Concurrency: Theory, Languages and Programming

– From Pi to Java and Back –

Session 8 – Dec 11th, 2002

Martin Odersky

EPFL-LAMP

From Pi to Java and Back

- CCS and π-Calculus are established formalisms for the *specification* and *study* of concurrent systems.
- When it comes to *programming*, most concurrent systems are written using a thread library with semaphores, monitors, etc.
- What is the relationship between the two idioms?
- We will answer that by
  - encoding imperative synchronization constructs in π-calculus
  - implementing π-calculus using traditional threads (that's what pilib does).

From Java to Pi

What follows are encodings of

- Semaphores
- Monitors
- Readers/writers locks as used in databases

in pi calculus.

To keep the presentation simpler, we actually use *pilib* instead of

π-calculus as target language.

Preliminaries: Signals

- All communication in *pilib* works over channels.
- An action without parameters as in CCS is modeled as a channel over which unit values `()` are sent.
- This can be expressed more directly by a *Signal*, defined as follows.

```
class Signal extends Chan[Unit] with {
  def send = write(());
  def receive = read;
}
```

Semaphores

- A semaphore implements two operations, *get* and *release*.
- "Critical regions" of processes are enclosed in calls to first *get*, then *release* of a semaphore.
- Between those two calls, a process is said to own a semaphore.
- The semaphore implementation ensures that at most one thread can own a semaphore at any given time.

A Semaphore Implementation

Here is an implementation of a semaphore, which uses a signal for synchronization.

```
class Semaphore with {
  private val busy = new Signal;
  def get = busy.send;
  def release = spawn < busy.receive >;
  release
}
```

Usage:

```
val s = new Semaphore;
s.get; ...; s.release;
```


Another Semaphore Implementation

Here is another implementation, which implements the synchronization policy in a recursive process.

```
class Semaphore with {
    private val busy = new Signal;
    private val free = new Signal;
    def get = busy.send;
    def release = free.send;
    def sema = { busy.receive; free.receive ; sema }
    spawn < sema >
}
```


Binary and Counting Semaphores

- **Question:** In both semaphore implementations, what happens if there is a *release* without preceding *get*?

```
val s = new Semaphore;
s.release;      // ?
```

- Two other possibilities:
  - The *release* should be forgotten, i.e.
    ```
    val s = new Semaphore;
    s.release; s.get; s.get;     // 2nd get blocks
    ```
  - The *release* should enable another subsequent *get*. i.e.
    ```
    val s = new Semaphore;
    s.release; s.get; s,get     // 2nd get continues
    ```
- The first behavior is called a *binary* semaphore, the second a *counting* semaphore.

Monitors

- A monitor encapsulates one or more procedures that need to be executed mutually exclusively.
- Monitors also offer a way to wait for a certain condition or to signal that a condition is established.
- Mutual exclusion can be implemented by a semaphore.
- However, waiting on conditions and mutual exclusion are not independent, since a waiting process has to release the monitor (to allow some other process to establish the condition).
- We now explain monitors in detail, using the Java implementation as example.

Monitors in Java

- A monitor in Java is represented by the *synchronized* language construct and the following three methods.

void notify()	Wakes up a single thread that is waiting on this object's monitor.
void notifyAll()	Wakes up all threads that are waiting on this object's monitor.
void wait()	Causes current thread to wait until another thread invokes the *notify*() method or the *notifyAll*() method for this object.

- The *synchronized* construct is written as follows:

  synchronized (*mobj*) { *block* }

  where *mobj* is a monitor and *block* is a sequence of statements that is executed under mutual exclusion.
- Alternatively, *synchronized* can also used as a method modifier; then the whole method body is treated as a synchronized block.
- Java treats every object as a potential monitor.

Monitors in Scala

Monitors in Scala are almost the same as in Java. There are only two differences.

1. Not every Scala object is a monitor. Monitor operations are available on objects of classes which inherit from *scala.Monitor*.
2. Instead of a *synchronized* language construct or modifier there is a predefined method of the same name in class *Monitor*.

```
class Monitor with {
  def synchronized[a](def b: ()a): a;
  def notify(): Unit;
  def notifyAll(): Unit;
  def wait(): Unit;
  def await(def cond: Boolean): Unit
}
```

Usage Example

Here is an example how *synchronized* is used in Scala.

```
class Counter extends Monitor with {
  private var x = 0;
  def increment ( ) = synchronized { x = x + 1 }
  def decrement ( ) = synchronized { x = x − 1 }
  def value ( ) = x
}
```


Usage Example (2)

As an example that also uses conditions, here is a counter which can never go negative.

```
class NonNegCounter extends Monitor with {
  private var x = 0;
  def increment ( ) = synchronized {
    x = x + 1;
    if (x == 1) notifyAll ( );
  }
  def decrement ( ) = synchronized {
    while (x == 0) { wait ( ) }
    x = x − 1
  }
  def value ( ) = x
}
```


Question: Why use *while* instead of ***if*** as a guard for the *wait*?

- In fact, the *while*−*wait* idiom is so common that it is encapsulated in a separate method in class *Monitor*.

```
def await (def cond : Boolean): Unit =
  while (!cond) wait ( )
```


- With *await*, the counter example can be written more concisely as follows.

```
class NonNegCounter extends Monitor with {
  private var x = 0;
  def increment ( ) = synchronized {
    x = x + 1;
    if (x == 1) notifyAll ( );
  }
  def decrement ( ) = synchronized {
    await (x != 0);
    x = x − 1
  }
  def value ( ) = x
}
```

Another Example: Bounded Buffer

Here is the implementation of a class for bounded buffers.

```
class Buffer[a](size: Int) extends Monitor with {
  var in = 0, out = 0, n = 0;
  val elems = new Array[a](size);
  def put(x: a) = synchronized {
    await(n < size);
    elems(out) = x;
    out = (out + 1) % size;
    if (n == 0) notifyAll();
    n = n + 1;
  }
```



```
  def get: a = synchronized {
    await(n > 0);
    val x = elems(in);
    in = (in + 1) % size;
    if (n == size) notifyAll();
    n = n − 1;
    x
  }
}
```


Coding Monitors in PiLib

- We now show how monitors can be implemented in *pilib*.
- In reality, it's the other way round – *pilib* is implemented using Java's monitor concept.
- But the present encoding is interesting since it gives a alternative account of monitors as higher-level synchronization constructs.

- The encoding uses two internal data structures
  - A lock to guarantee mutual exclusion
  - A list of waiting processes to be re-executed on a *notify* operation.

```
class JavaMonitor with {
  private val lock = new Semaphore;
  private var waiting: List[Signal] = [];
```

The *synchronized* implementation is straightforward:

```
  def synchronized[a](def s: a): a = {
    lock.get; val result = s; lock.release; result
  }
```

The *Wait* operation releases the monitor lock and waits for a private signal which is appended to the *waiting* list.

```
def Wait = {
  val s = new Signal;
  waiting = waiting ::: [s];
  lock.release;
  s.receive;
  lock.get;
}
```

(to avoid a conflict with Java's *wait* method, we have written *Wait* in upper case.)

The *Notify* operation wakes up the first process on the *waiting* list and removes the entry from the list.

```
def Notify =
  if (!waiting.isEmpty) {
    waiting.head.send;
    waiting = waiting.tail;
  }
```

The *NotifyAll* operation does the same to all processes on the list.

```
def NotifyAll =
  while (!waiting.isEmpty) {
    waiting.head.send;
    waiting = waiting.tail;
  }
```


A Limitation

- There is one aspect where the encoding of Java's monitors in pilib is not faithful.
- In Java, a thread owning a monitor is allowed to enter another synchronized block on the same monitor.
- **Question:** Using the *pilib* implementation of monitors and given the class:

  ```
  class Counter2 extends Counter with {
    def updown() = synchronized { increment(); decrement(); }
  }
  ```

  what is the effect of (***new*** *Counter2*).*updown*()?
- The Java behavior can be modeled in *pilib* only if one introduces process identifiers (which changes the signatures of operations).

Readers/Writers Locks

- A more complex form of synchronization distinguishes between *readers* which access a common resource without modifying it and *writers* which can both access and modify it.
- To synchronize readers and writers we need to implement operations *startRead*, *startWrite*, *endRead*, *endWrite*, such that:
  - there can be multiple concurrent readers, and
  - there can only be one writer at one time.
  - In addition it should be guaranteed that pending write requests are not delayed indefinitely (provided the process scheduler is *fair*).

Readers/Writers in π-calculus

- The idea is to model the operations by signals *sr* (start read), *er* (end read), *sw* (start write) and *ew* (end write).
- These signals are coordinated by process RW_n, where the n subscript indicates the number of readers in the system.

$$
\begin{aligned}
RW_0 &= sr.RW_1 + sw.ew.RW_0 \\
RW_n &= sr.RW_{n+1} + er.RW_{n-1} \quad (n > 0)
\end{aligned}
$$

Readers/Writers in PiLib

We package the π-calculus program in a Scala class as follows.

```
class ReadWriteLock with {
  val sr = new Signal, er = new Signal, sw = new Signal, ew = new Signal;
  def startRead = sr.send;
  def startWrite = sw.send;
  def endRead = er.send;
  def endWrite = ew.send;
  def RW(n: Int): Unit =
    if (n == 0)
      choice { sr * (x => RW(1)) + sw * (x => ew.receive; RW(0)) }
    else
      choice { sr * (x => RW(n+1)) + er * (x => RW(n-1)) }
  spawn < RW(0) >;
}
```


Question:

- Assume that the system is very busy: At any one time there are always processes that want to read and other processes that want to write.
- Assume that processes are scheduled randomly.
- What is the probability that a reader or a writer will never get the resource?

Avoiding Starvation

- How can we avoid the potential starvation of writers?
- An idea is to introduce another signal *ww*, which stands for “want write’ ”.
- A writer process will always execute *ww*, *sw*, and *ew* in that order.
- We then add queue process Q, which sequentializes *sr* and *ww* requests.

- A system with a process P using a readers/writers lock is then composed from

$$P \mid Q \mid RW'_0$$

where Q and RW' are given as follows.

$$\begin{aligned} RW'_0 &= sr'.RW_1 + sw'.ew.RW_0 \\ RW_n &= sr'.RW_{n+1} + er.RW_{n-1} \quad (n > 0) \\ \\ Q &= sr.\overline{sr'}.Q + ww.\overline{sw'}.sw.Q \end{aligned}$$

Readers/Writers Locks using Monitors

- Here is an alternative implementation of a readers/writers lock which uses a monitor.
- There are two counter variables.
  - One counts the number of active (i.e. reading or writing) processes.
  - The other counts the number of active or waiting writers.

```
class ReadWriteLock extends Monitor with {
    private var nactive : Int = 0;
    private var nwriters : Int = 0;
```


A reader can start only if there are no writers active or waiting:

```
def startRead = synchronized {
  await (nwriters == 0);
  nactive = nactive + 1;
}
```

A writer can start only if there are no active processes:

```
def startWrite = synchronized {
  nwriters = nwriters + 1;
  await (nactive == 0);
  nactive = 1;
}
```


Operations *endRead* and *endWrite* decrement counters and possibly notify waiting processes.

```
    def endRead = synchronized {
      nactive = nactive − 1;
      if (nactive == 0) NotifyAll;
    }

    def endWrite = synchronized {
      nwriters = nwriters − 1;
      nactive = 0;
      NotifyAll;
    }
}
```

Question:

- Assume that the system is very busy: At any one time there are always processes that want to read and other processes that want to write.
- Assume that processes are scheduled randomly.
- What is the probability that a reader or a writer will never get the resource?
- Is this acceptable?
- If not, how can it be fixed?