Multithreading and Interactive Programs

CS160: User Interfaces

John Canny

This time

Multithreading for interactivity – need and risks

Some design patterns for multithreaded programs

Debugging multithreaded programs

Why Multithreading?

Interactive programs need to respond **fast** to user input.

Direct manipulation assumes that objects onscreen move with the user's hand.

Why Multithreading?

But not all code returns from event-handling so fast:

- File access
- Network operations
- Database lookup
- Simulation

Why Multithreading?

We at least need to decouple the code processing screen events from the code that handles them.

But we often need to do more to make sure the code runs robustly, even in the presence of errors.

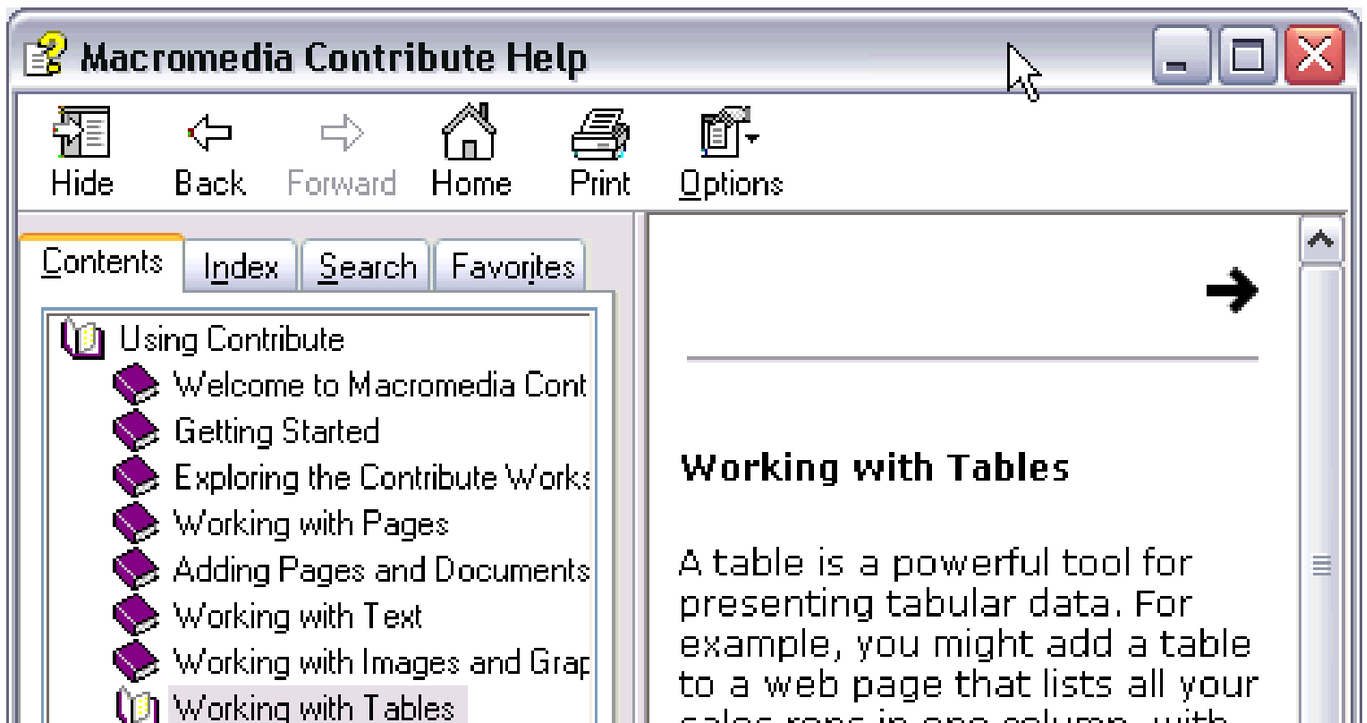
Why Multithreading?

Other processes that need to stay "live"

- Getting help
- Aborting (need to handle the abort operation)
- Doing something else while waiting for a long operation

Running these modules in a single thread gives users many aggravations:

Such as?



Why Multithreading?

More examples:

Multicore processors: use all the CPUs

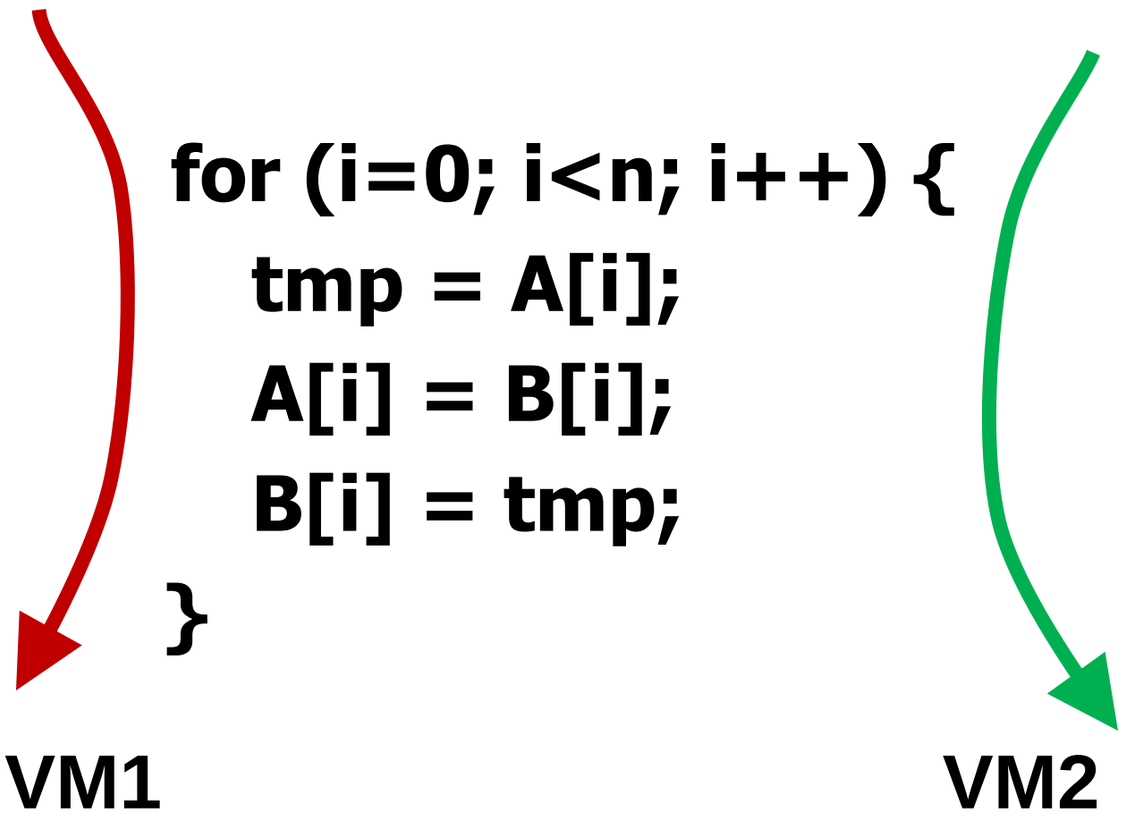
Dynamic widgets:

- Clocks
- Progress indicators
- Mail inbox…

PS your course projects don't **have** to be multi-threaded, they are interactive prototypes.

What is a Thread?

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.



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

What is a Process?

A **process** is a **complete virtual machine** with its own stack and heap.

Since processes don't share memory with each other, they need other mechanisms (e.g. system message queues) to communicate with each other.

Processes vs. Threads

i.e. processes are like dating but multithreaded programs are like living together.

Why Threads?

Because they share memory, threads support very efficient and versatile communication. In fact, such communication is basically free, and any data structure can be used.

Well, even if communication is free, code development certainly isn't. Multithreaded programming is probably the biggest productivity killer of all time.

Threading Hell

After a long and careful analysis the results are clear: 11 out of 10 people can't handle threads.”
— Todd Hoff

Why is it hard?

Threading primitives are lower-level than high-level language constructs.

Not standardized across OSes.

Nondeterminism.

Combinatorics:

Combinatorics

Consider 10 steps of a simple non-branching program.

```
A = B;
C = 4*A;
Y = exp(C);
...
Z = 5*R;
```

There is exactly one way that these statements can execute in a single-threaded program.

Combinatorics

But if two threads execute the same 10 statements, how many possible orderings of operations are there?

```
A = B;
C = 4*A;
Y = exp(C);
…
Z = 5*R;
```

A: 184756 = $\binom{20}{10}$ - but really more since there will be >> 10 instructions in the executable

Thread Safety

Code is **thread safe** if it can be called from multiple threads without interaction between them.

A simple C++ function or method works just fine:

```
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.

Thread un-Safety

But what happens if we change the variables to static (shared or global)?

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

Thread Safety and OOP

Object-Oriented Programming uses class instances, and these are usually heap-allocated. Heap storage is shared, and so class instances can be seen and manipulated by all threads.

You need to take extra care when writing OOP multi-threaded code for this reason.

Arbitrary sharing of state in objects is very likely to lead to chaos (Windows Vista Explorer crash)

Try to use a well-structured concurrent design pattern, like the ones we are going to describe.

Thread Coordination

Data primitives to allow one thread to process data without interference from others.

Include semaphores, mutexes, condition variables, monitors.

These are all low-level constructs and very error-prone.

Java, Objective-C and C# support a “synchronized” block wrapper.

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 trying this on the same object will block at the synchronized statement until the first thread exits this block.

Synchronized methods

```
Public class myInc {
    int val;
    public synchronized void incr() {
        if (val < 10) val = val + 1;
    }
}
```

Locks on the class instance.

This is like wrapping synchronized(self) around the method.

Reentrancy

```
Public class myFact {
    public synchronized int fact(int n) {
            if (n == 0) return 1;
            else return n * fact(n-1);
    }
}
```

Locks on the class instance the first time. Reentrancy allows the same thread to pass through the lock multiple times.

A count is kept of the levels of reentrance, so the lock releases at the right time.

Thread-Safety

By synchronizing all the methods of a class, we can usually achieve thread-safety.

You still need to watch out for:

- public instance variables
- Static variables (but you can lock on the class object)

And classes that use thread-safe classes still need more work.

Its surprisingly difficult to get it right. No need in most cases. Thread-safe classes exist for many important data structures (e.g. collections in Java).

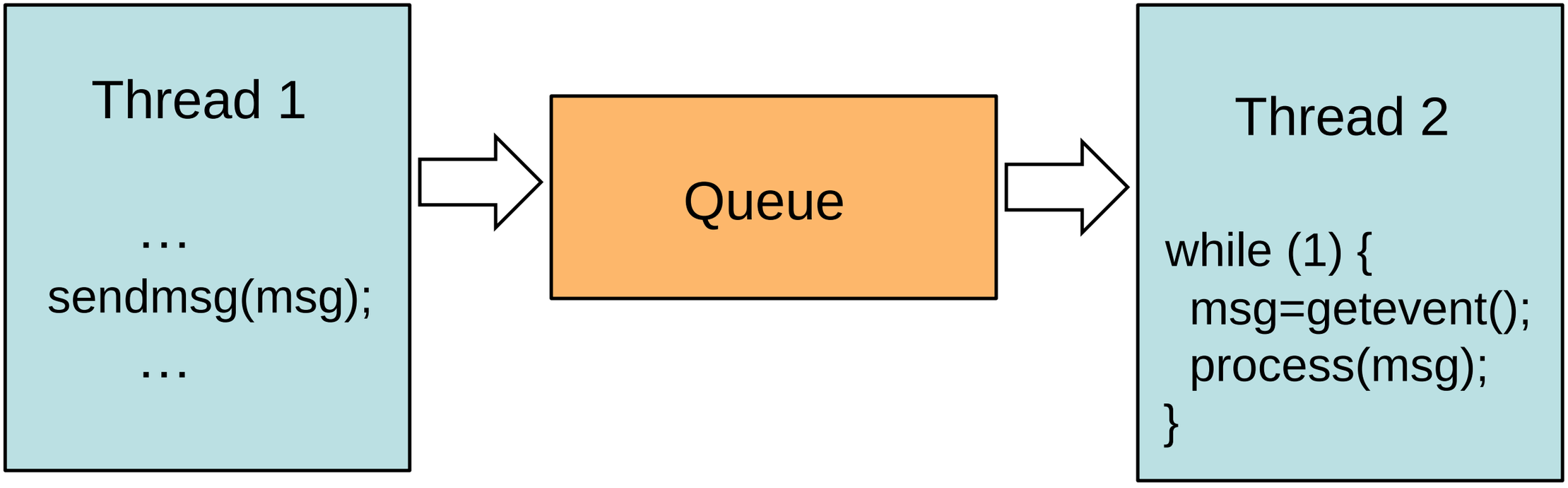
Design Patterns

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

Message Queues

Two threads (or processes) with limited communication can use a message queue.

This is a simple implementation which minimizes coordination and data-sharing between the two threads.



Message Queues

Examples:

- The event queue in almost every GUI toolkit
- The "Handler" queue for threads in Java
- .NET MessageQueue objects
- Thread-specific message queues in the Windows GUI API
- Java Message Service (JMS)

And similar primitives exist between processes:

- POSIX message queues (in Linux)
- CORBA asynchronous messaging

Message Queues

We can realize a message queue by synchronizing queue and dequeue operations in a standard queue. i.e.

```
public Class myQueue<T> {
   Queue<T> Q;
   public synchronized void enqueue(T v) {
      Q.enqueue(v);
   }
   public synchronized T dequeue() {
      return Q.dequeue();
   }
```

Message Queues

Advantages:

- Code is basically sequential in each thread. Much easier to develop and debug.
- Reusable queue libraries do all the hard work.

Disadvantages:

- Inefficient if too much communication, or if complex data structures are passed.
- Need message loop/dispatcher on receiving end – not very modular.

Runnables

A weakness with a simple message queue approach is that behavior is very limited – the receiver only responds to messages it already knows what to do with.

A much more powerful mechanism is to post Runnables:

```
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.

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).

Futures

But the method 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()

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
}
```

Design Patterns

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

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) are assigned to threads by the pool service. You don't normally need to know what is happening.

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

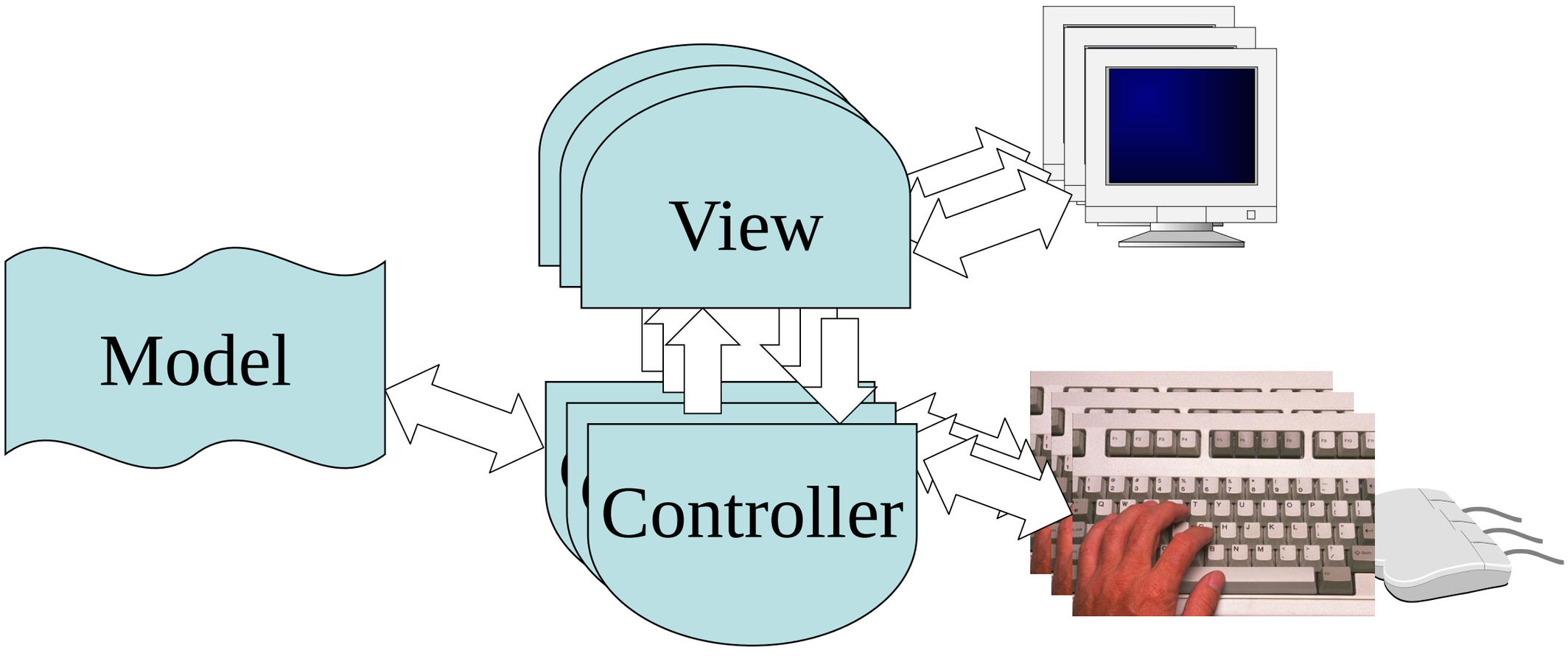
- Message queue
- GUI Thread / Worker thread pool
- Database / Model-View-Controller
- Actor

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

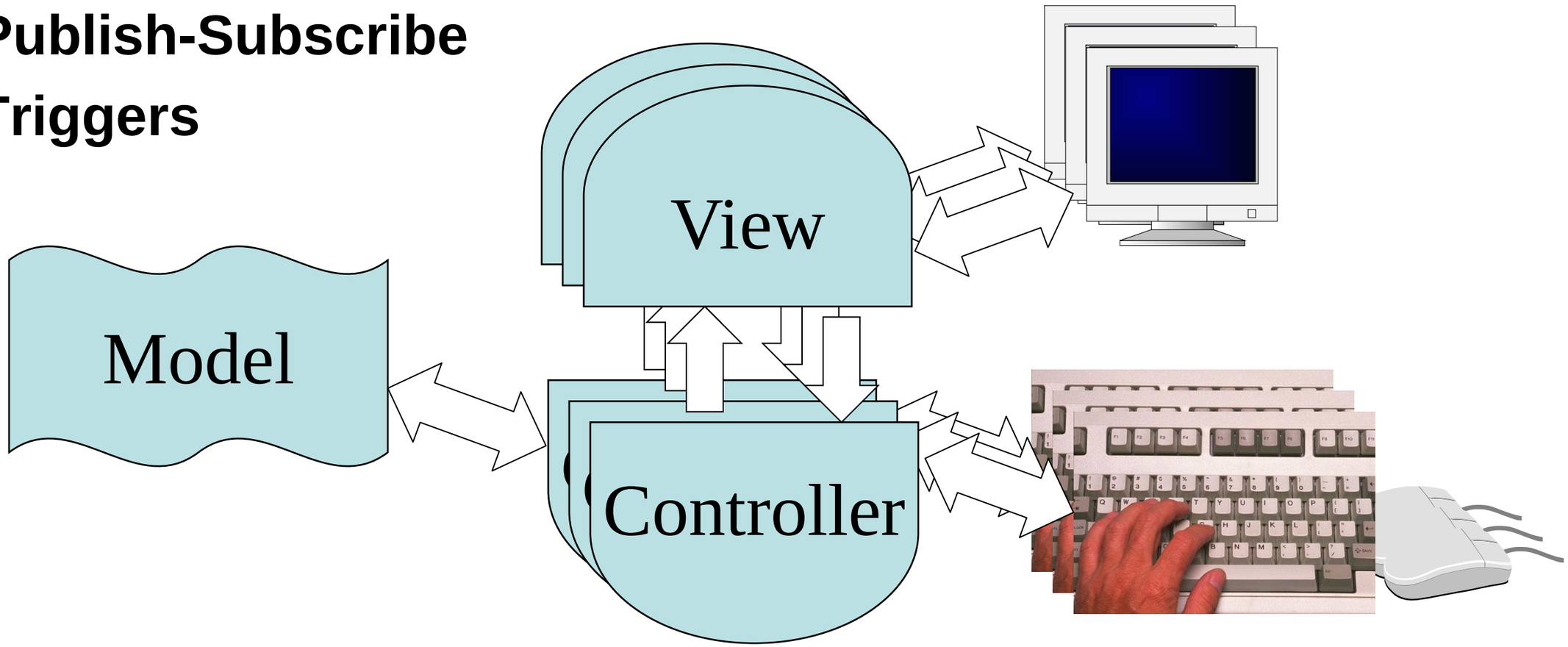
Databases provide an excellent backend for the model:

Transactions – complex updates are atomic.

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

Consistency constraints (relations).

Publish-Subscribe Triggers



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.

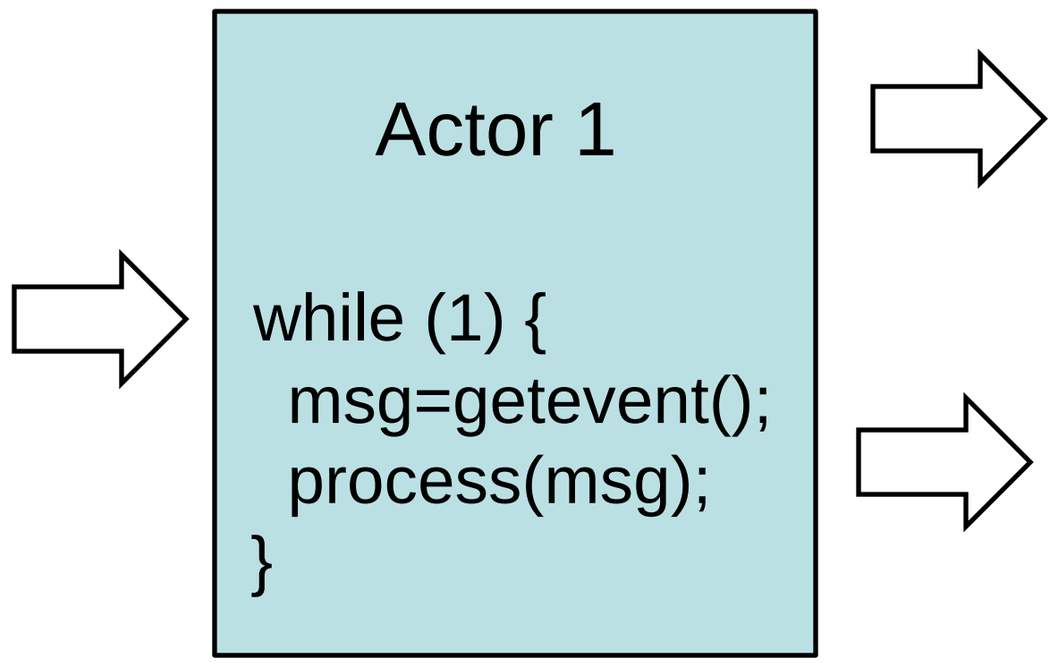
Actor

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.



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