Concurrency, Part 1

by Raúl Pardo Jimenez raup@itu.dk and Jørgen Staunstrup, jst@itu.dk, version Sept 1, 2022 The note starts by taking a helicopter view on concurrency where we ignore the details of a concrete programming language. Instead, the focus is qualitative, identifying a few core concepts that are fundamental for using concurrency.

Programming for concurrency is based on a number of concepts and abstractions used for writing software with multiple independent streams of statements/instructions[1]. This may be contrasted with sequential programming which deals with only a single stream of statements. In this note, we use the term *stream*[2] to denote software component with a single stream of statements e.g.

```
stream t = new stream(() -> {
  s1;s2;s3; ...
});
```

The body of a stream may contain all the well-known programming concepts e.g. loops, if-statements, variable declarations, functions etc. Almost anything, one would write in the `main method` of a Java program.

The syntax in this note is close to Java's syntax, but some of the code below, is not 100 % valid Java.

In this note, we consider concurrency an abstraction that has a number of concrete manifestations in programming languages, libraries, protocols and many other places including human interaction. This is similar to other abstractions e.g. an animal which has a number of properties common to a lot of living beings (a cat, a spider, a humming bird, etc.). Such abstractions are a common tool in almost all sciences (and many other places).

Programming languages for concurrency has not yet settled on a common set of concepts. This is in contrast to programming concepts such as functions/methods and parameters which are roughly the same in most programming languages.

Classification of concurrency

Most often concurrency concepts are discussed with a strong emphasis on performance. This is natural because concurrency is often used to improve performance. However, the performance bias often introduce extra complexity. For example, the interplay of the memory model and locking mechanisms in Java is quite complex. In this note, we focus on introducing a few abstract concurrency concepts ignoring the complexity of their efficient implementation.

The first such term is *stream* that we use as an abstraction for discussing the following three fundamental motivations for concurrency:

- User interfaces and other kinds of input/output *(Inherent)*.
- Hardware capable of simultaneously executing multiple streams of statements (*Exploitation*), a special (but important) case is communication and coordination of independent computers on the internet.
- Enabling several programs to share some resources in a manner where each can act as if they had sole ownership (*Hidden*)[3].

Concurrency programming has a number of additional challenges compared to sequential programming; first and foremost handling *coordination* when the independent streams need to collaborate, compete for resources or exchange information. The term *coordination* is used here as an abstraction of more low level terms like synchronization, locking, remote procedure calls, ... that one typically finds in writings about concurrency.

Concurrency and *nondeterminism* are two related but different concepts. Concurrent programs may certainly behave nondeterministicaly, for example, when different streams share variables and the result of the compu-

[1] stream implies that there is an ordering in the execution of the statements.
[2] you may think of a stream as a thread with no overhead. Not to be confused with the Java data type `Stream`.
[3] These terms were coined by the Norwegian computer scientist Kristen Nygaard.

tation differs because of the detailed timing of the statements in the streams. However, a nondeterministic computation may be completely sequential if a statement for throwing a dice (coin, random number, ...) is introduced.

The greatest challenge in concurrency programming is getting the streams to coordinate reliably in the face of this nondeterminism. Therefore, it is important not to prematurely focus on performance. This will only complicate getting the coordination correct.

A brief history of concurrency

The very first computers for general purpose were only meant for running one program at a time. However, they gradually became powerful enough that a small number of users could share the computer. The computer itself could still only execute a single sequence of commands/instructions. However, users had the illusion that they had the computer for themselves. This was called *timesharing* (and is analogous to the concept of timesharing an apartment). Today, we have the somewhat opposite situation where the hardware in a laptop or smart-phone is capable of running several streams simultaneously, because it has a small number of processors called *cores*.

In the meantime, the focus of concurrency programming has shifted several times and this has led to the terminology being somewhat confusing. For example, the first programming languages used the term *process* to denote a single independent stream of statements/instructions.

Going into more details with the three types of concurrency.

The three fundamental types of concurrency described above are not always viewed as instances of a more general concept. The thesis of this note is *that there are some common abstractions covering all three*. Performance details are set aside (because they often introduce a lot of low-level considerations). Performance may of course turn out to be important, but this is certainly not always the key challenge. Finding the right abstract program structure is often the key to writing good software. Historically, the importance of the fundamental motivations for concurrency has changed over time, and hence a number of programming language constructs introduced over the years have only addressed one or two of them well.

Hardware capable of simultaneously executing multiple streams of statements (Exploitation)

Today, most laptops, smart phones and servers have hardware capable of executing several streams of instructions. These are called multi-core computers and each core is an independent processor. To take advantage of multiple cores the software must be written in such a way that independent streams of statements can be separated and directed to the different cores/processors. In Java, this can be done by making each independent stream of statements a thread.

Example Below is an example of code where concurrency multiple streams are used to speed up a computation (running on a computer with multiple cores). Three streams collaborate on counting the number of primes less than 3 million.

```
stream t1= new stream(() -> {
  for (int i=0; i<999999; i++)
    if (isPrime(i)) counter.increment();
});

stream t2= new stream(() -> {
  for (int i=1000000; i<1999999; i++)
    if (isPrime(i)) counter.increment();
});

stream t3= new stream(() -> {
  for (int i=2000000; i<2999999; i++)
```

```
    if (isPrime(i)) counter.increment();
});
```

Interacting with the environment (Inherent)

Almost any computer needs to communicate with its environment, e.g. to get input or to show results. In sequential programs this may be done with statements like `read` or `write`:

```
n= read();
res= computeNoOfPrimess(n);
write(res);
```

However, this will force the computer executing the code to stop and wait for the user to provide input (and for the output to complete). This is not convenient, if there are many independent communication channels e.g. multiple buttons and text fields on a webpage, or on a smart-phone. Similarly, for a computer embedded in a robot that needs to listen for input from many different sensors to navigate safely.

So whenever there is an independent stream of external events that needs the computers attention, there should be an independent stream in its program handling the stream of events.

```
//Code for a user interface
stream searchField= new stream(() -> {
  newText= await(searchField);
      search(newText);
  });

stream stopApp= new stream(() -> {
      await(stopButton);
      exitApp;
  });
```

Resource sharing (Hidden)

Many applications may be active on a smartphone or computer, but most of the time they are idle waiting for something to happen e.g. an e-mail arriving, download finishing, alarm clock activated etc. Therefore, the underlying operating system will typically have many streams executing on the same processor (core). Since each of the streams is idle (waiting) most of the time, none of them will "notice" that other streams are running on the same processor.

```
//e-mail app

stream Email= new stream(() -> {
  do {
    await(email);
    notify(user);
    store(email_in_inbox);
  } forever
})

// Stopwatch app
stream stopWatch= new stream(() -> {
  do {
    alarmAt= await(start button);
    waitUntil(alarmAt);
    notify user;
  } forever
})
```

Interleavings

The key of our concurrency definition is "... multiple independent streams of statements/instructions". Now, consider a setup where we observe what is happening by adding two print statements.

Example Consider these two streams of statements (in Java like syntax):

```
int c= 0;print c;

stream t1=new stream(() -> { c= c+1;print c;s2;s3;... });
stream t2=new stream(() -> { s4;s5;c= c+1;print c;... });
```

What values of `c` are printed? Certainly 0 which is the initial value of the variable `c`. But what about:

```
0, 1, 2
0, 1, 1
0, 2, 2
```

The statements of the two streams are executed concurrently. This means that we cannot assume anything about when statements in one stream are executed in relation to execution of statements in the other stream. One would expect that this output (from the print statements) is possible: `0, 1, 2` because `c= c+1` is the first statements in sequence `t1`(incrementing `c` from 0 to 1). In stream `t2` the two statements `s4` and `s5` has to be executed before it increments `c`. A plausible assumption could be that `t1`has finished its increment long before `t2` has finished `s4` and `s5`. So, one may observe: `0, 1, 2`. However, the key part of our definition of concurrency is that the streams are *independent*, i.e., that no assumptions can be made about when the statements in the two streams are executed relatively to each other.

Therefore, another possible output of the two streams could be: `0, 1, 1`. This would happen if stream `t2` finished statements `s4, s5` very quickly so the two streams (almost) simultaneously read the initial value of `c` (0) then each stream does an increment from `0` to `1` and then both store the value `1` in `c`. One may think that this is a very contrived example that would never happen. **But it does.** In reality, the streams could be executed on two different computers connected by a network, on a modern laptop that have several cores (hardware capable of executing code simultaneously), or for a number of other reasons that you will learn about in this course.

Question: Would it also be possible that the program above prints: `0, 2, 2`?

The discussion above is based on a single observer (printing). What is observed is one (of many possible) *interleavings* of the statements from the different streams. A key challenge when programming with concurrency is that there are many possible interleavings that should all give results consistent with our specification of what the program is supposed to do. Below we go into more details with this challenge, before doing that we will look at different motivations for concurrency.

Syntax for interleavings

In order to have a common language for interleavings, we use the following syntax.

```
<stream>(<step>)
```

For instance, the interleaving showing that the program above prints `0,1,2` would be written as (assuming a `main` stream which executes the first two statements)

```
main(int c=0), main(print c), t1(c=c+1), t1(print c), t2(s4), t2(s5), t2(c=c+1), t2(print c), ...
```

To avoid writing program statements, it is useful to assign numbers to the steps of a program. Consider, the following rewriting of the programs above

```
int c= 0; // (0)
print c;  // (1)

stream t1=new stream(() -> {
```

```
    c= c+1;        // (0)
    print c;s2;s3; // (1)
    ...
});
stream t2=new stream(() -> {
    s4;      // (0)
    s5;      // (1)
    c= c+1;  // (2)
    print c; // (3)
    ...
});
```

Note that we use comments at the end of each line to give a number to each statement within a stream. Now, using this numbering, the interleaving printing `0,1,2` would be written as

`main(0), main(1), t1(0), t1(1), t2(0), t2(1), t2(2), t2(3), ...`

As we will see in the lectures, this syntax will be handy when reasoning about the correctness of concurrent programs.

Safety

The lack of synchronization in a concurrent program may lead to different results each time the program is executed. Recall the example we discussed in section interleavings. We saw a simple program with several interleavings (possible execution orders for the statements in the streams) that produces a different result for each interleaving.

Since we do not have control over the non-determinism that produces the different interleavings, it is hard to understand the behavior of concurrent programs. The uncontrolled interaction between streams may lead to undesired results.

Intuitively, in concurrency programming we say that a program is *safe* when the interaction between streams produces *only* the expected result. To illustrate this idea, consider a slightly modified version of the program in interleavings:

```
int c= 0;

stream t1=new stream(() -> { c= c+1; });
stream t2=new stream(() -> { c= c+1; });

print c;
```

Suppose the goal of this program is to increment `c` twice. In other words, the expected output of this program is to print 2. Is this concurrent program safe? As we saw earlier, the answer is no. The program may print 2, but it may also print 1 or even 0! We will go in greater depth about how these outputs are possible during the course. But, for now, simply remember that depending on the interleaving the output of the program may vary.

Producing different results depending on interleavings is well-known in concurrency programming, and it is known as a *race condition*. Precisely, a race condition is defined as:

> *A* **race condition** *occurs when the result of the computation depends on the interleavings of the operations*

Stream-safety

Now we are ready to precisely introduce our definition of *stream-safety* [4]

[4] In the course we will talk about threads (the Java instantiation of streams), so we will study *thread-safety*.

> *A* **program** *is said to be* **stream-safe** *if and only if no concurrent execution of its statements result in race conditions*

Checking that a program is stream-safe is a very difficult (or impossible in some cases) task. In the worst case, this problem requires exploring an infinite number of interleavings that a concurrent program may produce. During the course we will study techniques to test and formally verify stream-safety. However, checking whether a program is *not* stream-safe is easier task. We only need to find two interleavings that prodcue different outputs. In the example above, we found three interleavings producing 3 different outputs (0, 1 and 2).

Not all race conditions lead to undesired behaviour

Race conditions are a potential source of non-safety in concurrent programs, but note that not all programs containing race conditions are unsafe. Consider yet another modification of the program above to illustrate this

```
print "The shop is open"

stream t1=new stream(() -> { print "Jørgen entered the shop" });
stream t2=new stream(() -> { print "Raúl entered the shop" });
```

Suppose that this program simply needs to print `"The shop is open"` and later whether Jørgen or Raúl enter the shop. Here there are two possible interleavings:

```
1. "The shop is open" "Jørgen entered the shop" "Raúl entered the shop"
2. "The shop is open" "Raúl entered the shop" "Jørgen entered the shop"
```

Both interleavings produce correct results, but there is a race condition that determines the result.

This discussion hints that our definition of stream-safe programs is too strong. As we may be classifying programs that behave correctly as non-safe, even though the exhibit correct behavior. The key concept here is the meaning of *"correct behavior"*. So far we have only informally discussed correct behavior. The following section makes this notion precise.

Specifications

Note that all our examples above refer to the *"correct result"*. In general, in programming, and specifically in concurrency programming, it is of upmost importance to precisely define what is the correct/desired/expected result of a program. This way, we can precisely determine whether our concurrent program is safe.

A statement precisely defining the correct/desired/expected behavior of a program is typically referred to as its *specification*. We have seen examples of specifications in the programs above: *the program must increment* `c` *twice*, *the program must print whether Jørgen or Raúl enter the shop*. Other examples could be, *the program must output a sorted array*, *the program must output the number of primes in the range (x,y) and* after calling `q.queue(e)`, `e` must be added to the `q`*.

Providing a specification allow us to relax our definition of stream-safety as follows:

> *A* **program** *is said to be* **stream-safe** *with respect to a* **specification** *if and only if all concurrent execution of its statements satisfy the specification*

Note, that this new definition allows for race conditions which satisfy the specification.

Why two definitions of stream-safety? In the lack of specification, ensuring that the program has no race conditions helps, at least, to increase our confidence that concurrency bugs related to the order of execution of the streams are not introducing errors.

In practice, many developers talk about stream-safety without being explicit about the specification of their programs. This is a very bad practice, you should always make explicit and precise the specification of your programs (even for sequential ones). Furthermore, specifications will enormously help to test and understand your programs.

Stream coordination

If streams were completely independent, they would be of limited use. As with people, most of the time they go about their daily business independently of other people. However, every now and then they need to coordinate their activities, e.g. share some information, enter private areas (like a toilet), compete for limited resources etc. Similarly, streams also need to coordinate. This has led to the introduction of a number of programming concepts allowing streams to interact. As with the terminology for concurrency in general, the terminology used for coordination is often confusing and inconsistent.

There are two main types of coordination of streams in software: *sharing information* and *exchanging information*. These concepts can be also found in human coordination. An example of sharing information would be a bulletin board. Sending and receiving messages/letters is an example exchanging information.

For computers, there is also a very physical manifestation of the two types of coordination. In some cases, computers may physically share some or all of their memory. In other cases, they may be connected with a network, where information can only be exchanged via some form of communication link.

There have been numerous attempts to introduce programming languages strictly focusing on one of the two types of coordination. For example, Concurrent Pascal was an early object-oriented language where all coordination had to be done through shared objects. Other languages/concepts for concurrency were based on message passing, as the only way for streams to coordinate. An example would be the Http protocol.

Theoretically, sharing objects and message passing are equally powerful. One can always simulate one of them with the other. Therefore, the more general term *coordination* is used here.