Concurrency Part 3

by Jørgen Staunstrup, jst@itu.dk, draft version Feb 27, 2021

Resume of Part 1 and 2

The first two sections introduced the stream concept for concurrency programming and coordination of streams.

There has been numerous programming languages with mechanisms for concurrency programming. They all have concepts for defining the computations with independent streams of statements; but they differ significantly in the mechanisms they provide for coordinating the streams. However, these mechanisms fall in two classes (as discussed above): message passing anf shared objects.

Taking a Danish perspective two early examples of programming languages for concurrency are:

1.Søren Lauesen (1975): "A Large Semaphore-Based Operating System" Communications of the ACM 18, 7 (July 1975), pp. 377 - 389.

2.Brinch Hansen, Per (June 1975). "The programming language Concurrent Pascal" IEEE Transactions on Software Engineering (2): pp. 199–207.

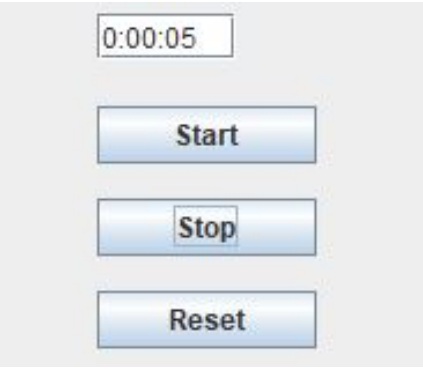
The first discusses the operating system for the RC4000 computer based on message passing (called semaphores in this paper), and the second a language based entirely on shared objects. Both takes a somewhat purist approach of using either one or the other. Many succeeding languages have been more pragmatic allowing a mix of message passing and shared objects. This is the approach taken in Java.

In this next section, we will discuss coordination of streams illustrated with code for a simple Stopwatch. Both a purely message based and a purely object sharing version are given in pseduo-code.

The two versions are then transformed to Java for Android, trying to get as close as possible to the "pure" versions.

The Stopwatch example

The two types of coordination (message passing and Shared objects) can be illustrated by a simple stopwatch.



The stopwatch has three buttons (Start, Stop and Reset) and a display for showing the running time. Following the principles outlined above, the Stopwatch can be programmed with three streams (one for each

of the buttons), and a stream for writing the display. Finally, we need a stream that emits a "tick" every second. All five streams are examples of intrinsic concurrency. A function `notify` is used for coordination. In the next section, we will explain how this coordination can be implemented using either shared objects or message passing.

```
stream startButton= new stream(() -> {
  await(startButton);
  notify(display);
});
stream stopButton= new stream(() -> {
  await(stopButton);
  notify(display);
});
stream resetButton= new stream(() -> {
  await(resetButton);
  notify(display);
});
stream display= new stream(() -> {
  private int t= 0; private boolean running= false;
  await{
    start: running= true;
    stop: running= false;
    reset: write(0); t= 0;
    tick: if (running) write(t++);
  }
});
stream clock = new stream(() -> {
  while true {
    await{1 second};
    notify(display);
  }
});
```

Coordination using message passing

When using (pure) message passing, it is not possible to share objects (memory). The state of the Stopwatch (`seconds` and `running`) is kept locally in the display stream. A call to `write(v)` writes the value `v` in the display.

```
stream startButton= new stream(() -> {
  await(startButton);
  sendMessage(display, start);
});
stream stopButton= new stream(() -> {
  await(stopButton);
  sendMessage(display, stop);
});
stream resetButton= new stream(() -> {
  await(resetButton);
  sendMessage(display, reset);
});
stream display= new stream(() -> {
  receiveMessage(m);
  swich m {
    start: running= true;
    stop:  running= false;
    reset: write(0); t= 0; running= false;
    tick:  if (running) write(t++);
  }
});
```

```
stream clock = new stream(() -> {
  while true {
    await{1 second};
    sendMessage(display, tick);
  }
});
```

You may find a message passing version in Java (for Android) here: https://github.itu.dk/jst/MMAD2021 (pich the StopWatchMP project).

Coordination using shared objects

To coordinate the streams using shared objects, we could use an object holding the state i.e. a counter (`int seconds`) and a boolean (`boolean running`).

```
object SecCounter {
  private int seconds= 0;
  private boolean running= false;
  setRunning(boolean running) { this.running= running;  }
  incr(){ if running { seconds++; } }
  reset(){ seconds= 0; running= false; }
}
```

Using this shared object, the five streams become:

```
stream startButton= new stream(() -> {
  await(startButton);
  SecCounter.setRunning(true);
});
stream stopButton= new stream(() -> {
  await(stopButton);
  SecCounter.setRunning(false);
});
stream resetButton= new stream(() -> {
  await(resetButton);
  SecCounter.reset();
});
stream display= new stream(() -> {
  await(SecCounter changed);
  write(SecCounter.seconds) ;
});
stream clock = new stream(() -> {
    await{1 second};
    SecCounter.incr();
});
```

The main difference in this example is really that this switching on the type of incoming message in the display disappears in the shared object version:

```
receiveMessage(m);
  swich m {
    start: running= true;
    stop:  running= false;
    reset: write(0); t= 0; running= false;
    tick:  if (running) write(t++);
  }
```

This may seem innocent in this small example, but could be a serious challenge in a large system where you want to receive messages on many channels. You may invent syntax to hide this but the implementation must inherently have some kind of poll mechanism where it listens on all the channels.

The Http protocol is probably the most prominent example of message passing. All Http traffic on the Internet is based on sending messages (GET and POST) between client and servers. In the Http protocol port numbers can be used to alleviate the need of a "switch" on the server side.

Java (Android) version based on shared objects

In the solution using shared objects, the implementation of `await(SecCounter changed)` is a bit awkward. It would be simpler to replace this stream with a method in the shared SecCounter object that is called when either `seconds` or `running` changes.

It is not possible in Android to make a version of the Stopwatch based completely on shared objects. The reason is that the display has to write on the screen. Only the so-called UI Thread is allowed to do that. The communication from the timer thread to the UI thread can be done by passing calling a (built in) method `post` on a `View` object, for example, like this:

```
timeView.post(() -> { setDisplay.write(); });
```

You may use the method `post` on any UI element. When doing that, the UI thread which will then do the write.

You may find a complete implementation of the Stopwatch using shared objects here: https://github.itu.dk/jst/MMAD2021

In the next section, it will be discussed how to construct objects than can be used safely by several threads.

Safe sharing of objects

Concurrency (in these notes) are defined as independent streams of statements/instructions. The independence assures that:

- it is possible to handle external events happening in an unpredictable order (inherent concurrency),
- underlying software (such as an operating system) has complete freedom in scheduling statements from different streams (hidden concurrency), and
- different streams can be executed simultaneous by the hardware without any assumptions about the speed of execution (exploitation).

But the independence does create a challenge when the streams share objects (even if the object is just a simple variable). Consider this example of two streams sharind the object (variable) `c` initialized to 100:

```
stream t1= new stream(() -> { s1; s2; ... c= c+20; sk; ....} );
stream t2= new stream(() -> { u1; u2; ... c= c-50; un; ....} );
```

Assume that no other statements than the two shown change the value of `c`; what would be the value of `c` when both of these streams terminate? 70, 120, 50, 217, 0, ... One could imagine different concrete implementations giving any of these results, although most would give one of the three first: 70, 120 or 50. With no further assumptions about the implementation we are not able to predict the result. This is clearly not desirable. One could be tempted to write the code in such a way that the statements preceeding `c= c+20` runs much faster or slower than the statements preceeding `c= c-50`. This is tricky and often a dangerous

practice. For example, one may later change the code in either `t1` or `t2` and forget about the timing assumptions. This is why, the concurrency abstraction stresses that the streams are *independent*.

Hence, the small program above must give the correct result no matter how slow or fast the two streams are executed. If this is the case the program is called *safe*.

Safety of programs can be made safe, byt making all the methods of the shared objects *atomic*. An atomic method is executed completely from start to end without interference from other streams. In the above example, if we make both `c= c+20` and `c= c-50` atomic, there are only one possible result: 70 (either `c` is first incremented by 20 and then decremented by 50 or the other way around).

An object where all methods are atomic is called a `monitor`:

```
object sharedc {
  private int c= 0;
  atomic increment() {c= c+20}
  atomic decrement() {c= c-50}
}
```

Monitors in Java

Java has a multitude of different ways of ensuring safety of shared objects. One of these is to declare all methods of a class `synchronized`:

```
class sharedc {
  private int c= 0;
  public synchronized void increment() {c= c+20;}
  public synchronized void decrement() {c= c-50;}
}
```

Any object of type `sharedc` will then behave as a monitor, i.e. all calls to `increment` or `decrement` on an object will be atomic. Note that it is the object that is a monitor. If we declare two different objects `sharedc c1, c2` then each of them will be a monitor with its own counter (`c`), but there are no constraints between the execution of the methods in `c1` and those in `c2`.

To summarize

Coordination of streams is necessary in any non-trivial application. Any programming language intended for concurrency must have some coordination constructs. Some languages (as the two cited in the resume of sections 1 and 2 above) take a purist stand and insist on making all coorodination message based or making it only via shared objects.

A challenge for the purely message based approach is to handle selection between messages from many sources. In the stop-watch it looked like this:

```
receiveMessage(m);
  swich m {
    start: running= true;
    stop:  running= false;
    reset: write(0); t= 0; running= false;
    tick:  if (running) write(t++);
  }
```

On the other hand, using shared objects, one needs to ensure atomicy when several streams operate on the same object. This will inherently force streams to wait for access with the possibility of decreased

performance.