non-responsive (e.g. blocked)
- Communication must be:
  - persistent
  - not transient

Communication about Shared Object

- Threads cannot see what other threads are doing
  - Can protocol:
    –Cans on Alice’s windowsill
    –Strings lead to Bob’s house
    –Bob pulls strings, knocks over cans
    •Gotchas
      –Cans cannot be reused
      –Bob runs out of cans
  - Interpretation
    - Interrupts do not work
      - Sender sets fixed bit in receiver’s space
      - Receiver resets bit when ready
      - Requires unbounded number of interrupt bits

Communication about Shared Object

- Flag protocol

  -
    ```
    Alice:
     raises flag
     waits for Bob's flag to be down
     enters critical section
     leaves critical section
     lowers flag
    ```

  -
    ```
    Bob:
     raises flag
     waits until Alice's flag is down
     enters critical section
     leaves critical section
     lowers flag
    ```

DANGER!

Communication about Shared Object

- Simple flag protocol:
  - Causes starvation if both see the other's flag before they enter the critical section

Communication about Shared Object

- Improved Flag Protocol

  -
    ```
    Alice:
     raises flag
     waits for Bob's flag to be down
     enters critical section
     leaves critical section
     lowers flag
    ```

  -
    ```
    Bob:
     raises flag
     if Alice's flag is up:
         lowers flag
         waits for Alice's flag to go down
         raises flag
     enters critical section
     leaves critical section
     lowers flag
    ```

Communication about Shared Object

- Can you show that two threads never end up in the critical section?

Communication about Shared Object

- Can you show that two threads never end up in the critical section?
  - Flag principle:
    - IF both raise their flag and then look at the other flag, one of them will see a flag raised

Communication about Shared Object

- Assume that both Alice and Bob are in the critical section
- When Alice looked last:
  - Her flag was up
    - She never lowers her flag unless she leaves the critical section
  - Bob's flag was down
- When Bob looked for the last time:
  - Alice's flag is down, so this is after Alice looked last
  - But then Alice's flag was raised. Contradiction!

Communication about Shared Object

- This protocol can be implemented with two Booleans
- Does not assume that setting or unsetting a Boolean is instantaneous

Communication about Shared Object

- Deadlock free:
  - One will eventually enter the critical section
- Proof:
  - Assume both want to enter
  - Both Alice and Bob raise their flags
    - If Bob looks before Alice raises her flag
      - He has raised his flag already, Alice sees the flag and waits, whereas Bob enters
    - If Bob looks after Alice raises her flag
      - He lowers his flag and does not enter

Communication about Shared Object

- Starvation free?
  - No: If Alice is busy entering and leaving, Bob is shut out

Communication about Shared Object

- Waiting:
    - If Alice is delayed after raising her flag, Bob cannot do anything

Producer Consumer Problems

- Bob produces items
- Alice processes (consumes) items
- Need to prevent Bob writing an item while Alice starts processing it

Producer Consumer Problems

- Can protocol
  - Alice:

```
waits until can is down
accesses section
leaves section
puts can up
```

  - Bob:

```
waits until can is up
accesses section
leaves section
pulls can down
```

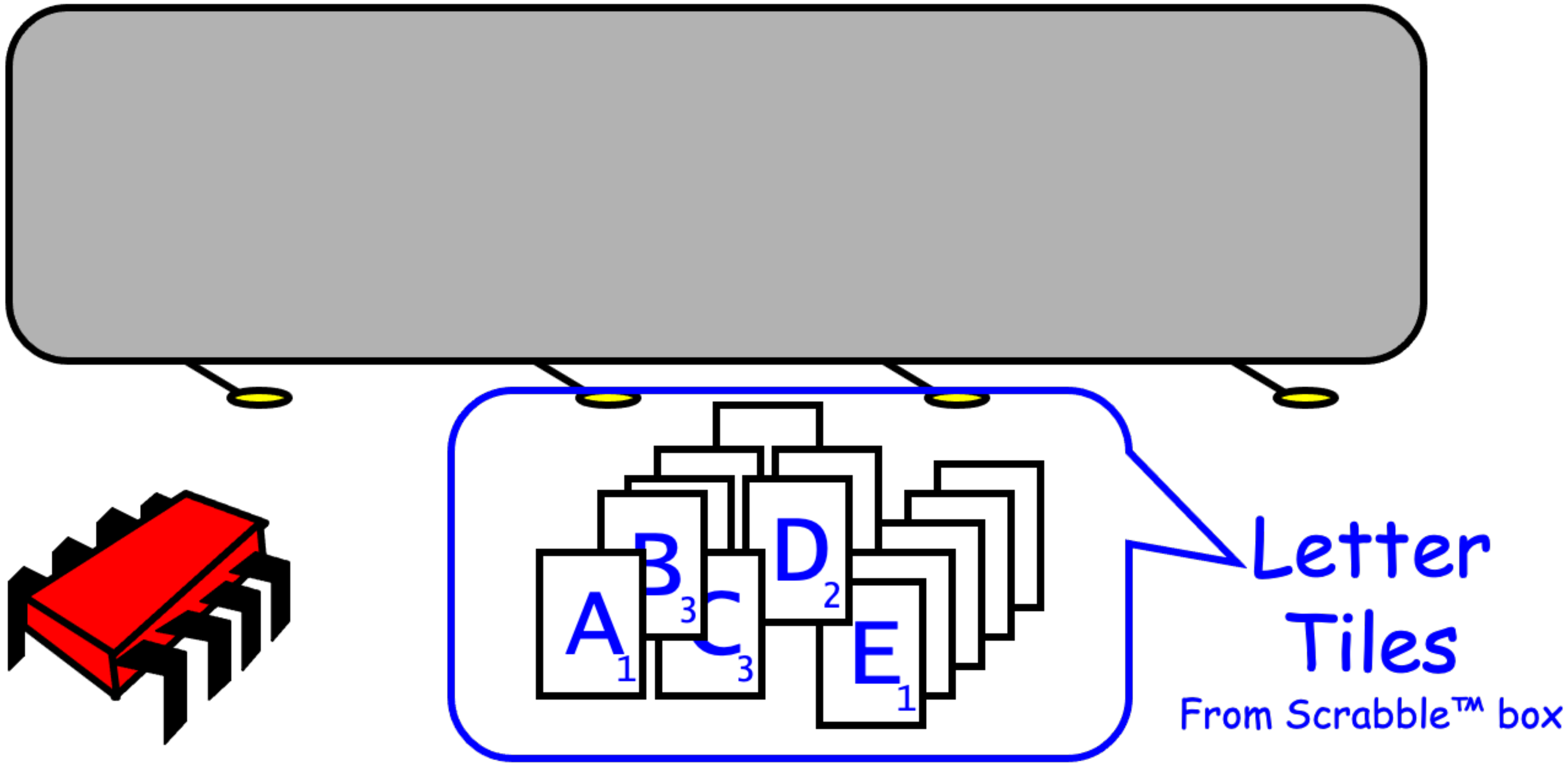
Producer Consumer Problems

- Mutual exclusion:
  - By induction:
    - At time 0: Nobody is in the critical section, can is up
    - Time *t* to *t*+1:
      - If can is up: Only Bob can enter, then leave, then pull can down
      - If can is down: Only Alice can enter, then leave, then put can up.
    - qed

Producer Consumer Problems

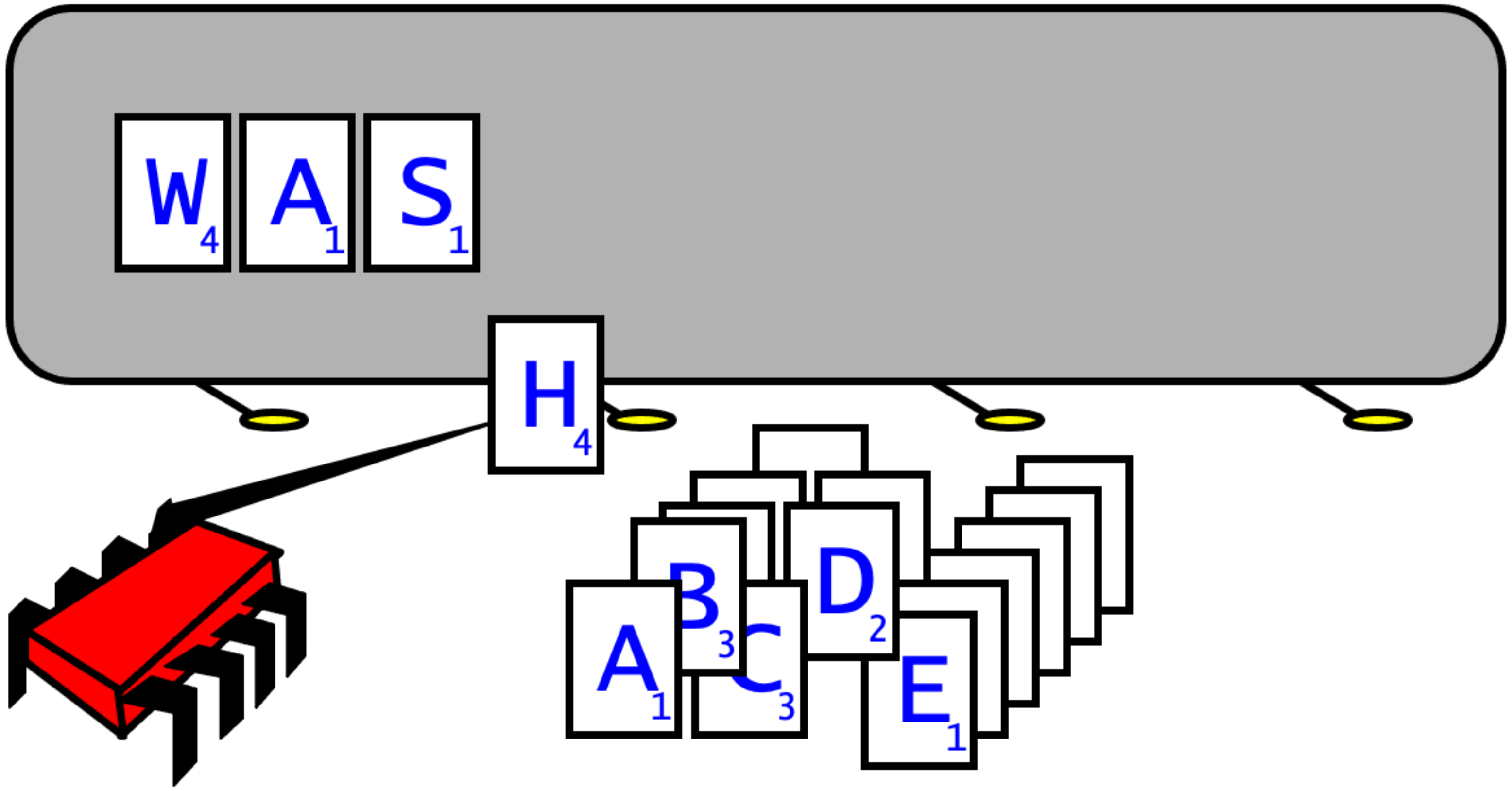
- Prove starvation freedom

Readers - Writers Problem



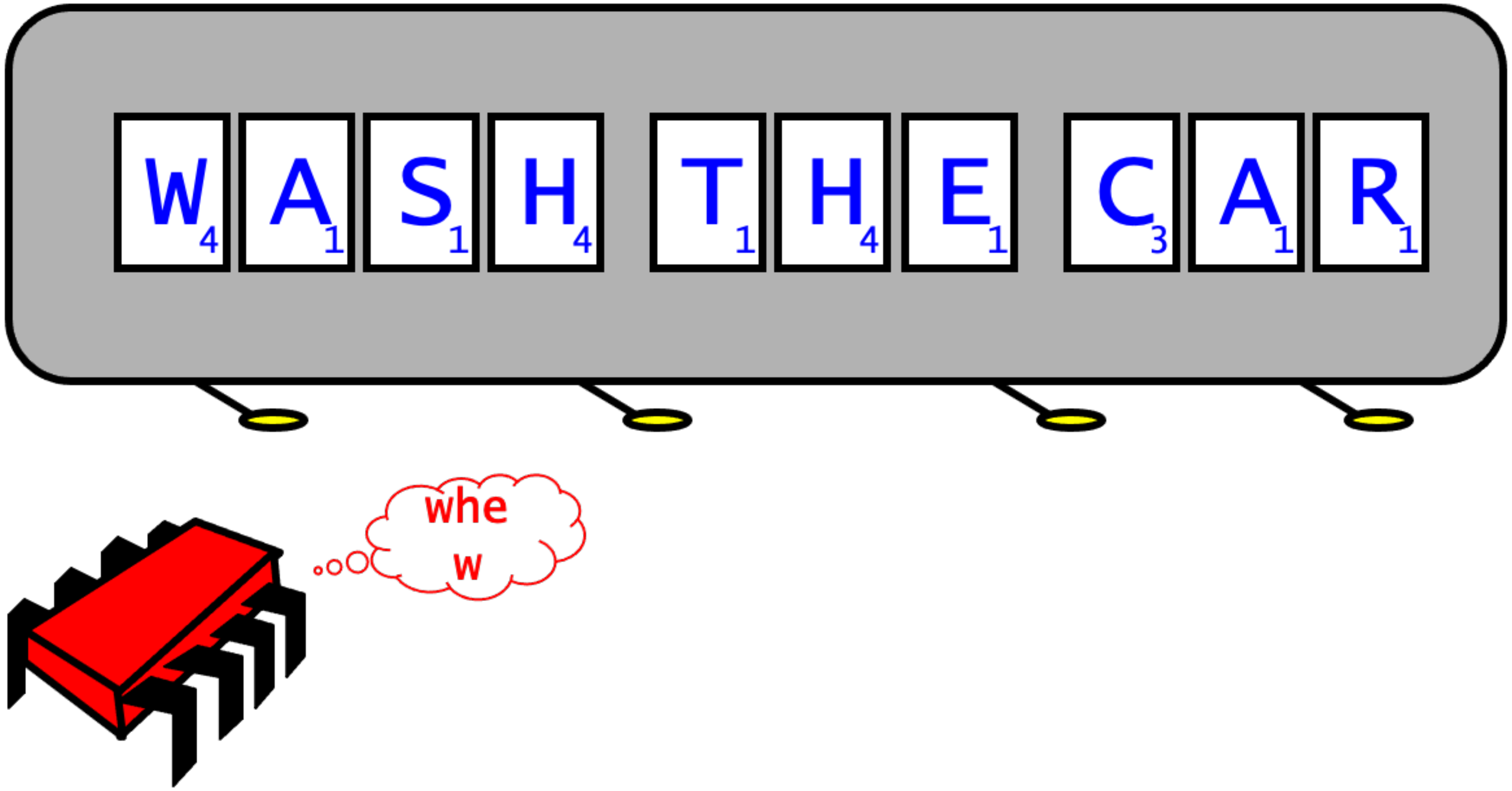
We are writing to a large bill-board

Readers - Writers Problem

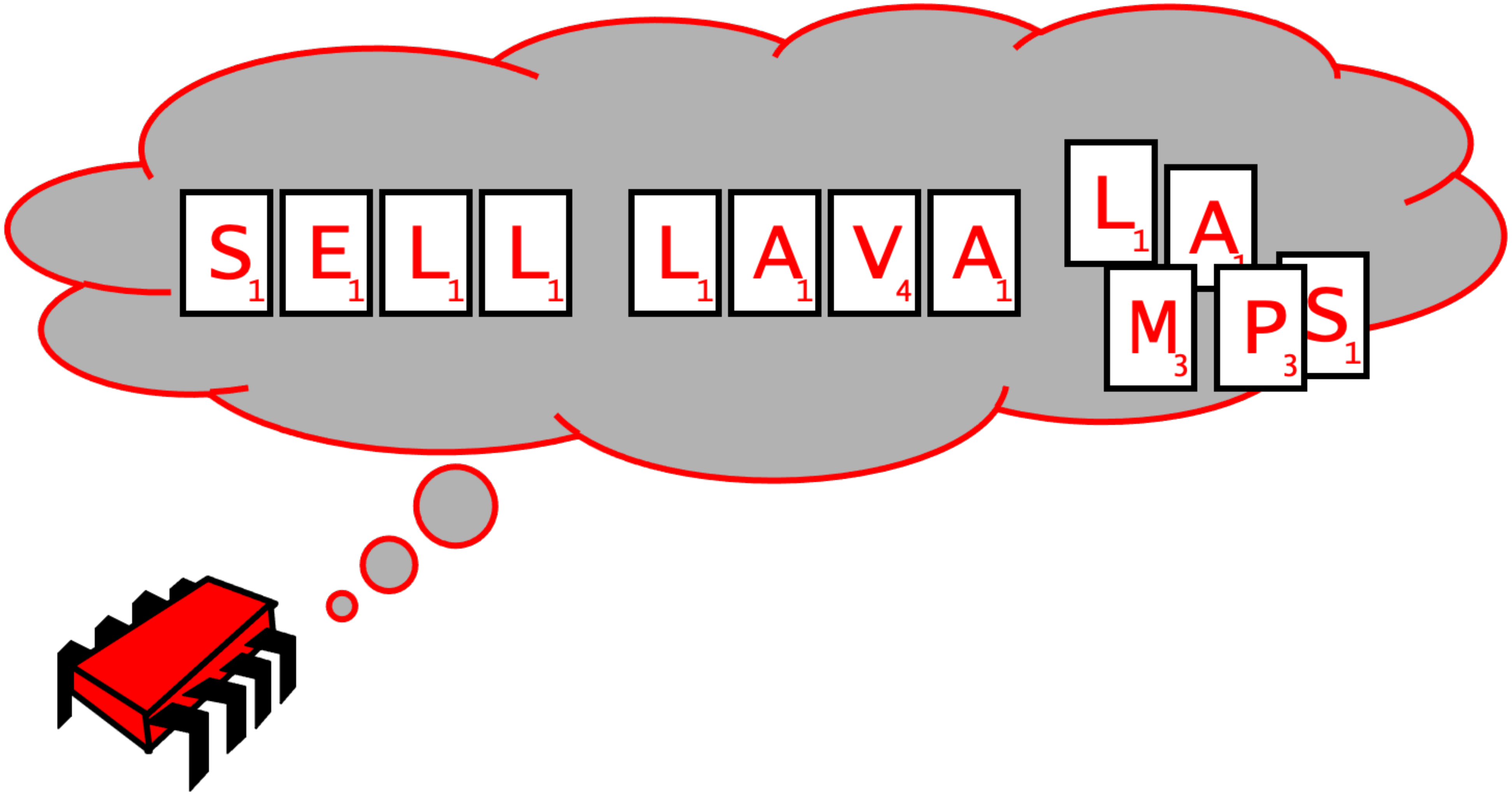


By necessity, we only write one letter at a time

Readers - Writers Problem

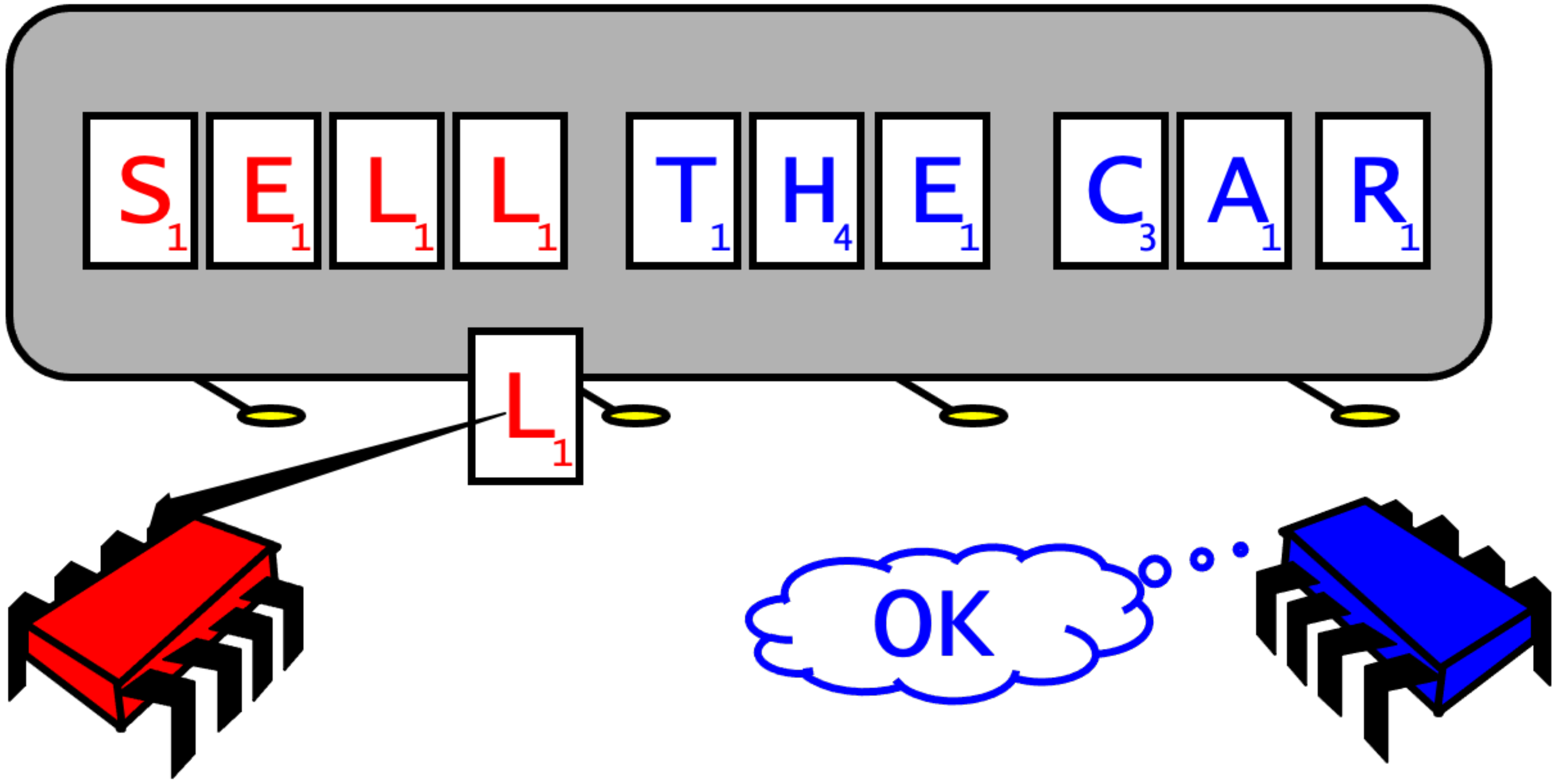


Readers - Writers Problem



Without coordination, bad things can happen

Readers - Writers Problem



Without coordination, bad things can happen

Readers - Writers Problem

- Simple solution:
  - Can use the flag protocol:
    - Only Alice or Bob can access the billboard
      - Alice can only read complete sentences
  - Can use the can protocol:
    - Bob produces complete sentences
    - Alice consumes complete sentences

Readers - Writers Problem

- But these protocols involve waiting
    - We can find several protocols that are *wait-free.*

Amdahls Law

- Speedup: $\dfrac{\text{Time before parallelization}}{\text{Time after parallelization}}$
- Assume *n* processors:
  - Ideal speed-up: *n* times
  - Assume portion ρ of program is parallelizable
  - Best speed-up possible is
  - $$\frac{1}{\frac{\rho}{n} + (1 - \rho)}$$

Amdahls Law

Amdahls Law

Amdahls Law