Chords

Concurrency Abstractions in C#
Motivation

Concurrency

- critical factor in behavior/performance
- affects semantics of all other constructs
- advantages of language vs. library
  - Compiler analysis/optimization
  - Clarity of syntax
- asynchronous communication
  - Occurs at various levels
  - Requires language support
Basic Constructs – Asynchronous Methods

Syntax:

```csharp
async postEvent (EventInfo data) {
    // method body using data
}
```

- calls to async method return “immediately”
- method body is scheduled for execution in another thread
- no return result is possible
- similar to sending message/event
Basic Constructs - Chords

Example:

```c#
public class Buffer {
    public string Get() & public async Put (string s) {
        return s;
    }
}
```

- illustrates a single chord with two methods
- chord body is executed only when all methods in the chord have been called
- non-async method call implicitly blocked/queued until chord completes
- async method calls are queued until matched (caller not blocked)
- at most one non-async method per chord
- non-deterministic selection of method calls matched by chord
- chord body executes in thread of non-async caller (unless all methods in chord are async methods, in which case a new thread is created)
Executing Chords

Invocation queues (one per method)

Methods

Bitmaps (one per chord)

execute \([a,b]\) chord

execute \([b,c]\) chord
“Counting” via Methods

class Token
{
    public Token (int initial_tokens) {
        for (int i=0; i<initial_tokens; i++) Release();
    }
    public int Grab (int id) & public async Release() {
        return id;
    }
}

- allows clients to Grab and Release a limited number of tokens
- argument on Grab returned to client
Recording “state” via Methods

```csharp
public class OneCell {
    public OneCell() { empty(); }

    public void Put(object o) & private async empty() {
        contains(o);
    }

    public object Get() & private async contains(object o) {
        empty();
        return o;
    }
}
```

- methods `empty` and `contains` are declared private
- methods `empty` and `contains` “carries” the state of the cell
Reader-Writer Example

class ReaderWriter
{
    ReaderWriter() { idle(); }

    public void Shared() & async idle() { s(1); }
    public void Shared() & async s(int n) { s(n+1); }
    public void ReleaseShared() & async s(int n) {
        if (n == 1) idle(); else s(n-1); }
    public void Exclusive() & async idle() {}
    public void ReleaseExclusive() { idle(); }
}
Active Object (Actor): Base Class

```csharp
public abstract class ActiveObject {
    protected bool done;
    abstract protected void ProcessMessage();
    public ActiveObject() {
        done = false;
        mainLoop();
    }
    async mainLoop() {
        while(!done) { ProcessMessage(); }
    }
}
```

- actor: thread per object; repeatedly processes received messages
- note: thread created by call to async mainLoop()
- abstract class creates basic actor pattern
**Active Object (Actor): Event Example**

```csharp
public class StockServer : ActiveObject {
    private ArrayList clients = new ArrayList();

    public async AddClient (Client c) 
        & override protected void ProcessMessage() { clients.Add(c); }

    public async WireQuote (Quote q) 
        & override protected void ProcessMessage() { 
            foreach (Client c in clients) { c.UpdateQuote(q); }
        }

    public async CloseDown() 
        & override protected void ProcessMessage() { done = true; }
}
```

- message reception/processing driven by ProcessMessage invocations in mainLoop
## Implementation Outline

<table>
<thead>
<tr>
<th>chord</th>
<th>bitmap, one bit for each method in the chord</th>
</tr>
</thead>
<tbody>
<tr>
<td>async method with argument(s) of type m</td>
<td>mQ: to hold message (e.g., intQ for integers)</td>
</tr>
<tr>
<td>async methods with no argument(s)</td>
<td>voidQ: a counter</td>
</tr>
<tr>
<td>synchronous method</td>
<td>threadQ: for blocking caller threads</td>
</tr>
</tbody>
</table>
## Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Test</th>
<th>operations/sec (thousands)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>polyphonic</td>
<td>non-polyphonic</td>
</tr>
<tr>
<td>single processor</td>
<td>ping pong</td>
<td>115</td>
<td>240</td>
</tr>
<tr>
<td></td>
<td>bounded buffer (1 prod/1 cons)</td>
<td>682</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>bounded buffer (2 prod/2 cons)</td>
<td>423</td>
<td>118</td>
</tr>
<tr>
<td>dual processor</td>
<td>ping pong</td>
<td>66</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>bounded buffer (1 prod/1 cons)</td>
<td>288</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>bounded buffer (2 prod/2 cons)</td>
<td>125</td>
<td>42</td>
</tr>
</tbody>
</table>
Syntactic Extension

class ReaderWriter {
    async idle();
    async s(int);

    ReaderWriter() { idle(); }
    public void Shared()
        when idle() { s(1); }
        when s(int n) { s(n+1); }

    public void ReleaseShared()
        when s(int n) { if n==1 idle(); else s(n-1); }

    public void Exclusive()
        when idle() {}

    public void ReleaseExclusive() { idle(); }
}