Multithreading II

CS160: User Interfaces

John Canny

This time

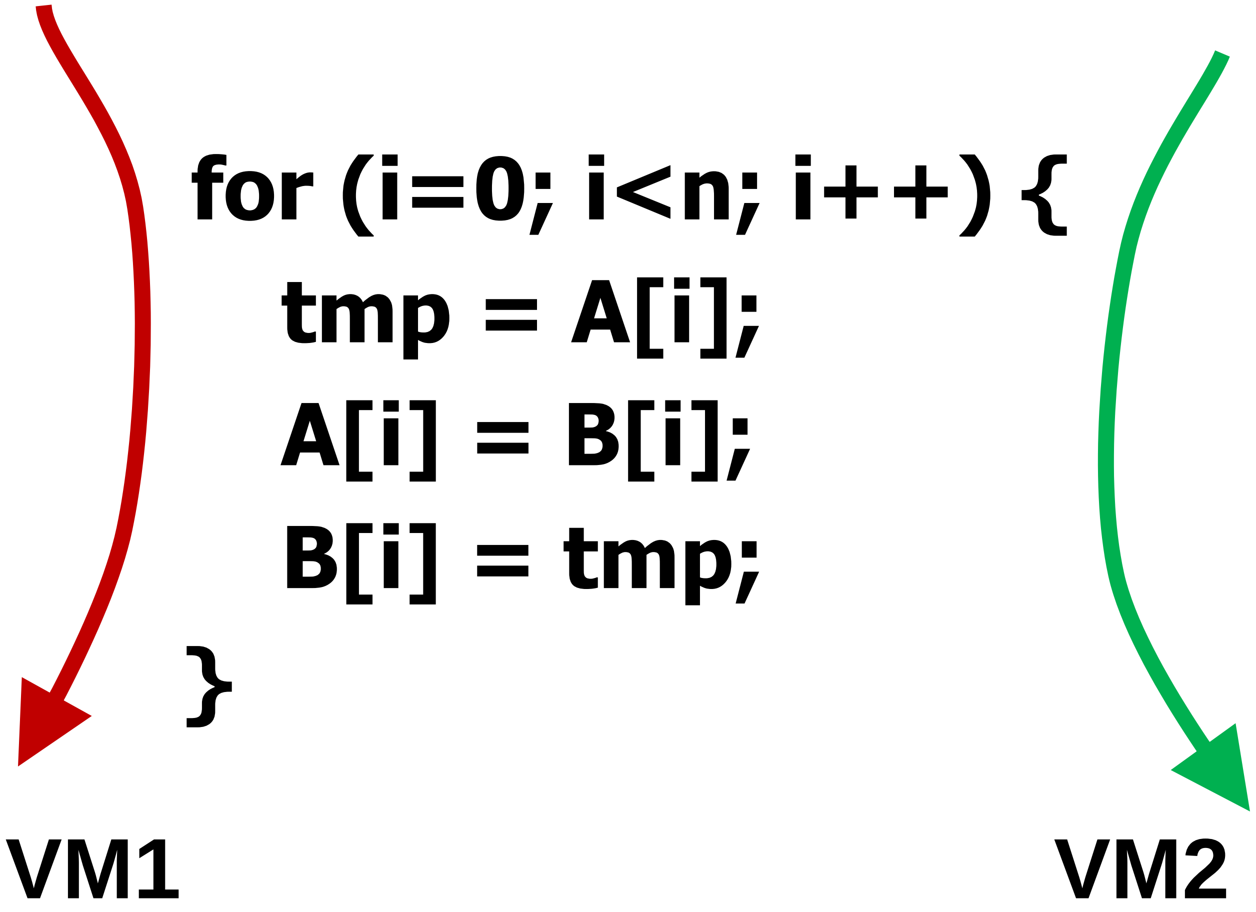
More on multithreaded programs

Debugging multithreaded programs

More examples

Threads - review

A **thread** is a **partial virtual machine**. Each thread has its own stack (and local variables), but shares its heap space and with other threads.



```
for (i=0; i<n; i++) {
   tmp = A[i];
   A[i] = B[i];
   B[i] = tmp;
}
```

* Threads may also have some private heap space, called Thread-Local Storage (TLS).

Thread Safety

Code is **thread safe** if it can be called from multiple threads without “breaking” the program.

```
int fact (int n) {
   int i, p;
   for (i=1,p=1; i<=n; i++)
      p*=i;
      return p;
   }
```

Separate ints i,p are created **on the stack** each time the function is called. Each thread has its own copy.

Java synchronized

The following code snippet protects a segment of code from access by more than one thread:

```
synchronized(someObject) {
        val = val + 1;
 }
```

The first thread to execute this obtains a lock on the object someObj.

Another thread to run any code locked by this same object will block at the synchronized statement until the first thread exits its block.

Concepts

- Message Queues – java “Handler”s
- Runnables and Callables
- Thread Pools
- Futures

Java Handler()

High-level interface to a MessageQueue

Two ways to use it:

- Send an android.os.Message() to it with Handler.sendMessage()

  override Handler.handleMessage() to take action.

- Send a Runnable or Callable object to the Handler.

Runnables

Simple message queues have limited functions – the receiver only responds to messages it already knows what to do with.

A much more powerful mechanism is to post Runnables to a message queue (Handler):

```
public class X implements Runnable {
   int y, z;
   public X(int y0, int z0) {y = y0; z = z0;} // Save y, z on create
   public void run() {
      // do something useful, using y, z at some later time
   }
}
```

Runnables

Runnables are class instances (Objects), and can be pushed into a queue like other messages.

When the message handler in the receiver dequeues a runnable, it recognizes it by type, and calls its run() method.

In this way, the runnable (which is created in an originating thread), gets executed in a different thread.

Callables

Very similar to Runnables, but return a value. E.g.

```
public Double Hanoi(int n) {
    return Hanoi(n-1) + 1.0 + Hanoi(n-1);
}

C = new Callable<Double>({
   public Double call() {return Hanoi(10);}
   });
```

Futures

Callables can return a **Future**, which is a handle on the allocated thread. With a Future F you can:

- Cancel the task, i.e. stop it asynchronously: F.cancel()
- Query the Future to see if the task is done: F.isDone()
- Get the return value after the task is complete: F.get()

GUI Thread + Worker ThreadPool

The GUI thread can only do one thing. A long operation (e.g. file read/write) has to run in another thread. We typically call those worker threads.

Creating/destroying threads is expensive, we don’t want to do it with each task. So we establish a **thread pool**, which is persistent and reusable.

Tasks (runnables and callables) are assigned to threads by the pool service. You don’t normally need to know what is happening.

Futures

Sending a task (runnable) to a threadPool is different from invoking a method in several ways:

- Arguments need to be saved as instance variables so they are available when the run() method is called.
- Starting the task returns immediately.
- There can be no return value (the method wasn’t called yet).

The Future provides a link to this running task, and allows the holder to check completion, get the result when its finished, or cancel it.

Callables - Arguments

```
public Double Hanoi(int n) {
    if (n == 1) return 1.0;
    else return Hanoi(n-1) + 1.0 + Hanoi(n-1);
}

Class runHanoi extends Callable<Double> {
   int n;  // Extend callable so instance variables can hold args
   public runHanoi(int n0) {n = n0;}
   public Double call() {return Hanoi(n);}
}
```

Callables and Futures

```
C = new runHanoi(10);                         // Save args
Future<Double> F = workers.submit(C); // put in the queue

// wait in main thread

while (!F.isDone()) {}
Double val = F.get();
```

Futures - Cancelling

The cancelled thread should receive a InterruptedException (Note: this may only happen in certain places, e.g. in Thread.sleep()).

The worker task should catch this exception (it has to), and then do any cleanup before finishing. i.e.

```
Try {
  // Normal worker code here
} catch (InterruptedException e) {
  // Cleanup here
}
```

Example App

Simulates:

- Long computation
- Passing in an argument and getting a result back

Runnables vs. RMI

Runnables look something like Java RMI (Remote Method Invocation). But there are big differences:

No serialization for runnables

Runnables and efficiency

Invoking runnable() methods should be no less efficient than invoking any other method on an object.

We used **new** to create new runnables and callables in the example code, but this was just for simplicity.

Its fine to allocate one runnable to implement a particular type of action, then modify its arguments each time it is posted.

Design Patterns

- Message queue
- GUI Thread / Worker thread pool
- Database / Model-View-Controller
- Actor

Example App

Simulates:

- File read and write
- Network connections
- A live help system

```
public class threadsDemo extends Activity {
        ExecutorService workers;   // The threadPool
        Handler GUIhandler;         // GUI thread’s message Q
        …

 @Override
    public void onCreate(Bundle savedInstanceState) {
        GUIhandler = new Handler();
        workers = Executors.newCachedThreadPool();
        …
```

```
void runReadBar () {
  // Simulates a file read. Gradually moves a progress bar
}

 @Override
   public void onCreate(Bundle savedInstanceState) {
     …
     // Define the onClick handler for the file read button
     start_button.setOnClickListener(new OnClickListener() {
       public void onClick(View v) {  // Standard onClick preamble
        …  // Callable is just like Runnable, but returns a Future
       readFuture = workers.submit(new Callable<String>() {
         public String call() {runReadBar(); return null;}
         });;}
  }
```

```
@Override
  public void onCreate(Bundle savedInstanceState) {
    … // This button cancels the running read task
    cancel_button.setOnClickListener(new OnClickListener() {
       public void onClick(View v) { // Standard onClick preamble
       if (!readFuture.isDone())  // Don’t cancel if its done
            if (readFuture.cancel(true)) { // see if cancel succeeded
            }
       }})
```

```
public class PBU implements Runnable { // Progress bar updater
  ProgressBar pb;  int i;  // Reference to a ProgressBar, new value
  public PBU(ProgressBar pb0, int i0) {pb=pb0; i=i0;}
  public void run() {pb.setProgress(i);} // Set the bar to its new val
}

public void runReadBar()   // incrementally fills the PB, then clears
  double completed = 0.0; // Fraction of completion
  try {
     while (completed < 1.0) { // Post runnable to GUI to update
        GUIhandler.post(new PBU(pb1,(int)(completed*bmax)));
        Thread.sleep(100);
        completed += 0.003;
        }
   } catch (InterruptedException e) {};  // Tidy up (nothing to do)
}
```

Demo

Design Patterns

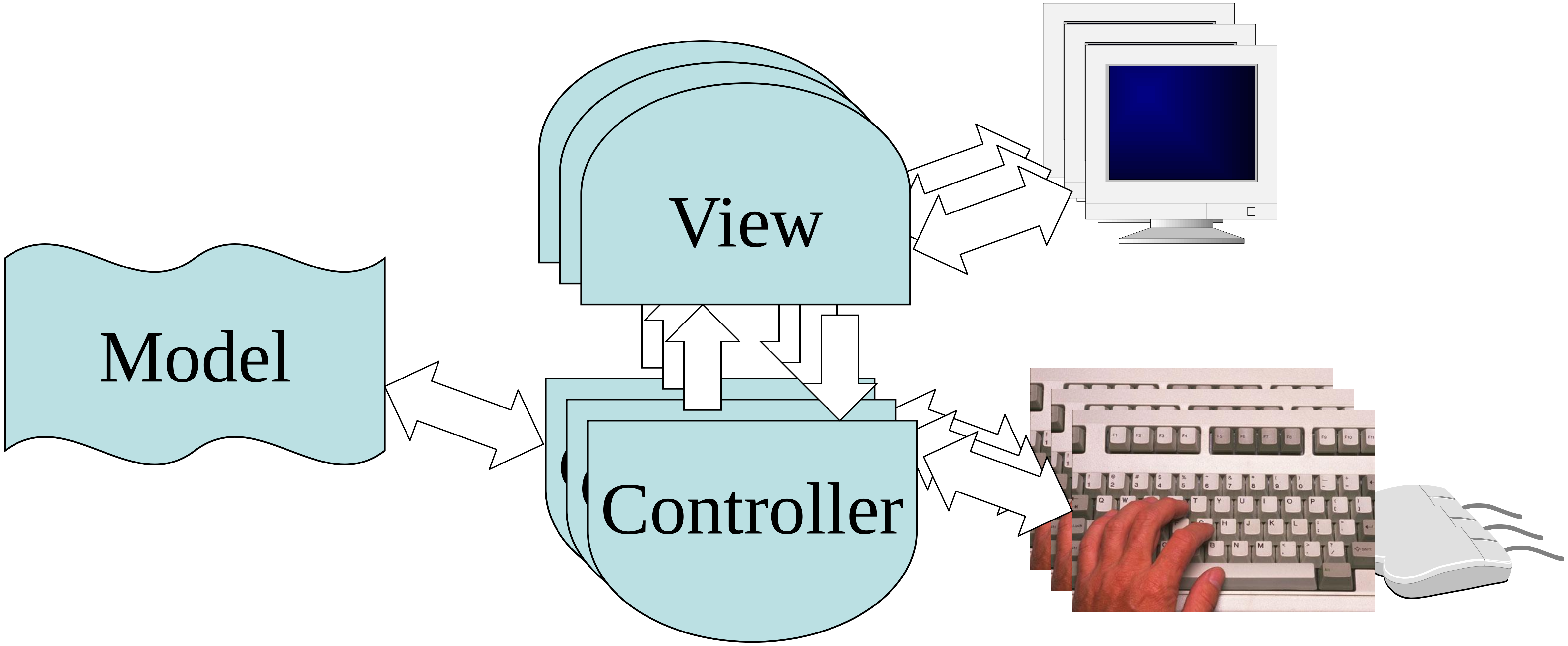
- Message queue
- GUI Thread / Worker thread pool
- Database / Model-View-Controller
- Actor

Model-View-Controller

MVC is an excellent pattern for concurrent programming:

State is centralized in the model, no other communication needed

Controllers+Viewers run independently, and each can have its own thread.



Model-View-Controller

Databases provide an excellent backend for the model:

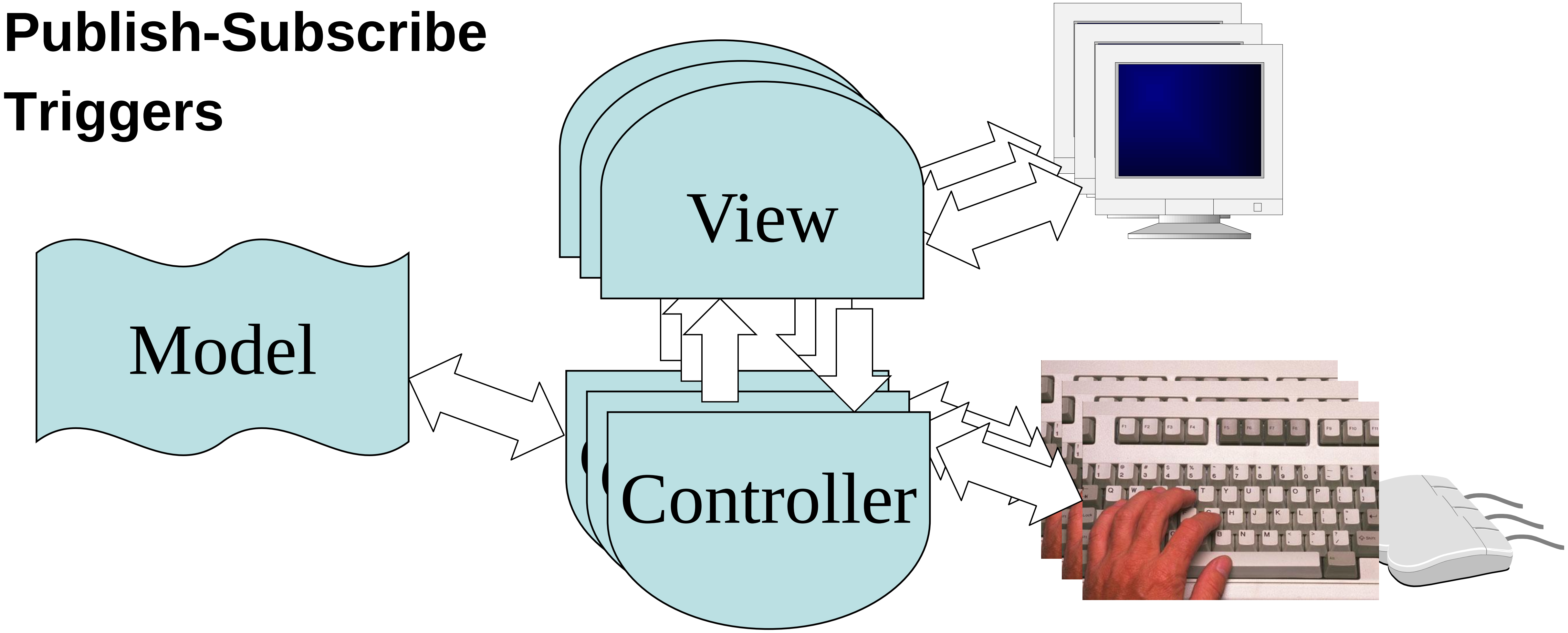
Transactions – complex updates are atomic.

Locking at different scales: an entire table or a row of a table.

Consistency constraints (relations).

Publish-Subscribe

Triggers



View

Model

Controller

MVC for multithreading

Advantages:

- Extensible, modular.
- Easy to develop and debug.
- Save much coding if a database is used.

Disadvantages:

- Heavy use of resources (space, time, memory).
- Discourages quick information flows.
- Can be very slow with many users if locks are too coarse.

Example

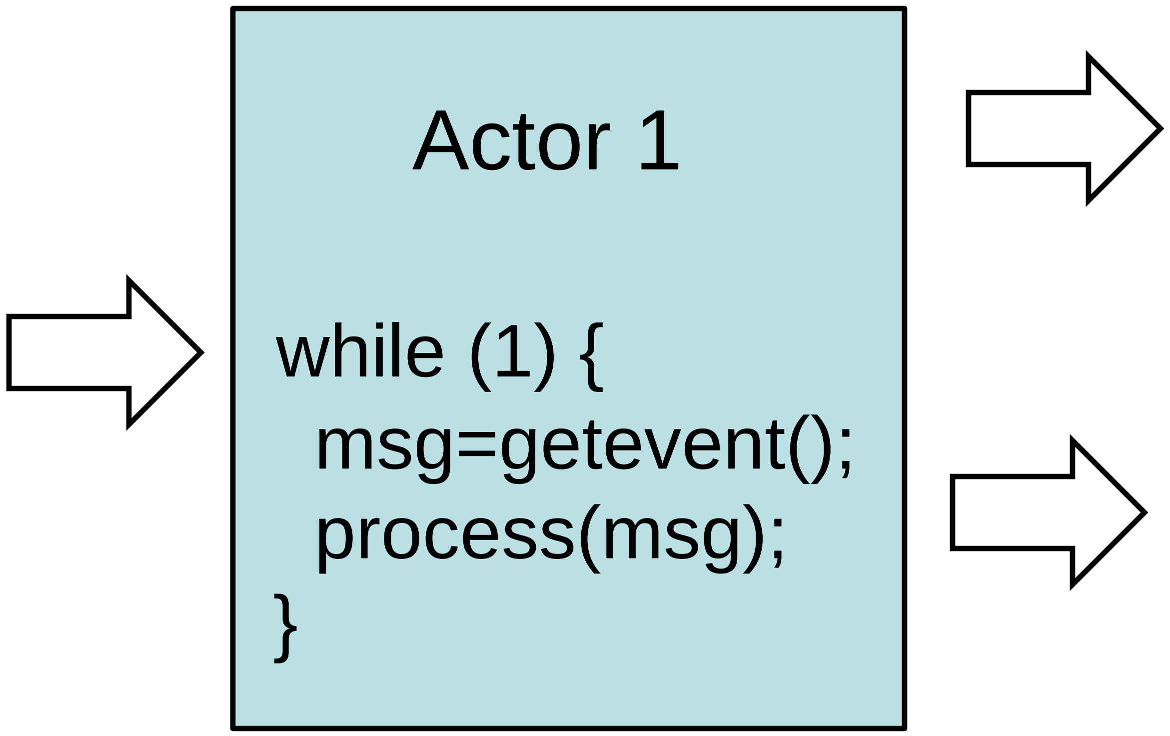
Actor

An actor is a class instance that runs its own thread.

Since data and methods are closely associated in a class, using a single thread to “run” the actor is very modular.

The actor will need an event loop to process incoming events.

Synchronized queues or mailboxes support communication.



```
while (1) {
  msg=getevent();
  process(msg);
}
```

Actor

Advantages:

Easy to design – like the sequential version of the class, but with the event loop added.

Good alignment between threads and data, minimizes contention and probability of inconsistency.

Exploits multicore processors.

Disadvantages:

A system of actors can be very complex to model.

Best to use a mixture of actors and “passive” classes.

A large multi-actor system is resource-intensive (memory, time,…)

Debugging

Not too difficult – similar to sequential debugging with a few extra operations:

Attach: attach the debugger to a running process.

List threads: list the running threads in the program.

Select a thread: Pick one to view or step through.

Thread-specific breakpoints: Stop the program when one specific thread reaches a program line.

Debugging

Note: The debugger normally runs **all** threads but checks when certain conditions (breakpoints or steps) are met by a particular thread.

So debugged execution is very similar to live execution, except for the pauses.

Review

Design patterns for multithreaded programs:

- Message queue
- GUI thread/Worker threadPool
- MVC
- Actor

Debugging multithreaded programs