Advances in Programming Languages

APL13: Concurrency

Ian Stark

School of Informatics
The University of Edinburgh

Thursday 28 February 2008
Semester 2 Week 8

Programming-Language Techniques for Concurrency

This is the first of three lectures presenting some programming-language techniques for managing concurrency.

- Java, Erlang
- Polyphonic C#
- Cautionary Tales

Programming-Language Techniques for Concurrency

This is the first of three lectures presenting some programming-language techniques for managing concurrency.

- Java, Erlang
- Polyphonic C#
- Cautionary Tales

Outline

Outline

Why Write Concurrent Programs?

Concurrent programming is about writing code that can handle doing more than one thing at a time.

Why Write Concurrent Programs?

Concurrent programming is about writing code that can handle doing more than one thing at a time. There are several reasons one might want to do this, such as:

- Efficient use of mixed resources (disk, memory, network)
- Responsiveness (GUI, hardware interrupts, managing those mixed resources)
- Speed (multiprocessing, hyperthreading, multicore)
- Multiple clients (database engine, web server)

Why Write Concurrent Programs?

Concurrent programming is about writing code that can handle doing more than one thing at a time. There are several reasons one might want to do this, such as:

- Efficient use of mixed resources (disk, memory, network)
- Responsiveness (GUI, hardware interrupts, managing those mixed resources)
- Speed (multiprocessing, hyperthreading, multicore)
- Multiple clients (database engine, web server)

Note that the aims here are different to *parallel programming*, which is generally about the efficient (and speedy) processing of large sets of data.

It's Hard to Walk and Chew Gum

Concurrent programming offers much, including entirely new problems.

Interference — code that is fine on its own may fail if run concurrently.

Liveness — making sure that a program does anything at all.

Starvation — making sure that all parts of the program make progress.

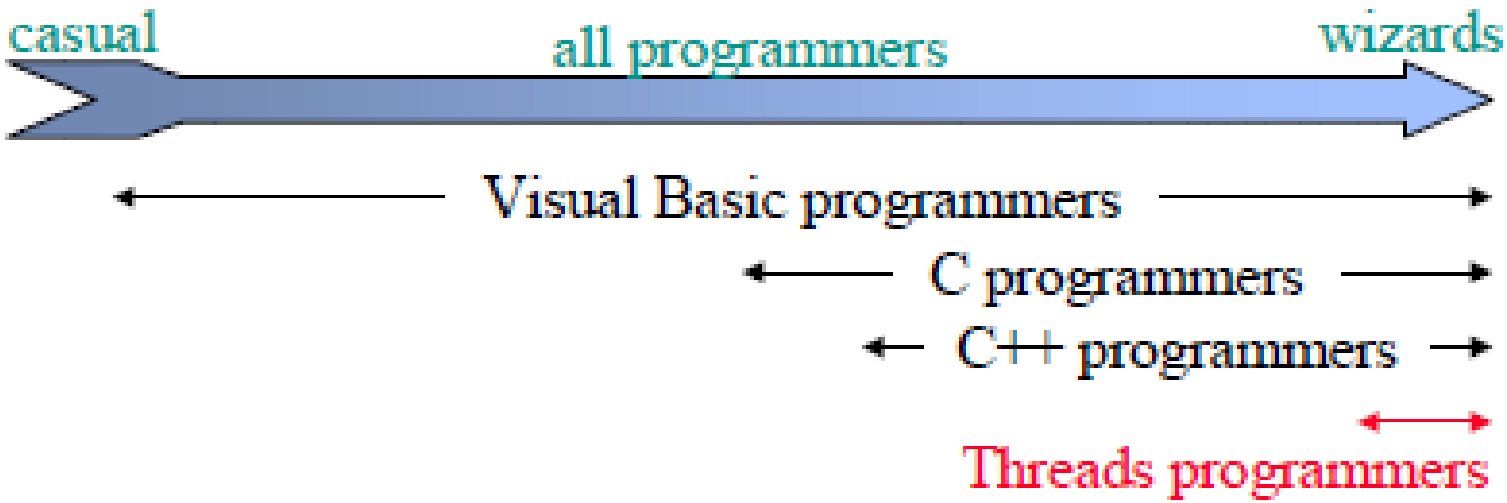
Fairness — making sure that everyone makes reasonable progress.

Safety — making sure that the program always does the right thing.

Specification — just working out what is "the right thing" can be tricky.

Concurrent programming is hard, and although there is considerable research, and even progress, on how to do it well, it is often wise to avoid doing it yourself unless absolutely necessary.

What's Wrong With Threads?



- **Too hard for most programmers to use.**
- **Even for experts, development is painful.**

John Ousterhout: *Why Threads Are A Bad Idea (for most purposes)*
USENIX Technical Conference, invited talk, 1996

Threads and Processes

The last slide notwithstanding, all operating systems and many programming languages provide some form of concurrent programming, usually through a notion of *processes* or *threads*.

The general idea is that a process/thread captures a single flow of control, and a concurrent program or environment will have many of these at a time.

A *scheduler* manages which threads are executing at any time, and how control passes switches between them.

There are many design tradeoffs here, concerning memory separation, mutual protection, communication, scheduling, signalling, ...

Usually "processes" are heavyweight and "threads" lightweight, but there is no hard-and-fast separation. Complete systems may include multiple layers of concurrency.

Critical Sections

A central issue with multiple explicit threads is to avoid interference through shared memory.

```
void moveBy(int dx, int dy) {
    System.out.println("Moving by "+dx+","+dy);
    x = x+dx;
    y = y+dy;
    System.out.println("Completed move");
}

void moveTo(int newx, int newy) {
    System.out.println("Moving to "+newx+","+newy);
    x = newx;
    y = newy;
    System.out.println("Completed move");
}
```

Critical Sections

A central issue with multiple explicit threads is to avoid interference through shared memory.

```
void moveBy(int dx, int dy) {
    .
    .
  x = x+dx;
  y = y+dy;
    .
    .
}
```

```
void moveTo(int newx, int newy) {
    .
    .
  x = newx;
  y = newy;
    .
    .
}
```

Critical Sections

A central issue with multiple explicit threads is to avoid interference through shared memory.

```
void moveBy(int dx, int dy) {
    .
    .
  x = x+dx;
  y = y+dy;
    .
    .
}
```

```
void moveTo(int newx, int newy) {
    .
    .
  x = newx;
  y = newy;
    .
    .
}
```

Because both methods access the fields x and y, it is vital that these two *critical sections* of code are not executing at the same time.

Locks and More

There are many ways to ensure critical sections do not interfere, and refinements to make sure that these constraints do not disable desired concurrency.

- Locks
- Mutexes
- Semaphores
- Condition variables
- Monitors *etc.*

These themselves need an underlying locking mechanism, either in hardware (test-and-set, compare-and-swap,. . .) or software (spinlock, various busy-wait algorithms).

Outline

Concurrency in Java

Java supports concurrent programming as an integral part of the language: threading is always available, although details of its implementation and scheduling will differ between platforms.

There is a class Thread, and threads can be explicitly created and set running on arbitrary code. Threads have unique identifiers, names, and integer priorities.

Parent code can spawn multiple child threads, and then wait for individual children to terminate.

Synchronized Methods

Java provides mutual exclusion for critical sections through the **synchronized** primitive.

```
synchronized void moveBy(int dx, int dy) {
    System.out.println("Moving by "+dx+","+dy);
    x = x+dx;
    y = y+dy;
    System.out.println("Completed move");
}

synchronized void moveTo(int newx, int newy) {
    System.out.println("Moving to "+newx+","+newy);
    x = newx;
    y = newy;
    System.out.println("Completed move");
}
```

Synchronized Methods

Two move methods cannot now execute at the same time on the same object.

```
synchronized void moveBy(int dx, int dy) {
    System.out.println("Moving by "+dx+","+dy);
    x = x+dx;
    y = y+dy;
    System.out.println("Completed move");
}

synchronized void moveTo(int newx, int newy) {
    System.out.println("Moving to "+newx+","+newy);
    x = newx;
    y = newy;
    System.out.println("Completed move");
}
```

Synchronized Methods

Each **synchronized** method must *acquire* a lock before starting and *release* it when finished.

```
synchronized void moveBy(int dx, int dy) {
    System.out.println("Moving by "+dx+","+dy);
    x = x+dx;
    y = y+dy;
    System.out.println("Completed move");
}

synchronized void moveTo(int newx, int newy) {
    System.out.println("Moving to "+newx+","+newy);
    x = newx;
    y = newy;
    System.out.println("Completed move");
}
```

Synchronized Expressions

Every Java object has an implicit associated lock, used by its **synchronized** methods. This can also be used to arrange exclusive access to any block of code:

```
void moveBy(int dx, int dy) {
  System.out.println("Moving by "+dx+","+dy);
  synchronized(this) {
    x = x+dx;              // Only this section of the
    y = y+dy;              // code is critical
  }
  System.out.println("Completed move");
}
```

The locking object need not be **this**, and careful use of multiple lock objects can give finer-grained concurrency.

Condition Variables

Java refines critical regions with basic *condition variables*.

- Synchronized code that finds things are not suitable for it to proceed may wait() on the condition variable associated with its lock. This blocks the code and releases the lock.
- Another thread can acquire the lock and do some work. Because this may change the situation for the other thread, it should notify() or notifyAll() other threads of this.
- Threads waiting on the condition variable will be made runnable again, and can check to see if they are now ready to proceed.

Having threads block saves on busy waiting, and ensures that they only wake when there is something to check.

A Simple Blocking Method

```
class Pigeonhole {

  private Object contents = null;

  synchronized void put (Object o) {

    while (contents != null)    // Wait until the pigeonhole is empty
      try { wait(); }
      catch (InterruptedException ignore) { return; }

    contents = o;               // Fill the pigeonhole
    notifyAll();                // Tell anyone who might be interested
  }
  ...
}
```

Java Concurrency

Summary:

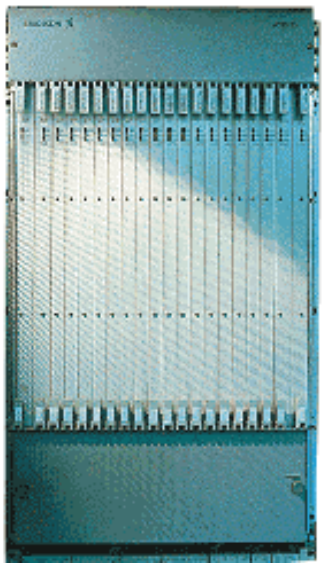
- Java provides concurrency within the language.
- Explicit spawning of multiple threads.
- Threads communicate through shared memory.
- Critical regions can be **synchronized**
- Condition variables with wait and notify to control shared resources.

The library package java.util.concurrent provides several advanced and flexible concurrency operations, requiring more explicit management (and more expertise).

Outline

Erlang

The *Erlang* programming language was originally created by Ericsson for telecommunications equipment. However, it is a general-purpose concurrent and distributed language, now with an open-source implementation and a number of industrial users.



Ericsson AXD 301 multiservice 10–160Gbit/s switch (l,c)

Nortel 8661 SSL Acceleration Ethernet Routing Switch (r)

Some Erlang Features

Courtesy of http://www.erlang.org/faq :

- Declarative, functional language
- Dynamic types
- Pattern matching, list comprehension
- Concurrency, thousands of lightweight threads (at least)
- Distributed, transparently
- Hot code upgrading
- Robust, fault tolerant
- Soft realtime, millisecond tolerance
- Mnesia distributed in-memory database
- OTP, Open Telecoms Library

Concurrency in Erlang

Erlang uses *share-nothing* concurrency, where threads have no shared mutable store.

Instead, communication is by *message-passing*, based on the *Actor* model, and similar to some concurrent object-oriented languages and process calculi.

Every thread has a *mailbox*, to which other threads can send messages. The thread sifts through received messages by pattern-matching.

Messages may include arbitrary data, including the names of other mailboxes. This allows the communication topology between threads to change during execution.

Erlang Mailboxes

Message server from `http://www.erlang.org/doc/getting_started/`

```
% User_List names users and says which client node they are at
server(User_List) ->
    receive
        {From, logon, Name} ->
            New_User_List = server_logon(From, Name, User_List),
            server(New_User_List);
        {From, logoff} ->
            New_User_List = server_logoff(From, User_List),
            server(New_User_List);
        {From, message_to, To, Message} ->
            server_transfer(From, To, Message, User_List),
            server(User_List)
    end.
```

Erlang Concurrency

Summary:

- Erlang provides concurrency within the language.
- Explicit spawning of multiple threads.
- Threads communicate through message-passing
- No shared memory means no critical regions
- Pattern-matching on mailboxes for coding interaction

Summary

- Concurrency is useful (efficiency, responsiveness, speed)
- But can be tricky (interference, deadlock, fairness)
- Concurrency can be in the language, in libraries, or in both
- Java provides shared-memory concurrency
- Java manages concurrency with locks and condition variables
- Erlang provides message-passing concurrency
- Erlang manages concurrency with pattern-matching mailboxes