CS 160: Interactive Programming

Professor John Canny

Outline

- Callbacks and Delegates
- Multi-threaded programming
- Model-view controller

callbacks and delegates

Multi-threaded programming

Model-view controller

Callbacks

Window System

Event Queue

Mouse & Keyboard

Myclass

…data

…method1

…method2

Widget1

Myclass2

…

Widget2

Widget3

Callback Registration

Your code

Myclass

…data

…method1

…method2

Widget1

Mouse & Keyboard

Myclass2

…

Widget2

Widget3

Callbacks: Typical uses

C++: One of the options to a Windows Form Constructor is usually a callback function.

C#: Within the main form class:

private System.Windows.Forms.MenuItem GameExit;

...during form initialization...

this.GameExit.Click += new System.EventHandler(myclass.GameExit_Click);

...defined among the myclass methods...

public void GameExit_Click(object sender, System.EventArgs e)
Callbacks and Delegates

In C#, method pointers are discouraged. Instead, a class instance representing the method is used. This class instance is called a delegate.

Method registration from the example looked like this:
```csharp
this.GameExit.Click += new System.EventHandler(myclass.GameExit_Click);
```

Here `myclass.GameExit_Click()` is the method, and `new System.EventHandler(…)` creates a delegate for it.

C#’s event model only permits registration of delegates.

Delegates

The delegate class normally overloads `operator()` with the same arguments as the method its based on.

So
```csharp
delegate(a, b, c)
```

Has the same effect as
```csharp
method(a, b, c)
```

But this means the delegate class must be redefined if the argument types to the method change.

This is a type-safe way to refer to methods.

The (minor) disadvantage is that we are using heap storage for each method reference.

To Thread, or not to Thread?

That is the question...

When thou must thread

Use separate threads for any operations that can occur asynchronously:

* Large file operations - use separate threads if you need to be updating and large files.
* Network communication (sockets): use one thread for each connection.
* Use a thread for each other I/O device, e.g. one each for reading from or writing to the sound card
* Timers: if you schedule events to happen later, you need a thread to trigger that action

When thou shoulds’t thread

There are few more reasons:

* Your computer has many cores (CPUs), and threads are the easiest way for the OS to keep them busy
* Providing progress indicators for long operations
* Keeping an interactive help system alive while your app is running
When thou hast not a choice

- In C#, there is always a garbage collection thread running, or trying to.
- Normally you don't "see" this thread (it waits until all other threads suspend), but you should know that its there, and what it does (moving objects).
- Code you put in "finalizers" (which do clean up before an object is garbage collected) runs in its own thread.

How many Multithreaded Apps?

- Multithreading is the norm for interactive, networked apps, may 95% of all applications.
- Just about all the 160 projects should be multi-threaded, at least in production versions.

What's in a thread?

A thread is a partial virtual machine (with its own stack) that runs your program. Threads share heap storage and static variables. (processes don't share memory)

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

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 Unsafe-ty

- What would happen if two threads tried to execute?:
  ```
  int fact(int n) {
    static int i, p;
    for (i=1, p=1; i<=n; i++) p*=i;
    return p;
  }
  ```

Thread Local Storage

- As a general rule, you should try to use different class instances in each thread to minimize conflicts.
- C# and Java have some support for this, and allow the same name to refer to different storage in each thread.
- Using thread local storage is a fast track to thread safety, and can greatly simplify multithreaded programming, e.g. separate thread local state for each remote network connection or file operation...
Thread Communication

1. Of course, the whole point of threads is to allow fast communication through shared memory. Everything can't be thread local, or the threads could never communicate.

2. Threads communicate through various shared objects. But whenever they share an object, we must be careful about how they do it to avoid problems.

3. Let's start with a method we have already seen...

Message Queues

1. The window system and processes managed by the OS communicate using message queues:
   * Event queues and sockets are examples of message-queue primitives.
   * One process can push data into a queue or socket at any time.
   * Another process can poll the queue at its convenience and read data when its available.

Shared data

1. You could certainly implement a queue class with instances which are shared between threads to allow them to communicate. This is a sensible approach but sometimes too expensive.

2. Any piece of shared data can be used for communication. But we must be sure that changes made by one thread are fully complete before another thread sees them. This is the synchronization problem.

3. Note: message queues need synchronization too...

Synchronization

Shared-memory communication poses challenges. If you rely on “mailbox” primitives, things can go wrong:

0 <blank>

Data

Flag to show this thread has written new data

Intuitively, threads that want to write should:

wait until thread_id = 0;
set thread_id = 1;
write data;

But thread switching can happen anytime, e.g.

wait until thread_id = 0;
set thread_id = 1;
write data;
Synchronization

A switch between checking the flag and setting it allows both threads to (incorrectly) write the flag and their data. To prevent this, we define critical sections of the code that cannot be interrupted.

Monitors

In C#, critical sections are described with monitors on specific class instances:

```csharp
Monitor.Enter(classinstance);
...critical section code updating classinstance...
Monitor.Exit(classinstance);
```

Which is a "lock" on the classinstance instance. No other thread can execute that code section on that instance while it is locked.

Monitors without blocking

When a thread attempts to execute `Monitor.Enter(classinstance);` on an instance that is already locked, it will block until the other thread has released the lock.

```csharp
if thread_id == 0:
    set thread_id = 1;
write data;
```

Critical section, thread can't be pre-empted.

Monitors and Exceptions

You need to be very careful when using monitors because if there is an exception in locked code, the class instance may remain locked (see readings). If an exception is possible, there should be a try...finally block around the critical section, and the `Monitor.Exit()` call should be in the finally block.

See also the lock() statement which does exception handling automatically.
Threading Do's

1. Do use threads in interactive applications to deal with asynchronous events: network, files, media etc.
2. Do keep threads as independent as possible by creating separate class instances (or separate classes) for each thread.
3. Use shared variables for communication, and choose appropriate primitives: add buffering (queues) if tight synchronization is not needed or desired.

3/8/2006 31

Threading Do's

4. Do use monitors to localize critical sections to particular class instances.
5. If you use a large shared datastore (a "database"), consider dividing it into small class instances ("records") that can be updated independently.

3/8/2006 32

Threading Don't's

1. Don't attempt to communicate from one thread to another by "calling" the other thread's methods – it is not thread-safe.
2. Don't share too many class instances between threads and attempt to synchronize them all. This leads to many kind of disaster.
3. Don't overdo monitor'ed code, the more code that's locked, the harder it is for other threads to run, and you may cause a deadlock.

3/8/2006 33

Model-View-Controller

1. Architecture for interactive apps
   * Introduced by Smalltalk developers at PARC
   * Partitions application in a way that is
     * Scalable
     * Maintainable

3/8/2006 35

Model-View-Controller

1. Microsoft version:
   Document/View project type (MFC) available in Visual Studio
2. Creating one of these initializes a "view" class and a "document" (model) class.
Example Application

Model

- Information the app is trying to manipulate
- Representation of essential data
  * Circuit for a CAD program
  * Shapes in a drawing program

View

- Implements a visual display of the model
- May have multiple views
  * e.g., shape view and numerical view

Multiple Views

Controller

- Receives all input events from the user
- Decides what they mean and what to do
  * Communicates with view to determine which objects are being manipulated (e.g., selection)
  * Calls model methods to make changes on objects
    + model makes change and notifies views to update
**Controller**

- Blue circles: 4
- Cardinal squares: 2

---

**Controller**

- Blue circles: 4
- Cardinal squares: 2

---

**Relationship of View & Controller**

"pattern of behavior in response to user events (controller issues) is independent of visual geometry (view issues)"

- Controller must contact view to interpret what user events mean (e.g., selection)

---

**Combining View & Controller**

- View and controller are tightly intertwined
  - Lots of communication between the two
- Almost always occur in pairs
  - i.e., for each view, need a separate controller
- Many architectures combine into a single class (e.g., MS MFC)

---

**Why MVC?**

- Combining MVC into one class or using global variables will not scale
  - Model may have more than one view
  - Each view is different and needs update when model changes
- Separation eases maintenance
  - Easy to add a new view later
  - New model info may be needed, but old views still work
  - Can change a view later, e.g., draw shapes in 3-d

---

**Adding Views Later**

- Blue circles: 4
- Cardinal squares: 2
**Event Flow**

Creating a new shape

---

**Event Flow (cont.)**

Assume blue circle selected

---

Press mouse over tentative position

Windowing system identifies proper window for event

Controller for drawing area gets mouse click event

Checks mode and sees "circle"

Calls models AddCircle method with new position

---

AddCircle adds new circle to model's list of objects

Model then notifies list of views of change

- Drawing area view and text summary view

Views notifies windowing system of "damage"

- Both views notify WS without making changes yet!

---

Views return to model, which returns to controller

Controller returns to event handler

Event handler notices damage requests pending and responds

If one of the views was obscured, it would be ignored

---

Event handler calls view's Redraw methods with damaged area

Views redraw all objects in model that are in damaged area
Summary

- Callbacks and Delegates
- Multi-threaded programming
- Model-view controller