Concurrency I

by Raúl Pardo (raup@itu.dk) and Jørgen Staunstrup (jst@itu.dk)

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 when programming for concurrency.

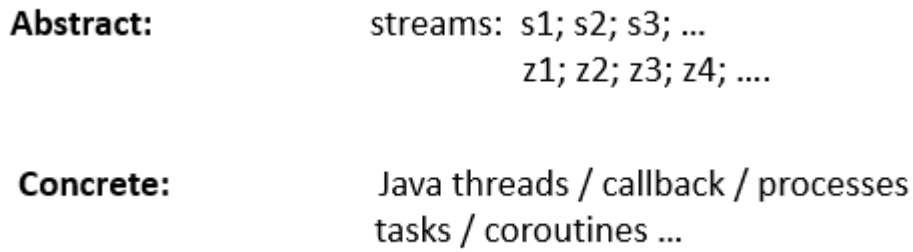
Programming for concurrency is based on a number of concepts and abstractions used for writing software with multiple independent streams of statements/instructions[1]. This may be contrasted with sequential programming which deals with only a single stream of statements/instructions. In this note, we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

```
stream t = {
  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 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 hummingbird, etc.). Such abstractions are a common tool in almost all sciences (and many other places).



Programming languages for concurrency have not yet settled on a common/standardized 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. In the section "Concurrency concepts" we present and discuss some of the many ways streams and stream coordination are realized in various programming languages. Except for that section, we focus on Java in these notes.

In the rest of these notes we will use the term *concurrent program* for code that have two or more independent streams.

Classification of concurrency

Most often concurrency concepts are discussed with a strong emphasis on performance. This is natural because concurrency is often utilized to improve performance. However, the performance bias often

[1] stream implies that there is an ordering in the execution of the statements. In Java there is an interface "stream" which has nothing to do with our use of the term stream in these notes.

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*)[2].

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, message passing … that one typically finds in writings about concurrency.

Concurrency and *nondeterminism* are two related but different concepts. Concurrent programs may behave nondeterministicaly, for example, when different streams share variables, and the result of the computation depends on the detailed timing of the statements in the streams. However, a nondeterministic computation may be completely sequential e.g. 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 and terminology

The very first computers for general purpose were only meant for running one program (stream) 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, because the computer switched its attention between the users. This was called *timesharing* (analogous to the concept of timesharing an apartment) and addressed the exploitation motivations. 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* that can each run a stream.

To write the software for supporting timesharing (exploitation) the first concurrency programming concepts evolved. They made it possible to write programs with several streams (at that time called tasks or processes). The streams also needed mechanisms for coordinating their computations e.g. to access external devices (such as the printer) or exchange data. Almost immediately, these mechanisms came in two different flavors: message passing and shared memory. These two approaches are discussed in the section Stream coordination of these notes.

The focus of concurrency programming has shifted several times since the early days of timesharing. For example, the use of computers for simulation led to the programming language SIMULA67. Simulations typically have a (large) number of streams each representing some activity in the real world: people, physical phenomenon like the weather, productions lines in a factory etc. SIMULA67 has a stream concept used for describing such activities: co-routines. This concept has since been re-invented several

[2]These terms were coined by the Norwegian computer scientist Kristen Nygaard.

times e.g. in the Kotlin programming language that is now the the primary language for Android development.

Very early, computers became connected in a network and used for telecommunication. The interaction between the computers was achieved via. telecommunication lines and hence message passing was an obvious coordination mechanism. This led to programming languages with constructs for sending and receiving messages. The Internet protocols introduced at that time are also based on message passing. These (message-passing) protocols are still the foundation of the Internet.

Over the years, many programming languages supporting concurrency have been introduced often reinventing the fundamental concepts: streams and coordination.

Hardware details relevant for concurrency

Modern hardware have several features that supports and exploits concurrency. First and foremost the CPU may have several processors that can execute code independently of each other. These processors are most often called cores. The CPU of a modern laptop typically have from 4 (and upwards) such cores. This is how you may detect the number of cores on your computer:

linux: `nproc`

windows: Open task manager (Ctrl + Shift + Esc) and select Performance tab

Mac: Hit Command+Spacebar to open Spotlight, then type "System Information" and hit return (or type "System Profiler" for earlier MacOS versions) Click "Hardware" to see the Hardware Overview, and find the "Total Number of Cores"

Most of the material in these notes (Concurrency Notes) focus on how to take advantage of such multi-core CPU's. However, the organization of the computers memory may also influence the running time of your code.

Memory hierarchy

This section gives a short overview of the most important aspects of the memory in modern hardware.

This is an illustration of the memory components (caches) of an Intel i7-4870HQ CPU. The L1 and L2 caches are local to a core; the L3 cache and RAM are shared between cores.

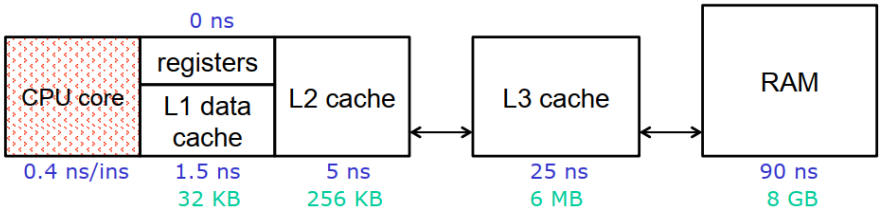


Figure 1: Memory hierarchy example. Copied from A multicore performance mystery solved by Peter Sestoft (sestoft@itu.dk)

The numbers (in blue) indicates the time it takes to fetch (or store) data in the different parts of the memory hierarchy. These numbers illustrate that it is more than an order of magnitude faster to reach memory in the local memory of a core compared to reaching memory in the RAM.

The disadvantage of having data in the local memory of a core is that changes are not visible from other cores.

Fortunately, programming languages include mechanisms to control where variables are placed in the memory hierarchy. For example, in Java, the modifier `volatile` ensures (among other things) that variables are stored in memory shared by all cores (and hence variable writes performed by one core are visible by the other cores).

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 concurrency)

Today, most laptops, smart phones and servers have hardware capable of executing several streams of instructions simultaneously. 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*.

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= {
  for (int i=0; i<999999; i++)
    if (isPrime(i)) counter.increment();
};

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

stream t3= {
  for (int i=2000000; i<2999999; i++)
    if (isPrime(i)) counter.increment();
};
```

Interacting with the environment (Inherent concurrency)

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);
  ...
```

However, this will force the computer executing the code to stop when reaching the `read()` and wait for the user to provide input. 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 computer's attention, there *should be an independent stream in its program* handling the stream of events.

To illustrate inherent concurrency, consider an it-system to keep track of the number of visitors to Tivoli. At each entrance there is a turnstile:

Each of these turnstiles contains some electronics (probably a small computer) that signals whenever a visitor goes through the turnstile.

```
stream turnStile1= {
  await(visitor); // signalling a visitor at turnstile 1
  visitors1= visitors1 + 1;
};

stream turnStile2= {
  await(visitor); // signalling a visitor at turnstile 2
  visitors2= visitors2 + 1;
);

stream turnStile3= {
  await(visitor); // signalling a visitor at turnstile 3
  visitors3= visitors3 + 1;
};
```

The details of how the value of the three counters are communicated to the central computer is not important here (however we return to this later in the section on message passing). The key point is that the arrival of visitors to the three turnstiles are independent events and hence require a separate stream.

Resource sharing (Hidden concurrency)

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 while they are waiting.

```
//e-mail app

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

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

Programming for Concurrency

The stream concept used above can be implemented in most commonly used programming languages. In this section we briefly discuss a few of these.

Java

Most of the code presented in these nodes will be in Java where the stream:

```
stream t = {
  s1;s2;s3; ...
};
```

can be implemented like this:

```
new Thread() {
  @Override
  public void run() {
    s1;s2;s3; ...
  };
}.start();
```

There is a lot more to concurrency in Java than this some of which will be covered in later sections.

Java programs are run by a virtual machine called JVM(more details in section …). When many threads are created the JVM ensures that all threads can progress.

Kotlin (coroutines)

Kotlin is one among many languages where the switch between running one stream and another must be explicitly stated in the code by calling a yield function which stops the stream and allow another stream to run:

```
stream t = {
  s1; yield(); s2;s3; yield(); s4;  .... yield(); ....
};
```

Such streams are called coroutines. It was actually among the first constructs to implement concurrency e.g. in Simula67[…]

Callbacks

JavaScript

More to come

Realtime

All of the above mechanisms for ensuring that all streams progress have the drawback that it is difficult/impossible to ensure realtime constraints such as: "An answer must be give within xxx milliseconds". Such constraints are for example important in code controlling external devices e.g. a robot see e.g. https://en.wikipedia.org/wiki/Real-time_Java

Realtime programming is not discussed in these notes.

Interleavings

The key to 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= { c= c+1;print(c);s2;s3;... };
stream t2= { s4;s5;c= c+1;print(c);... };
```

What values of `c` are printed? Certainly 0 which is the initial value of the variable `c` that is printed before the two streams start. 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` must 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 or 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 the specification of what the program is supposed to do. Below we go into more details with this challenge.

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= {
    c= c+1;        // (0)
    print c;s2;s3; // (1)
    ...
};
stream t2= {
    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), ...>`

This syntax will be handy when reasoning about the behavior of concurrent programs.

Java Scheduler

Consider again the Tivoli turnstile example. To avoid using threads, one may code it like this:

```
do {
  read(turnstile, 1);
  visitors= visitors +1;

  read(turnstile, 2);
  visitors= visitors +1;

  read(turnstile, 3);
```

```
  visitors= visitors +1;
} while (true);
```

There are several reasons why this is (very) bad code, but is does illustrate the overall task of the central computer.

First of all, in each iteration the loop shown above blocks when reading from the first turnstile. This means that the subsequent reads from the two other turnstiles will be blocked.

Secondly the doloop is a "busy waiting" loop. This considered a very bad programming practice because it may block any other programs running on the computer.

In Java, the solution to preventing both problems is to introduce *threads*. Using threads, the code for handling one turnstile could look like this:

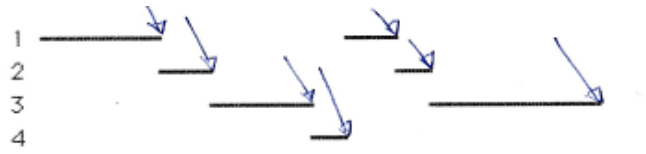
```
new Thread() {
  @Override
  public void run() {
    read(turnstile, 1);
    visitors= visitors +1;
  };
}.start();
```

To complete the program two more threads handling turnsstile 2 and 3 must be started. For a more thorough introduction to Java threads see https://www.geeksforgeeks.org/java-threads/ (or one of the many other tutorials on Java threads).

Java code is executed by the JVM (Java Virtual Machine) which transforms the Java code to the underlying hardware. For example, the statement `visitors= visitors +1` is transformed to at least three hardware instructions: (1) reading the value of `visitors` from memory, (2) incrementing it, and storing the value of `visitors` back into memory.

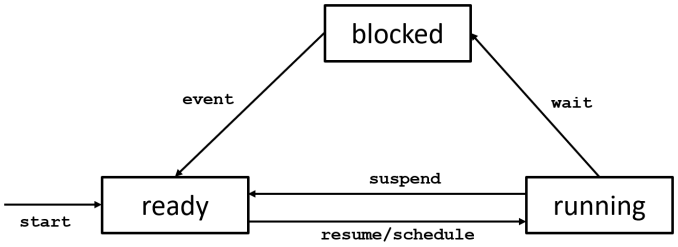
Another task of the JVM is to start, stop and delay threads. A Java program may have many threads and the JVM interleaves the code in these threads allowing all of them to progress even if the underlying hardware only executes one instruction at a time.

The solid lines in this diagram illustrates time periods where a thread progresses. Such a time period is ended by the scheduler (the small arrows on the diagram). When this happens the scheduler allows another thread to progress.



The time periods when a threads progresses is called a *time-slice*. These need not be of the same length and the scheduler may decide in which order the time-slices are executed; but it is the responsibility of the scheduler that *all threads progress*.

When a thread is started, the scheduler (in the JVM) will register it as started (in internal data structure). At some (unpredictable) time the scheduler is moved to the running state. In this state, the hardware executes the code of the thread (e.g. incrementing `visitors`). The scheduler may at any time move the thread back to the ready state and move some other thread to the running state.



When a running thread executes a statement like `read`, it may have to wait for some event (external to the thread). If this is the case, the scheduler moves the thread to the blocked state. When in the blocked state, the thread cannot progress. At some point (e.g. when the read blocking the thread has occurred) the scheduler moves the thread back to the ready state (from which it can progress).

For more details on the Java scheduler see section 20 of Sestoft: Java Precisely.

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 (fully) 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= { c= c+1; };
stream t2= { 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* [3]

[3] 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 an easier task. We only need to find two interleavings that produce 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 behavior

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= { print("Jørgen entered the shop") };
stream t2= { 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 they 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.

Syntax for concurrent executions, sequential executions and linearizations

This section introduces techniques for reasoning/analyzing concurrent executions. This can be challenging because different executions of the same program may behave differently. Therefore, the techniques introduced here have the overall goal of *systematic reasoning* and hence to *increase the confidence* that a given program behaves as expected.

But first we introduce concepts and notation that can be used for *reasoning/analysis of concurrent executions.* Although the concepts and notations may be used for formal/mathematical proofs, here they are used more informally (but supplied with references to the formal theory behind our approach).

First, we will introduce the notations/syntax we use: *concurrent executions, sequential executions and linearizations.* This is done by example (rather than rigorous formal definitions) .

Concurrent executions

Consider a program with two streams `A` and `B` that both access two shared objects `p` and `q`. Below is an example of a concurrent execution of this program:

```
A: -|  q.enq(x)  |-------|  p.enq(y)  |---->
B: -----|  q.deq(x)  |------|  p.enq(y)  |->
```

Time is represented horizontally, either as —- (nothing happens) or as | ooooo | some action (ooooo) is excuted. The text in between `|  |` denotes method calls. The left `|` denotes the point in time when the method was invoked. The right `|` denotes the point in time when the response of the method call is received (in other words, when the method call finished). The width of `|  |` denotes the duration of the method call.

In the execution shown above,stream `A` first inserts `x` in `q` and then `y` in `p`. Concurrently, stream `B` retrieves `x` from `q` and then `y` from `p`.

This notation was introduced in: Maurice P. Herlihy and Jeannette M. Wing. ….

Sequential mapping

An concurrent execution can be mapped (projected) to a (number of) sequential executions e.g.

```
S: -| q.enq(x) |----| p.enq(y)  |----| q.deq(x) |------|  p.enq(y)  |->
```

or

```
T: -| q.enq(x) |----| q.deq(x)  |----| p.enq(y) |------|  q.deq(y)  |->
```

These sequential execution can also be written using the interleaving notation introduced above:

```
<q.enq(x),p.enq(y),q.deq(x),p.deq(y)>
```

In a sequential execution, the sequence of method calls are listed in the (sequential) order of execution. Recall that real-time is irrelevant for sequential executions. We are only interested on whether a method call happens before another.

Sequential consistency

In sequential programs, the compiler and CPU are allowed to re-order instructions as long as they produce a valid result (according to the specification of the object).

(JSt comment): I find this sentence confusing: The the compiler and CPU has no knowledge of the specification. Should it be something like: the compiler and CPU are allowed to re-order instructions as long as the result is not inconsistent with the semantics of the programming language) ????

Hence, these orderings are all potentially possible:

```
<q.enq(x),p.enq(y),q.deq(x),p.deq(y)>
<p.enq(y),q.enq(x),p.deq(y),q.deq(x)>
<q.enq(x),p.deq(y),q.deq(x),p.enq(y)>
```

However, the last interleaving is problematic because `p.deq(y)` appears before `p.enq(y)` . This is inconsistent with the sequential specification of the behavior of a queue. Therefore, we require that all concurrent executions are sequentially consistent.

Definition

For executions of concurrent objects, an execution is sequential consistency iff:

1. Method calls appear to happen in a one-at-a-time, sequential order
2. Method calls should appear to take effect in program order

Linearization

Linearization is a rigorous method for reasoning about concurrent behavior (e.g. of an object) of code that does not use locks or `synchronized` to ensure atomicity. Here, we focus on code using CAS (compare and swap) operations to coordinate the execution of several streams. The linearization method is based on the assumption that there instead of considering all possible interleavings, we need only consider all interleavings of the CAS operations. For example:

```
public T dequeue() {
  while (true) {
    Node<T> first = head.get();
    Node<T> last = tail.get();
    Node<T> next = first.next.get(); // D3
    if (first == head.get()) { // D5
      if (first == last) { // D6
        if (next == null) // D7
          return null;
        else
          tail.compareAndSet(last, next);
      } else {
        T result = next.item;
        if (head.compareAndSet(first, next)) // D13
          return result;
      }
    }
  }
}
```

Linearizability extends sequential consistency by requiring that each method call appears to take effect instantaneously at some moment, called the linearization point, between its invocation and response.

The statements D3 and D13 defines the instant in the execution when the method call "takes effect" and hence may cause inconsistencies, they are *the linearization points* of this method. Instead of considering interleavings of all statements (D1-D13) with all interleavings of other parts of the program, we only consider interleavings of D3 and D13 with linearization points in the remaining part of the code. Doing this, we will have considerable fewer interleavings to analyze.

The linearization method is based on first coming up with a list of possible interleavings (of linearization points) and secondly rigorous arguments why all of these interleavings produce a correct result. **Note that "a list of possible interleavings" is not the same as a formal proof.**

Linearizability extends sequential consistency by requiring that each method call appears to take effect instantaneously at the linearization point(s). In the sequential mappings below the linearization point are marked with * .

As an example, consider this execution:

```
-| * q.enq(x)  |------|  * p.enq(y)  |---->
- |    q.deq(x) *|-----|  p.deq(y) *  |->
```

The linearization points are marked with `*`. It can be mapped to a sequential execution (the last line):

```
-| * q.enq(x)  |------|  * p.enq(y)  |---->
- |:   q.deq(x) *|-----| : p.deq(y) *  |->
   *            *        *          *
```

Which gives this linearization:

```
<q.enq(x),q.deq(x),p.endq(y),p.deq(y)>
```

This is consistent with our specification.There are many other possible linearizations e.g.

```
<q.enq(x),p.enq(y),q.deq(x),p.deq(y)>
```

For each of these we must argue that it is consistent with the specification. Note that

```
<q.enq(x),p.enq(y),q.deq(x),p.deq(y)>
```

is not a possible linearization because it is not consistent with the real time ordering where `q.deq(x)` is finished before `p.enq(y)` is started.

Finally, consider this execution:

```
-|  q.enq(x)  |------| q.deq(y)|---->
------------|  q.enq(y) |----------->
```

This is not linearizable, because `q` is a FIFO queue, hence the first `deq` operation must return `x` which is the first element enqueued.

Examples. In this section we define and exemplify objects that are linearizable. In the previous section linearizability was presented as a property of executions which is the basis of the definition of *Linearizable Concurrent Objects*.

A concurrent object is linearizable iff:

1. All executions are linearizable, and
2. All linearizations satisfy the sequential specification of the object

Most often proving the first property formally is hard. Here we focus on a more pragmatic approach where we argue (informally) about the behavior of the linearizations. More precisely:

To argue that an object is linearizable we identify all CAS operations (assuming these are the relevant linearization points). We then argue that all possible sequences of the CAS operations satisfy the sequential specification of the object.

Below are two examples of Java classes based on CAS operations. Both of these are analyzed using the linearization technique introduced above.

The first example is an implementaion of a FIFO queue called `WaitFreeQueue` (that turns out to not be linearizable) whereas the second implementation called `MSQueue` is linearizable).

WaitFreeQueue Here is the Java code for the `WaitFreeQueue`.

```
class WaitFreeQueue<T> {
  int head = 0, tail = 0;
  T[] items;

  public WaitFreeQueue(int capacity) {
    items = (T[]) new Object[capacity]
  }

  public void enq(T x) throws Exception {
    if (tail - head == items.length) // E1
      thrown new Exception(); // E2
    items[tail % items.length] = x; // E3
    tail++; // E4
  }

  public void deq() throws Exception {
    if (tail - head == 0)  // D1
      thrown new Exception(); // D2
    T x  = items[head % items.length]; // D3
    head++; // D4
    return x;
  }
}
```

The linearization points for `enqueue` are: - E4 is the point when an enqueue is completed if the queue is not full - E1 is the point when an enqueue is completed if the queue is full

and for `dequeue`: - D4 is the point when a dequeue is completed if the queue is not empty - D1 is the point when a dequeue is completed when queue is empty

Now consider this interleaving: E1,E3,E1,E3,E4,E4 . Here E1,E3 are done twice before incrementing `tail` hence the second execution of `E3` overwrite the first. Hence, this linearization is possible:

`<q.enq(x),p.endq(y),q.deq(y)>`

This violates the specification of a FIFO queue !!! so the `WaitFreeQueue` is *not* linearizable.

MSQueue The MSQueue is (yet another) implementation of a FIFO queue. Here is the Java code for the enqueue/dequeue methods.

```
public void enqueue(T item) {
  Node<T> node = new Node<T>(item, null);
  while (true) {
    Node<T> last = tail.get();
```

```
    Node<T> next = last.next.get();
    if (last == tail.get()) { // E7
       if (next == null)  { // E8
         // In quiescent state, try inserting new node
         if (last.next.compareAndSet(next, node)) { // E9
        // Insertion succeeded, try advancing tail
         tail.compareAndSet(last, node);
         return;
          }
       } else
       // Queue in intermediate state, advance tail
       tail.compareAndSet(last, next);
    }
  }
}

public T dequeue() {
  while (true) {
    Node<T> first = head.get();
      Node<T> last = tail.get();
    Node<T> next = first.next.get(); // D3
      if (first == head.get()) { // D5
        if (first == last) { // D6
          if (next == null) // D7
          return null;
        else
          tail.compareAndSet(last, next);
      } else {
        T result = next.item;
        if (head.compareAndSet(first, next)) // D13
          return result;
      }
    }
  }
}
```

Enqueue has one linearization point: - E9 – if successfully executed, the element has been enqueued

Dequeue has two linearization points - D3 - if the queue is empty. After its execution, the evaluation of D7 is determined and whether the method will return null. - D13 - if successfully executed, the element has been dequeued

Below we argue that all possible sequences of the CAS operations satisfy the sequential specification of the object.

- If two streams execute enqueue concurrently before tail and next are updated, then only one of them succeeds in executing E9 (and possible update the tail). The other fails and repeats the enqueuing.
- If a stream executes enqueue after another stream updated the tail, then E7 fails and it repeats the enqueue.
- If a stream executes enqueue after another stream updated next, then E8 fails, the stream tries to advance the tail, and it restarts the enqueue.

- If two streams execute dequeue concurrently before the head is updated (D5 succeeds for both) and the queue is not empty (D6 fails), then D13 succeeds for only one of them. The other restarts the dequeue.
- If a stream executes dequeue after another stream updated the head, then D5 fails and it restarts the dequeue
- If a stream execute dequeue while another stream executed enqueue (E9) and before the enqueuing stream updates the tail, then D7 fails, the dequeuing stream tries to update the tail and restarts the dequeue.

Based on this, we can conclude that all linearizations MSQueue satisfy the specification of a FIFO queue.

Linearizability is a compositional property i.e.

- If two objects are linearizable, any concurrent execution involving the two objects remains linearizable

This means that reasoning can be split into single objects that can then be safely composed.