Concurrency Abstractions in C#

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:

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

- calls to async methods return “immediately”
- method body scheduled for execution in another thread
- no return result
- similar to sending message/event

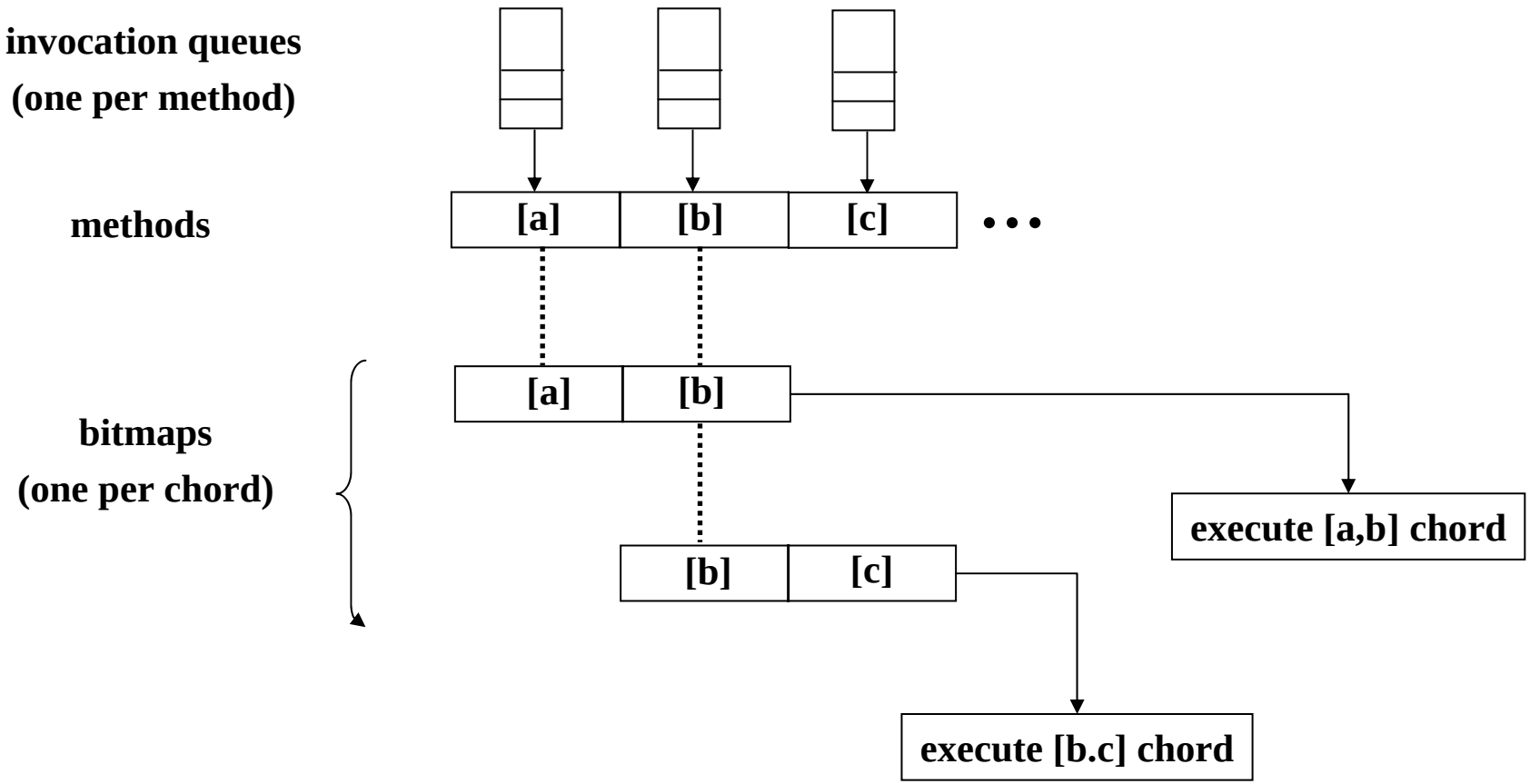
Basic Constructs - Chords

Example:

```
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 complete
- 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



“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

```
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();
                returns o;}
}
```

- methods *empty* and *contains* are declared private
- methods *empty* and *contains* “carries” the state of the cell

Reader-Write 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

```
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 infrastructure/pattern

Active Object (Actor): Event Example

```
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

chord	bitmap, one bit for each method in the chord
async method with argument(s) of type m	mQ: to hold message (e.g., intQ)
async method with no arguments	voidQ: a counter
synchronous method	threadQ: for blocking caller threads

Performance

Benchmark	Test	operations/sec (thousands)	
		polyphonic	**non-polyphonic**
single processor	ping pong	115	240
	bounded buffer (1 prod/1 cons)	682	115
	bounded buffer (2 prod/2 cons)	423	118
dual processor	ping pong	66	70
	bounded buffer (1 prod/1 cons)	288	250
	bounded buffer (2 prod/2 cons)	125	42

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 (); }
}
```