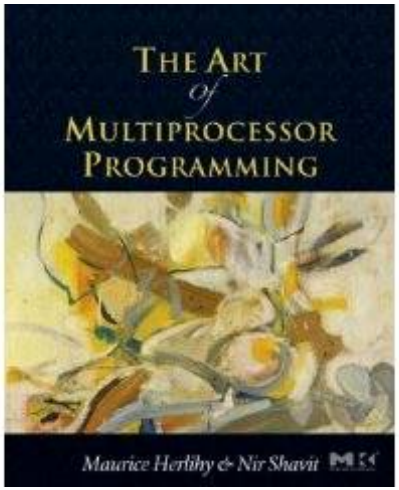
Programming Language Basics



Companion slides for

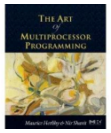
The Art of Multiprocessor Programming

by Maurice Herlihy & Nir Shavit

With some very minor changes by APS

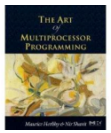
Languages for Multiprocessor Programming

- Java
- PThreads
  - C and C++
- C#
- MPI
- Etc…



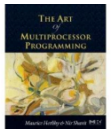
Threads

- Execution of a sequential program
- You can tell a thread
  - What to do
  - When to start
- You can
  - Wait for it to finish
- Other stuff:
  - Interrupt it, give it priority, etc.



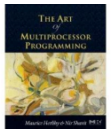
Threads in Java

- Class **`java.lang.Thread`**
- Each thread has a method
  - **`Void run()`**
- Executes when it starts
- Thread vanishes when it returns
- You must provide this method



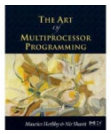
Creating a Thread

- Create a **`Runnable`** object
  - **`Runnable`** is an interface
  - Provides **`run()`** method
- Pass **`Runnable`** object to thread constructor



A Runnable Class

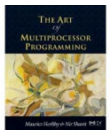
```
public class Hello implements Runnable {
  String message;
  public Hello(m) {
    message = m;
  }
  public void run() {
    System.out.println(message);
  }
}
```



A Runnable Class

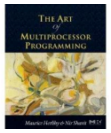
```
public class Hello implements Runnable {
  String message;
  public Hello(m) {
    message = m;
  }
  public void run() {
    System.out.println(message);
  }
}
```

Runnable interface



Creating a Thread

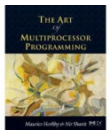
```
String m = “Hello from ” + i;
Runnable h = new Hello(m);
Thread t = new Thread(h);
```



Creating a Thread

```
String m = “Hello from ” + i;
Runnable h = new Hello(m);
Thread t = new Thread(h);
```

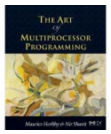
Create a Runnable object



Creating a Thread

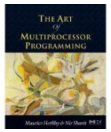
```
String m = “Hello from ” + i;
Runnable h = new Hello(m);
Thread t = new Thread(h);
```

Create the thread



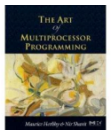
Syntactic Help

- Defining a single-use class like Hello can be a nuisance
- Java provides special syntax
- Anonymous inner classes
  - May be more trouble than it's worth
  - You should recognize it



Anonymous Inner Class

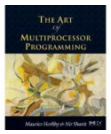
```
t = new Thread(
        new Runnable() {
          public void run() {
            System.out.println(m);
          }
        }
      );
```



Anonymous Inner Class

```
t = new Thread(
      new Runnable() {
        public void run() {
          System.out.println(m);
        }
      }
    );
```

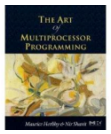
Creates object of anonymous Runnable class



Anonymous Inner Class

```
t = new Thread(
        new Runnable() {
          public void run() {
            System.out.println(m);
          }
        }
      );
```

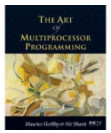
Calls Thread constructor
with anonymous object



Starting a Thread

```
t.start();
```

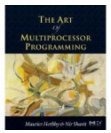
- Starts the new thread
- Caller returns immediately
- Caller & thread run in parallel



Joining a Thread

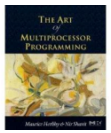
```
t.join();
```

- Blocks the caller
- Waits for the thread to finish
- Returns when the thread is done



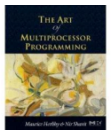
Monitors

- Each object has an implicit lock
- Managed by **`synchronized`** modifer
  - Methods
  - Code blocks
- OK for easy cases
- Not always for hard cases



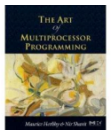
Call Center Scenario

- Calls arrive faster than they can be answered
  - Play recorded message
    - “your call is very important to us …”
  - Put call in queue
    - Play insipid music …
  - Operators dequeue call when ready



Bad Queue Implementation

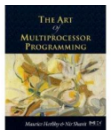
```
class Queue<T> {
  int head = 0, tail = 0;
  T[] items = new T[QSIZE];
  public enq(T x) {
    items[(tail++) % QSIZE] = x;
  }
  public T deq() {
    return items[(head++) % QSIZE]
  }}
```



Bad Queue Implementation

```
class Queue<T> {
  int head = 0, tail = 0;
  T[] items = new T[QSIZE];
  public enq(T x) {
    items[(tail++) % QSIZE] = x;
  }
  public T deq() {
    return items[(head++) % QSIZE]
  }}
```

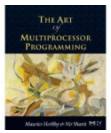
Works for generic type T



Bad Queue Implementation

```
class Queue<T> {
  int head = 0, tail = 0;
  T[] items = new T[QSIZE];
  public enq(T x) {
    items[(tail++) % QSIZE] = x;
  }
  public T deq() {
    return items[(head++) % QSIZE]
  }}
```

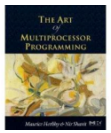
Array of T items



Bad Queue Implementation

```
class Queue<T> {
  int head = 0, tail = 0;
  T[] items = new T[QSIZE];
  public enq(T x) {
    items[(tail++) % QSIZE] = x;
  }
  public T deq() {
    return items[(head++) % QSIZE]
  }}
```

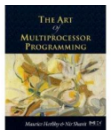
next slot to dequeue, 1st empty slot



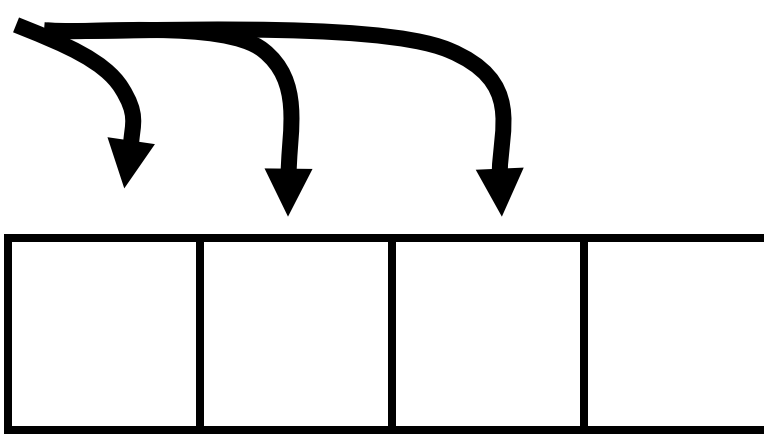
Bad Queue Implementation

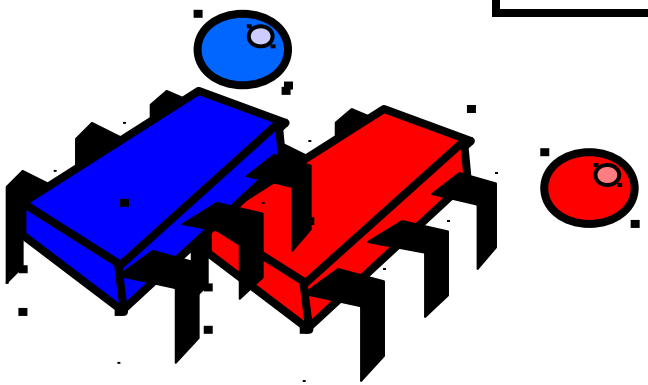
```
class Queue<T> {
  int head = 0, tail = 0;
  T[] items = new T[QSIZE];
  public enq(T x) {
    items[(tail++) % QSIZE] = x;
  }
  public T deq() {
    return items[(head++) % QSIZE]
  }}
```

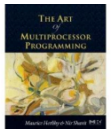
Put in empty slot, advance head



Of course, this doesn't work

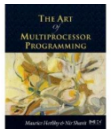




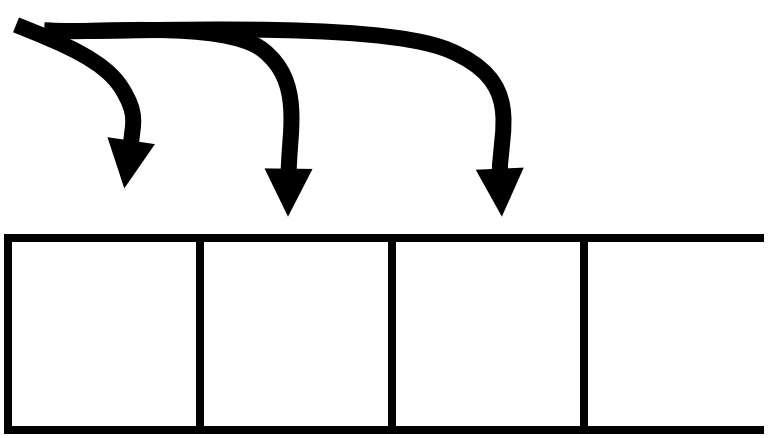


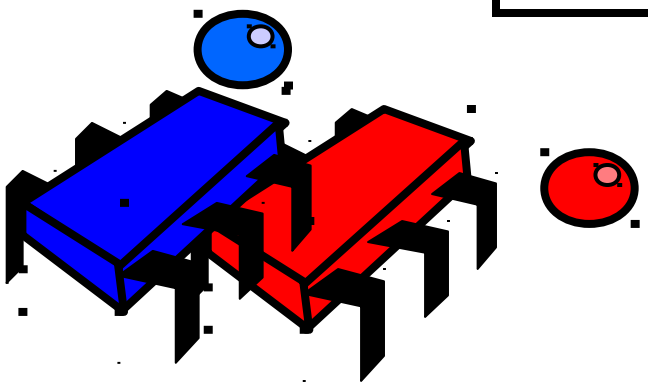
Mutual Exclusion

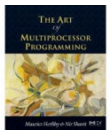
- Only one thread modifying queue fields at a time
- Use **`synchronized`** methods
  - Locks object on call
  - Releases lock on return



Mutual Exclusion

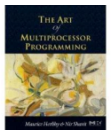






Synchronized Method

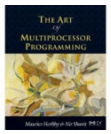
```
class Queue<T> {
  …
  public synchronized enq(T x) {
    items[(tail++) % QSIZE];
  }
  ...
}
```



Synchronized Method

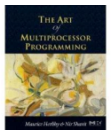
```
class Queue<T> {
  …
  public synchronized enq(T x) {
    items[(head++) % QSIZE];
  }
  ...
}
```

Lock acquired on entry, released on exit



Syntactic Sugar

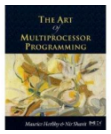
```
class Queue<T> {
  …
  public enq(T x) {
    synchronized (this) {
      items[(tail++) % QSIZE];
    }
  }
  ...
}
```



Syntactic Sugar

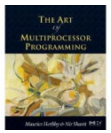
```
class Queue<T> {
  …
  public enq(T x) {
    synchronized (this) {
      items[(tail++) % QSIZE];
    }
  }
  ...
}
```

Same meaning, more verbose



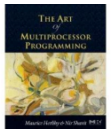
Vocabulary

- A **`synchronized`** method locks the object
- No other thread can call another **`synchronized`** method for that same object
- Code in middle is critical section



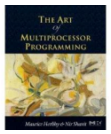
Re-entrant Locks

- What happens if you lock the same object twice?
  - In Java, no deadlock
  - Keeps track of number of times locked and unlocked
  - Unlock occurs when they balance out



Still Doesn't Work

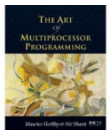
```
class Queue<T> {
  …
  public synchronized enq(T x) {
    items[(tail++) % QSIZE] = x;
  }
  ...
}
```



Still Doesn't Work

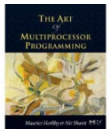
```
class Queue<T> {
  …
  public synchronized enq(T x) {
    items[(tail++) % QSIZE] = x;
  }
  ...
}
```

What if the array is full?



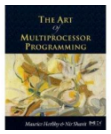
Waiting

- What if
  - Enqueuer finds a full array?
  - Dequeuer finds an empty array?
- Throw an exception?
  - What can caller do?
  - Repeated retries wasteful
- Wait for something to happen



Waiting Synchronized Method

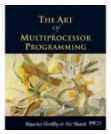
```
class Queue<T> {
  …
  public synchronized enq(T x) {
    while (head == tail) {};
    items[(tail++) % QSIZE] = x;
  }
  ...
}
```



Waiting Synchronized Method

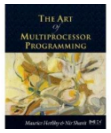
```
class Queue<T> {
  …
  public synchronized enq(T x) {
    while (head == tail) {};
    items[(tail++) % QSIZE] = x;
  }
  ...
}
```

Spin while the array is full



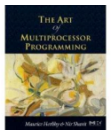
Deadlock

- Enqueuer is
  - Waiting for a dequeuer
  - While holding the lock
- Dequeuer
  - Waiting for enqueuer to release lock
- Nothing will ever happen



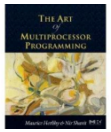
Waiting Thread

- Release lock while waiting
- When “something happens”
  - Re-acquire lock
  - Either
    - Re-release lock & resume waiting
    - Finish up and return



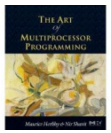
Styles of Waiting

- Spinning
  - Repeatedly retest condition
- Blocking
  - Ask OS to run someone else



Styles of Waiting

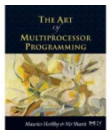
- Spinning
  - Good for very short intervals
  - Expensive to call OS
  - Works only on multiprocessors!
- Blocking
  - Good for longer intervals
  - Processor can do work
- Clever libraries sometimes mix



The wait() Method

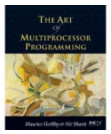
```
q.wait();
```

- Releases lock on q
- Sleeps (gives up processor)
- Awakens (resumes running)
- Reacquires lock & returns



The wait() Method

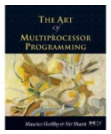
```
class Queue<T> {
  …
  public synchronized enq(T x) {
    while (head == tail) {
      wait();
    };
    items[(head++) % QSIZE] = x;
  }
  ...
}
```



Waiting Synchronized Method

```
class Queue<T> {
  …
  public synchronized enq(T x) {
    while (head == tail) {
      wait();
    };
    items[(head++) % QSIZE] = x;
  }
  ...
}
```

Keep retesting condition

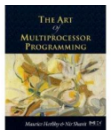


Waiting Synchronized Method

```
class Queue<T> {
  …
  public synchronized enq(T x) {
    while (head == tail) {
      wait();
    };
    items[(head++) % QSIZE] = x;
  }
  ...
}
```

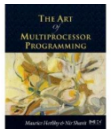
Keep retesting condition

Release lock & sleep



Wake up and Smell the Coffee

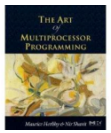
- When does a waiting thread awaken?
  - Must be notified by another thread
  - when something has happened
- Failure to notify in a timely way is called a "lost wakeup"



The wait() Method

```
q.notify();
```

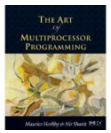
- Awakens one waiting thread
- Which will reacquire lock & returns



The wait() Method

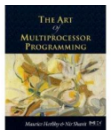
```
q.notifyAll();
```

- Awakens all waiting threads
- Which will reacquire lock & return



The wait() Method

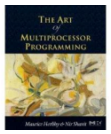
```
public synchronized enq(T x) {
  while (head == tail) {
    wait();
  };
  items[(head++) % QSIZE] = x;
  if ((head-tail+QSIZE) % QSIZE == 1) {
    notify();
  }
}
```



The wait() Method

```
public synchronized enq(T x) {
  while (head == tail) {
    wait();
  };
  items[(head++) % QSIZE] = x;
  if ((head-tail+QSIZE) % QSIZE == 1) {
    notify();
  }
}
```

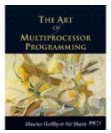
Wait for empty slot



The wait() Method

```
public synchronized enq(T x) {
  while (head == tail) {
    wait();
  };
  items[(head++) % QSIZE] = x;
  if ((head-tail+QSIZE) % QSIZE == 1) {
    notify();
  }
}
```

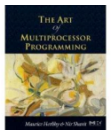
Stuff item into array



The wait() Method

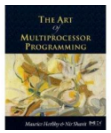
```
public synchronized enq(T x) {
  while (head == tail) {
    wait();
  };
  items[(head++) % QSIZE] = x;
  if ((head-tail+QSIZE) % QSIZE == 1) {
    notify();
  }
}
```

If the queue was empty, wake up a dequeuer

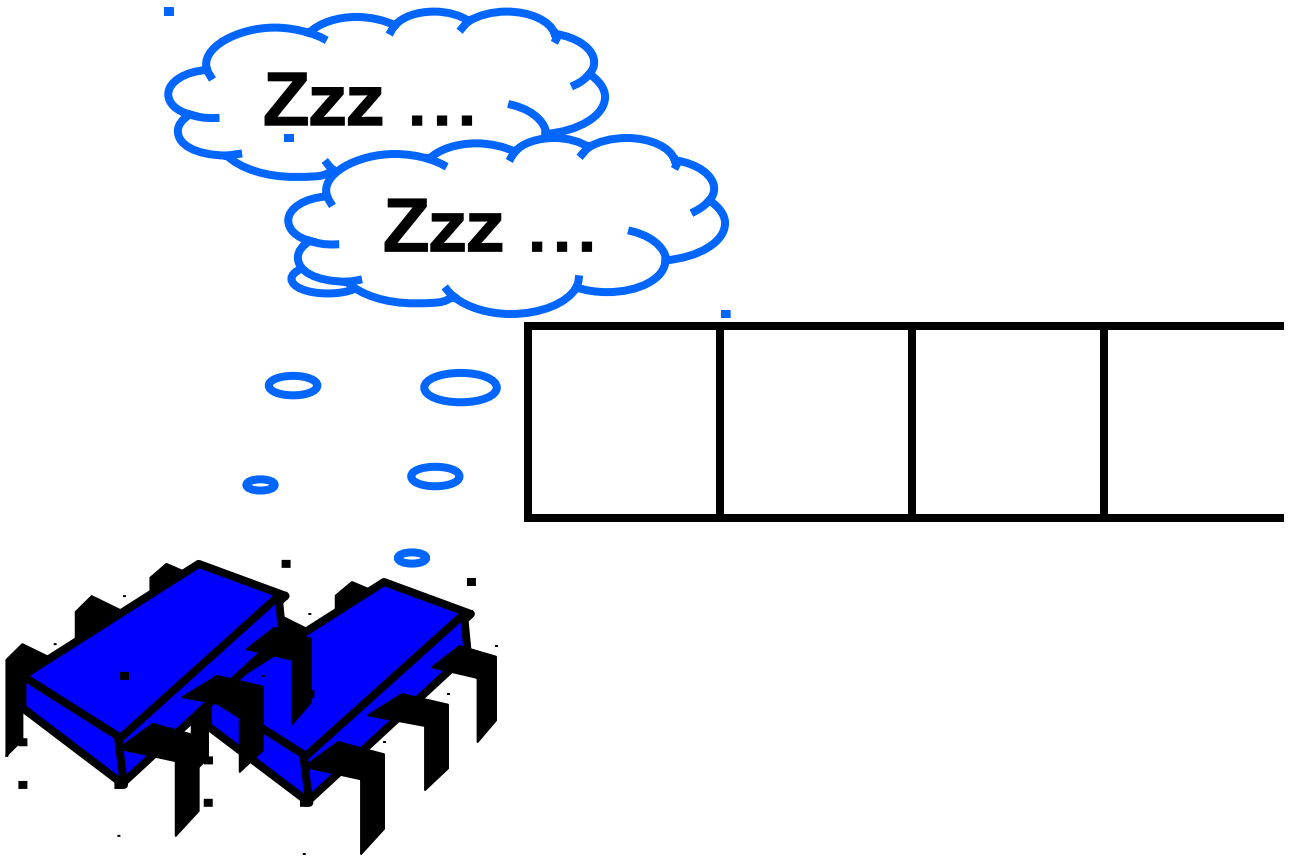


Lost Wakeup

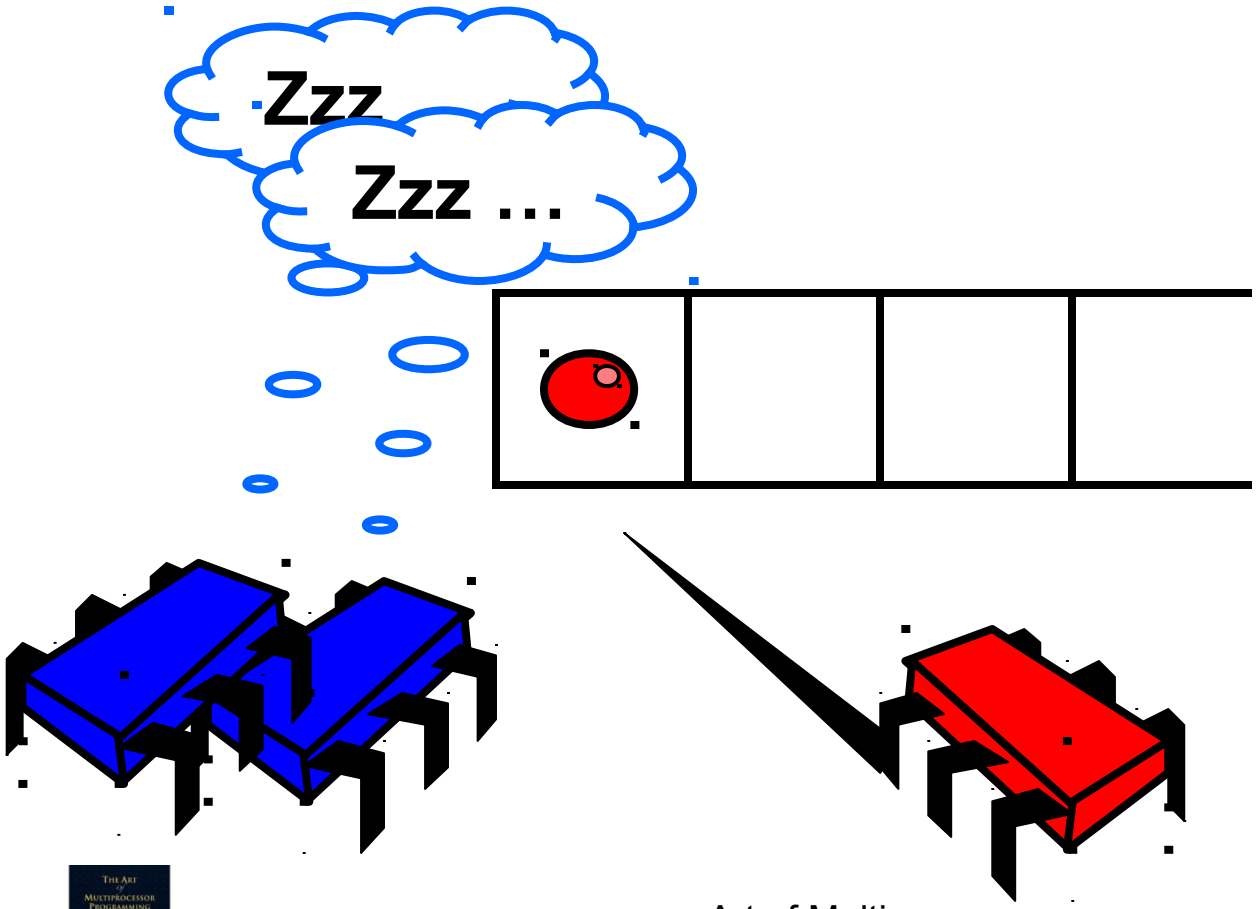
- This code has a lost wakeup bug
- Possible to have
  - Waiting dequeuer
  - Non-empty queue
- Because not enough threads awakened



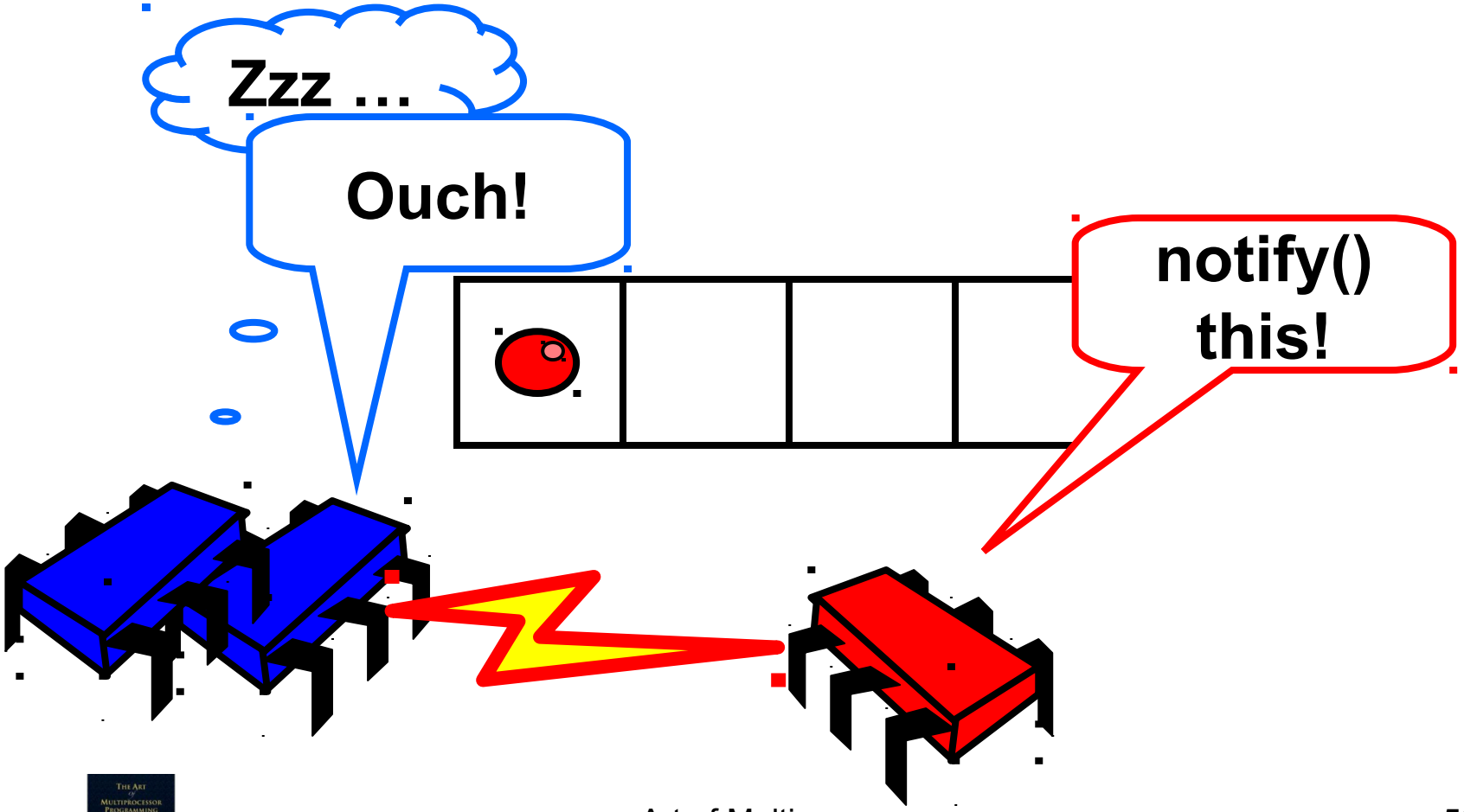
Empty queue, waiting dequeuers

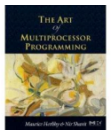


Enqueuer puts item in queue

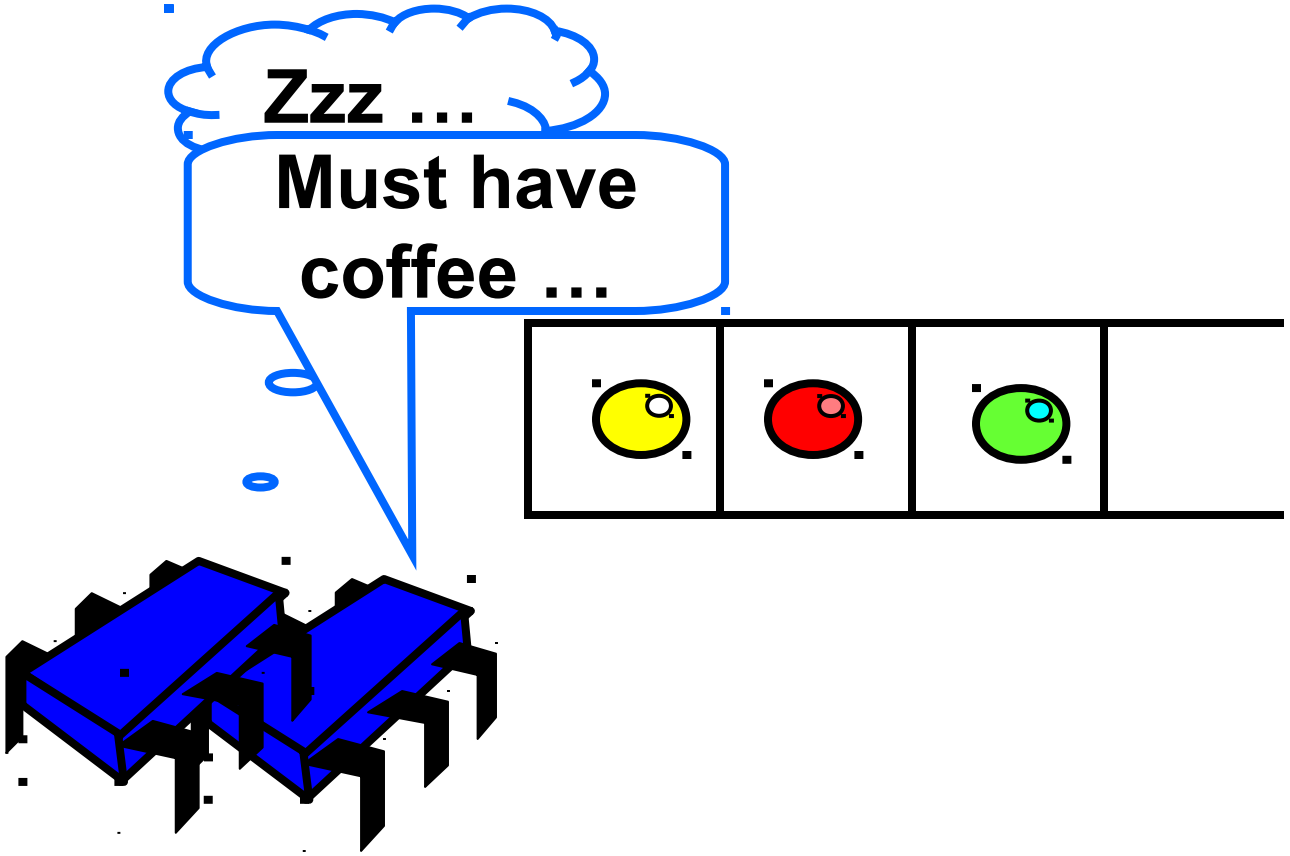


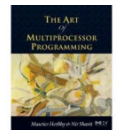
Since queue was empty, wakes dequeuer



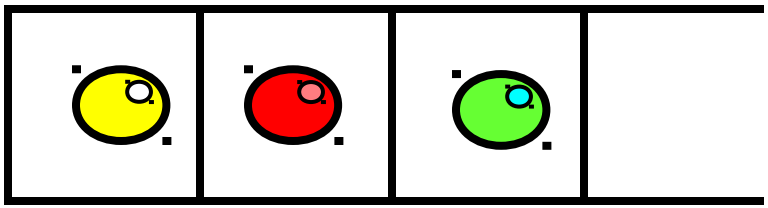


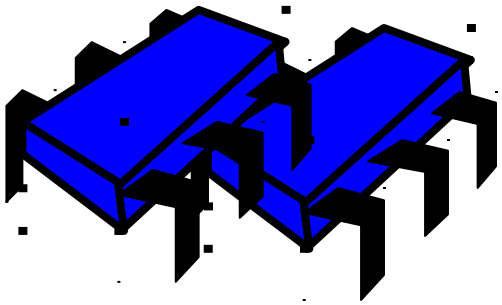
1st Dequeuer slow, overtaken by enqueuers

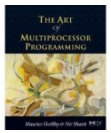




1st Dequeuer finishes

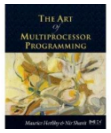






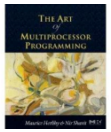
Solutions

- Don’t write buggy code d’oh!
- Always call **`notifyAll()`**
- Can also use timed waits
  - Wake up after specified time



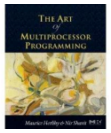
Thread-Local Data

- In many of our examples we assume
  - Threads have unique ids
  - In range 0,…,n-1
- Where do they come from?
  - Long-lived data
  - Unique to a thread



Thread-Local Data in Java

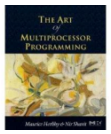
- **`ThreadLocal<T>`** class
- No built-in language support
- Library classes
  - Syntax is awkward
  - Very useful anyway



ThreadLocal methods

```
ThreadLocal<T> local;
T x = …;
local.set(x);
```

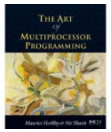
- Changes calling thread's version of object
- Other threads' versions unaffected



ThreadLocal methods

```
ThreadLocal<T> local;
T x = local.get();
```

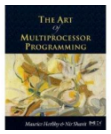
- Returns calling thread's version of object



Initializing ThreadLocals

```
T x = local.initialValue();
```

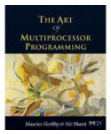
- Called by **`get()`** method the first time the thread-local variable is accessed.



Example

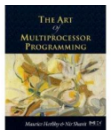
```
int me = ThreadID.get()
```

- Return unique thread id
- Take a number first time called



Thread-Local IDs

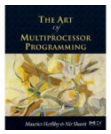
```
public class ThreadID {
 private static volatile int nextID = 0;
 private static LocalID threadID =
    new LocalID();
  public static int get() {
    return threadID.get();
  }
  … // define LocalID here
}
```



Thread-Local IDs

```
public class ThreadID {
 private static volatile int nextID = 0;
 private static LocalID threadID =
    new LocalID();
  public static int get() {
    return threadID.get();
  }
  … // define LocalID here
}
```

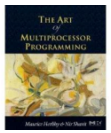
Next ID to assign



Thread-Local IDs

```
public class ThreadID {
 private static volatile int nextID = 0;
 private static LocalID threadID =
     new LocalID();
   public static int get() {
     return threadID.get();
   }
   … // define LocalID here
}
```

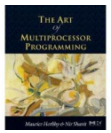
Declare & initialize thead-local ID



Thread-Local IDs

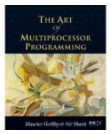
```
public class ThreadID {
 private static volatile int nextID = 0;
 private static LocalID threadID =
    new LocalID();
 public static int get() {
    return threadID.get();
 }
  … // define LocalID here
}
```

Return value of thread-local ID



The Inner Class

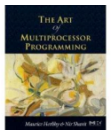
```
private static class LocalID
    extends ThreadLocal<Integer> {
      protected synchronized Integer
        initialValue() {
          return nextID++;
        }
}
```



The Inner Class

```
private static class LocalID
    extends ThreadLocal<Integer> {
      protected synchronized Integer
        initialValue() {
          return nextID++;
        }
}
```

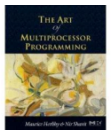
Subclass of ThreadLocal<Integer>



The Inner Class

```
private static class LocalID
    extends ThreadLocal<Integer> {
  protected synchronized Integer
    initialValue() {
      return nextID++;
    }
}
```

Overrides initialValue()



Summary

- Threads
  - And how to control them
- Synchronized methods
  - Wait, notify, and NotifyAll
- Thread-Local objects