Australian
National
University

SYNCHRONIZED DATA

Week 9 Laboratory for Concurrent and Distributed Systems

Uwe R. Zimmer

Pre-Laboratory Checklist

- ❑ **You have read this text before you come to your lab session.**
- ❑ **You understand and can utilize message passing.**
- ❑ **You have a firm understanding of memory based synchronization.**
- ❑ **You understand and can apply implicit concurrency.**
- ❑ **You can create and control tasks.**

Objectives

This lab will focus on another approach to concurrent system design, namely synchronized access to or atomic operations on primitive types like Boolean, Integer or Float. You will learn about the challenges and benefits of this approach.

Interlude: Explicit concurrency

During last lab you learnt about ways to program concurrently without ever mentioning tasks. As powerful as this is, there are also problems which require synchronized access to data and explicit control of concurrent entities. From your Ada background you know already ways to declare a data structure as protected and to coordinate all access to such a data structure. Any write access becomes automatically mutual exclusive to any other write access and you are on safe grounds.

The question is though which data structure do you protect with a dedicated lock? More fine grained locking implies a larger number of locks and a potentially more complex form of coordination. More coarse grained locking might mean that some parts of the data structure might become locked unnecessarily. Let’s take it to the extreme and find out what happens if we add a lock to every scalar entity.

Atomic operations

You remember that atomic operations are provided by the hardware for certain cases, e.g. **atomic test-and-set**. In your programming languages so far you could of course define a synchronization mechanism on any level and define an atomic test-and-set operation on e.g `Integers` or `Floats`. If we want to work with fine grained locks in this lab, then it would be useful though if some of those operations would already be predefined. And indeed Chapel (similar to C++ since version C++11) provides such atomic operations.

If you define for instance:

```
No_of_data_points : atomic uint;
```

you can then no longer access this variable with standard assignments, yet instead you gain a set of atomic operations — among the most commonly used are[1]:

```
proc (atomic t).read (): t
proc (atomic t).write (v: t)

proc (atomic t).peek (): t  // relaxed memory order - do not use to sync tasks!
proc (atomic t).poke (v: t) // relaxed memory order - do not use to sync tasks!

proc (atomic t).exchange (v: t): t
proc (atomic t).compareExchange (e: t, v: t): bool

proc (atomic t).add (v: t)
proc (atomic t).sub (v: t)
proc (atomic t).or  (v: t)
proc (atomic t).and (v: t)
proc (atomic t).xor (v: t)

proc (atomic t).fetchAdd (v: t): t
proc (atomic t).fetchSub (v: t): t
proc (atomic t).fetchOr  (v: t): t
proc (atomic t).fetchAnd (v: t): t
proc (atomic t).fetchXor (v: t): t

proc (atomic t).waitFor(v: t)

proc (atomic bool).testAndSet () : bool
proc (atomic bool).clear ()
```

You could therefore safely increment such a value in a concurrent environment without loosing a beat with:

```
No_of_data_points.add (1);
```

or if you want to atomically check what the current value is and also increment it in the same go:

```
Previous_No_of_data_points : uint = No_of_data_points.fetchAdd (1);
```

which increments the variable and gives you the exact value before you incremented it. Keep those operations in mind for a moment, while we add some more interesting constructs.

Synchronized scalars

Instead of using pre-defined atomic operations we can also declare:

```
No_of_data_points : sync uint = 0;
```

which provides a lock representing "empty" and "full" for this variable. It can only be written and read in alternate sequence and any access out of order will result in this task being blocked until another task will performs the currently available operation. If the usage of this variable in multiple tasks would be for instance:

```
No_of_data_points = No_of_data_points + 1;
```

or short:

```
No_of_data_points += 1;
```

you will see that there is always one read operation, followed by one write operation. The important part of this process is that after one task read the variable as part of this statement, the variable is locked down for further reads until somebody performed a write operation on it. Which means here that those increment operations become automatically atomic (unless yet other tasks are reading and writing this variable in a different pattern).

1 see the Chapel reference manual for details about those operations.

Ok, this was not yet spectacular so far (compared to last week's lab), so let's try something more interesting:

```
config const Buffer_Size  : uint = 10,
             Writes_Reads : uint = 30;

// Domain chosen such that modulo operations work on the index:

var Buffer : [0 .. Buffer_Size - 1] sync real; // only the elements are sync'ed!

proc Producer (Id : uint) {

  writeln ("-- Producer ", Id, " starting up");

  var Write_Index : index (Buffer.domain) = Buffer.domain.low,
      Data        : real                  = Id + 0.01;

  for i in 1 .. Writes_Reads {
    Buffer [Write_Index] = Data;
    Write_Index = (Write_Index + 1) % Buffer_Size;
    Data = Data + 0.01;
  }
}

proc Consumer (Id : uint) {

  writeln ("-- Consumer ", Id, " starting up");

  var Read_Index  : index (Buffer.domain) = Buffer.domain.low,
      Data        : real                  = 0.0;

  for i in 1 .. Writes_Reads {
    Data = Buffer [Read_Index];
    write (Data, " ");
    Read_Index  = (Read_Index + 1) % Buffer_Size;
  };
}

writeln ("----- Starting data communication");

cobegin {
  Producer (1);
  Consumer (1);
}

writeln ();
writeln ("----- All data communicated");
```

`cobegin` is a keyword which you did not see before, but you are very familiar with the concept: You create a number of tasks within a scope and are blocked at the end of the scope until all those tasks complete. Here we are creating two tasks – one which executes Producer and one which executes Consumer – and wait at the end of the `cobegin` scope until both tasks complete.

Yet the interesting line in the code above is:

```
var Buffer : [0 .. Buffer_Size - 1] sync real;
```

Which defines an array where every element is synchronized individually. The array is *not* initialized which means that the locks are currently all in state "empty". The first operation on each array element therefore has to be a write operation. In case that the `Consumer` is swift on its feet and reaches the first

```
Data = Buffer [Read_Index];
```

statement before the `Producer` could write some data, then the `Consumer` will be blocked until the `Producer` has caught up and executed its

```
Buffer [Write_Index] = Data;
```

If the `Producer` would be the one which is many times faster then the `Consumer` then it will fill all elements in the array until it wraps around and finds an element which it already wrote to and will be blocked until the `Consumer` caught up and reads out some data.

Now comes the essence of this interlude: convince yourself that as long as the buffer as a whole is neither empty nor full, then both, the `Producer` and the `Consumer` will operate **safely and concurrently** on this buffer without blocking on each other. This was only possible because we applied synchronization to the elements themselves instead of the whole array.

Exercise 1: Multiple producers and consumers

Once you create more consumers or more producers, for instance like such:

```
cobegin {
  Producer (1);
  Producer (2);
  Consumer (1);
  Consumer (2);
  Consumer (3);
}
```

you will find that the program does not work any more (does not terminate any more). First analyse why this is. What goes wrong if we have multiple instances of those tasks?

What is supposed to happen is that 30 data items are produced across all producers and are consumed across all consumers. In other words, the producers and the consumers have to coordinate such that the right number of items in total is produced and consumed.

Try to find a way to synchronize the now out-of-sync tasks in a way which does not involve a global lock which would nail down everything to a single operation on the array at any given time - if we would have wanted that, then we could have just wrapped the array into a protected object.

There are multiple ways of solving this exercise. You can use atomic operations or synchronized variables, or a combination of both. Keep the goal of maximal concurrency in mind though.

This exercise is not easy and at the end I want you to reflect on how much additional concurrency you gained in exchange for how much additional programming and how many synchronization points.

Submit your program by direct copy-and-paste to the *SubmissionApp* under “Lab 9 Fine Grained Synchronization” for code review by your peers and us.

Interlude: Barriers

There is another form of synchronized variables offered in Chapel which does not enforce alternating writes and reads, but enforces a single write followed by an unbound number of reads. These are called `single` variables in Chapel. If we for instance declare

```
var Gate : single bool; // considered “empty” after declaration
```

we can have as many tasks blocked on this gate by reading from it as we want, until one tasks writes a value to this variable (“fills” it) and automatically released all blocked tasks. An attempt to write to it a second time is considered a program error as there is no waiting queue defined for write operations on this variable (as the variable is not supposed to change back to “emtpy” ever again – well, unless we use some special operations to reset it). Note that it is irrelevant what value is written into the variable, as the value itself is not related to the locking state. This example may illustrate this point better:

```
var Poll_Result : single uint; // considered “empty” after declaration
```

We can now have many tasks waiting on the poll result while we can write into the variable whatever we like – it will always have the effect of releasing all reading tasks.

Exercise 2: Kickstarter

Write a Chapel program from scratch which has a number of tasks which are simulating potential share-holders in a kickstarter project. Each of the tasks will provide a certain funding to a central account. A central tasks will look at the accumulated amount of money and will start the project (release all the share-holders) and tells them how much money was accumulated when the threshold was reached or surpassed. In short: *n* tasks add a discrete amount to a central variable and then block themselves until the value of this central variable exceeds a preset value. After release they are all informed about the amount of money which accumulated when the preset value was exceeded.

This should only take a few lines of Chapel code if you took the hints from today’s lab on board.

Submit your program by direct copy-and-paste to the *SubmissionApp* under “Lab 9 Kickstarter” for code review by your peers and us.

**MAKE SURE YOU LOGOUT
TO TERMINATE YOUR SESSION!**

Outlook

Next you will be looking at your second assignment where you will implement a core algorithm of a router.