Distributed Systems

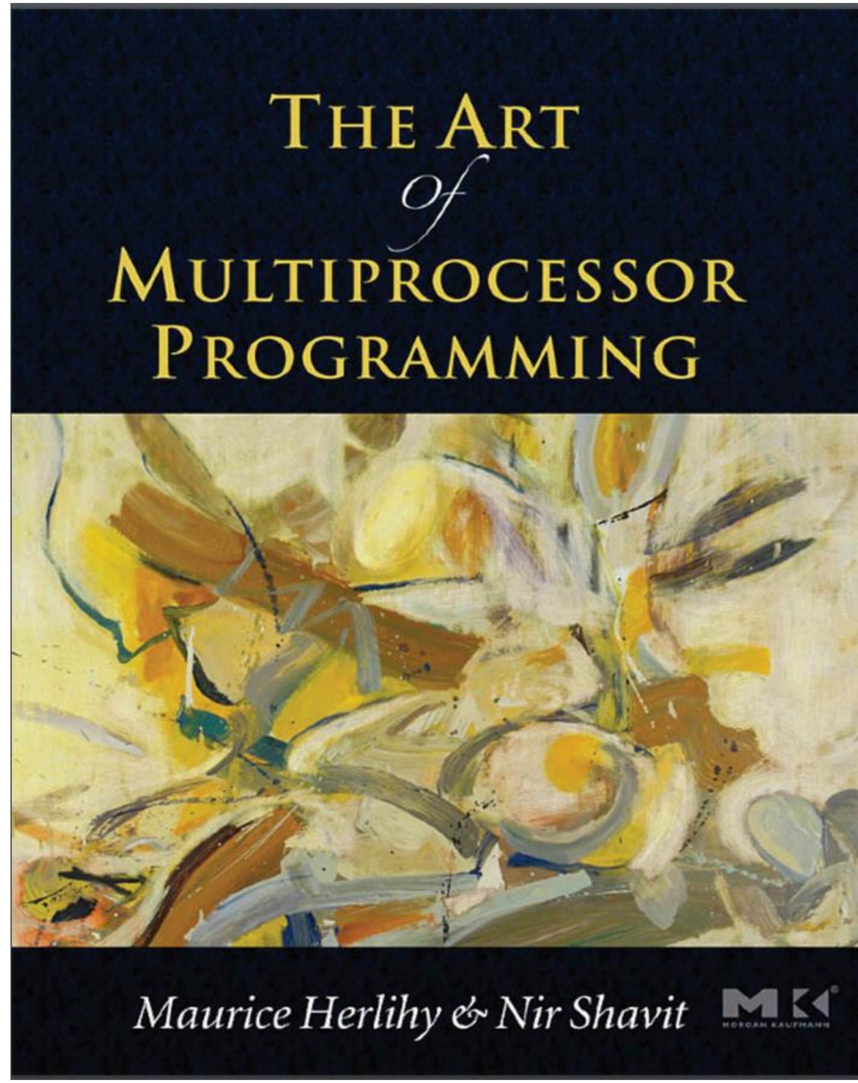
The Art of Multiprocessor Programming

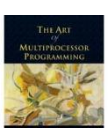
Dept. of CSE
염 헌 영
Spring 2019

- Textbook
  - by Maurice Herlihy & Nir Shavit

- This material is a slight modification from the

- Course Material in
  http://cs.brown.edu/courses/csci1760/lectures.shtml

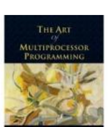




Teaching Staff

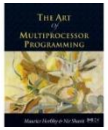
- Instructor : Yeom, Heonyoung (염헌영)
  - Email : yeom AT snu.ac.kr
  - Office : 302-321
  - Office Hours : T Th 1-2 pm

- TAs
  - 임희락 rockylim AT snu.ac.kr
  - Office : 302-311-2



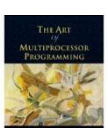
When you need help

- Class eTL
  - Lecture slides
  - Q & A



수업시간

- 화/목 11:00-12:15 (302동 106호)
- 6/6(현충일) 수업합니다.
- 중간고사1   4/4(목)
- 중간고사2   5/14(21)(화)
- 기말고사   6/18 (화)



평가

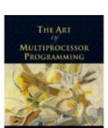
- 출석/태도 : 15%
- 숙제 (6-7) : 15%
- 중간고사 : 15% + 15%
- 기말고사 : 20%
- 텀프로젝트 : 20%
  - 수업시간에 배운 내용을 토대로 기존의 OS/Application 중에서 multicore/multiprocessor Scalability 문제가 있는 부분을 파악해서 이를 고치는 방법을 제시하고 구현할 것.

- 조교 : 임희락 (rockylim@snu.ac.kr)

- ETL 에 접속해서 수업자료/숙제 확인할 것.

Class Rules

- No open laptops
  - Unless you are asked to do something…
- No cell phones in my sight
  - Better have them mute if they are in your possesion.
- Use common senses !
- No more begging for grade changes !
  - I have to report the incidents to the dean !!!



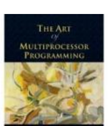
2016 성적

- 최초수강신청인원 : **40**
- 수강신청변경후 인원 : **45**
- 수강취소 : **12**
- 최종인원 : **33**

A+ : 3　　A0 : 3　　A- : 6

B+ : 5　　B0 : 10　　B- : 4

C0 : 2



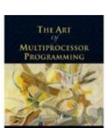
2017 성적

- 최초수강신청인원 : **54**
- 수강신청변경후 인원 : **50**
- 수강취소 : **10**
- 최종인원 : **40**

A+ : 5   A0 : 7   A- : 7

B+ : 7   B0 : 6   B- : 7

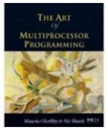
C0 : 1



2018 성적

- 최초수강신청인원 : **42**
- 수강신청변경후 인원 : **37**
- 수강취소 : **9**
- 최종인원 : **28**

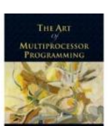
A+ : 3	A0 : 6	A- : 6
B+ : 6	B0 : 3	B- : 4



2019

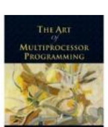
- 최초수강신청인원 : 38
- 수강신청변경후 인원 : ??
- 수강취소 :
- 최종인원 :

A+ :	A0 :	A- :
B+ :	B0 :	B- :
C+ :	C0 :	

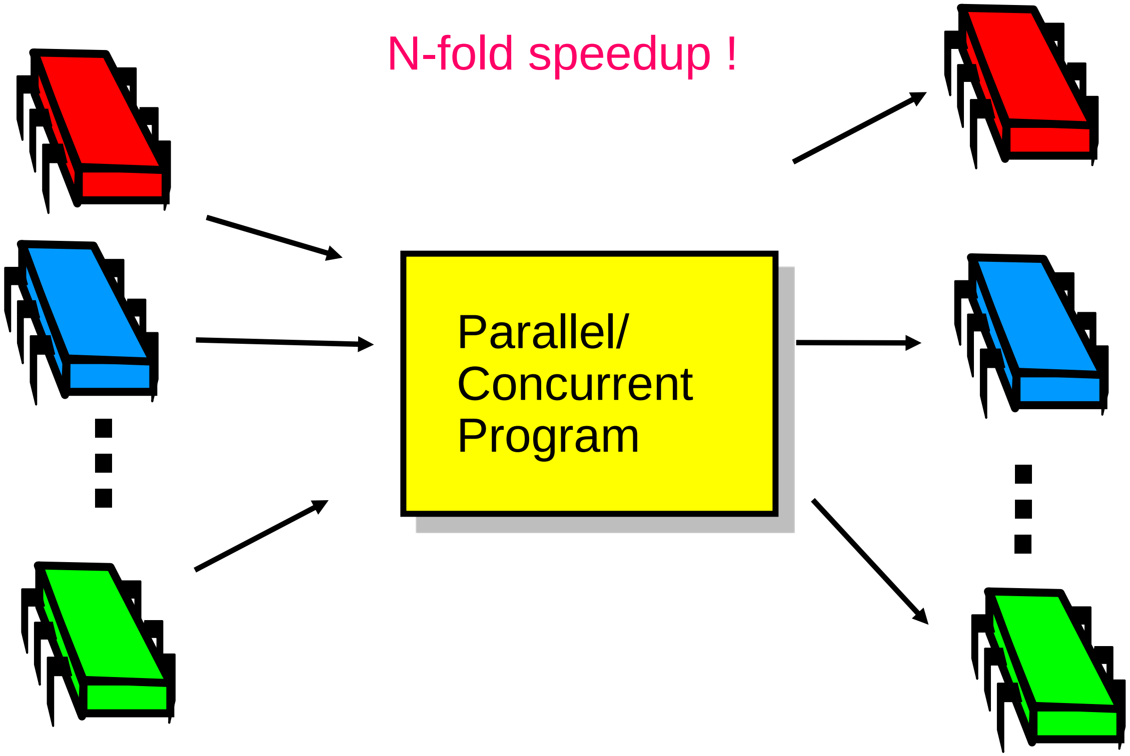


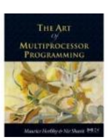
숙제 1 (~3/14)

- 이번 학기에 사용할 수 있는 **multi-core** 기계를 하나 골라서 **(the more cores, the better . . .)**
- CPU 사양, 시스템 사양 을 조사할 것.
- 이 CPU에서 제공하는 여러가지 **synchronization operation**들을 파악한다. (https://software.intel.com/en-us/node/506090)
  - Compare and Swap, Fetch and Add, . . .
  - Memory Barrier
- 이런 **operation**들의 성능을 측정해본다.
  - 하나의 long integer variable을 하나의 core 에서 1,000,000,000회 증가시키는 시간과 여러 개의 core를 사용하고 synchronization을 해서 같이 작업할 경우의 시간 측정

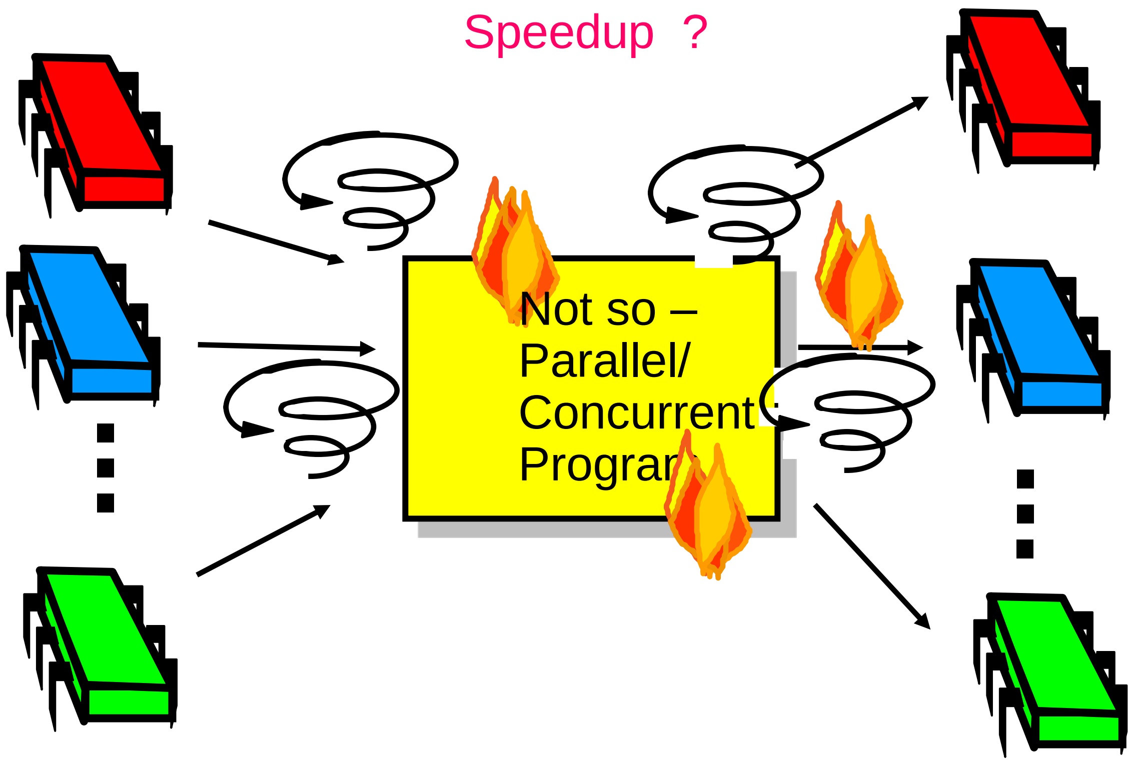


Ideal Multiprocessor

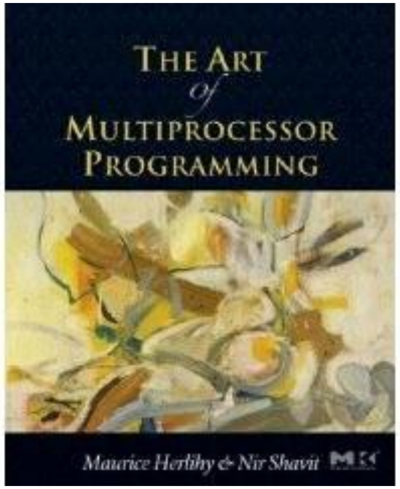




In Reality

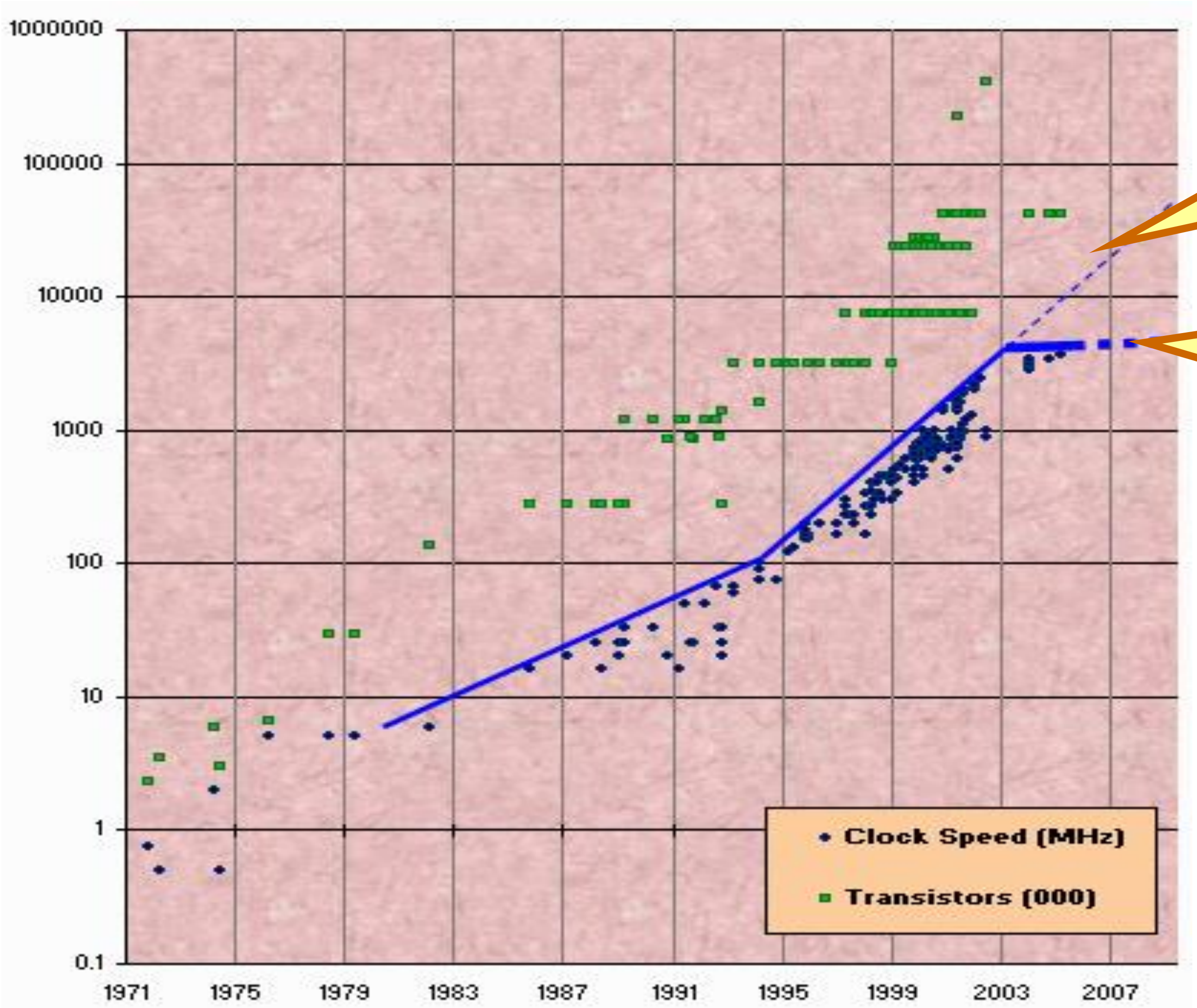


Introduction

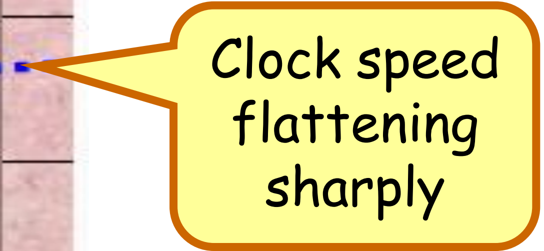


Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

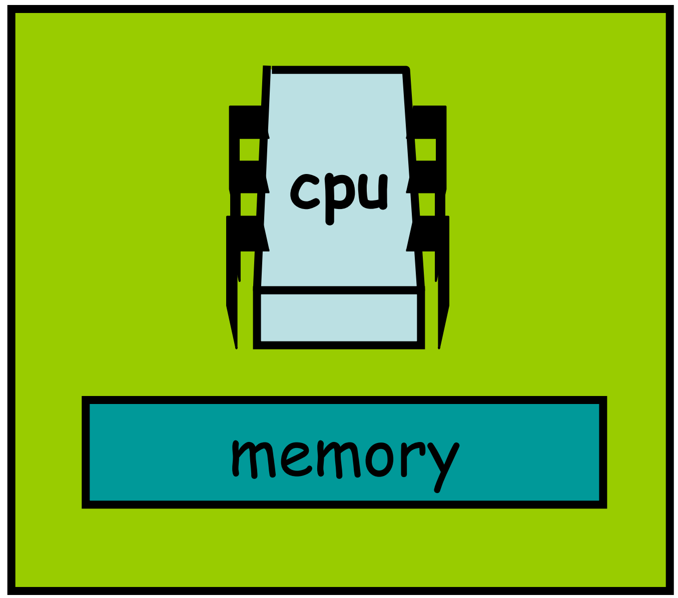
Moore's Law

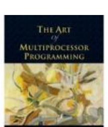

1000000
100000
10000
1000
100
10
1
0.1
1971
1975
1979
1983
1987
1991
1995
1999
2003
2007
Clock Speed (MHz)
Transistors (000)


Transistor count still rising

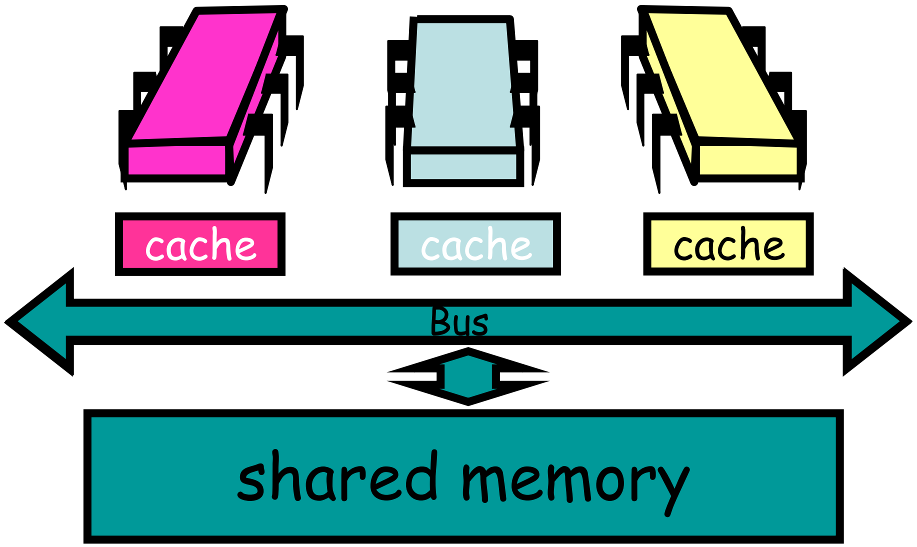

Clock speed flattening sharply

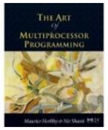
Vanishing from your Desktops: The Uniprocesor





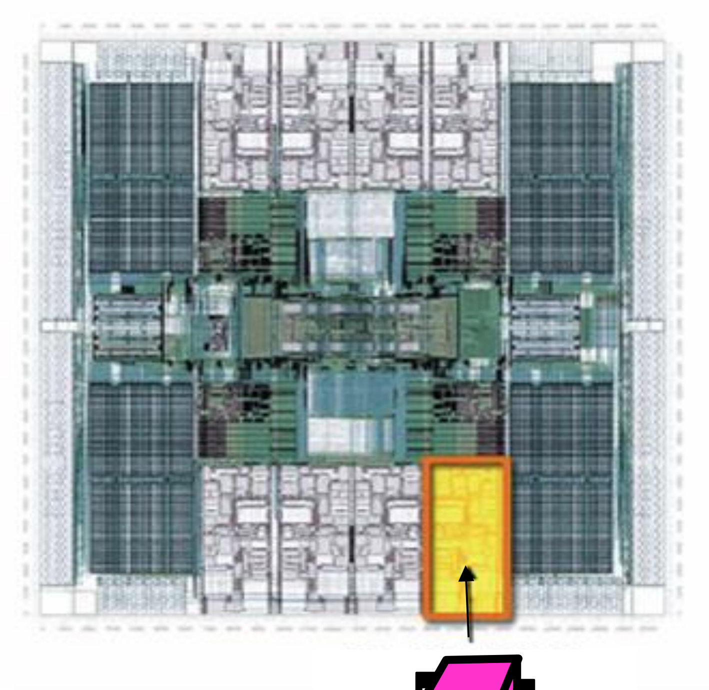
Your Server: The Shared Memory Multiprocessor (SMP)

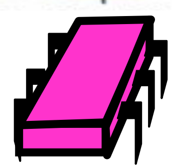




Your New Server or Desktop: The Multicore Processor (CMP)

All on the same chip



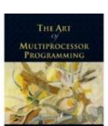


Sun T2000 Niagara

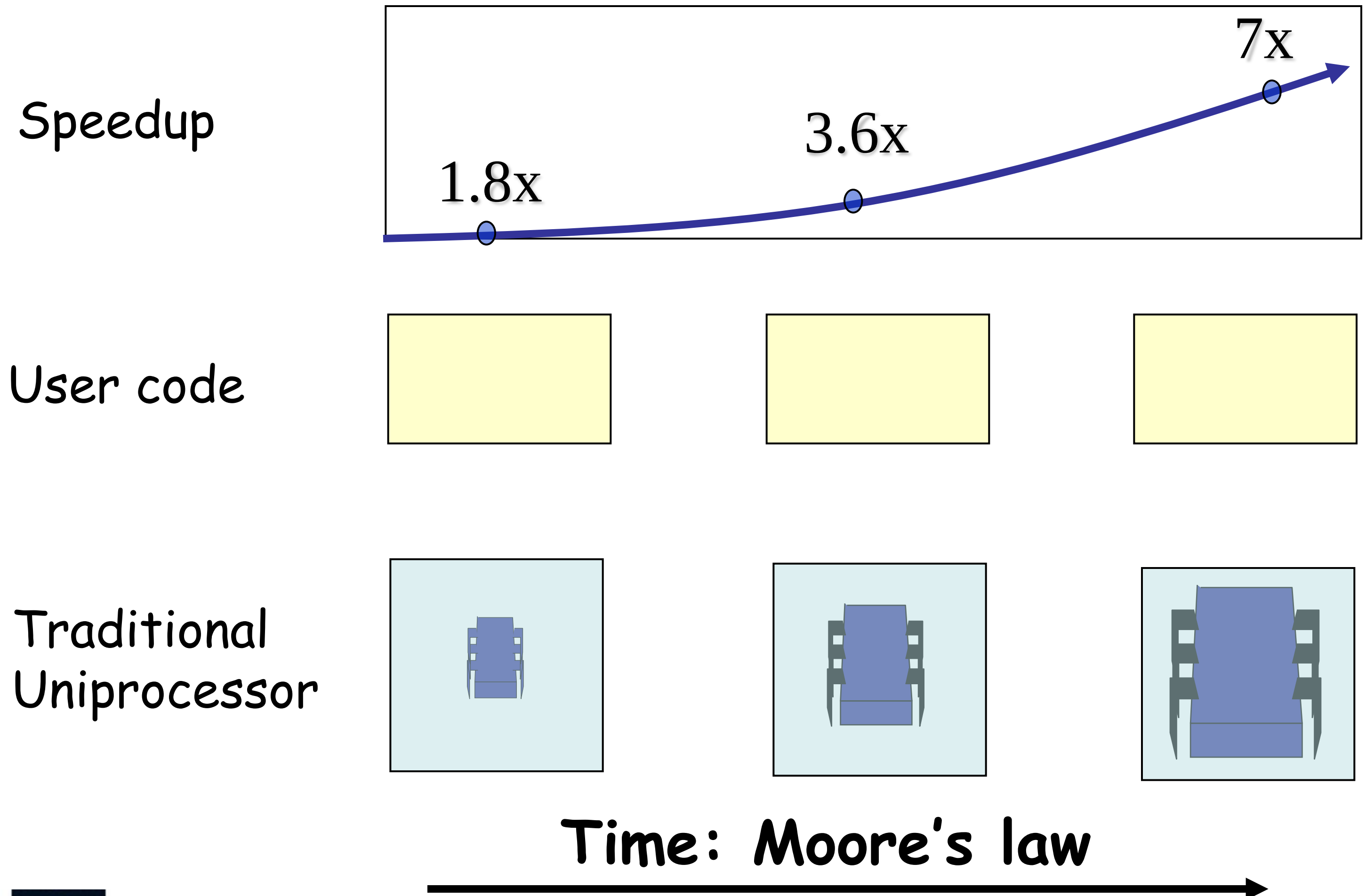
From the 2008 press…

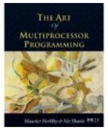
…*Intel* has announced a press conference in San Francisco on November 17th, where it will officially launch the Core *i7* Nehalem processor…

…Sun's next generation Enterprise T5140 and T5240 servers, based on the 3rd Generation UltraSPARC T2 Plus processor, were released two days ago…

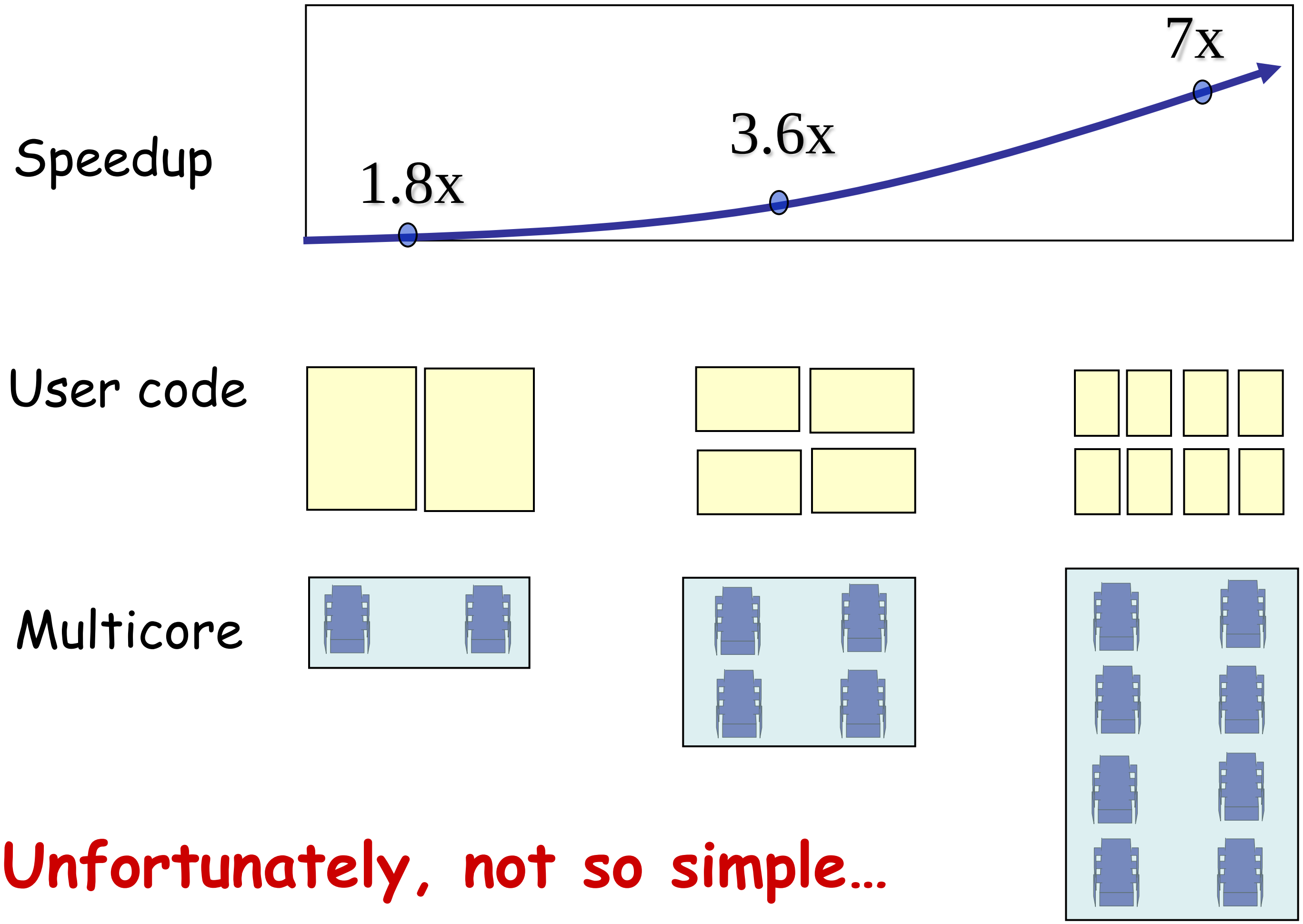


Traditional Scaling Process



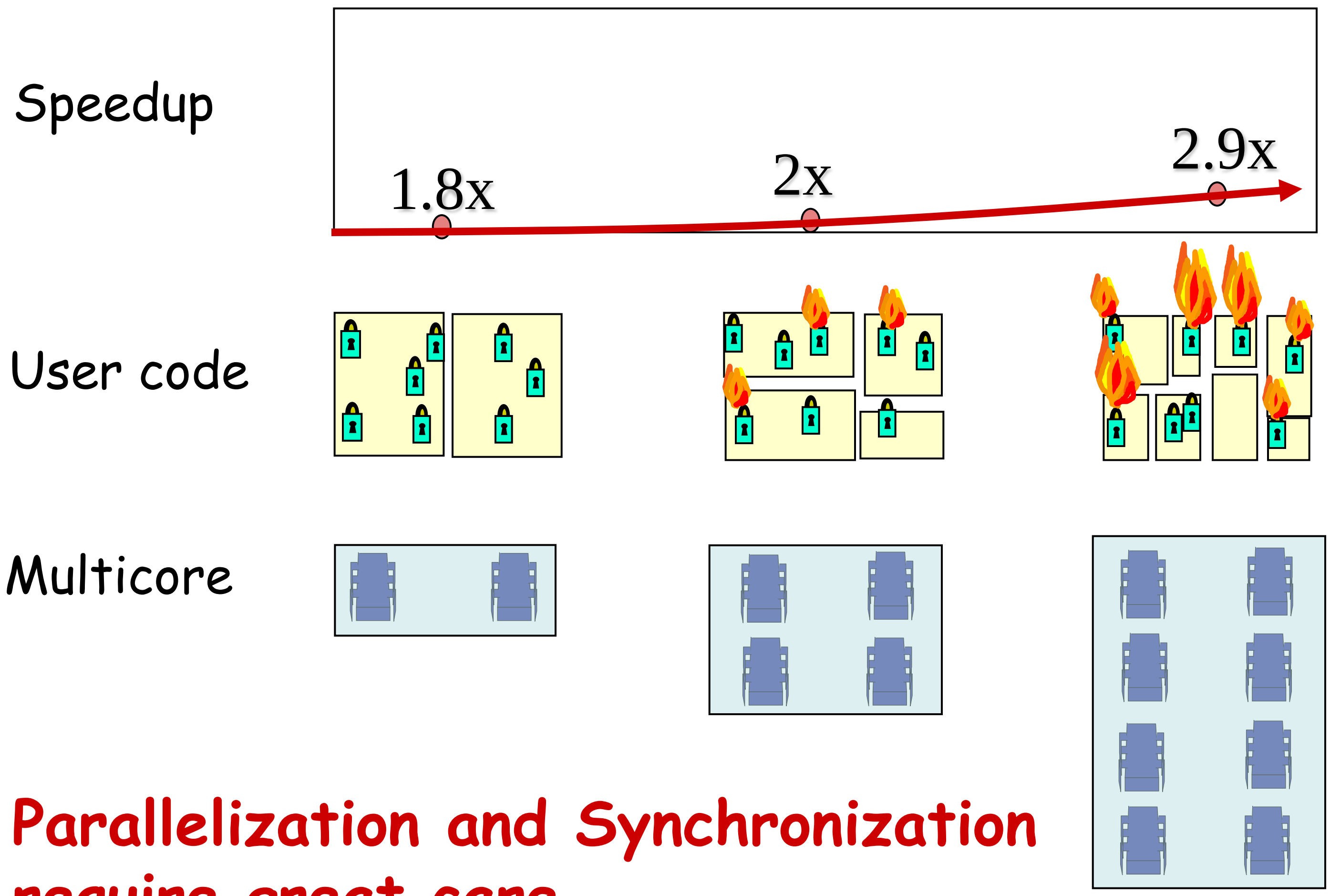


Multicore Scaling Process

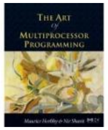


Unfortunately, not so simple…

Real-World Scaling Process

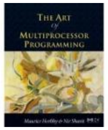


Parallelization and Synchronization require great care…



Multicore Programming: Course Overview

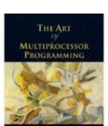
- Fundamentals
  - Models, algorithms, impossibility
- Real-World programming
  - Architectures
  - Techniques



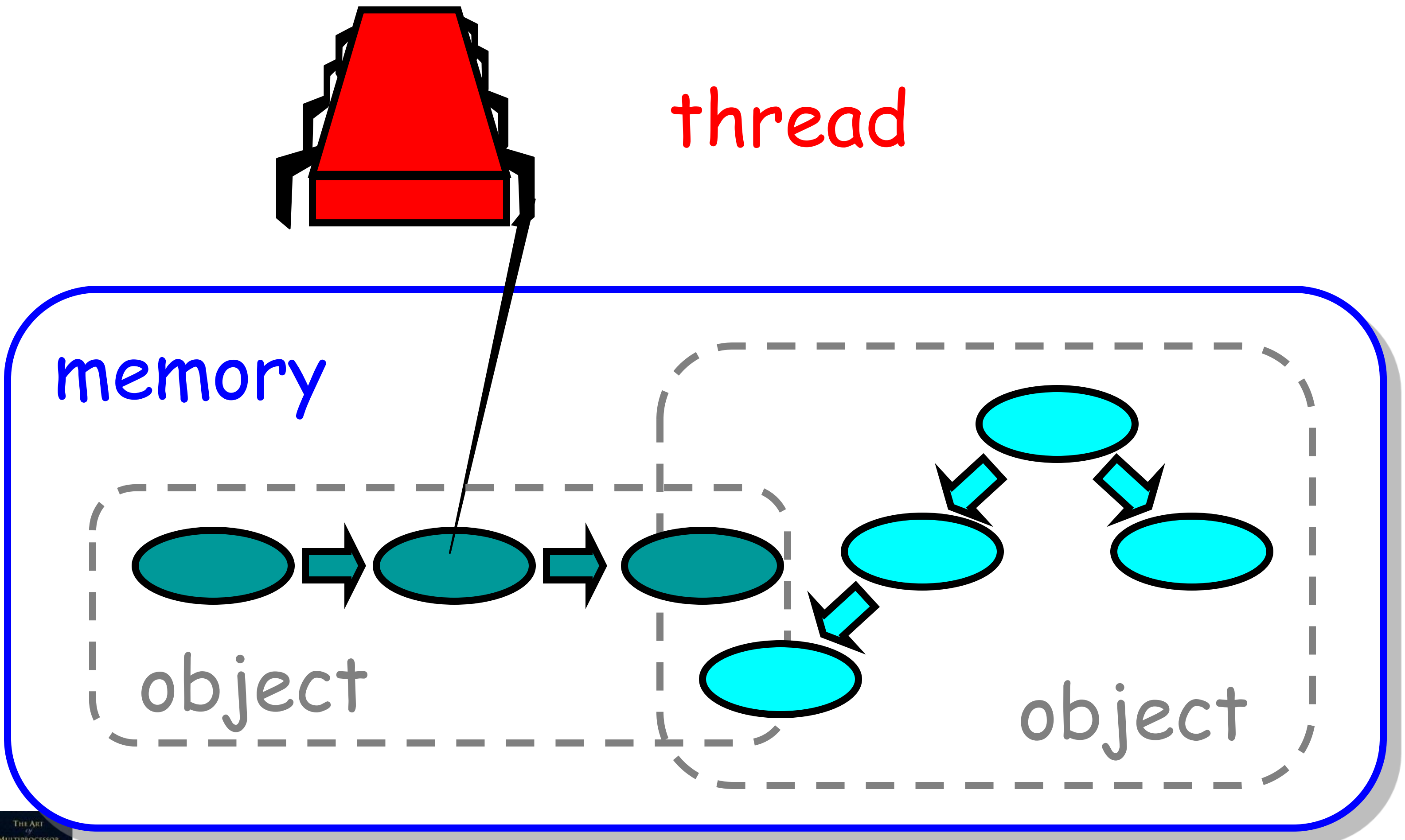
Multicore Programming: Course Overview

- Fundamentals
  - Models, algorithms, i
- Real-World pr
  - Architect
  - Techniques

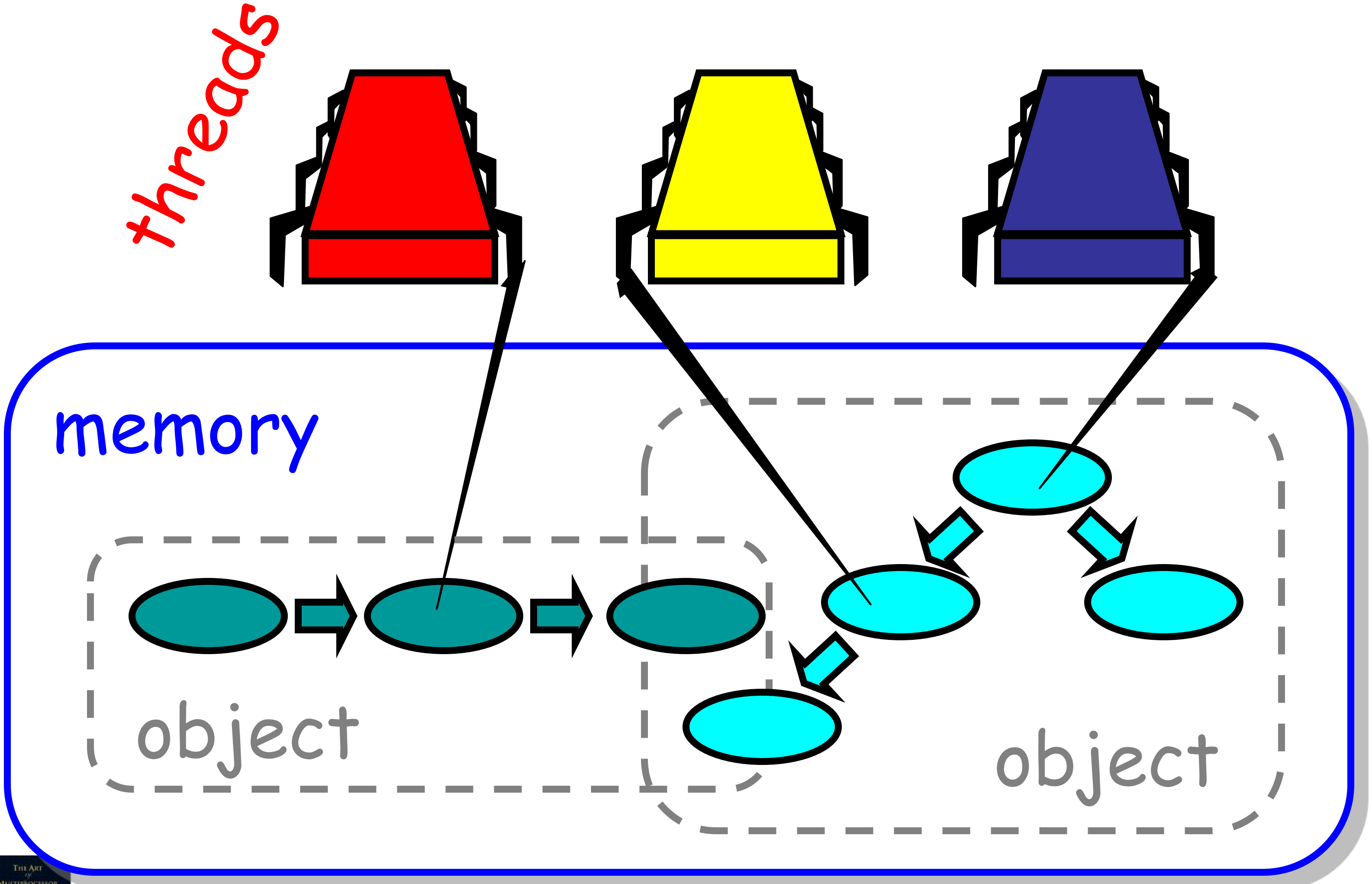
We don't necessarily want to make you experts…



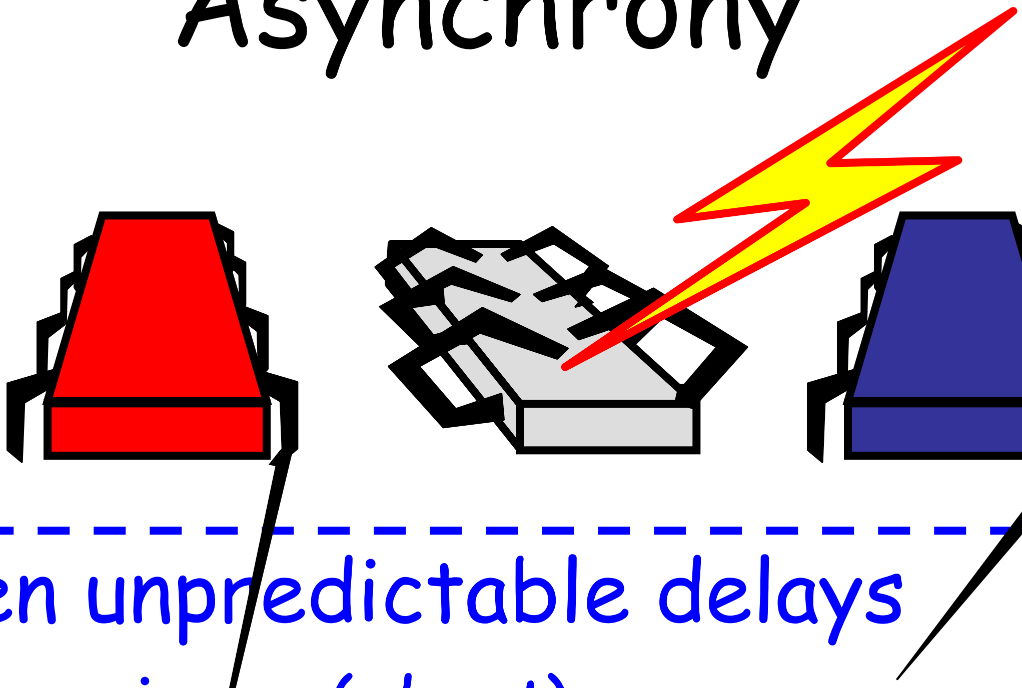
Sequential Computation



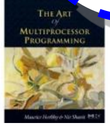
Concurrent Computation



Asynchrony

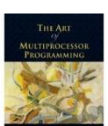


- Sudden unpredictable delays
  - Cache misses (*short*)
  - Page faults (*long*)
  - Scheduling quantum used up (*really long*)



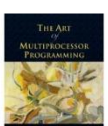
Model Summary

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



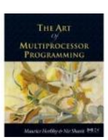
Road Map

- 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"



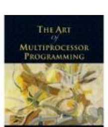
Concurrency Jargon

- Hardware
  - Processors
- Software
  - Threads, processes
- Sometimes OK to confuse them, sometimes not.

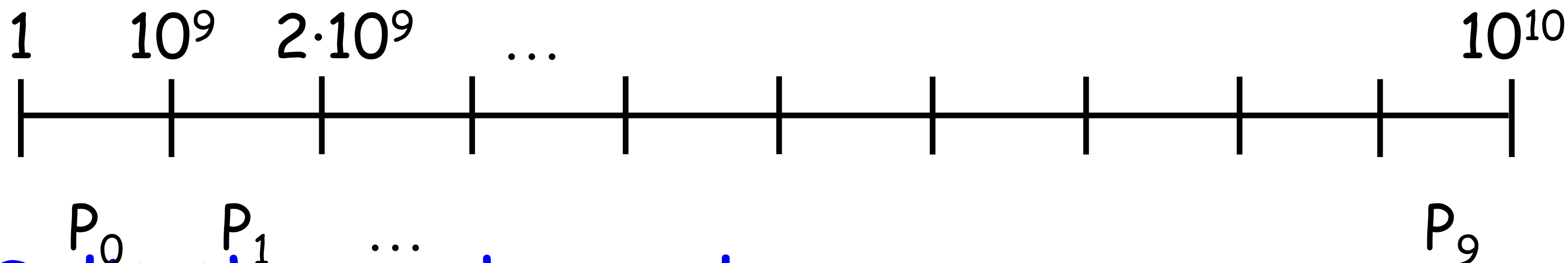


Parallel Primality Testing

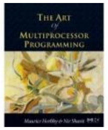
- Challenge
  - Print primes from 1 to 10^{10}
- Given
  - Ten-processor multiprocessor
  - One thread per processor
- Goal
  - Get ten-fold speedup (or close)



Load Balancing

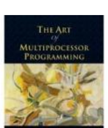


- Split the work evenly
- Each thread tests range of 10^9



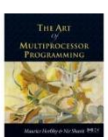
Procedure for Thread *i*

```
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

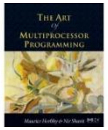
- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict



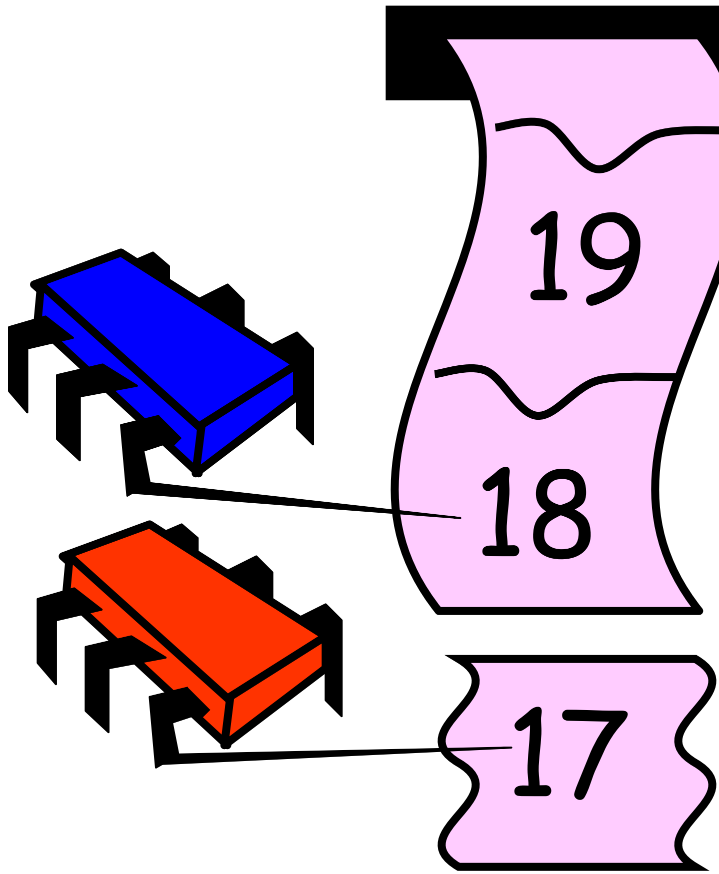
Issues

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

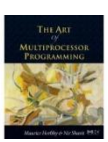
rejected



Shared Counter

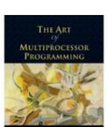


each thread
takes a number



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);
  }
}
```



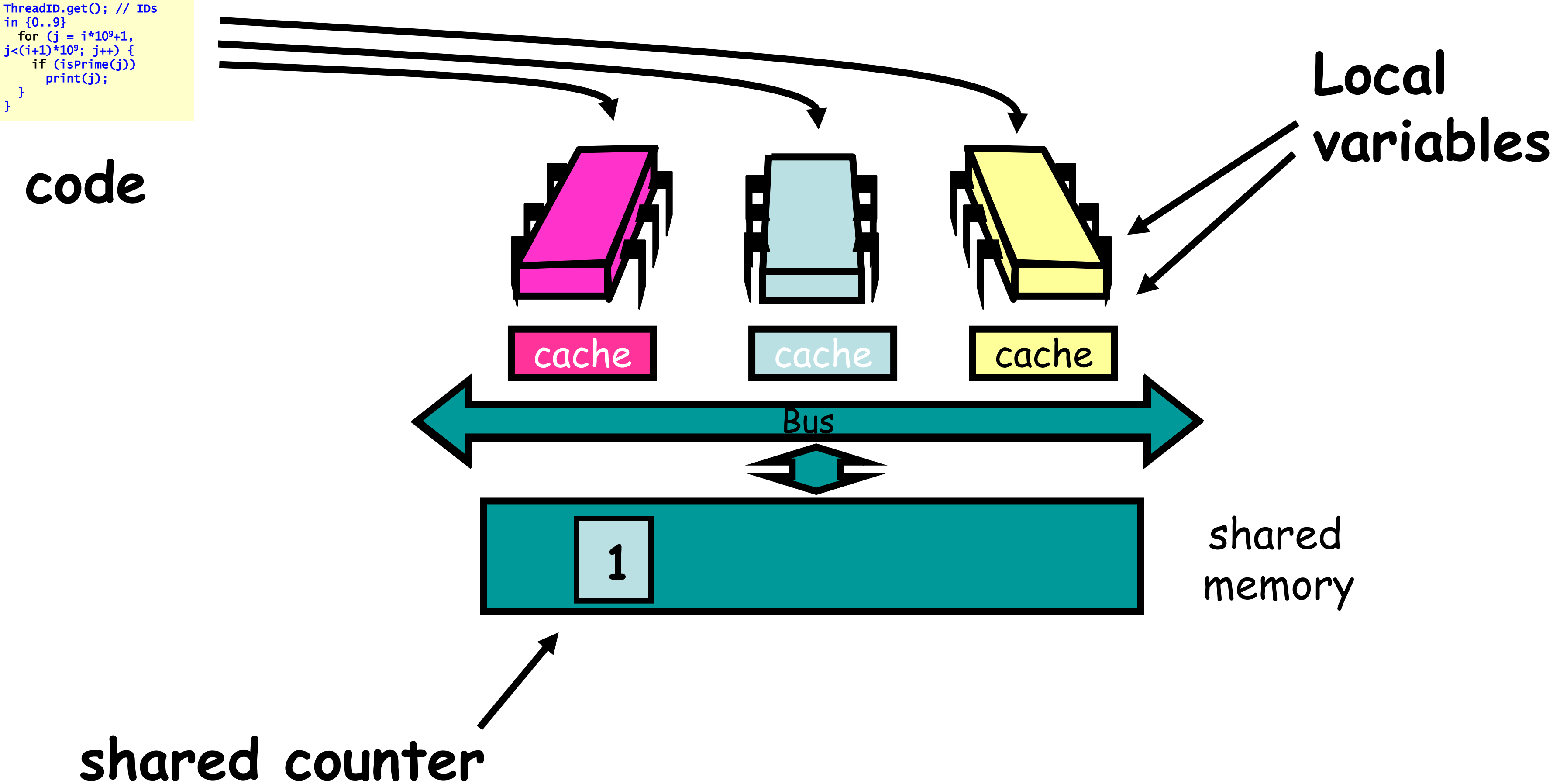
Procedure for Thread *i*

```
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

```
void primePrint {
  int i =
ThreadID.get(); // IDs
in {0..9}
  for (j = i*10⁹+1,
j<(i+1)*10⁹; j++) {
    if (isPrime(j))
      print(j);
  }
}
```



Procedure for Thread *i*

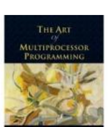
```
Counter counter = new Counter(1);

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

Stop when every value taken

Procedure for Thread *i*

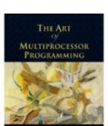
```
Counter counter = new Counter(1);

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

Increment & return each new value



Counter Implementation

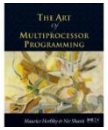
```
public class Counter {
  private long value;

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



Counter Implementation

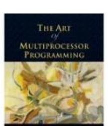
```
public class Counter {
  private long value;

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

OK for single thread,
not for concurrent threads



What It Means

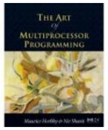
```
public class Counter {
  private long value;

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



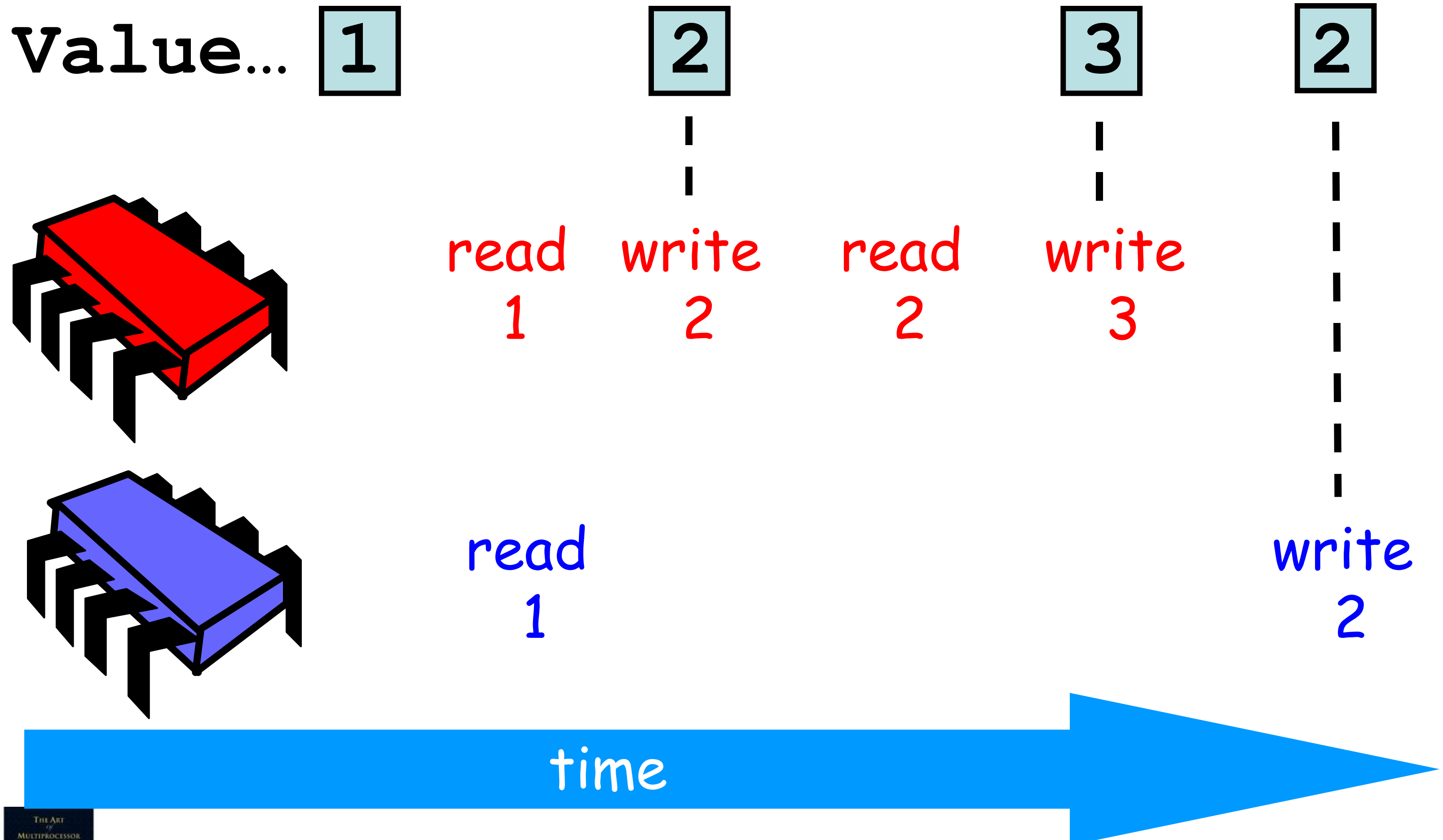
What It Means

```
public class Counter {
  private long value;

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

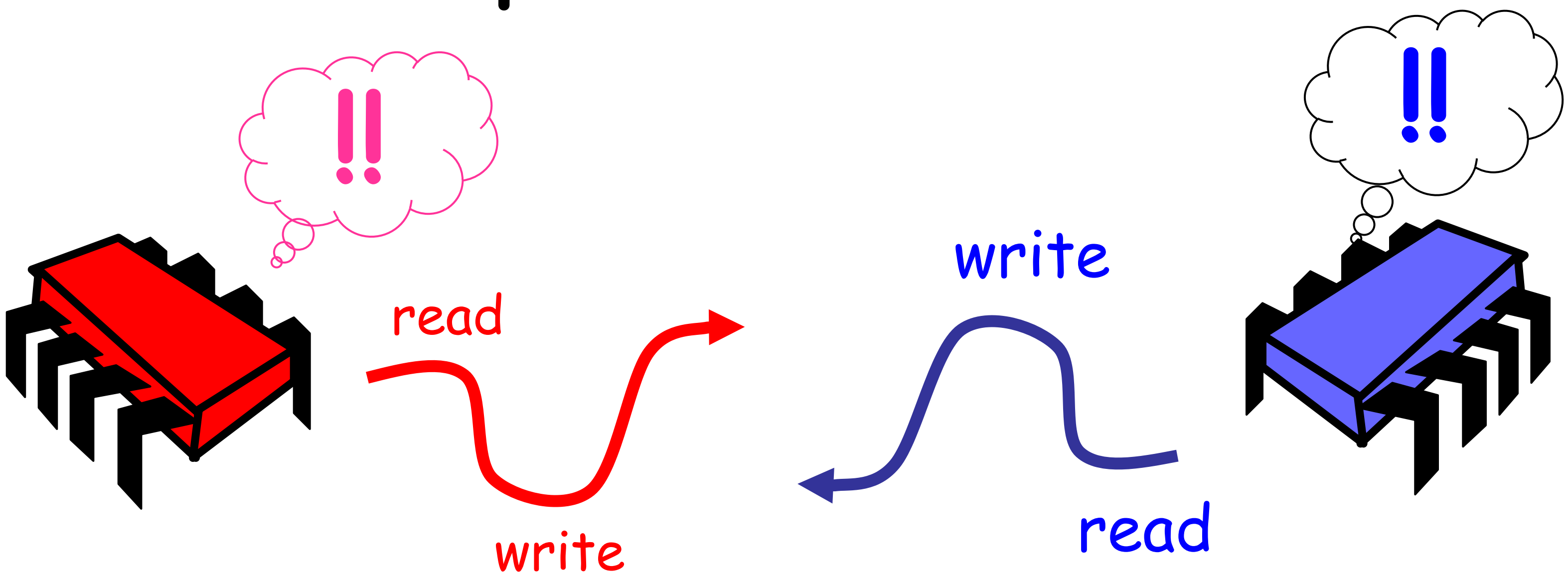
```
temp  = value;
value = temp + 1;
return temp;
```



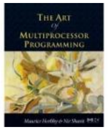
Not so good…



Is this problem inherent?

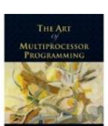


If we could only glue reads and writes together…



Challenge

```
public class Counter {
  private long value;

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



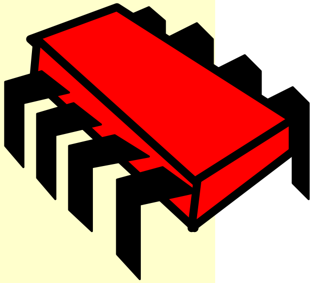
Challenge

```
public class Counter {
  private long value;

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

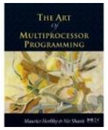
Make these steps atomic (indivisible)

Hardware Solution

```
public class Counter {
  private long value;

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

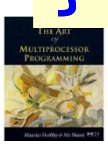


ReadModifyWrite() instruction



An Aside: Java™

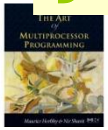
```
public class Counter {
  private long value;

  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

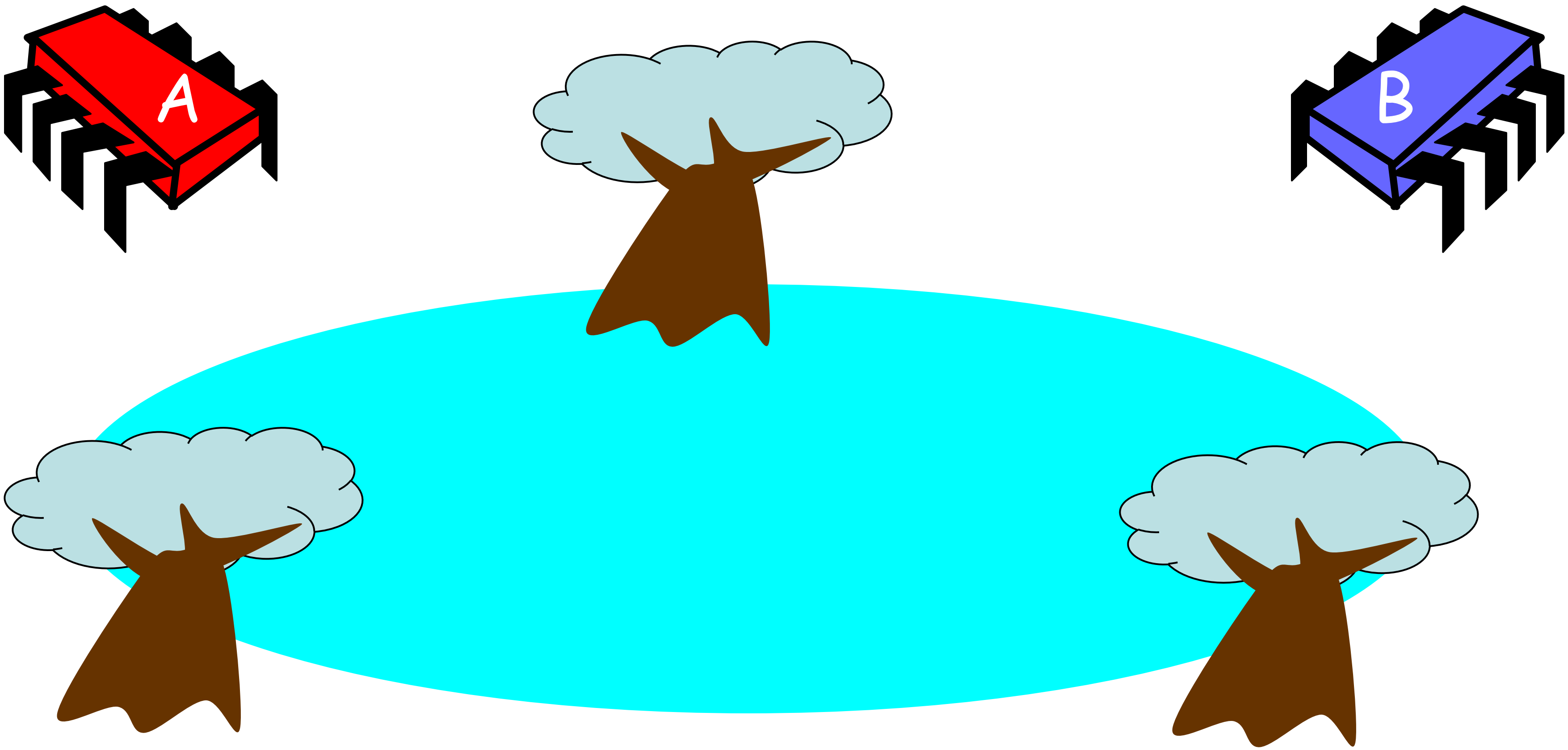


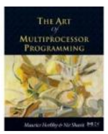
An Aside: Java™

```
public class Counter {
  private long value;

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

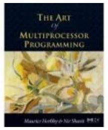
Mutual Exclusion

Mutual Exclusion or "Alice & Bob share a pond"

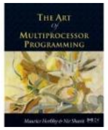




Alice has a pet



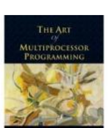
Bob has a pet



The Problem

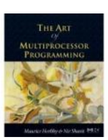
Formalizing the Problem

- Two types of formal properties in asynchronous computation:
- Safety Properties
  - Nothing bad happens ever
- Liveness Properties
  - Something good happens eventually



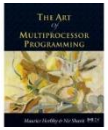
Formalizing our Problem

- Mutual Exclusion
  - Both pets never in pond simultaneously
  - This is a ***safety*** property
- No Deadlock
  - if only one wants in, it gets in
  - if both want in, one gets in.
  - This is a ***liveness*** property



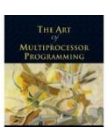
Simple Protocol

- Idea
  - Just look at the pond
- Gotcha
  - Not atomic
  - Trees obscure the view



Interpretation

- Threads can't "see" what other threads are doing
- Explicit communication required for coordination

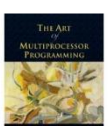


Cell Phone Protocol

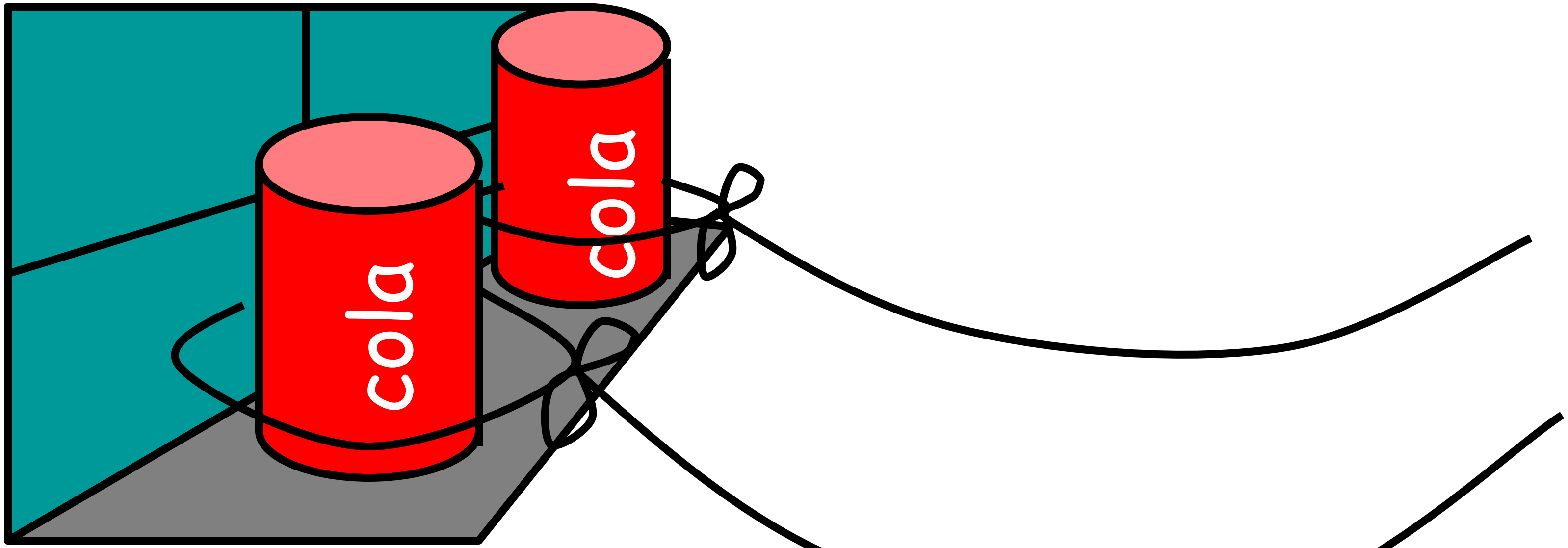
- Idea
  - Bob calls Alice (or vice-versa)
- Gotcha
  - Bob takes shower
  - Alice recharges battery
  - Bob out shopping for pet food …

Interpretation

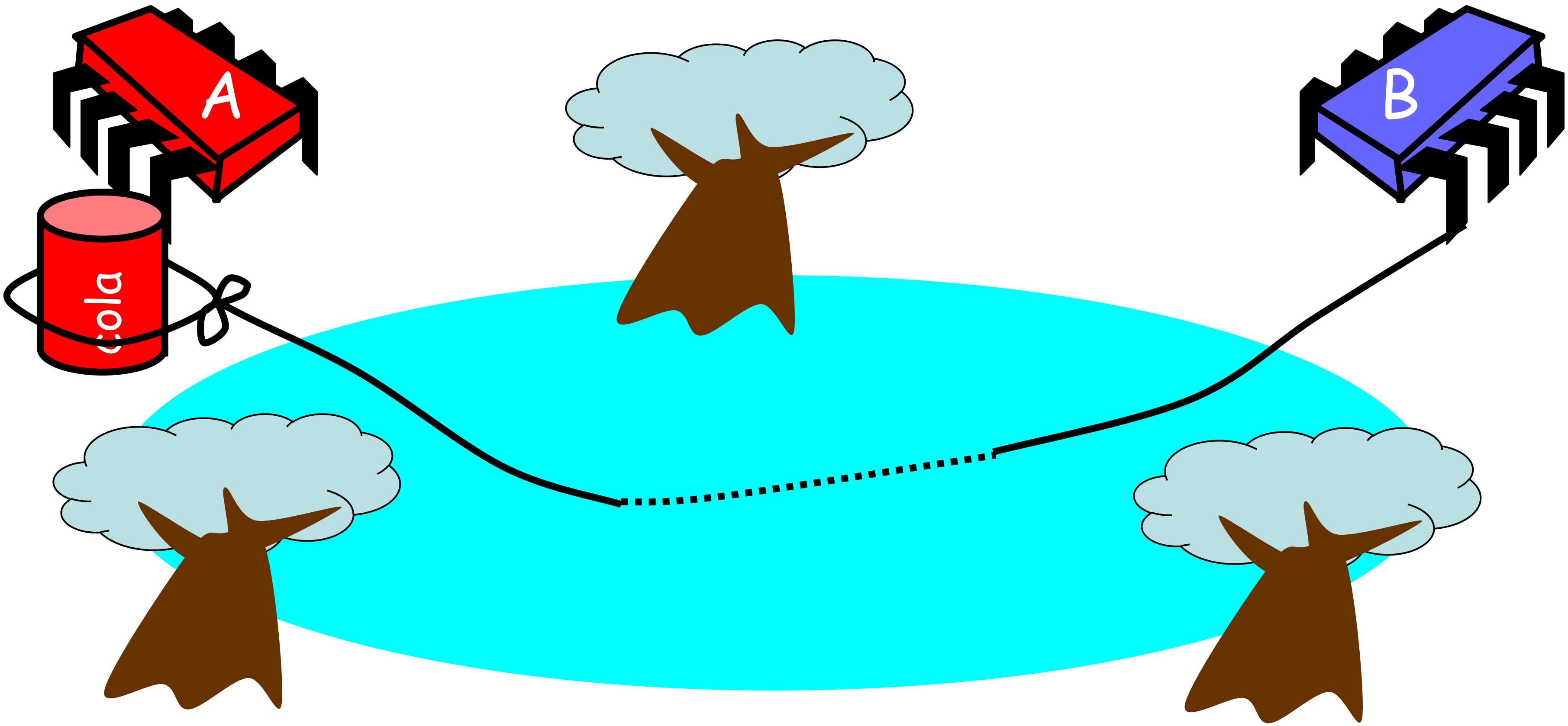
- Message-passing doesn't work
- Recipient might not be
  - Listening
  - There at all
- Communication must be
  - Persistent (like writing)
  - Not transient (like speaking)

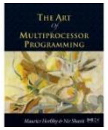


Can Protocol

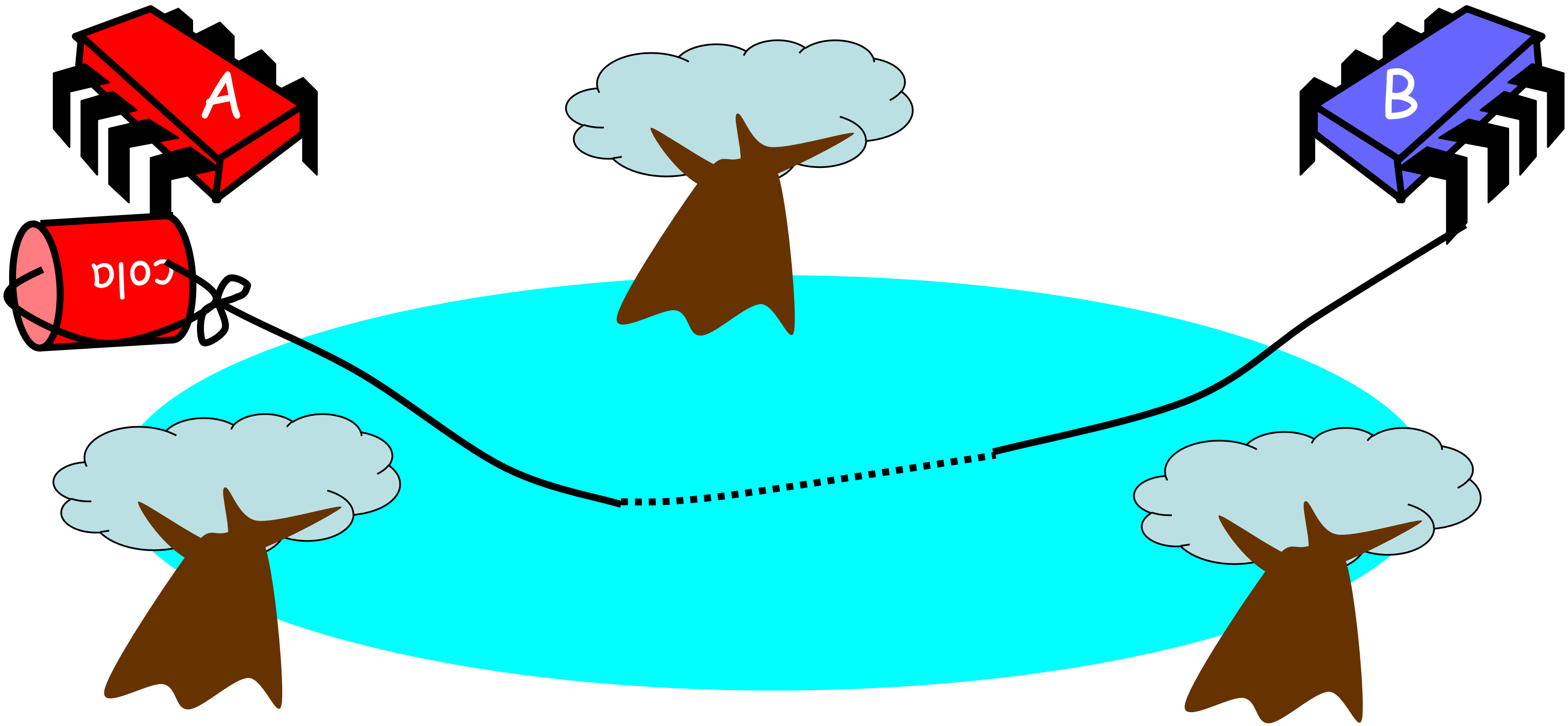


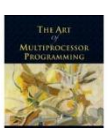
Bob conveys a bit





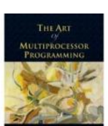
Bob conveys a bit





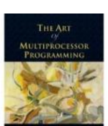
Can Protocol

- Idea
  - Cans on Alice's windowsill
  - Strings lead to Bob's house
  - Bob pulls strings, knocks over cans
- Gotcha
  - Cans cannot be reused
  - Bob runs out of cans

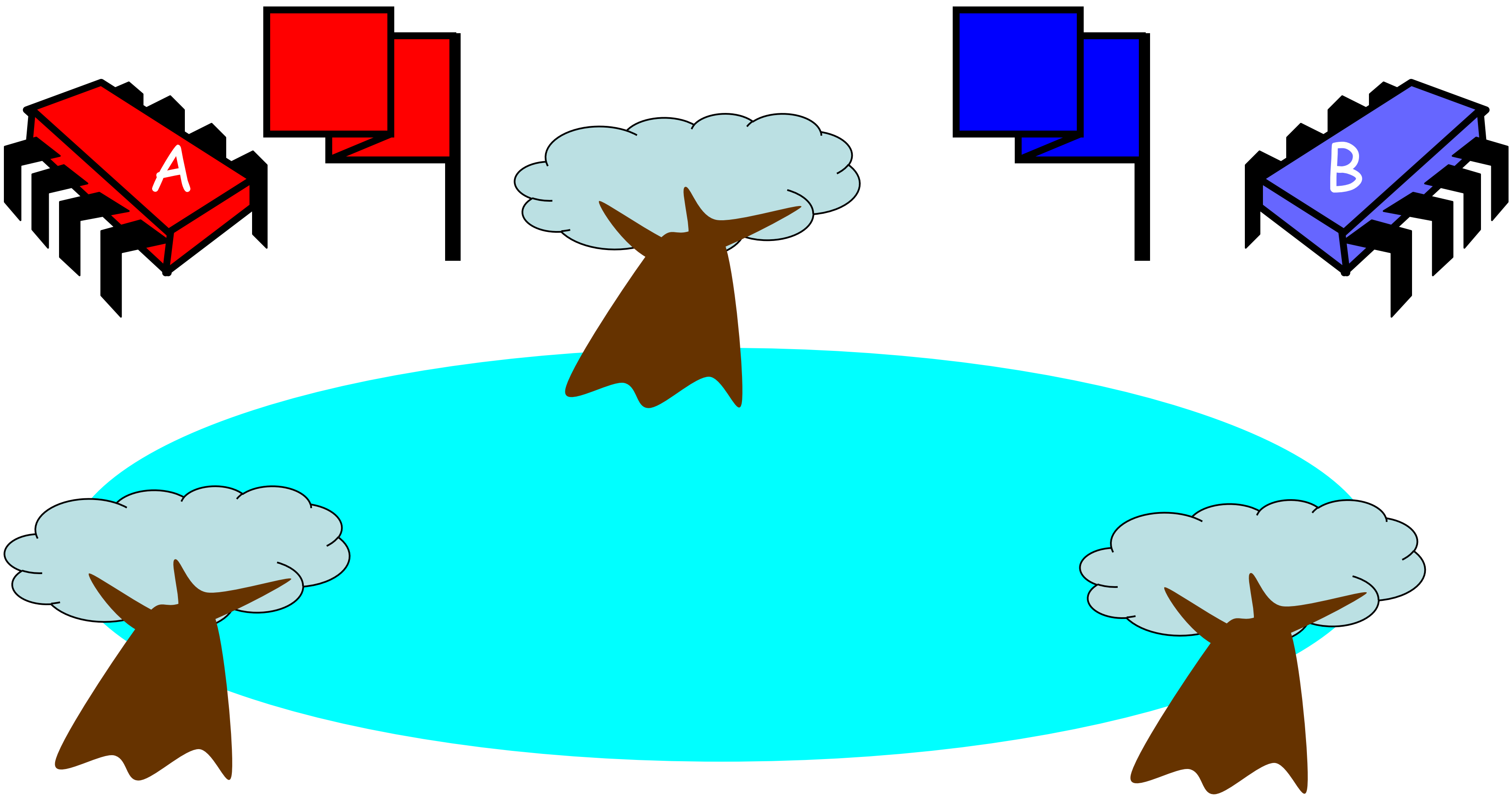


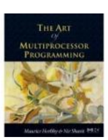
Interpretation

- Cannot solve mutual exclusion with interrupts
  - Sender sets fixed bit in receiver's space
  - Receiver resets bit when ready
  - Requires unbounded number of interrupt bits

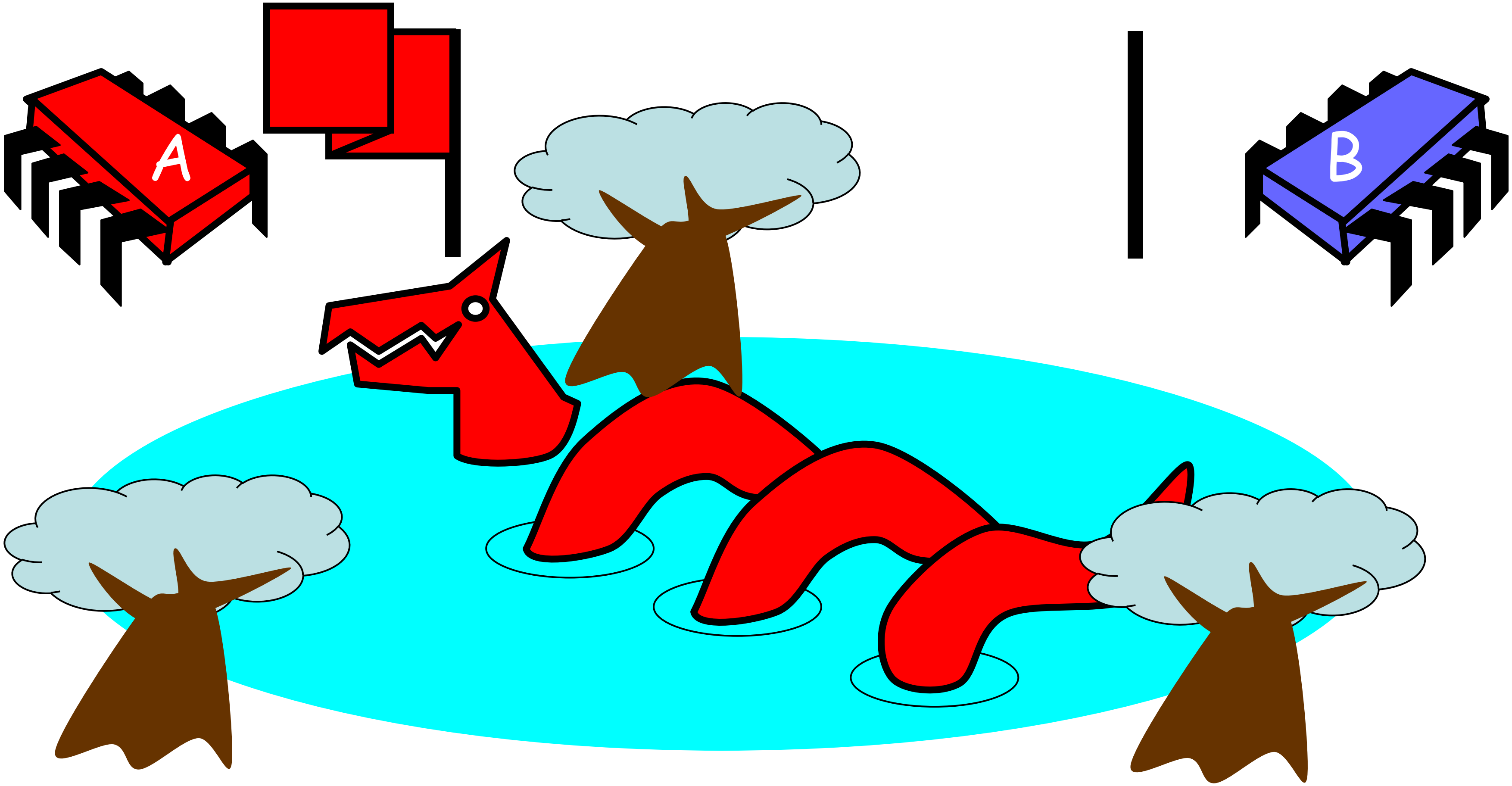


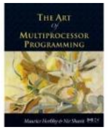
Flag Protocol



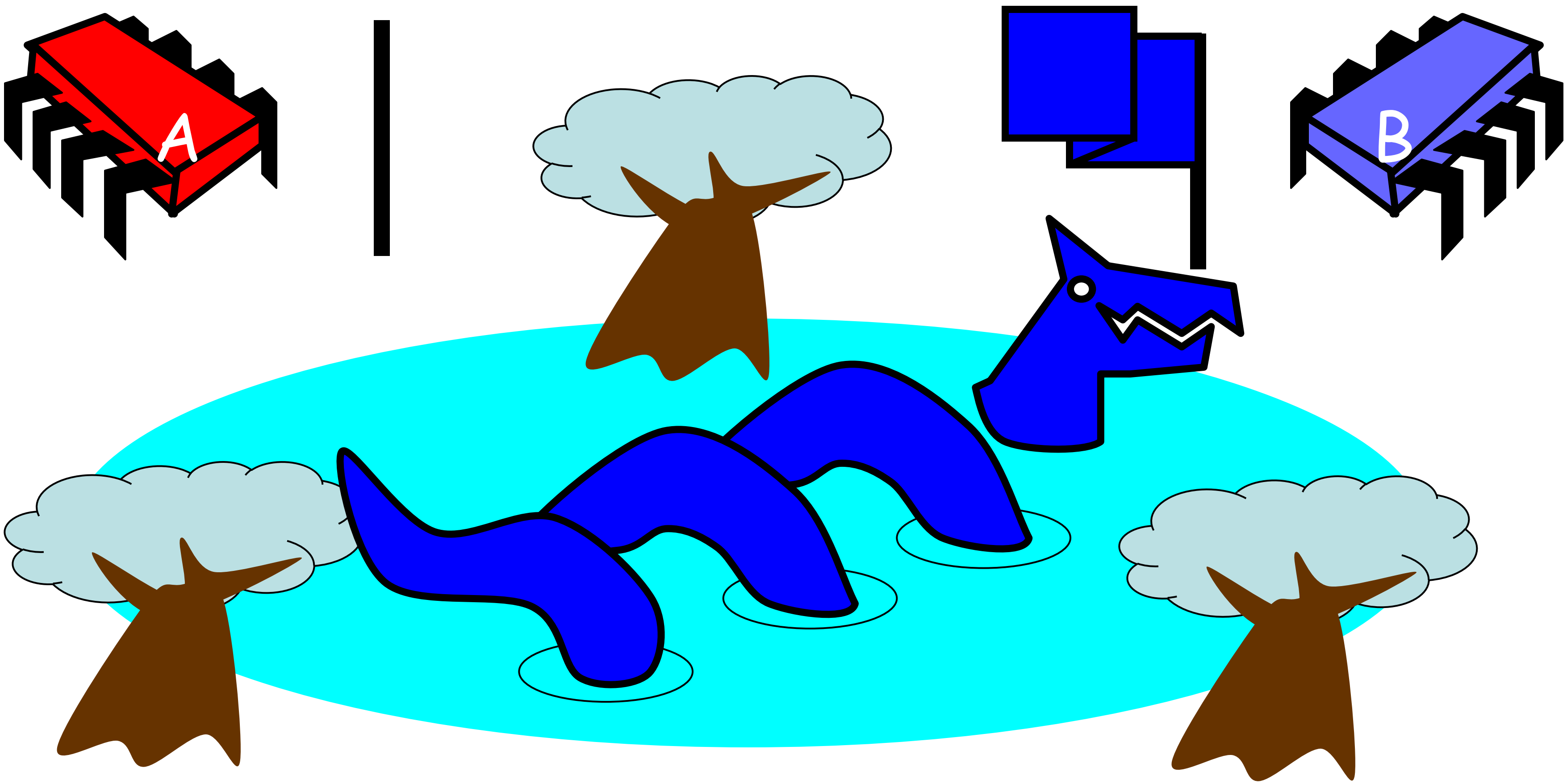


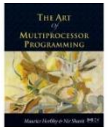
Alice's Protocol (sort of)





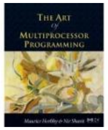
Bob's Protocol (sort of)





Alice's Protocol

- Raise flag
- Wait until Bob's flag is down
- Unleash pet
- Lower flag when pet returns



Bob's Protocol

- Raise flag
- Wait until Alice's flag is down
- Unleash pet
- Lower flag when pet returns

danger!

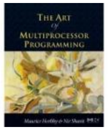
Bob's Protocol (2nd try)

- Raise flag
- While Alice's flag is up
  - Lower flag
  - Wait for Alice's flag to go down
  - Raise flag
- Unleash pet
- Lower flag when pet returns

Bob's Protocol

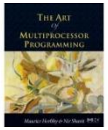
- Raise flag
- While Alice's flag is up
  - Lower flag
  - Wait for Alice's flag to go down
  - Raise flag
- Unleash pet
- Lower flag when pet returns

Bob defers to Alice



The Flag Principle

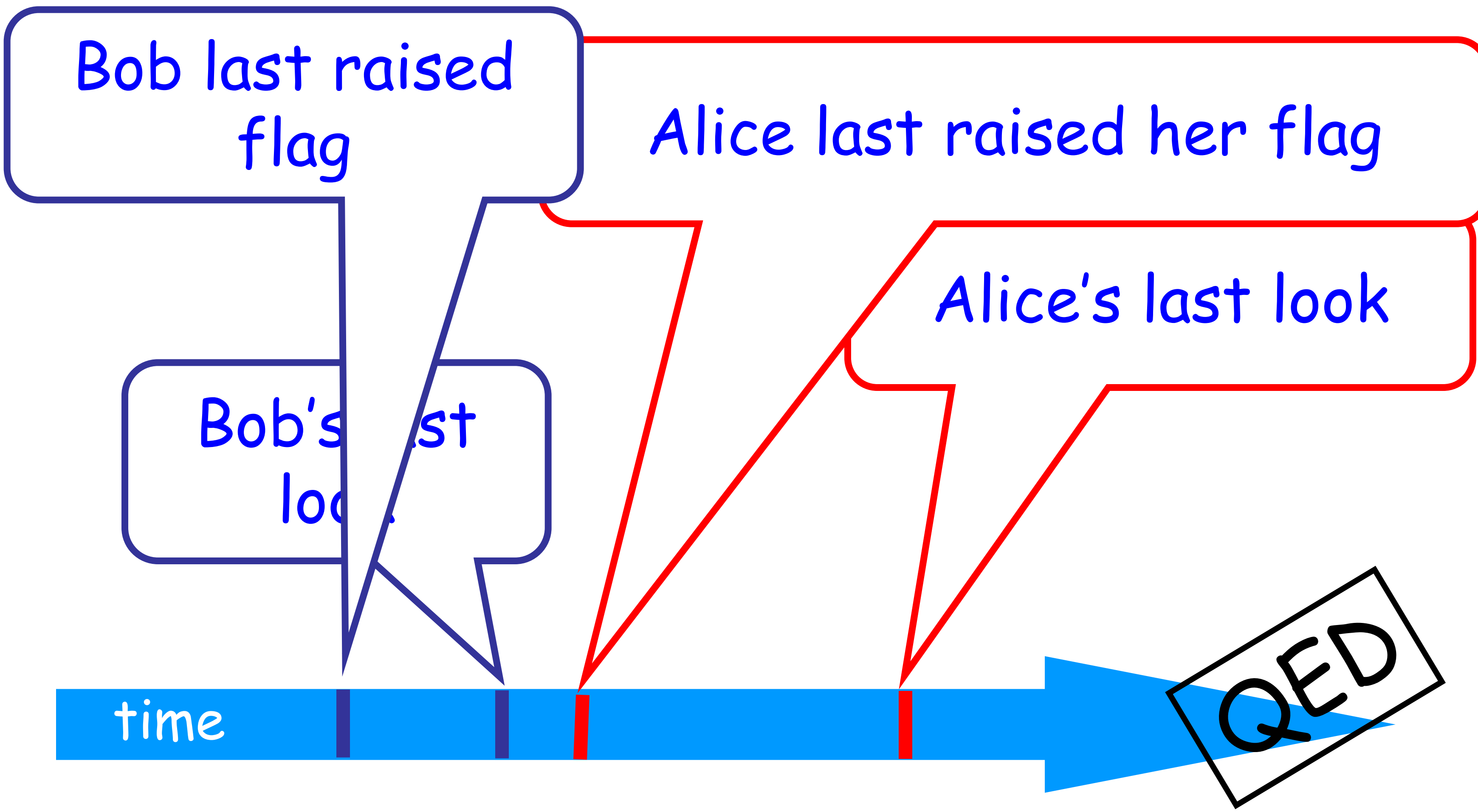
- Raise the flag
- Look at other's flag
- Flag Principle:
  - If each raises and looks, then
  - Last to look must see both flags up



Proof of Mutual Exclusion

- Assume both pets in pond
  - Derive a contradiction
  - By reasoning **backwards**
- Consider the last time Alice and Bob each looked before letting the pets in
- Without loss of generality assume Alice was the last to look…

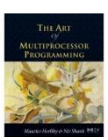
Proof



Alice must have seen Bob's Flag. A Contradiction

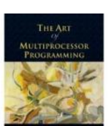
Proof of No Deadlock

- If only one pet wants in, it gets in.



Proof of No Deadlock

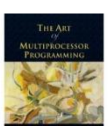
- If only one pet wants in, it gets in.
- Deadlock requires both continually trying to get in.



Proof of No Deadlock

- If only one pet wants in, it gets in.
- Deadlock requires both continually trying to get in.
- If Bob sees Alice's flag, he gives her priority (a gentleman…)

QED



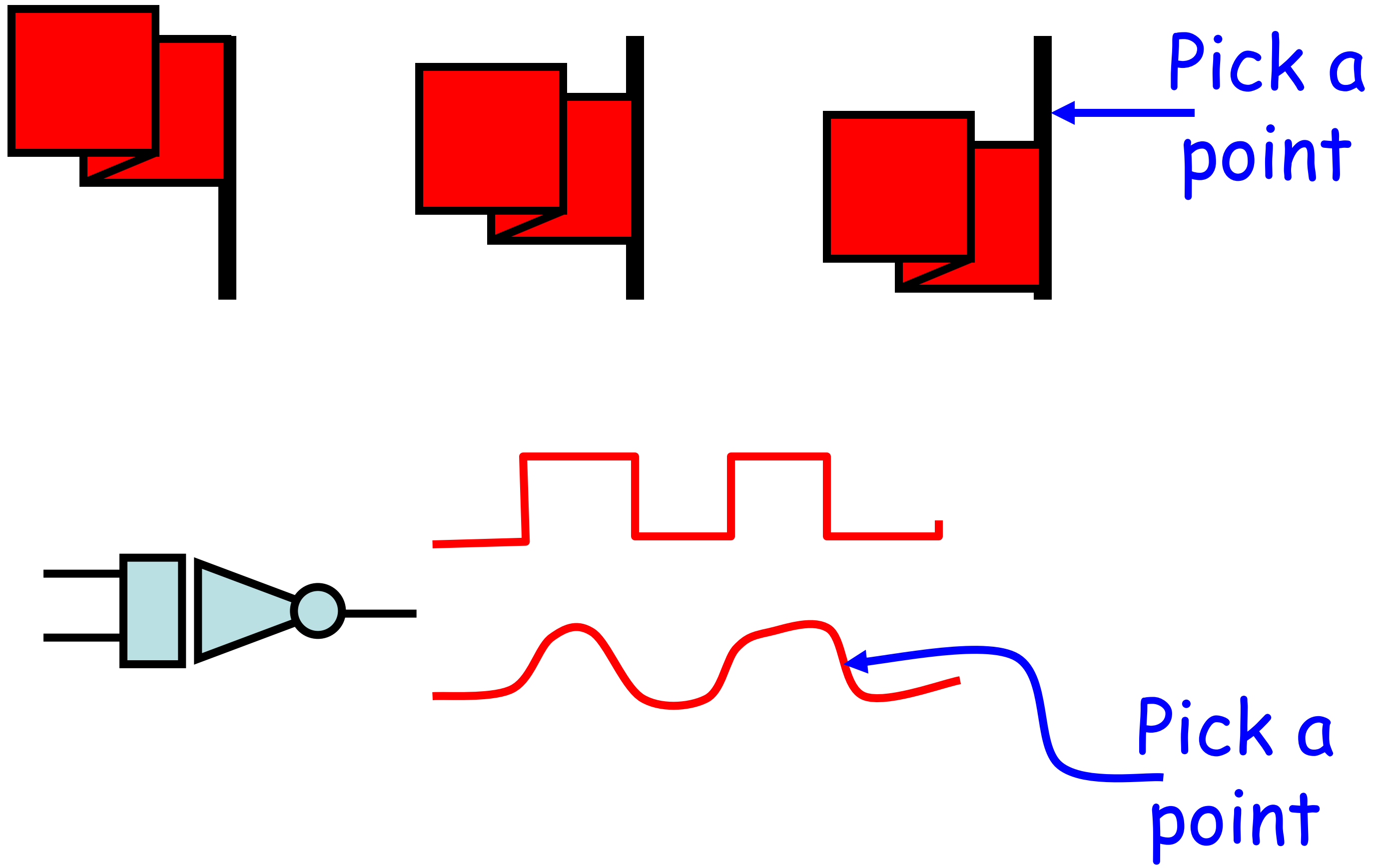
Remarks

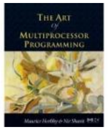
- Protocol is *unfair*
  - Bob's pet might never get in
- Protocol uses *waiting*
  - If Bob is eaten by his pet, Alice's pet might never get in

Moral of Story

- Mutual Exclusion cannot be solved by
  - transient communication (cell phones)
  - interrupts (cans)
- It can be solved by
  - one-bit shared variables
  - that can be read or written

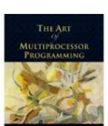
The Arbiter Problem (an aside)





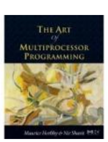
The Fable Continues

- Alice and Bob fall in love & marry



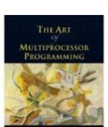
The Fable Continues

- Alice and Bob fall in love & marry
- Then they fall out of love & divorce
  - She gets the pets
  - He has to feed them

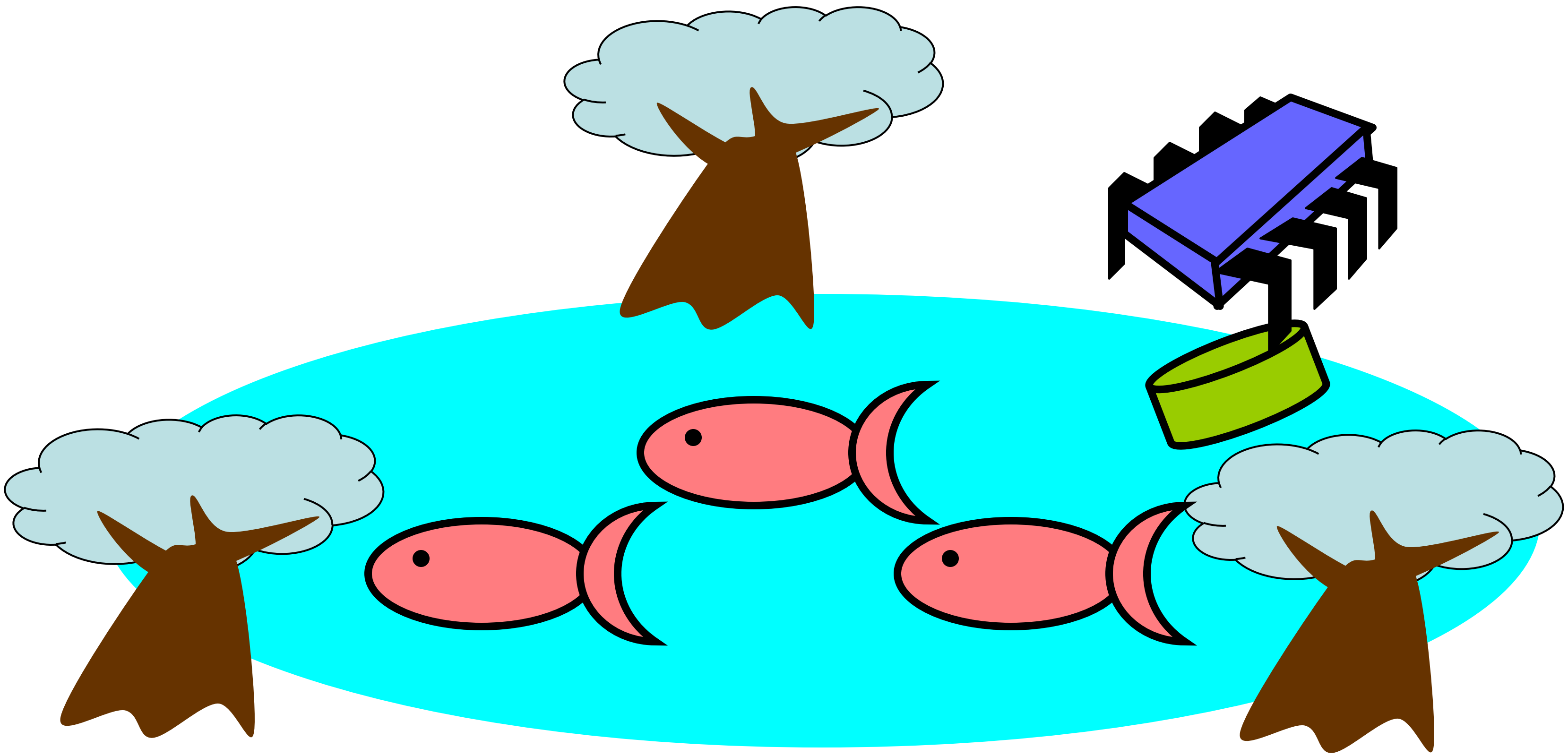


The Fable Continues

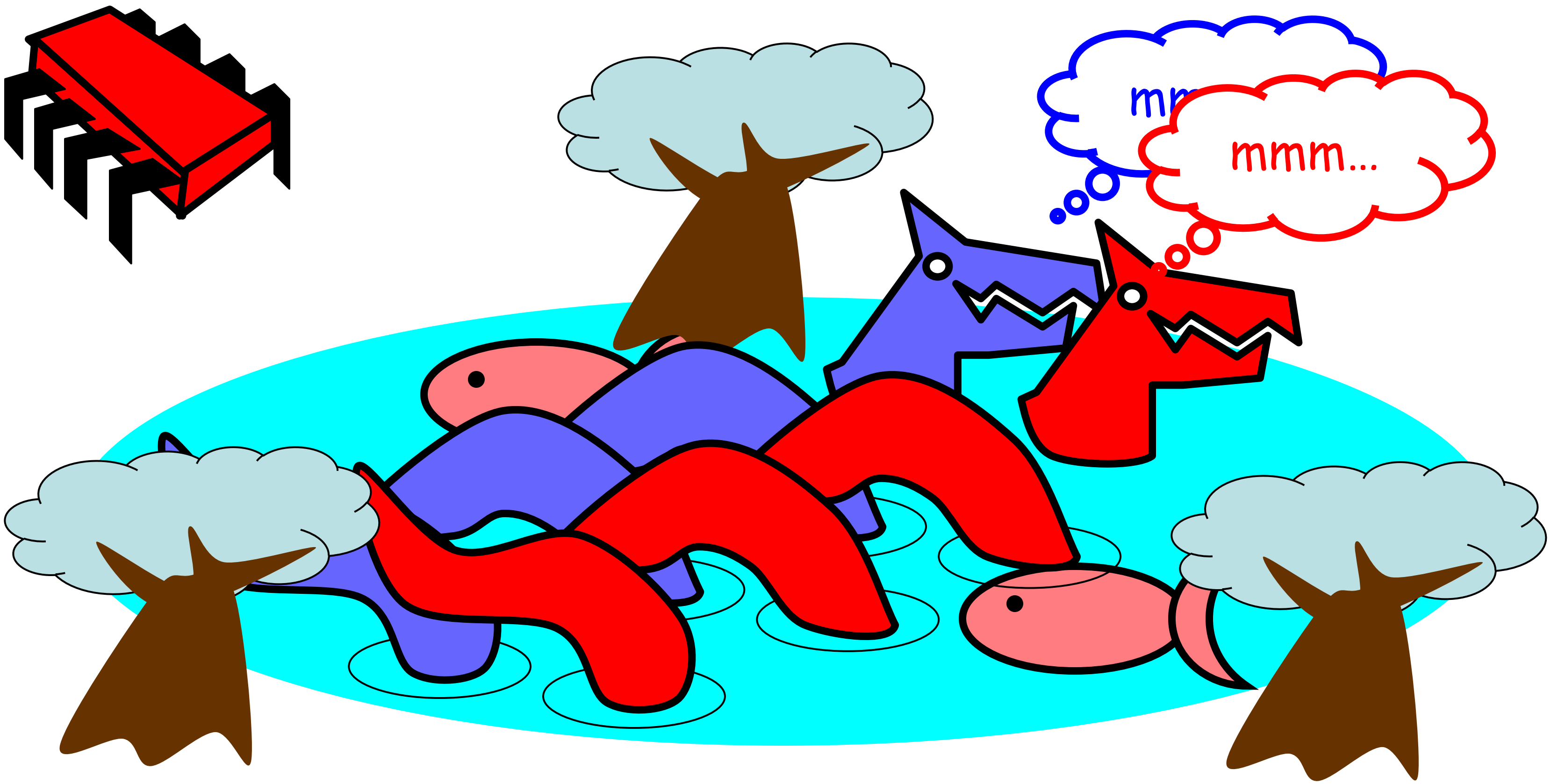
- Alice and Bob fall in love & marry
- Then they fall out of love & divorce
  - She gets the pets
  - He has to feed them
- Leading to a new coordination problem: Producer-Consumer

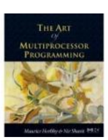


Bob Puts Food in the Pond



Alice releases her pets to Feed



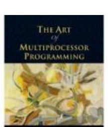


Producer/Consumer

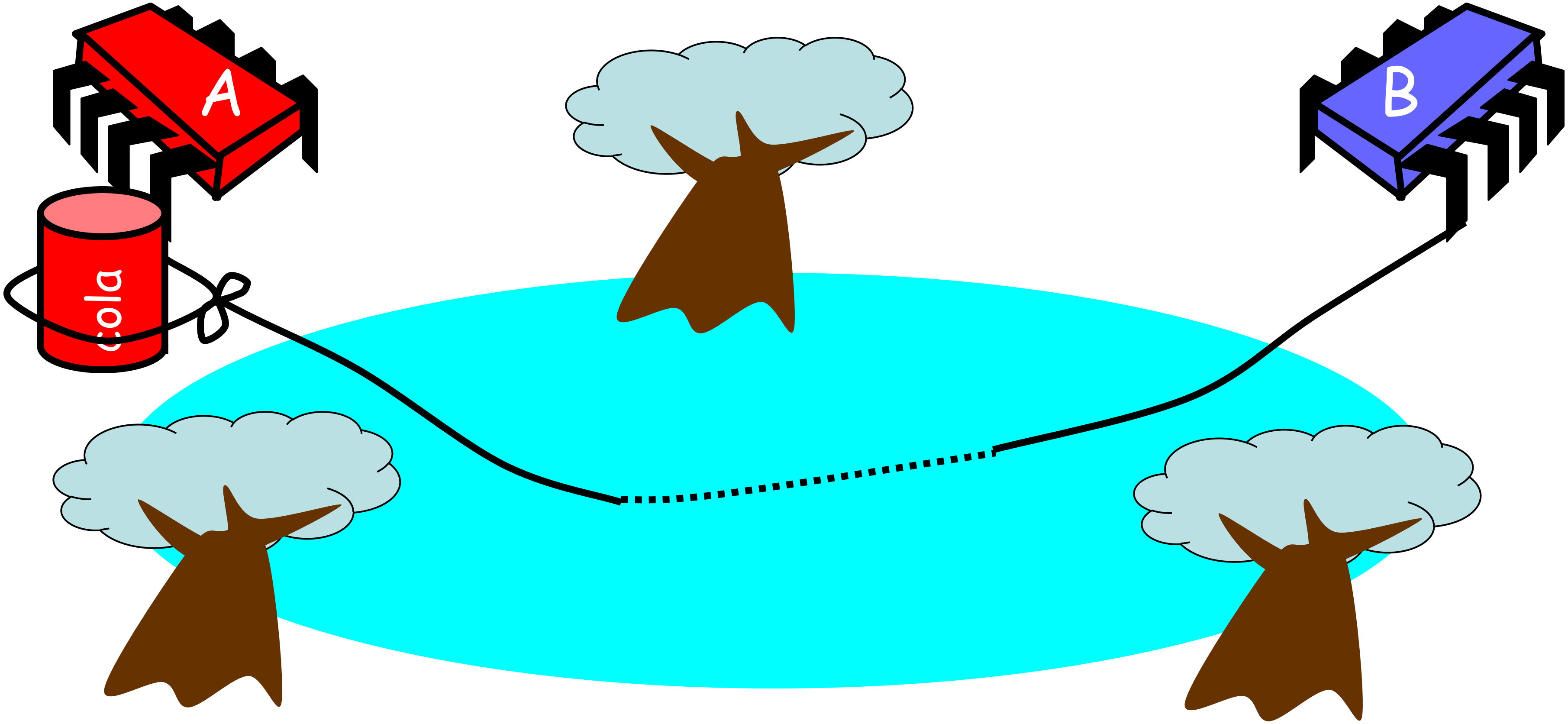
- Alice and Bob can't meet
  - Each has restraining order on other
  - So he puts food in the pond
  - And later, she releases the pets
- Avoid
  - Releasing pets when there's no food
  - Putting out food if uneaten food remains

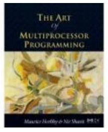
Producer/Consumer

- Need a mechanism so that
  - Bob lets Alice know when food has been put out
  - Alice lets Bob know when to put out more food

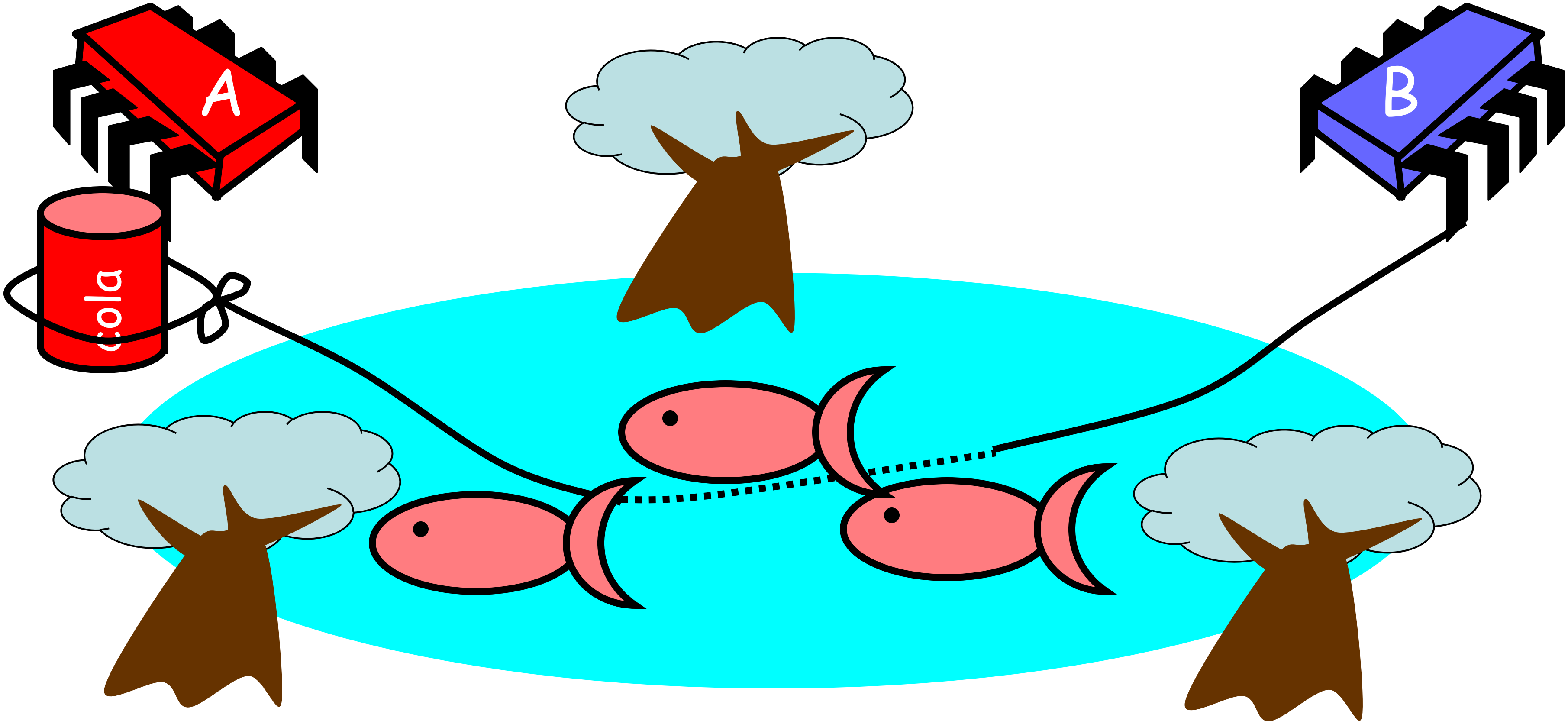


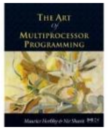
Surprise Solution



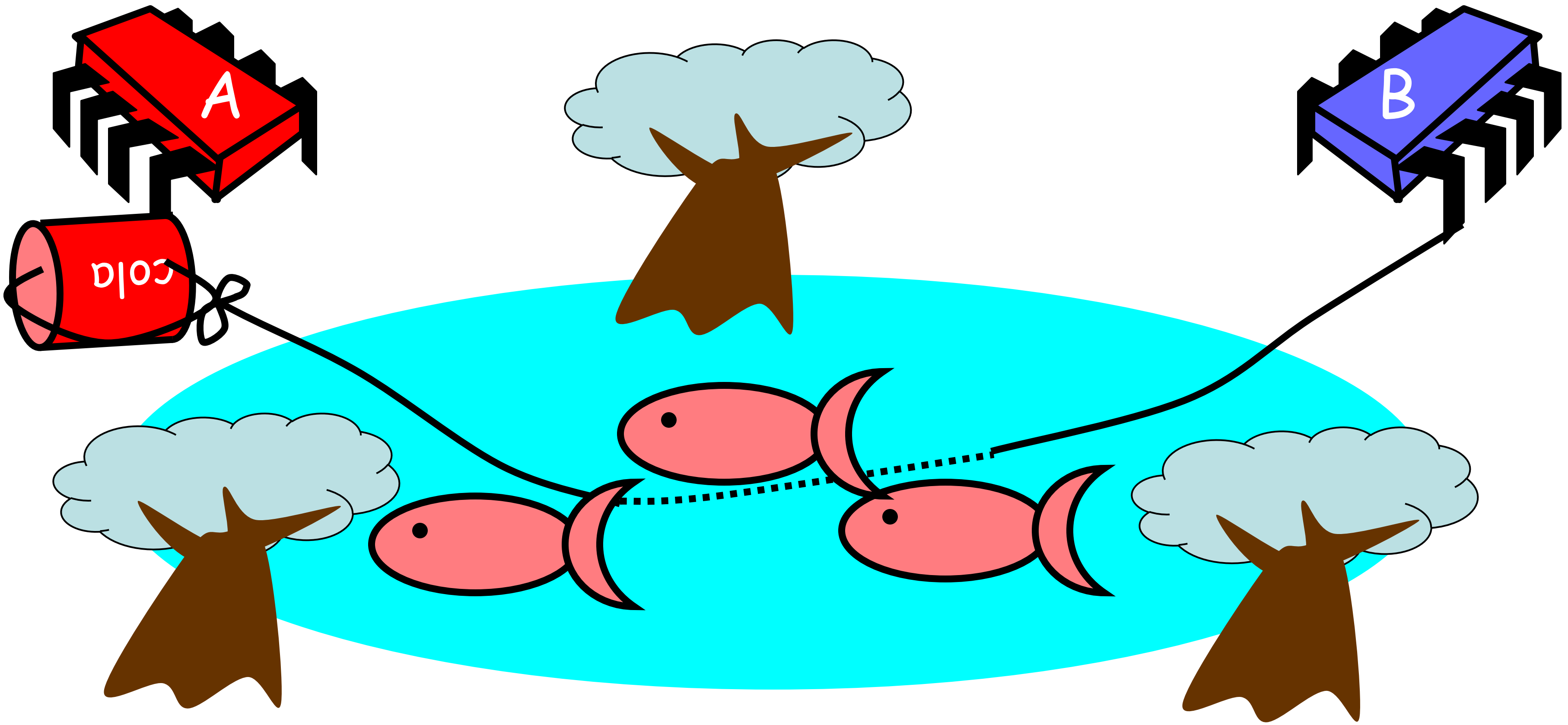


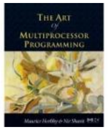
Bob puts food in Pond



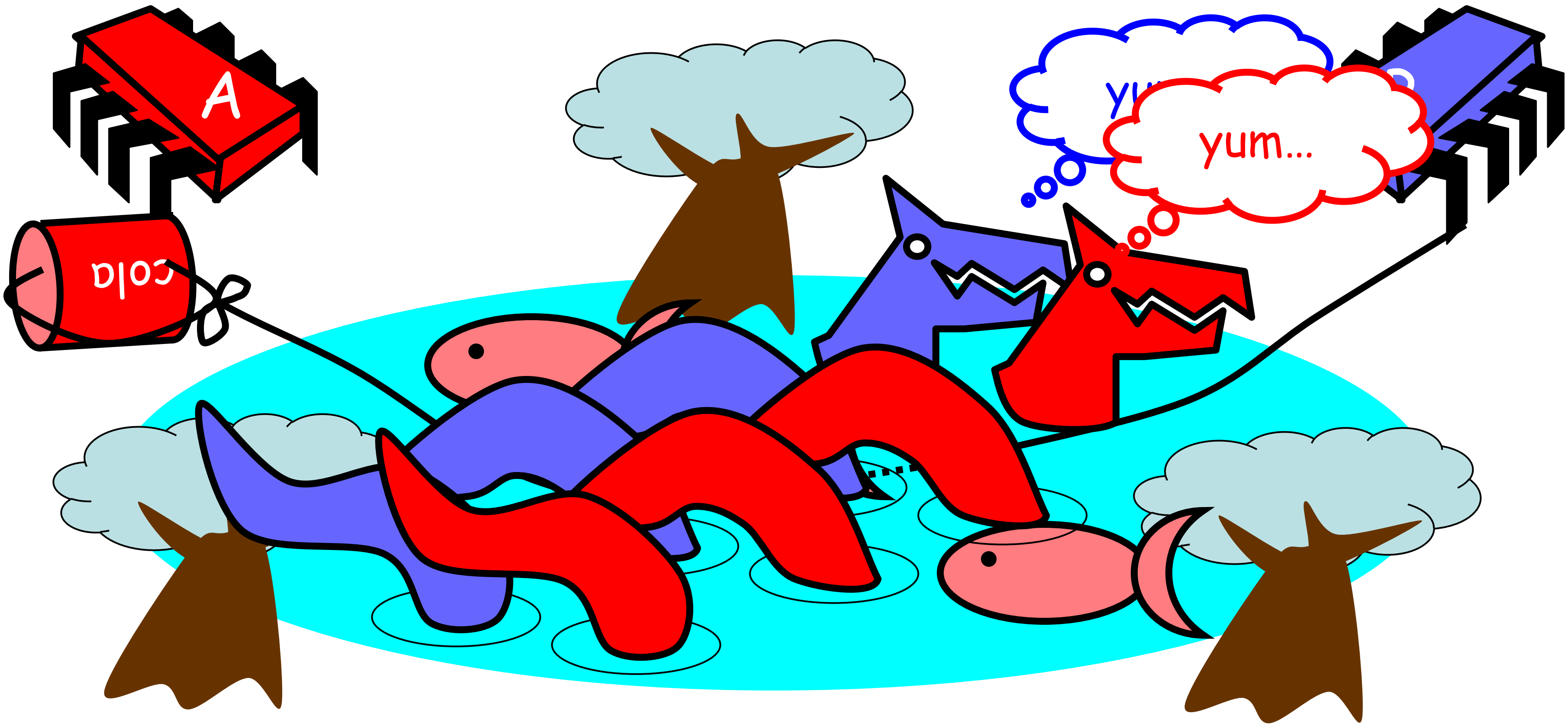


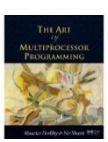
Bob knocks over Can



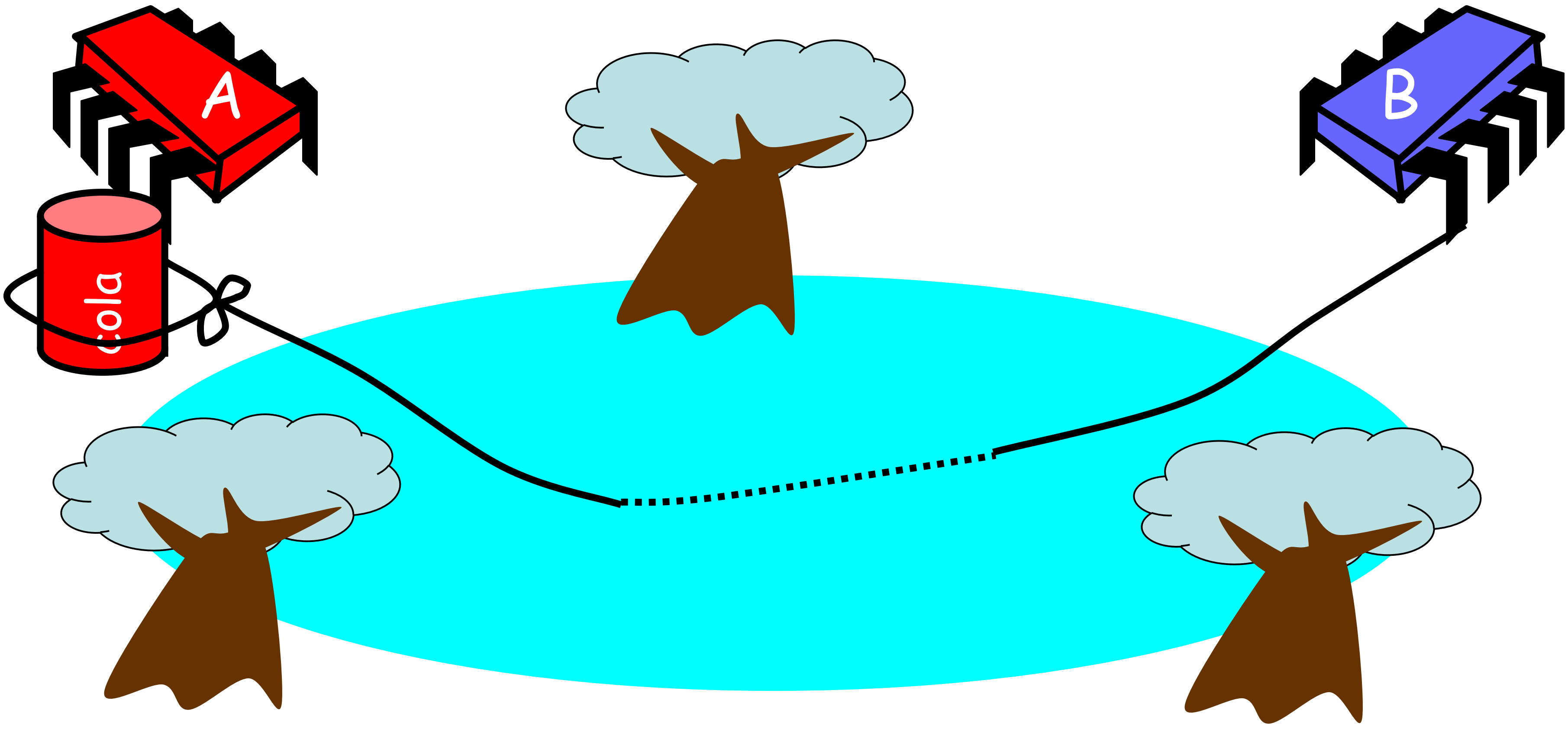


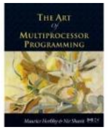
Alice Releases Pets





Alice Resets Can when Pets are Fed

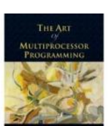




Pseudocode

```
while (true) {
  while (can.isUp()){};
  pet.release();
  pet.recapture();
  can.reset();
}
```

Alice's code



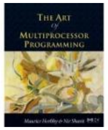
Pseudocode

```
while (true) {
  while (can.isUp()){};
  pet.release();
  pet.recapture();
  can.reset();
}
```

Alice's code

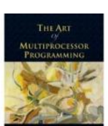
Bob's code

```
while (true) {
  while (can.isDown()){};
  pond.stockWithFood();
  can.knockOver();
}
```



Correctness

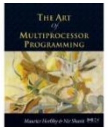
- Mutual Exclusion
  - Pets and Bob never together in pond



Correctness

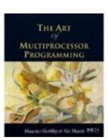
- Mutual Exclusion
  - Pets and Bob never together in pond
- No Starvation

  if Bob always willing to feed, and pets always famished, then pets eat infinitely often.

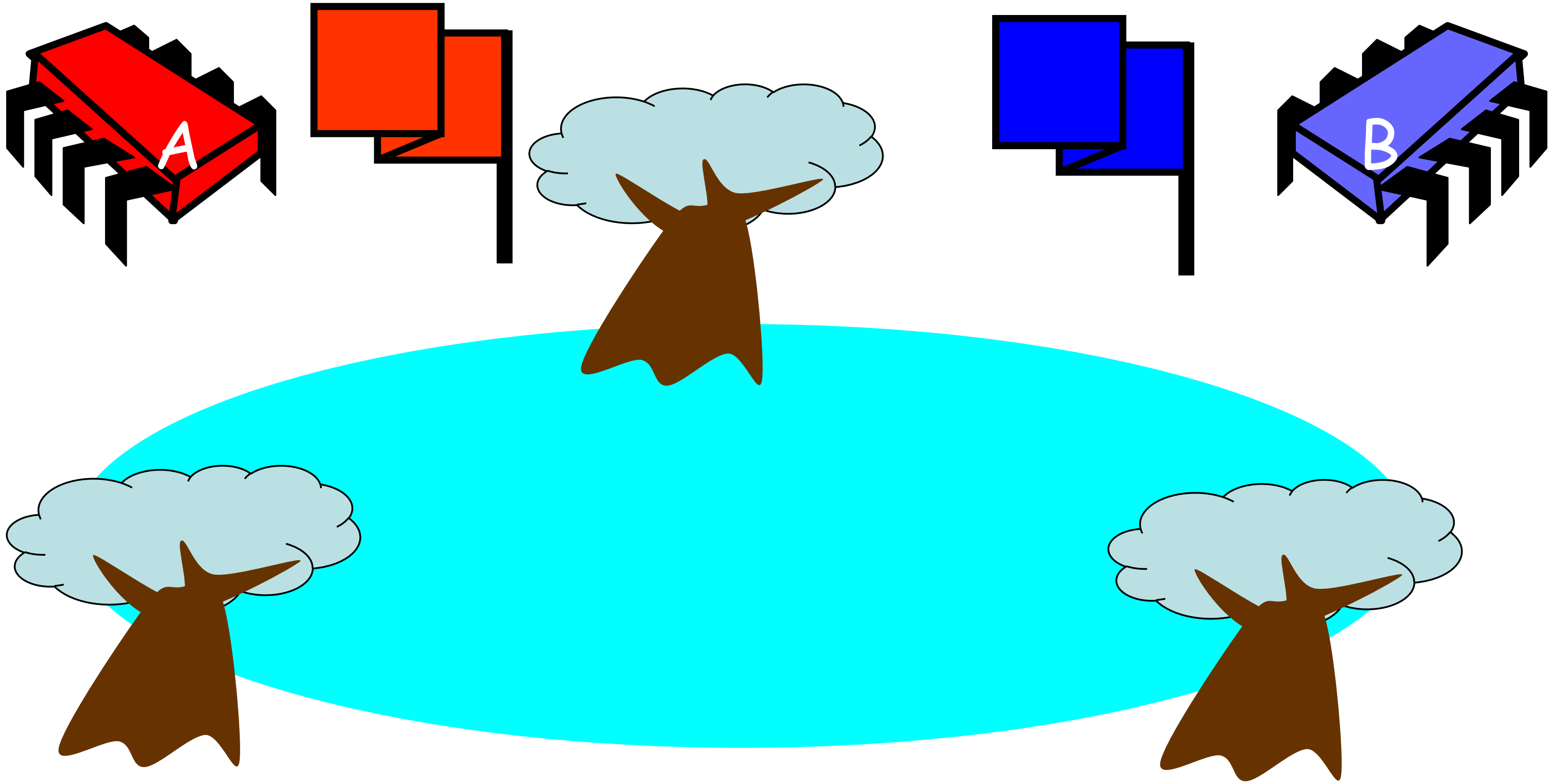


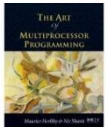
Correctness

- Mutual Exclusion — safety
  - Pets and Bob never together in pond
- No Starvation — liveness
  - if Bob always willing to feed, and pets always famished, then pets eat infinitely often.
- Producer/Consumer — safety
  - The pets never enter pond unless there is food, and Bob never provides food if there is unconsumed food.



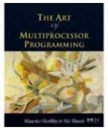
Could Also Solve Using Flags





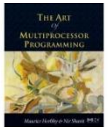
Waiting

- Both solutions use waiting
  - `while(mumble){}`
- In some cases waiting is ***problematic***
  - If one participant is delayed
  - So is everyone else
  - But delays are common & unpredictable



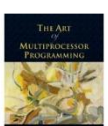
The Fable drags on …

- Bob and Alice still have issues



The Fable drags on …

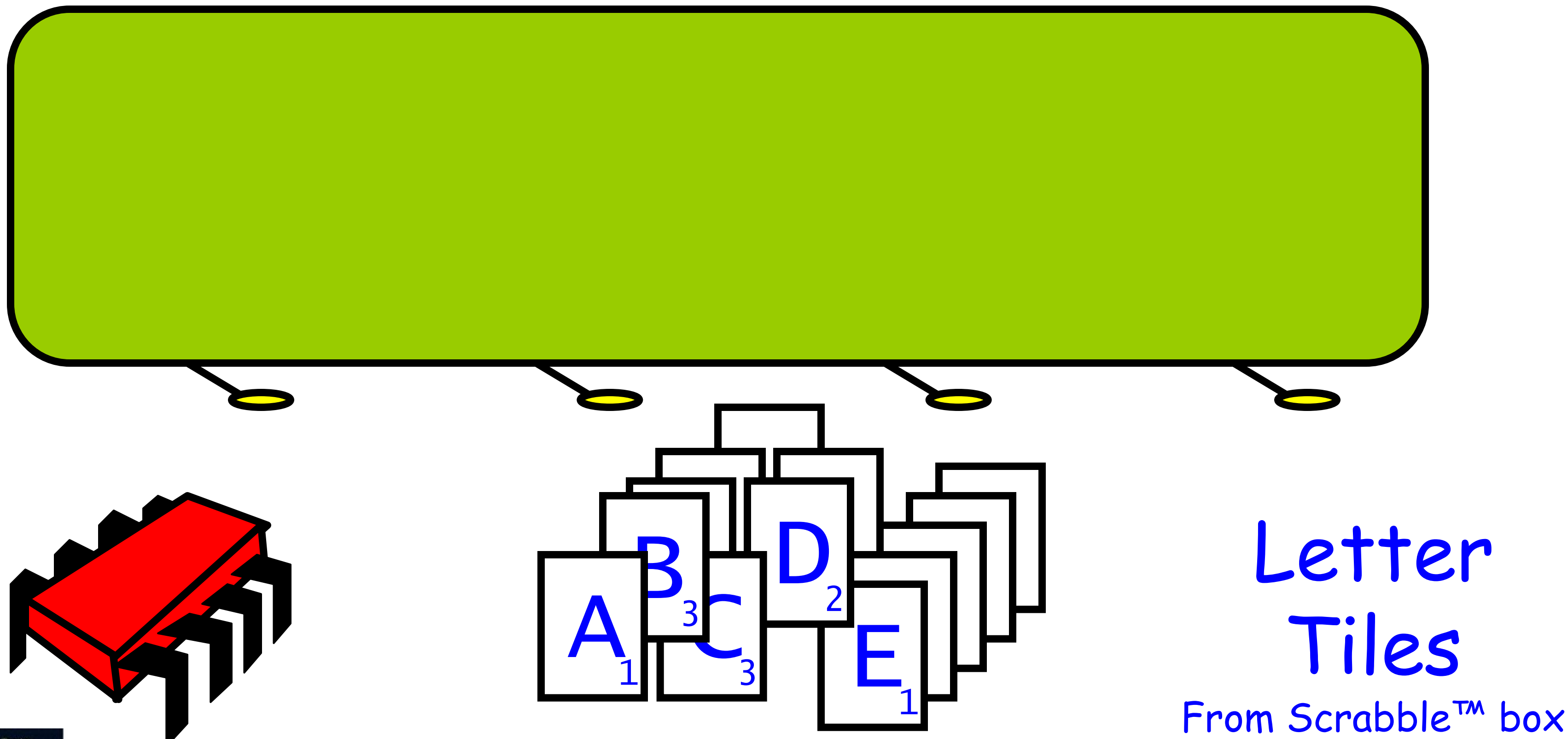
- Bob and Alice still have issues
- So they need to communicate

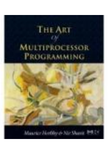


The Fable drags on …

- Bob and Alice still have issues
- So they need to communicate
- They agree to use billboards …

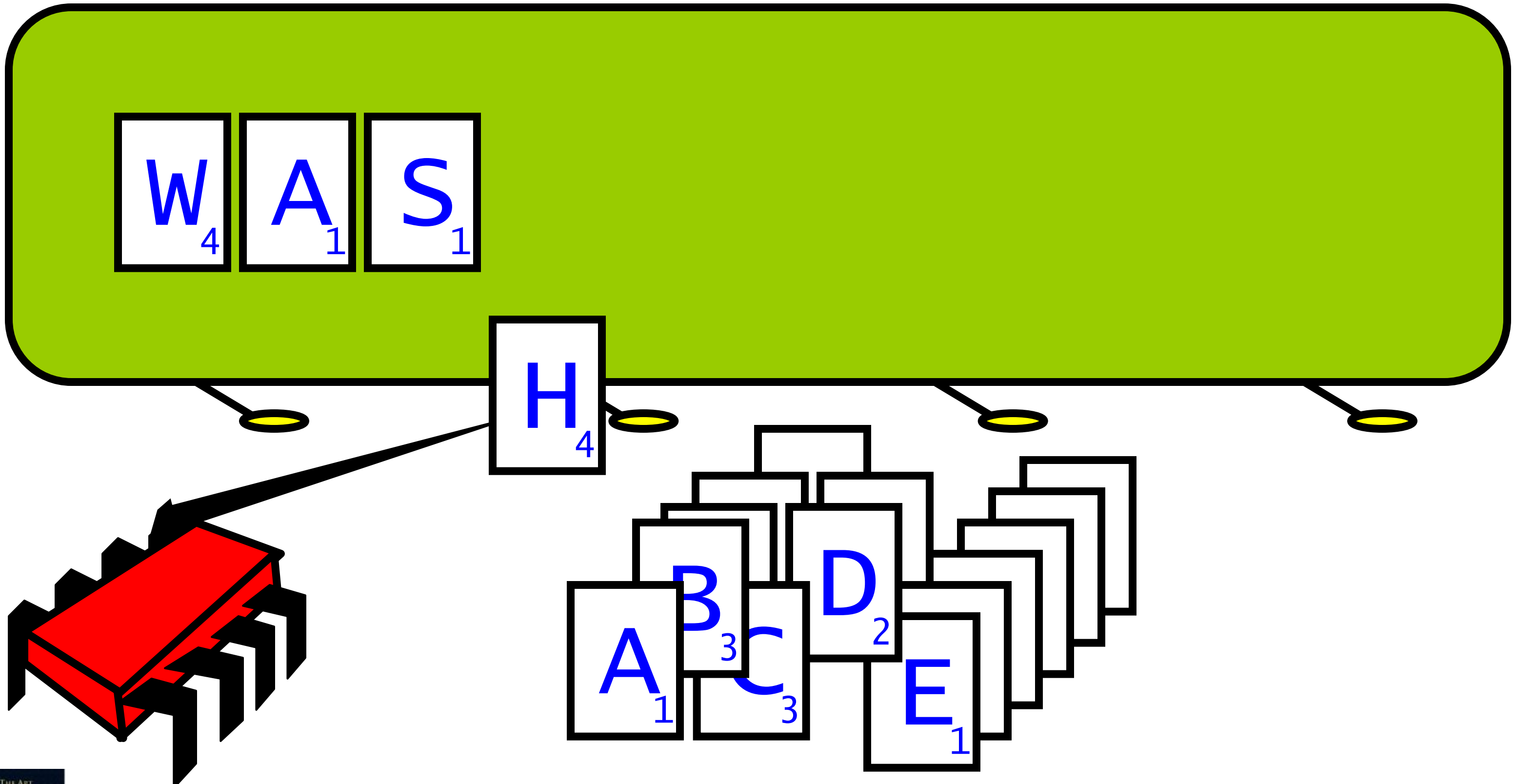
Billboards are Large

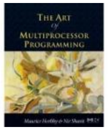




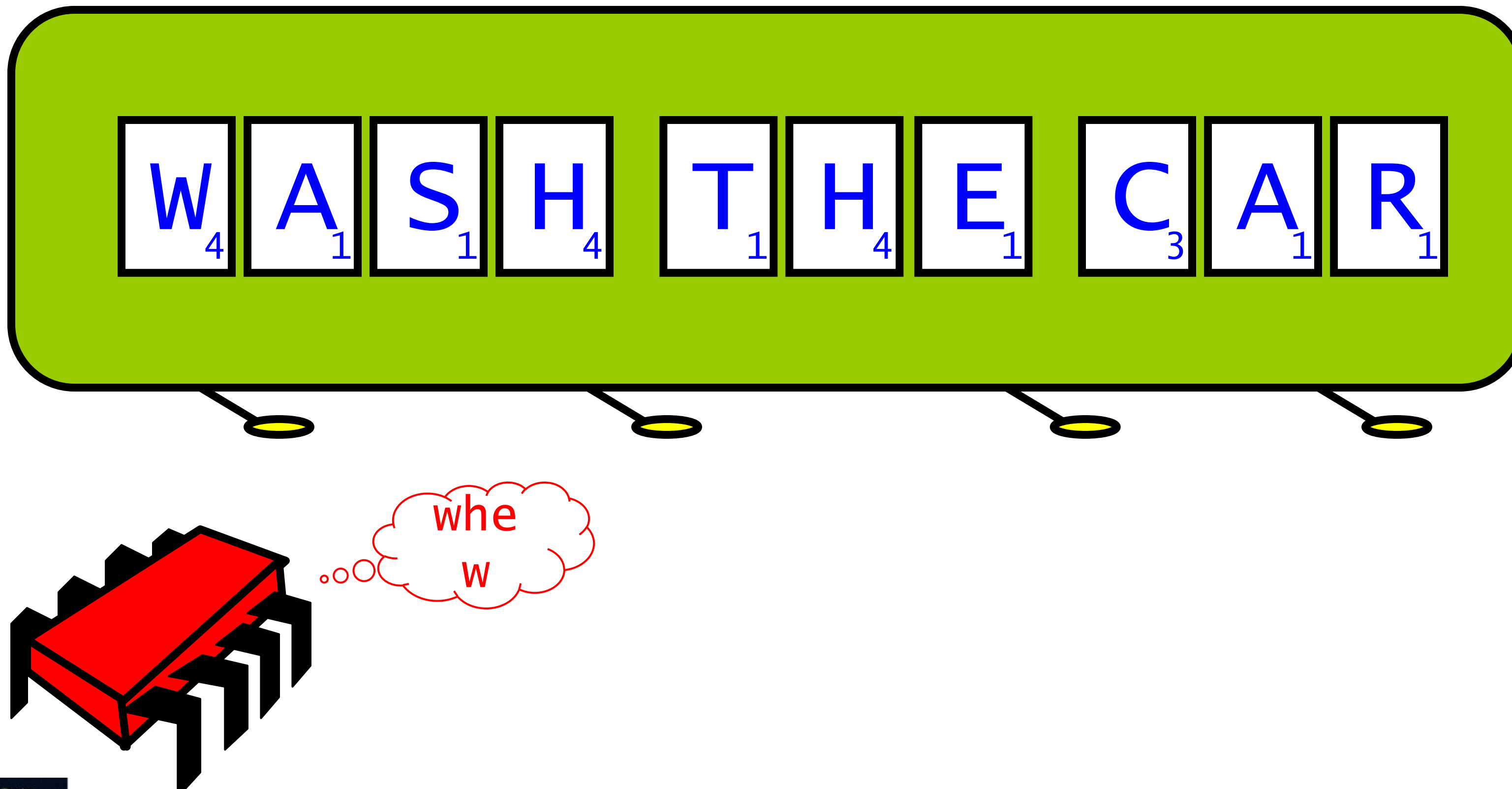
Art of Multiprocessor Programming

Write One Letter at a Time …

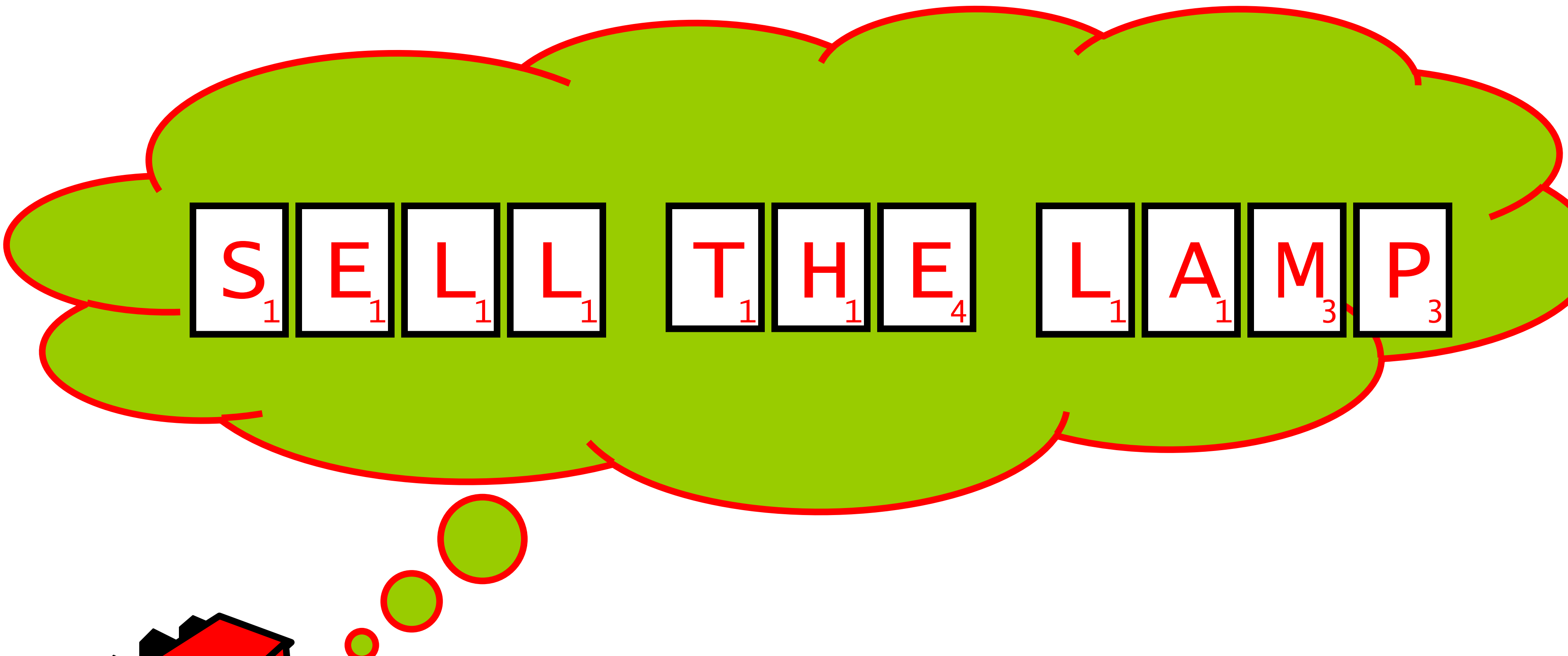




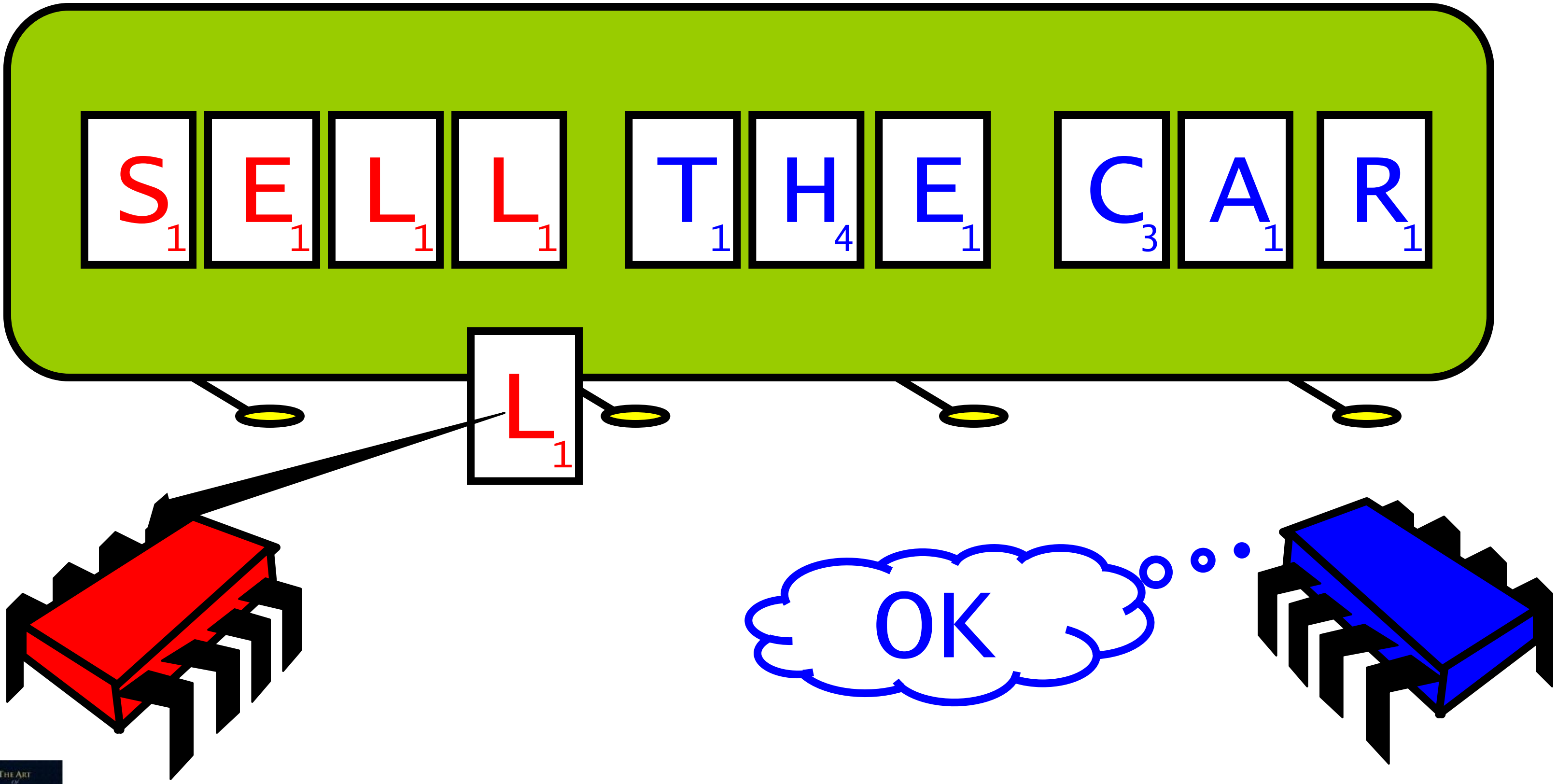
To post a message

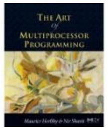


Let's send another message



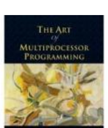
Uh-Oh





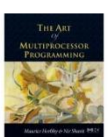
Readers/Writers

- Devise a protocol so that
  - Writer writes one letter at a time
  - Reader reads one letter at a time
  - Reader sees “snapshot”
    - Old message or new message
      - sell the lamp / wash the car
    - No mixed messages



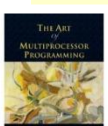
Readers/Writers (continued)

- Easy with mutual exclusion
- But mutual exclusion requires waiting
  - One waits for the other
  - Everyone executes sequentially
- Remarkably
  - We can solve R/W without mutual exclusion



Lock-free Reader-Writer

```
public class LockFreeRW {

  int head = 0, tail = 0;
  Item[QSIZE] items;

  public void write(Item x) {
    while (tail-head == QSIZE); // busy-wait
    items[tail % QSIZE] = x; tail++;
  }
  public Item read() {
     while (tail == head);     // busy-wait
     Item item = items[head % QSIZE]; head++;
     return item;
}}
```



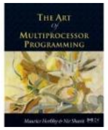
Esoteric?

- Java container `size()` method
- Single shared counter?
  - incremented with each `add()` and
  - decremented with each `remove()`
- Threads wait to exclusively access counter

performance bottleneck

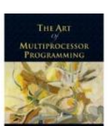
Readers/Writers Solution

- Each thread i has `size[i]` counter
  - only it increments or decrements.
- To get object's size, a thread reads a "snapshot" of all counters
- This eliminates the bottleneck



Why do we care?

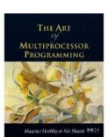
- We want as much of the code as possible to execute concurrently (in parallel)
- A larger sequential part implies reduced performance
- Amdahl's law: this relation is not linear…



Amdahl's Law

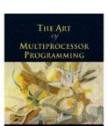
$$\text{Speedup} = \frac{\text{OldExecutionTime}}{\text{NewExecutionTime}}$$

…of computation given **n** CPUs instead of **1**



Amdahl's Law

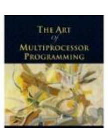
$$\textbf{Speedup} = \frac{1}{1 - p + \frac{p}{n}}$$



Amdahl's Law

$$\textbf{Speedup} = \frac{1}{1 - p + \frac{p}{n}}$$

Parallel fraction

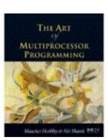


Amdahl's Law

$$\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}$$

Sequential fraction

Parallel fraction



Amdahl's Law

Sequential fraction

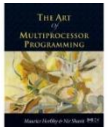
Parallel fraction

$$\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}$$

Number of processors

Example

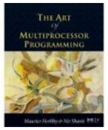
- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?



Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 2.17 = \frac{1}{1-0.6+\frac{0.6}{10}}$$



Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 3.57 = \frac{1}{1 - 0.8 + \frac{0.8}{10}}$$

Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = 5.26 = \frac{1}{1 - 0.9 + \frac{0.9}{10}}$$

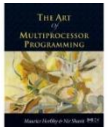
Example

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

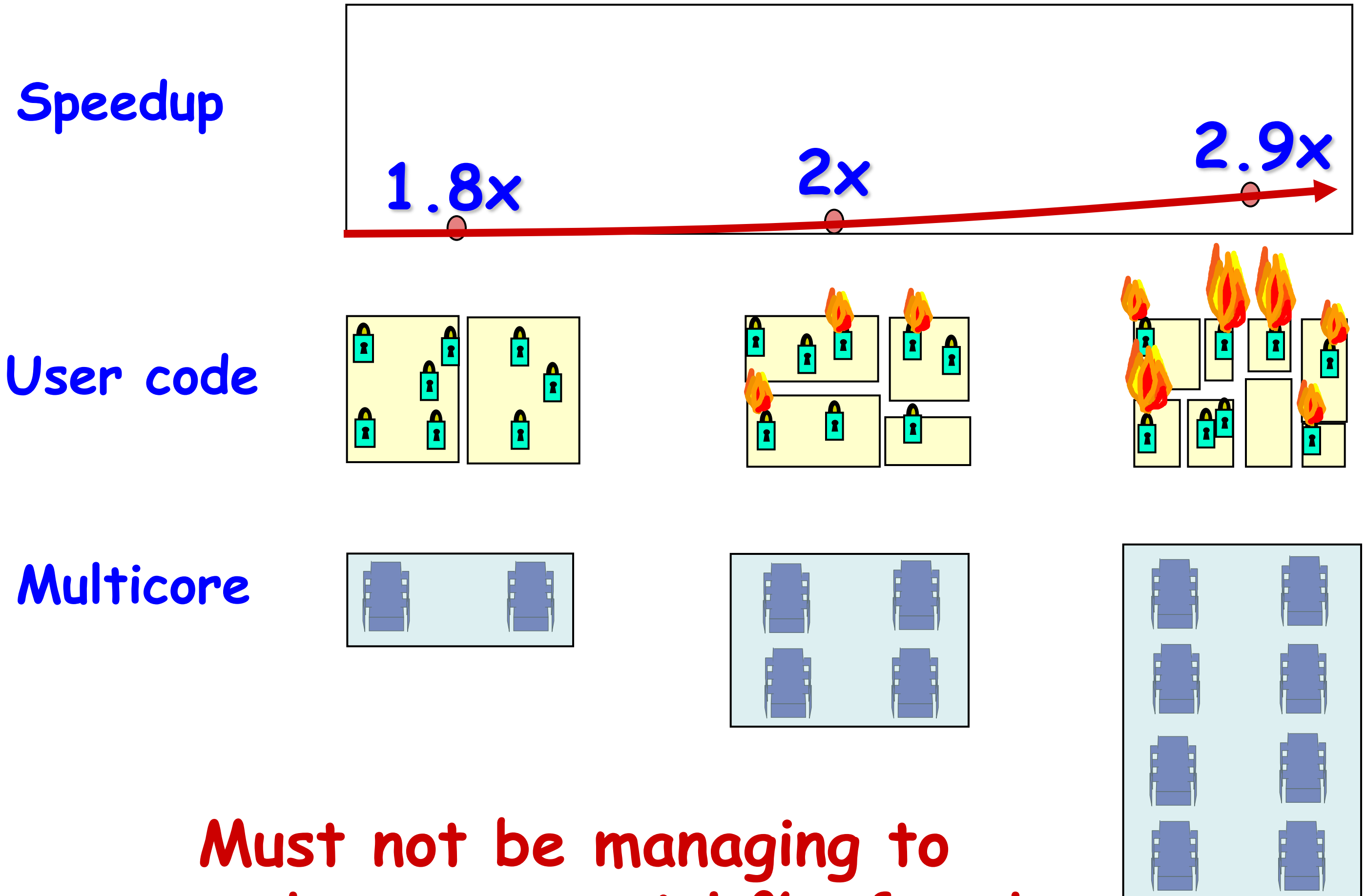
Example

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

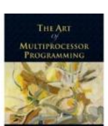
$$\text{Speedup} = 9.17 = \frac{1}{1 - 0.99 + \frac{0.99}{10}}$$



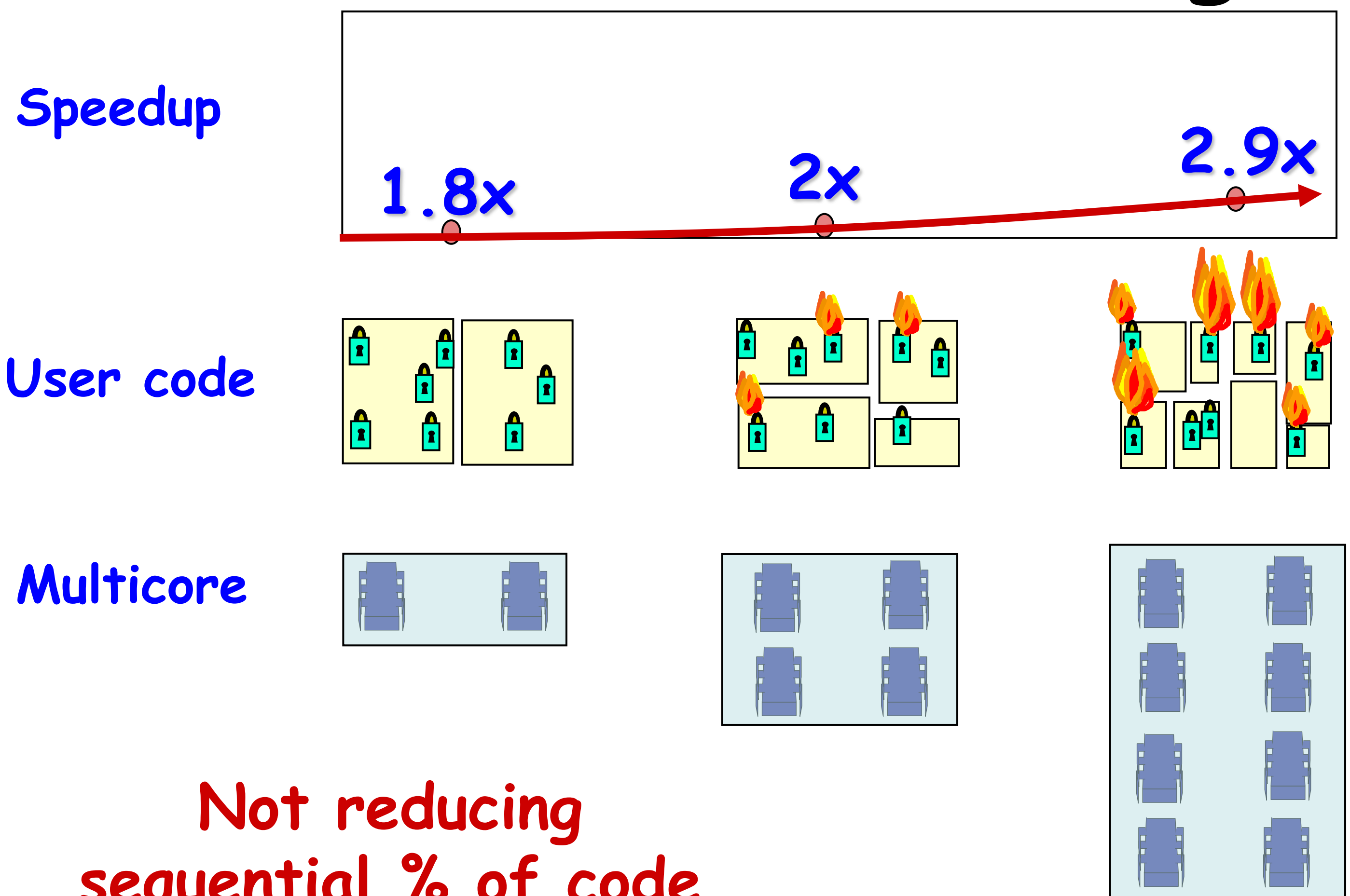
Back to Real-World Multicore Scaling



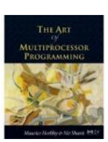
Must not be managing to reduce sequential % of code



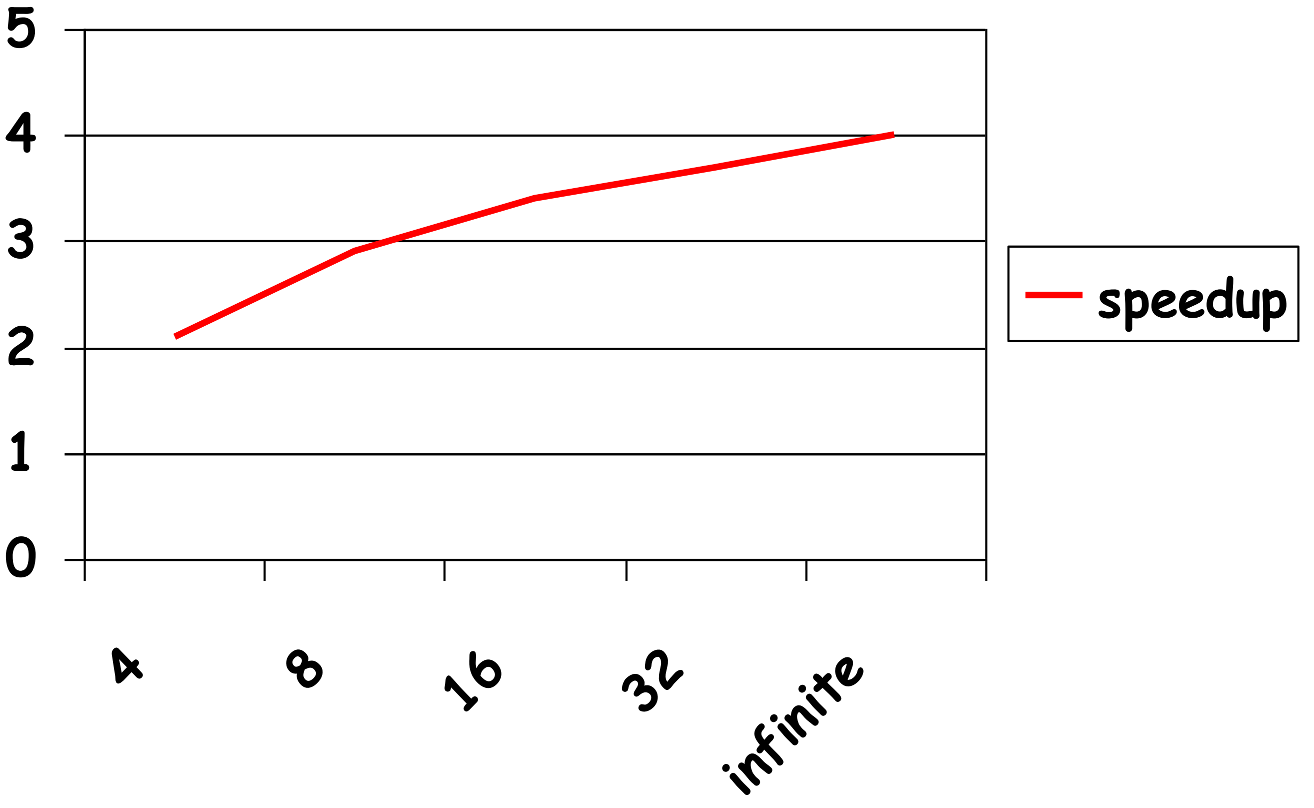
Back to Real-World Multicore Scaling

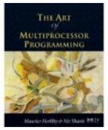


Not reducing sequential % of code



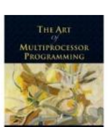
Diminishing Returns

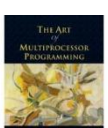




Multicore Programming

- This is what this course is about…
  - The % that is not easy to make concurrent yet may have a large impact on overall speedup
- Next:
  - A more serious look at mutual exclusion





What can we do ?
What should we do ?

- Understand the nature of the problem
  - Through formal models
- Synchrony vs. Asynchrony
  - Interrupts, cache-miss, pre-emption
  - pipeline flushing due to Branch mis-prediction