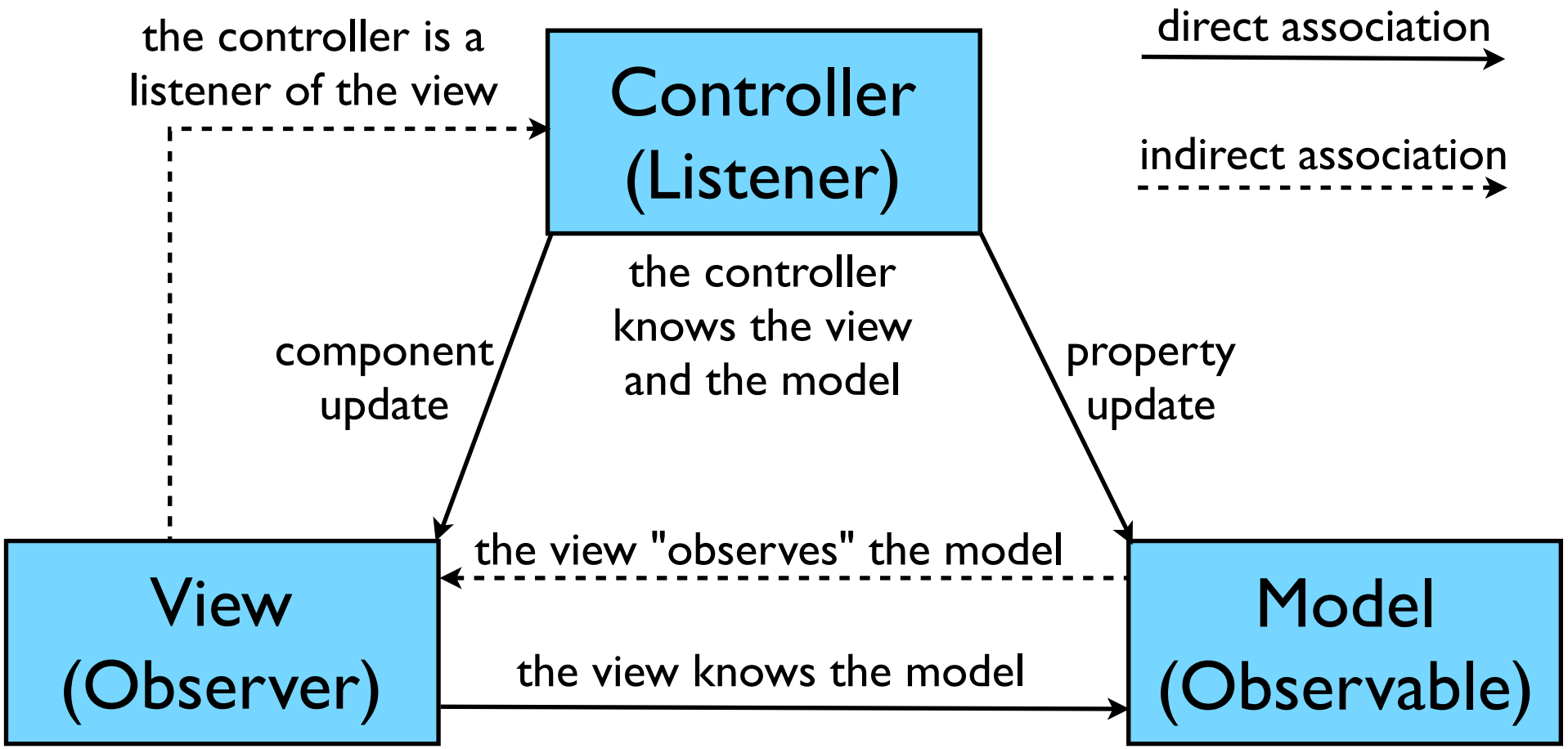
Model-View-Controller

- Applications are generally complex
  - so are their graphical interfaces
- Difficult to handle both application and GUI logic in one, big program
- Need to separate them
  - handle them *independently*

Model-View-Controller

- The GUI class
  - description of the graphical interface and its components
  - the view
- The application class
  - the data and the methods which describe the main functionality of the application
  - the model
- The control class
  - the set of listeners which announce the model when some events occur via the view
  - the controller

Model-View-Controller



Programming with MVC

- Java provides some useful *interfaces* to map applications onto this model
- `java.util.Observer`
  - used by classes that want to be informed of changes in `Observable` objects
  - `update()` - called whenever the observed object is changed.
- `java.util.Observable`
  - an observable object
  - `addObserver(Observer o), notifyObservers(), setChanged()`

Programming with MVC

- Swing provides a set of predefined models and corresponding "delegate" components
- Model classes
  - data structures - *models* - that can generate events when data are modified
  - ex: `DefaultButtonModel, DefaultListModel, DefaultTableModel, DefaultTreeModel`
- Delegate components
  - define the *view* (by extending `JComponent`) AND the *control* (by implementing the appropriate interfaces)
  - ex: `AbstractButton` (implements `ItemSelectable`)

Using MVC with a list

```
Vector colors = new Vector();
colors.add("red"); colors.add("blue");
JList list = new JList(colors);
colors.add(“green”);
```

- Modifying the initial vector's components doesn't reflect on the list
    - use `setListData(Vector v);`

```
DefaultListModel model = new DefaultListModel();
model.addElement("red");
model.addElement("blue");
JList list = new JList(model);
model.addElement("green");
```

- Modifying the model's components automatically reflects on the list

Concurrent Programming in Java

Outline

- Processes vs. threads
- Creating and using threads
- Thread synchronisation
- Threading support in Java 1.5
- Some classical synchronisation problems

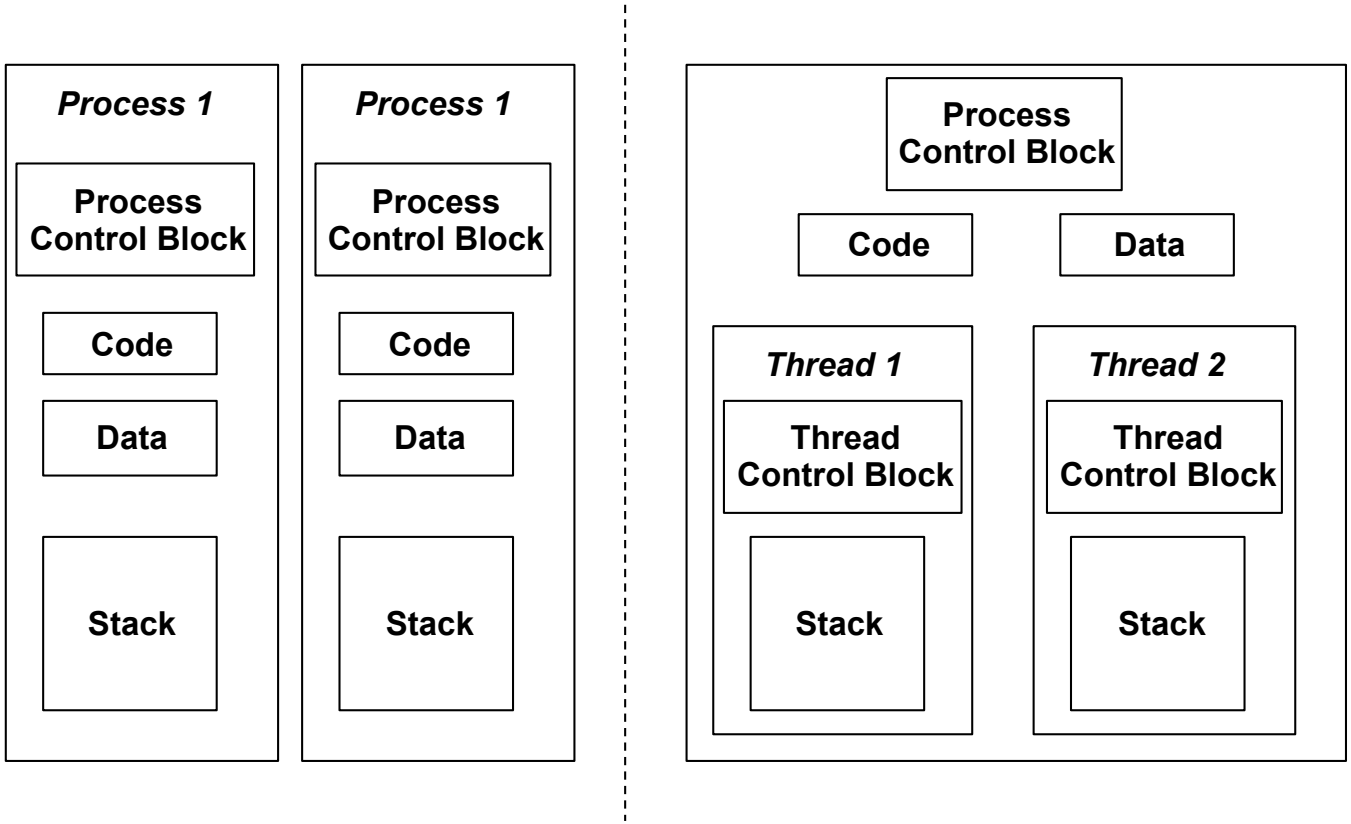
Using threads

- Improves efficiency
  - simultaneous use of all cores
  - simultaneous use of resources
- Creating and executing threads is "lighter" than processes
- Better program structures: several execution units

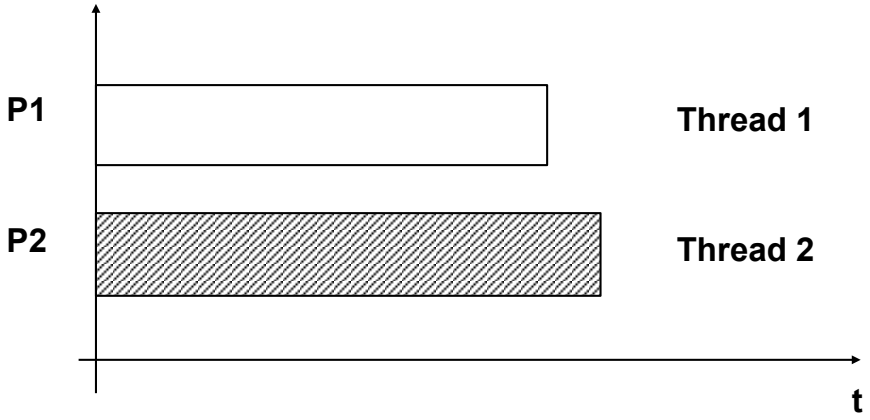
Processes and threads

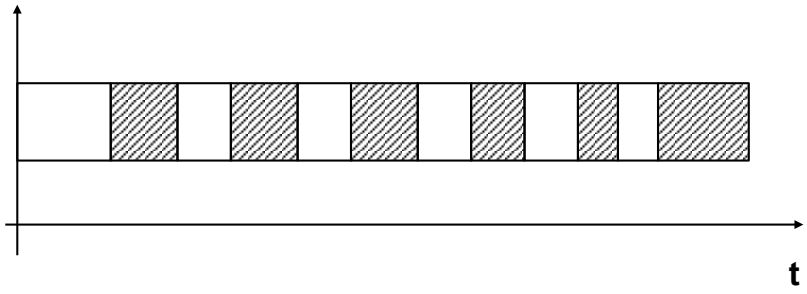
- Process
  - instance of a program in execution
  - the OS allocates for each process:
    - some memory space (program code, data buffer, stack)
    - the control of some resources (files, I/O)
- Thread
  - sequential control flow *within a process*
  - a process can have several threads
  - threads share: the program code and the data buffers in the memory, the resources of the process
  - the stack is specific to each thread: represents its current status

Multi-process vs. multi-threading



Threads execution





- Multi-core systems
  - parallel execution
- Single-core systems
  - scheduling done by the OS
  - policies: time slice, priorities

Java threads

- Java is the first important programming language to support threads ("built in", not as an OS dependent library)
- Starting with Java 1.5: a large set of classes for concurrent programming
- Handling threads:
  - `java.lang.Thread` class
  - `java.lang.Runnable` interface
  - methods of the `Thread` class: `start()`, `sleep()`, `getPriority()`, `setPriority()`
  - methods of the `Object` class: `wait()`, `notify()`, `notifyAll()`

Creating a thread

```
class MyThread extends Thread {

  public void run() {
    …
    sleep(100);
    …
  }
}

…
MyThread mt = new MyThread();
…
mt.start();
```

```
class MyModule extends Module
implements Runnable {
  public void run() {
    …
    Thread t = Thread.currentThread();
    t.sleep(100);
    …
  }
}
…
MyModule mm = new MyModule();
Thread mmt = new Thread(mm);
…
mmt.start();
```

Threads' states

- Created: the object was created with the `new()` operation; the `start()` method can be called
- Ready to execute: the `start()` method was called, the thread can execute
- Suspended: `sleep()` or `wait()` were called
- Terminated: the `run()` method completed

Thread synchronisation

- Two scenarios:
  - *Concurrency*
  - *Collaboration*
- The locking mechanism in Java:
  - each object has an associated lock
- Synchronisation
  - using objects' locks: `synchronized(object)`
  - `synchronized` keyword attached to a method or to some code sequence: *monitor*
  - during the execution of a synchronized sequence the lock is secured: mutual exclusion

Concurrent access to a resource

Concurrent access to a resource

```
class MyThread extends Thread {

    static int account = 0;
    static Object o = new Object();

    MyThread (String name) {
        super(name);
    }

    void deposit() {
        System.out.println(getName() + " account=" +
      account);
        account = account + 1000;
    }
```

Concurrent access to a resource

```
    public void run() {
        for (int i = 0; i < 3; i++) {
          synchronized(o) {
            deposit();
          }
        }
        System.out.println("DONE!");
    }
}

class Test {
    public static void main (String args[]) {
        MyThread Alice = new MyThread("Alice");
        MyThread Bob = new MyThread("Bob");
        Alice.start();
        Bob.start();
        System.out.println("DONE main!");
    }
}
```

Using locks

```
class Thingie {
    private static Date lastAccess;
    public synchronized void setLastAccess(Date date){
          this.lastAccess = date;
    }
}
class MyThread extends Thread {
    private Thingie thingie;
    public MyThread(Thingie thingie) {
          this.thingie = thingie;
    }
    public void run() {
          thingie.setLastAccess(new Date());
    }
}
public class Test {
   public static void main() {
          Thingie thingie1 = new Thingie();
          Thingie thingie2 = new Thingie();
          (new MyThread(thingie1)).start();
          (new MyThread(thingie2)).start();
 }
 }
```

Remember!

`synchronized`:

lock on the **object/instance**

Handling threads

- `wait(), notify(), notifyAll()`
- Can be invoked only inside monitors
- To execute any of these methods, you must be holding the lock for the associated object

Waiting

- Inside a monitor, a process can block waiting for some condition
- Wait Set
  - associated to each object in Java
  - a set of threads blocked on that object
- When `wait()` is called for an object `m` by a thread `t`:
  - if `t` doesn't have the lock for `m`: `IllegalMonitorStateException`
  - `t` is added to the wait set of `m` and it unlocks the lock associated with `m`
  - `t` stops executing until is removed from the wait set of `m` by:
    - a call to `notify()` / `notifyAll()` from another thread
    - the expiration of the waiting time
    - an interruption with `Thread.interrupt()`
    - an implementation dependent action

Notifications

- `notify()` and `notifyAll()`
- When a notification is called for an object `m` by a thread `t`:
  - if `t` doesn't have the lock for `m`: `IllegalMonitorStateException`
  - `notify()`: *one thread* is picked randomly and removed from the wait set of `m`
  - `notifyAll()`: *all threads* are removed from the wait set of `m` - but only one will regain the lock

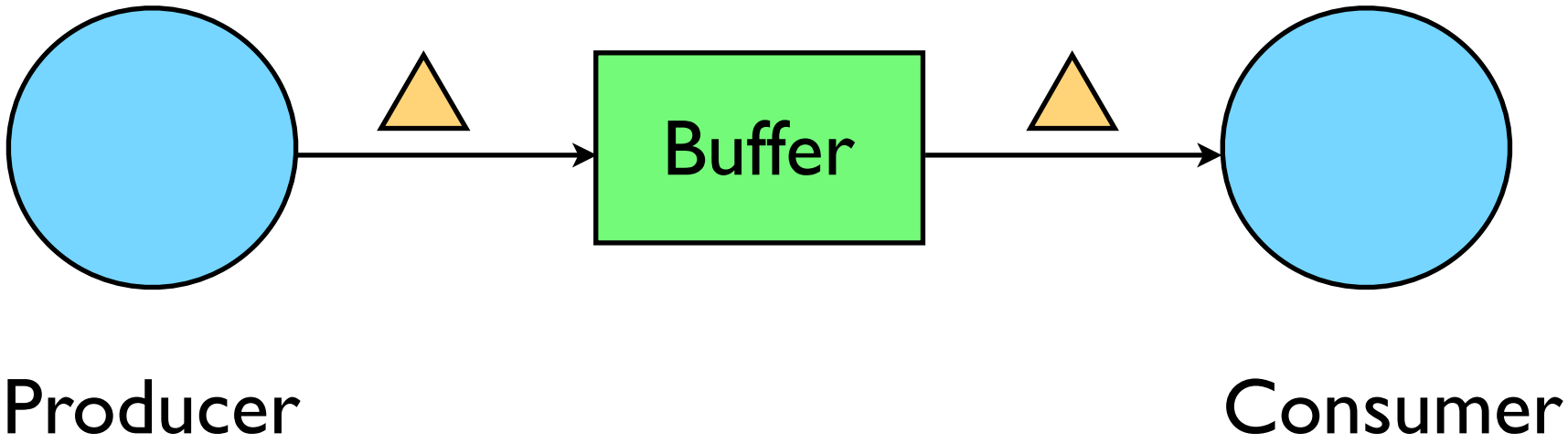
Notifying several threads

- You can use several blocking conditions:

```
while (!condition) {
  wait();
}
```

Unlocked only when the appropriate condition is true

Collaboration: Producer / Consumer



Collaboration: Producer / Consumer

```
//1 Producer, 1 Consumer, Buffer of size 1
class Producer extends Thread {
      private BufferArea Buffer;
      private int ID;
      public Producer(BufferArea z, int ID) {
        Buffer = z; this.ID = ID;
      }
      public void run() {
        for (int i = 0; i < 10; i++) {
         Buffer.send(i);
         System.out.println("Producer " + ID + " has sent: "+
           i);
         try {
            sleep((int)Math.random() * 100));
           } catch (InterruptedException e) {}
        }
      }
}
```

Collaboration: Producer / Consumer

```
class Consumer extends Thread {
      private BufferArea Buffer;
      private int ID;

      public Consumer(BufferArea z, int ID) {
        Buffer = z;
        this.ID = ID;
      }

      public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
          value = Buffer.receive();
          System.out.println("Consumer "+ID+" has taken " + value);
        }
      }
}
```

Collaboration: Producer / Consumer

```
class BufferArea {
  private int value; // current value in the buffer
  private boolean available = false; // is a value ready for Consumer
  public synchronized int receive() {
    if (!available) {
      try { wait();} catch (InterruptedException e) {}
    }
    available = false;
    notify();
    return value;
  }
  public synchronized void send(int value) {
    if (available) {
      try { wait();} catch (InterruptedException e) {}
    }
    this.value = value;
    available = true;
    notify();
  }
}
```

Concurrency support in JDK 5.0

- New package: `java.util.concurrent`
- Changes within the JVM: exploits the new instructions available in modern CPUs
- Basic classes: locks, atomic variables
- High level classes: semaphores, barriers, thread pools

Useful classes

- `Semaphore`
  - classic Dijkstra counting semaphore
- `Mutex`
  - mutual exclusion
- `CyclicBarrier`
  - reusable barrier
- `CountdownLatch`
  - similar with the barrier, but arriving is separated from waiting (not blocking)
- `Exchanger`
  - two-way exchange between two cooperating threads

Low level synchronization

- `Lock`
- `ReentrantLock`
- `Conditions`
- `ReadWriteLock`
- Atomic variables: `AtomicInteger, AtomicLong, AtomicReference`

Semaphore

- A variable / abstract data type that controls access to a shared resource
- Simple counters that indicate the status of a resource
- Usage:
  - if counter > 0, then the resource is available
  - if counter <= 0, then that resource is busy or being used by someone else.
- This counter is a protected variable and cannot be accessed by the user directly, only through special methods:
  - P() - wait on the semaphore variable (decrement counter, block if the value becomes negative); Java: `acquire()`
  - V() - signals the semaphore variable (increment counter); Java: `release()`
- Problem: P() and V() are scattered among several threads:
  - It is difficult to understand their (side) effects
  - Usage must be correct in all threads
  - One bad (or malicious) thread can fail the entire collection of threads

Example: using a Semaphore

```
final private Semaphore s = new Semaphore(1, true);

s.acquire(); //acquire semaphore token: P()

try {
   balance=balance+10; //protected value
} finally {

s.release(); //return semaphore token: V()
}
```

`CyclicBarrier`

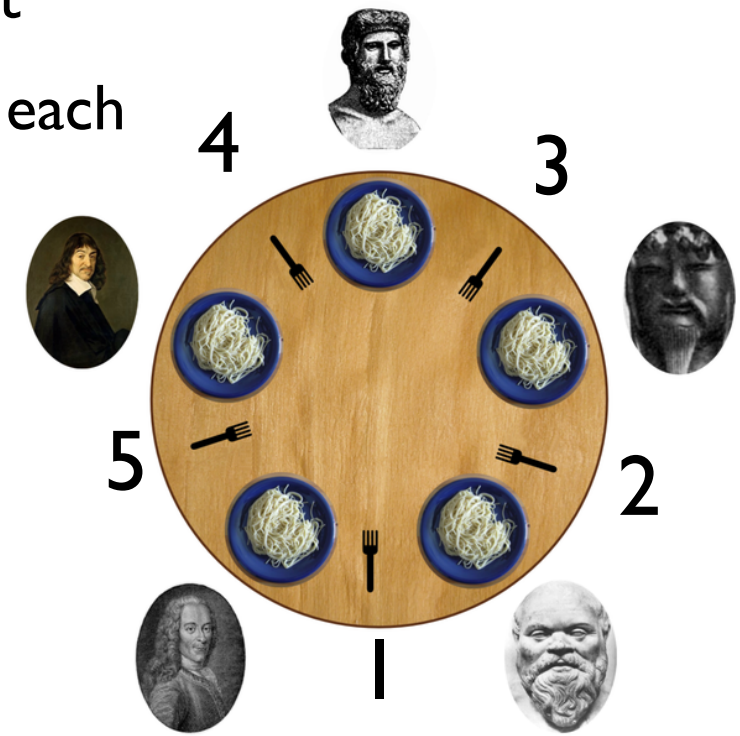
- A synchronization tool
- Any thread must stop at the barrier and cannot proceed until all other threads reach the barrier
- Useful to control loops with dependencies
- `await()`, `getNumberWaiting()`

Synchronization problems

- Classical problems to illustrate synchronization issues
- *Producers-Consumers*
- *Dining philosophers*
- *Readers-Writers*
- *Sleeping barber*

Dining philosophers

- Each philosophers needs two forks to eat
- Design a concurrent algorithm such that each philosopher won't starve
- One (bad) approach:
  - think until the *left* fork is available; when it is, pick it up
  - think until the *right* fork is available; when it is, pick it up
  - eat
  - put the *left* fork down
  - put the *right* fork down
  - repeat from the start
- Illustrates the problem of avoiding *deadlocks*



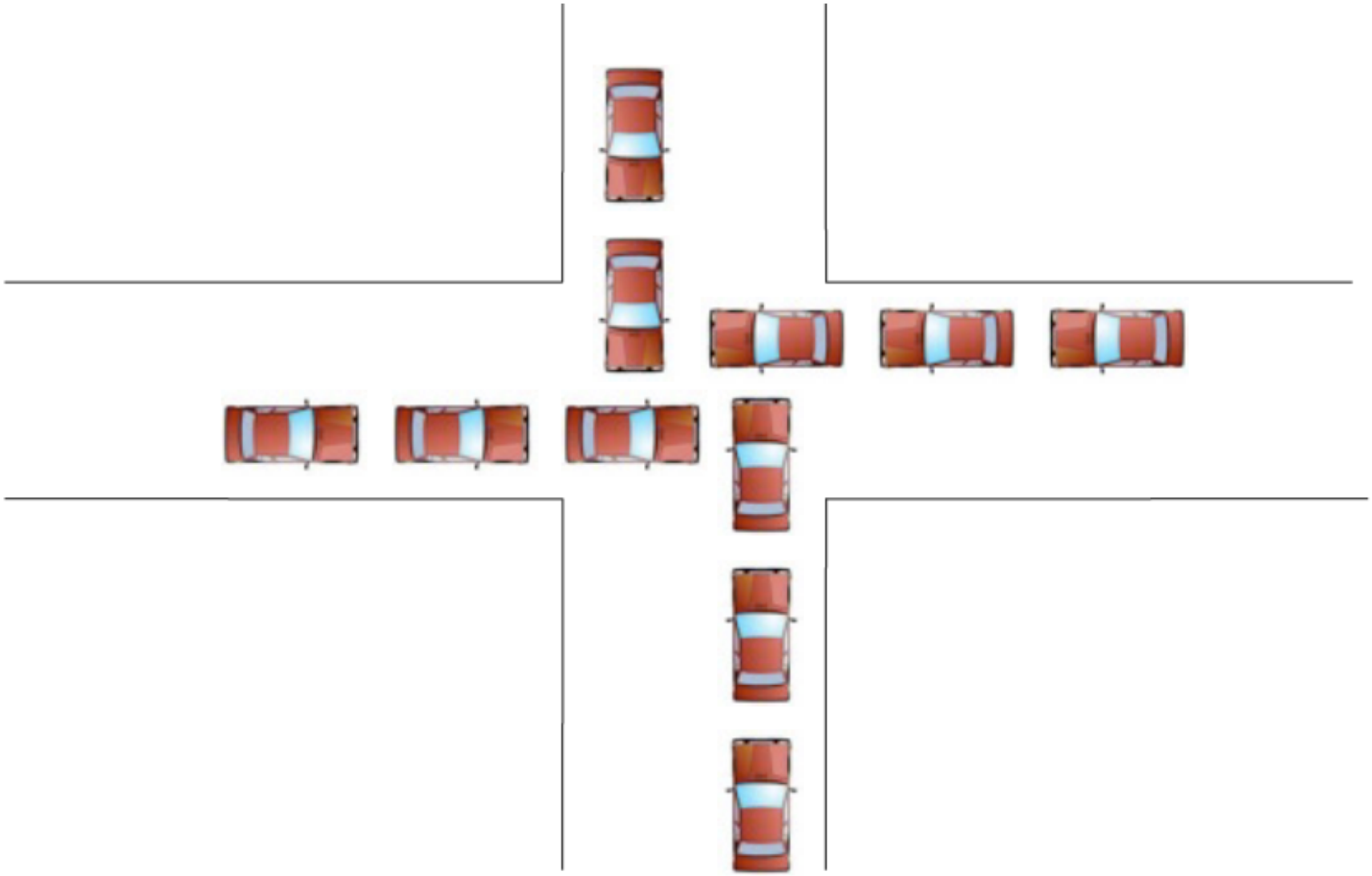
One solution

- pick *lower-numbered* fork
- pick *higher-numbered* fork
- eat
- put *higher-numbered* fork
- put *lower-numbered* fork

```
P(lower-numbered);
P(higher-numbered);
//eat
V(higher-numbered);
V(lower-numbered);
```

Assign semaphores to each fork!
Use asymmetry to prevent deadlocks!

Deadlocks



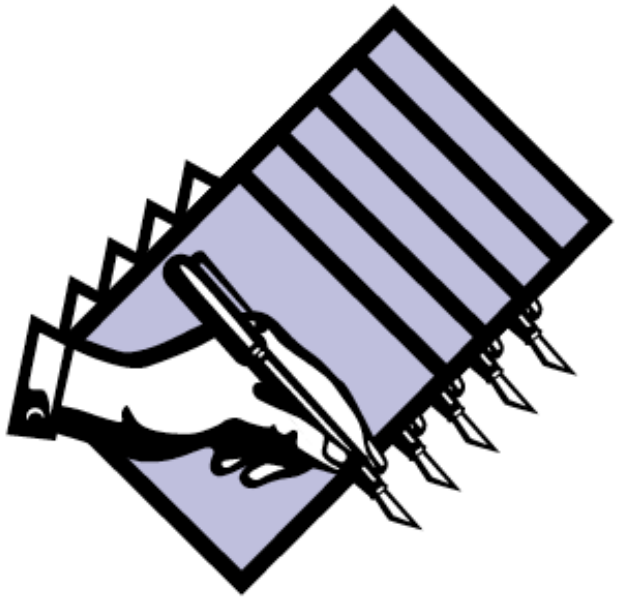
Deadlocks

- A thread is deadlocked if it is waiting for an event that will never occur.
- Typically, but not necessarily, more than one thread will be involved together in a deadlock
  - A set of threads where all threads in the set are waiting for an event that can only be caused by another thread in the set (which is also waiting!).
- Dining Philosophers is a perfect example
  - Each holds one chopstick and will wait forever for the other.

Conditions for deadlocks

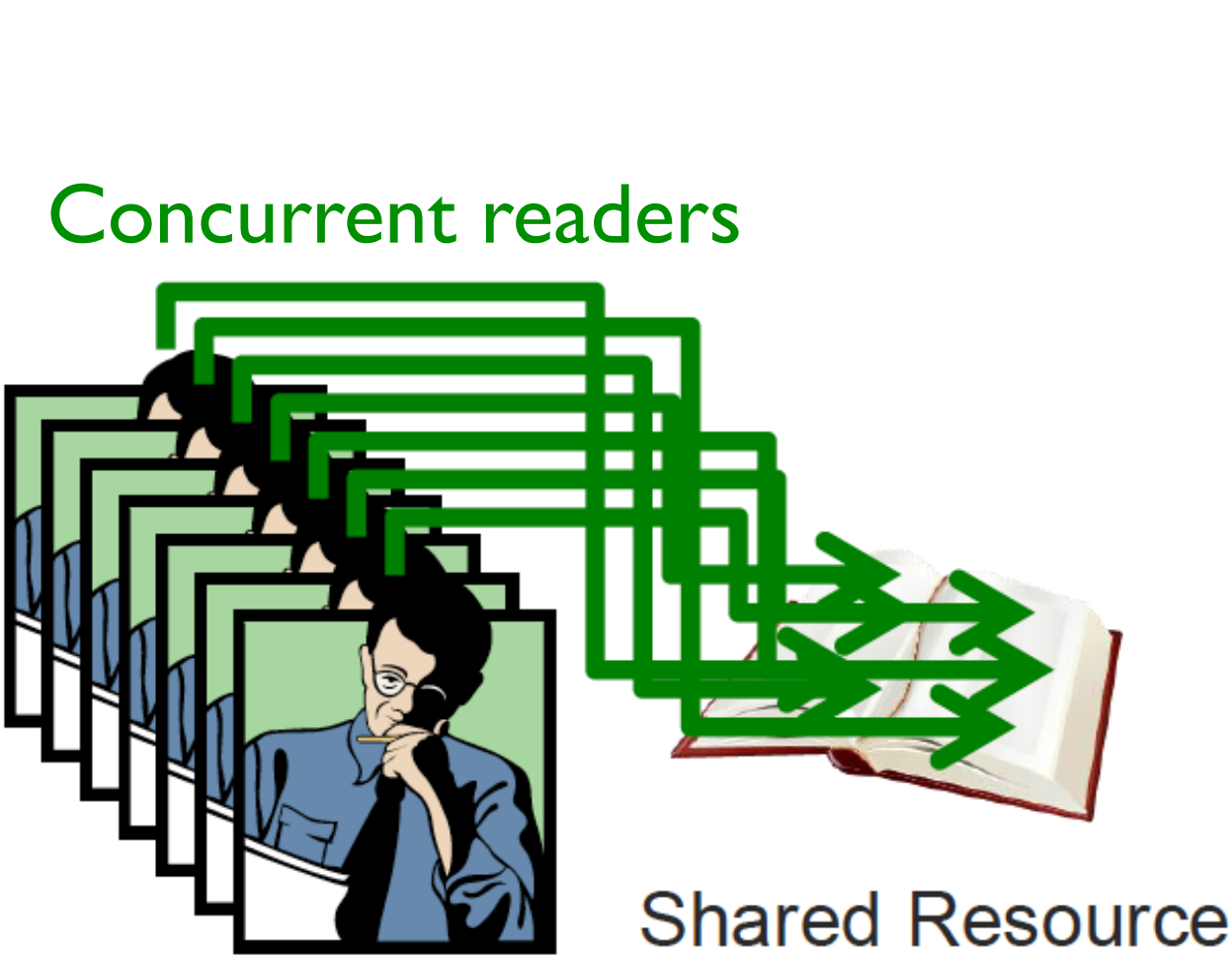
- *Necessary* and *sufficient* conditions for deadlocks:
  - Mutual Exclusion – some resource must be held exclusively
  - Hold and Wait – some thread must be holding one resource and waiting for another
  - No Preemption – resources cannot be taken away from a thread involuntarily (until completion)
  - Circular Wait – circular chain of threads in which each thread holds one or more resources that are requested by the next thread in the chain
- To break deadlocks, avoid any of these!
  - Example: when a resource request occurs, see if granting it will create a cycle; if so, delay granting that resource; run another thread instead

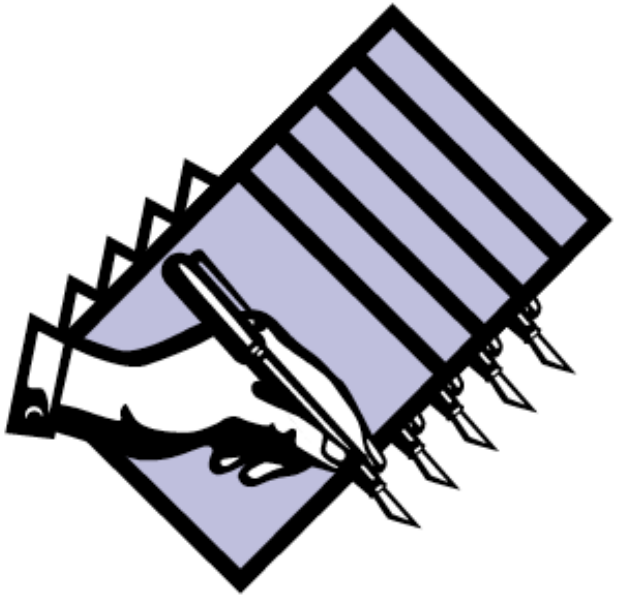
Readers and Writers



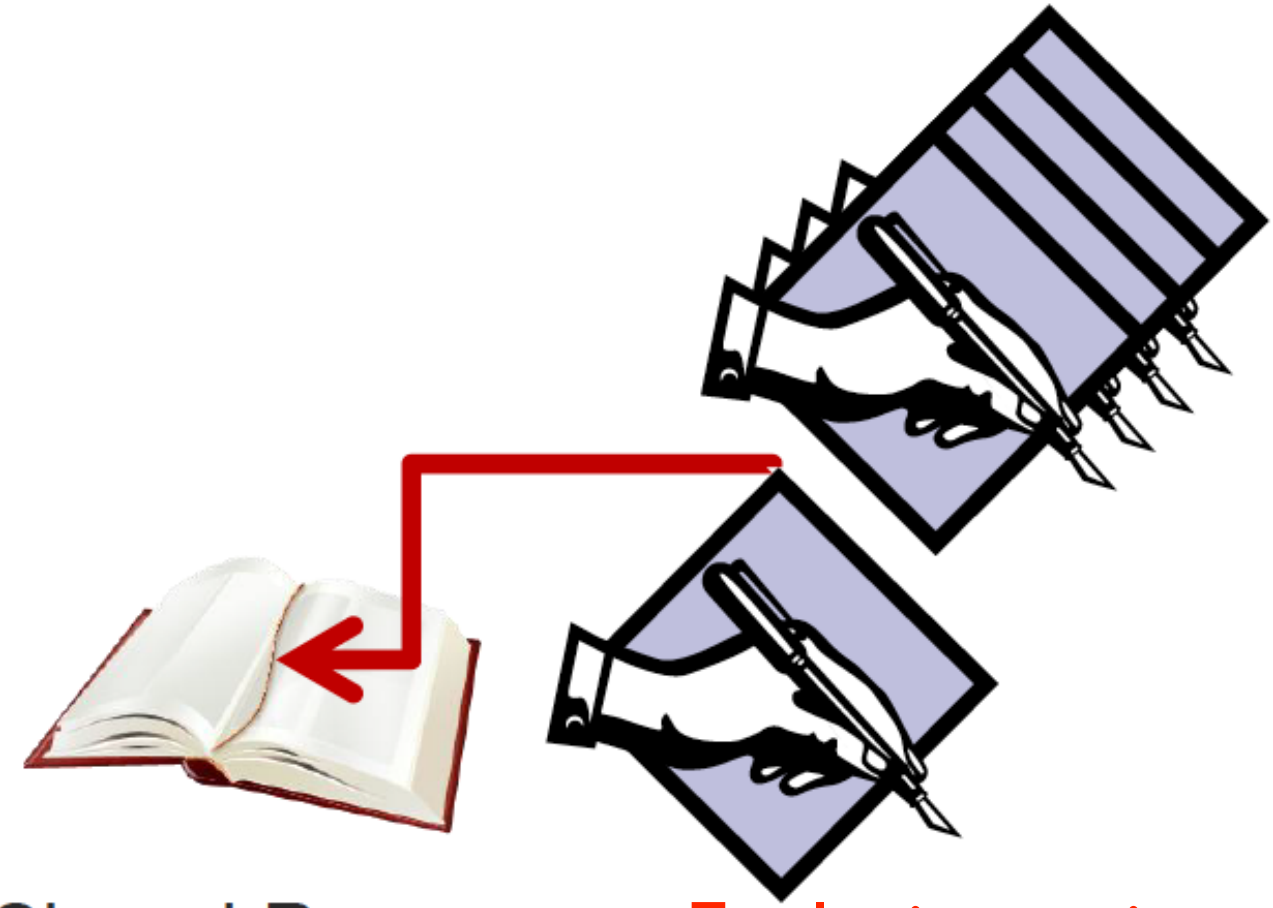
Shared Resource

Readers and Writers





Readers and Writers



Readers and Writers

- Locking table: whether any two can be in the critical section simultaneously

	Reader	Writer
Reader	OK	No
Writer	No	No

- Solution sketch:
  - keep track of how many readers there are
  - the first reader locks the shared resource for all subsequent readers
  - a writer locks the shared resource

First solution

```
//Reader
while(true){
    mutex.acquire();
    readCount++;
    if(readCount == 1)
      writeBlock.acquire();
    mutex.release();
    /* Critical section */
    access(resource);
    mutex.acquire();
    readCount--;
    if(readCount == 0)
      writeBlock.release();
    mutex.release();  }
```

```
static int readCount=0;
Semaphore mutex = new
Semaphore(1, true);
Semaphore writeBlock =
new Semaphore(1, true);
```

```
//Writer
while(true){
    writeBlock.acquire();
    /*Critical section*/
    access(resource);
    writeBlock.release();
}
```

Second solution

```
static int readCount=0;static int writeCount=0;
Semaphore mutex1 = new Sempahore(1, true);
Semaphore mutex2 = new Sempahore(1, true);
Semaphore readBlock = new Sempahore(1, true);
Semaphore writeBlock = new Sempahore(1, true);
```

```
//Reader
while(true){
    readBlock.acquire();
    mutex1.acquire();
    readCount++;
    if(readCount == 1)
      writeBlock.acquire();
    mutex1.release();
    readBlock.release();
    /* Critical section */
    access(resource);
    mutex1.acquire();
    readCount--;
    if(readCount == 0)
      writeBlock.release();
    mutex1.release();  }
```

```
//Writer
while(true){
    mutex2.acquire();
    writeCount++;
    if(writeCount == 1)
      readBlock.acquire();
    mutex2.release();
    writeBlock.acquire();
    /* Critical section */
    access(resource);
    writeBlock.release();
    mutex2.acquire();
    writeCount--;
    if(writeCount == 0)
      readBlock.release();
    mutex2.release();  }
```

Sleeping barber

- One barber, *n* waiting chairs
- When there are no clients, the barber sleeps
- When a client arrives, he either wakes up the barber or waits if the barber is busy
- If all the chairs are occupied, the client leaves

One solution

```
Semaphore custReady = new Semaphore(0,true);     //if 1, at least one customer ready
Semaphore barberReady = new Semaphore(0, true);
Semaphore mutex = new Semaphore(1, true);  //if 1, the # of seats can be inc. or dec.
int numberOfFreeWRSeats = N;     // total number of seats in the waiting room
```

Barber:

```
  while(true){                    // Run in an infinite loop.
    custReady.acquire();             // Try to acquire a customer - if none, go to sleep
    mutex.acquire();   // Awake,try to get access to modify # of available seats
    numberOfFreeWRSeats++;        // One waiting room chair becomes free.
    barberReady.release();          // I am ready to cut.
    mutex.release();       // Don't need the lock on the chairs anymore.
    // (Cut hair here.)}
```

Customer:

```
  while(true){                    // Run in an infinite loop.
    mutex.acquire();         // Try to get access to the waiting room chairs.
    if(numberOfFreeWRSeats > 0) // If there are any free seats:
      numberOfFreeWRSeats--;      //   sit down in a chair
      custReady.release();          //   notify the barber, who's waiting a customer
      mutex.release();     //   don't need to lock the chairs anymore
      barberReady.acquire();          //   wait until the barber is ready
      // (Have hair cut here.)
    else                        // otherwise, there are no free seats; tough luck
      mutex.release();     // but don't forget to release the lock on the seats!
      // (Leave without a haircut.)}
```

Further reading

- Brian Goetz, “Introduction to Java threads” – http://www.ibm.com/developerWorks
- Brian Goetz, “Concurrency in JDK 5.0” – http://www.ibm.com/developerWorks
- Threads: Basic Theory and Libraries - http://www.cs.cf.ac.uk/Dave/C/node29.html