# Lecture 14
# Concurrent Programming

9th October 2003

*In theory, there is no difference between theory and practice. In practice, there is.* – Yogi Berra

## Discussion: Priority Inversion

What did your program look like? What did it do? Was the result repeatable? Did you try it on different JVMs? What do you conclude about the JVM(s) that you used?

## Explicit-communication based concurrent programming

Up until now we have been looking at concurrent programming based on a shared-memory model.

We move now to looking at programming using explicit communication without an assumption of shared memory. There is an incredible variety of mechanisms occupying this space. We will look at some of the dimensions on which they vary, and some of the details of implemented mechanisms found in real systems.

It is somewhat unfortunate that the book treats all of these in a section entitled "Distributed Programming". While it is true that these mechanisms find natural application in distributed programming they also have a natural place in non-distributed settings. There is, I claim, some virtue in programming without using shared memory, even if it is available and provides the underlying implementation of the communication mechanisms: using only communication mechanisms really forces you to think about the interactions between processes.

In the second class project (to be handed out later) you will implement communications using shared-memory mechanisms.

**Dimensions of variation**

| Synchronization | End-point naming |
|---|---|
| **synch recv/asynch send** | **channels(static/dynamic)** |
| **synch send/recv** | **processes** |
| **synch (send+recv) [RPC client]** | **processes+ports** |

| Process/msg relationship | language integration |
|---|---|
| existing process recvs | part of language |
| new process recvs | library |
| | |

| Choice |
|---|
| none |
| input only |
| input/output |

## Asynchronous message passing

In an asynchronous message passing mechanism, send operations succeed immediately. The values being sent are buffered by the mechanism. We start with the assumption that buffering is unbounded – relaxing the assumption leads to additional synchronization (making senders wait sometimes) or unreliability (drop messages for which there is no room).

## Synchronous message passing

In a synchronous message passing system, both send and recv operations are blocking. Each send is paired with a recv, and the pair are performed simultaneously. In theory, synchronous message passing is what you get with a blocking bounded buffer of size 0 (not 1!). The sense one has of what is happening is quite different between asynchronous and synchronous systems. In the asynchronous system we would try to design things so that blocking due to either a full or empty buffer was rare and we'd optimized the non-blocking case. In the synchronous system blocking always occurs and we need to optimize for it.

### Choosing between synchronous or asynchronous message passing

Synchronous MP with general choice is more powerful – with synchronous MP with general choice you can implement asynchronous MP with abstract general choice but not vice versa.

Synchronous MP is easier to reason about.

Synchronous MP typically fails fast when done incorrectly, while asynchronous won't fail until all buffers (all available memory?) are full.

Synchronous MP is lousy with large geographic separation. Why? Asynchronous MP with message drops models pretty well the situation in true distributed computing. A good deal of work, both theoretical and practical :-), has gone into working out algorithms and protocols for successful operation in this model (If you want to know more, take 464/564 Distributed Systems Concepts and Programming next semester). In this class we'll primarily concern ourselves with reliable synchronous and asynchronous mechanisms.

## Program example

### Active Monitor

```
chan request(int clientID, types of input values);
chan reply[n](types of results);
process Server {
  int clientID;
  declarations of other permanent variables;
  initialization code;
  while (true) { ## loop invariant MI
    receive request(clientID, input values);
    code from body of operation op;
    send reply[clientID](results);
  }
}
process Client[i = 0 to n-1] {
 ...
 send request(i, value arguments); # "call" op
 receive reply[i](result arguments); # wait for reply
 ...
}
```

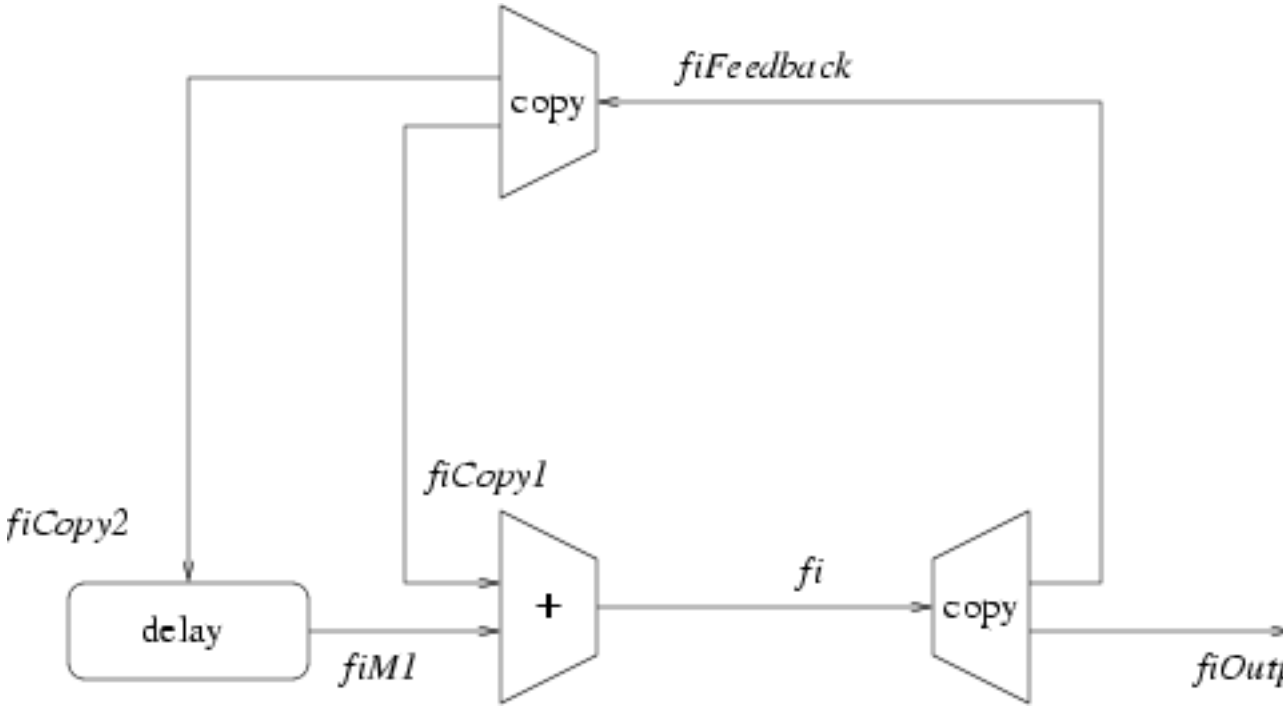
What does the Server process resemble? An event loop, you say? For your consideration: programming with explicit communication is one way to follow Ousterhout's suggestion to use threads sparingly.

Does it matter whether communication is synchronous or asynchronous?

### Process network – Fibonacci

Recall the fibonacci numbers:

$$\begin{array}{rcl} fib_1 & = & 1 \\ fib_2 & = & 1 \\ fib_{i+1} & = & fib_{i-1} + fib_i \end{array}$$




We can create a *process network* whose output consists of *n* fibonacci numbers:

```
// all of the following channels are int channels
chan fi, fiOutput, fiFeedback, fiCopy1, fiCopy2, fiM1;
process copyOutput {
   while 1 {
      receive fi (i);
      send fiOutput (i);
      send fiFeedback (i);
   }
}
process copyFeedback {
   while 1 {
      receive fiFeedback (i);
      send fiCopy1 (i);
      send fiCopy2 (i);
   }
}
process delayFeedback {
   int delayed = 0;
   while 1 {
      receive fiCopy2 (current);
      send fiM1 (delayed);
```

```
      delayed = current;
   }
}
process sum {
   while 1 {
      receive fiM1 (addend1);
      receive fiCopy1 (addend2);
      send fi (addend1+addend2);
   }
}
send fi (1)
```

## Assignment

Reading: Chapter 7

Exercises: 7.2, 7.6, 7.7, 7.18 ; due next time