13.3 Other issues in parallel programming

Description

As mentioned, parallel programming is a complicated and tricky endeavour and there are many pitfalls in practicing parallel programming. This is due to the fact that a complex system with many parallel activities is difficult to grasp and the state-of-art of methods, techniques and programming languages is not that well developed. We strongly advise readers of the book to take one or more courses in parallel programming before starting to practice it. Below we touch upon some of the issue of parallel programming, which may be taken as a starting point.

Critical regions

One common problem with parallel programming is to avoid that two or more parallel objects access the same *resources* at the same time since this may lead to an undefined result. A resource in this sense may be data stored in other objects often referred to as *shared objects*. It may also be external devices like the console object that prints strings in a window on the screen.

The statements of a parallel object accessing shared objects is often referred to as a *critical region* or *critical section*. The situation where two or more parallel objects access the same shared objects implying an undefined result is referred to as a *race condition* as introduced in section .

The term *mutual exclusion* is a property of parallel programming that is introduced in order to prevent race conditions. Mutual exclusion must ensure that a parallel object never enters a critical region while another parallel object is in a critical region modifying the same shared data and/or using a shared resource like a console.

The next example shows a program with shared objects, critical regions and a situation with a race condition:

```
raceSystem: obj BasicSystem
   N: var integer   <<< shared data
   P1: obj BasicProcess
      ...
      N := N + 1    <<< critical region
      ...
   P2: obj BasicProcess
      ...
      N := N + 2    <<< critical region
      ...
   N := 10
   P1.start
   P2.start
```

The variable `N` is an example of data shared between the parallel objects `P1` and `P2`. `P1` and `P2` both update the value of `N` and since this may happen at the same time, the resulting value of `N` is undefined. At the end of the program the value of `N` may be one of 11, 12 or 13.

This is due to the fact that execution of an assignment like `N := N + 1` by `P1` is not an atomic operation in the sense that `P2` may access `N` while `P1` is executing `N := N + 1`. Execution of `N := N + 1` takes place in a number of steps as shown below:

```
R: var integer
R := N
R := R + 1
N := R
```

P1 reads the value to a a variable R, which in practice may be a register of the core unit executing P1. It then increments R and stores the value of R back into N.

Execution of N := N + 1 takes place in a similar way and this may be lead to the following scenario:

```
P1.R := N
P2.R := N
P2.R := P2.R + 2
N := P2.R         <<< now N = 12
P1.R := P1.R + 1
N := P1.R         <<< P1 overrides the value of N and now N = 11
```

In this scenario the resulting value of N is 11. The above statements may be interleaved in a number of other ways leading to different results as mentioned above.

In order to avoid the race conditions, the critical regions must be protected by mutual exclusion and this might be done by declaring N within a Monitor-object and defining methods for updating N.

With respect to accessing shared objects, there is a difference between parallel objects reading the values of shared objects without modifying the shared objects, and writing new values into shared objects. It is in general harmless to have multiple parallel objects reading shared objects at the same time. If, however, a parallel object is writing into and thus modifying a shared object, the mutual exclusion mechanism should ensure that no other parallel objects are reading or writing the shared objects since this may lead to race conditions.

Parallel objects reading shared objects are often referred to as *readers* and those writing shared objects as *writers*. The problem of handling multiple readers and at most one reader is called the r*eaders-writers* problem.

In section 8.1, the searcher-objects store matching records during search in the Set-object matches. Here the critical sections are the statement matches.insert(P). The object matches is an example of a shared object, and since it is encapsulated in the collector-object, which is a Monitor, mutual exclusion is guaranteed when the searcher-objects access matches.

The records-object used by the searcher-objects in the same section ~~o~~is also an example of a shared object~~s~~. Here multiple searcher-objects may access records at the same time, but since they only read data and do not modify the objects, this is safe and does not lead to race conditions.

This is, however, at the risk of the programmer. The code in the example might as well modify the records without the compiler of the language complaints. In general, it is a great advantage if the language mechanisms guarantees mutual exclusion.

As mention, the monitor is one language mechanism that may be used to ensure mutual exclusion. Below we will mention some of the most common mechanism used in mainstream programming languages.

High-level synchronization

In a previous section, we have introduce monitors as an example of a high-level synchronisation and communication abstraction. There are many other examples of such high-level abstractions and here we will give one more example.

A common form of communication between parallel processes is message-passing where a process may send a message directly to another process. The form of a message varies depending on the system, library, and/or language being used. A message may be a text, a reference to an object, or a value. For the difference between reference to an object and a value, see the Chapter Objects and Values.

Another difference between message-passing systems is whether or not the ‘sending of the message’ is synchronous or asynchronous just as it differs what sending of message actually means.

In this book, sending of a message is defined as invocation of method of a parallel object. In general, a method invocationis *synchronous* in the sense that the invoker of the method is blocked (waits) until the message has been executed by the receiver whereafter it returns to the invoker.

In systems with parallel objects, synchronous method invocation may imply that a calling parallel object may wait unnecessarily for a method to be executed by the receiver. For this reason, so-called *asynchronous* method invocations may be used.

In asynchronous method invocations, the method object is usually inserted into a queue at the receiving object, and the receiver then executes the method objects from the queue. The exact way of how this is done varies from system to system.

Here we will use as an example, a simple asynchronous message passing system where each method object is inserted in a queue at the receiver, which then executes them in the order they arrive.

We use a variant of the search example from section where the `searcher`-objects send a matching `Person` record directly to the `printer`-object and not via a `Monitor`-object:

```
asynchSystem: obj SimpleAsynchSystem
   records: obj IndexedRef(100,Person)
   search(first: var integer, last: var integer):
      current: ref Person
      for (first) :to(last):repeat
          if ((18 <= records.get[inx].age)
              && (records.get[inx].age <= 24)) :then
             current := records.get[inx]
             inner(search)
   Person(name: ref String, age: var integer):
      ...
   Searcher: Process
      inner(Searcher)
   searcherA: obj Searcher("SearcherA")
      search(1,33)
         printer.add(current)
   searcherB: obj Searcher("SearcherB")
      search(34,66)
         printer.add(current)
   searcherC: obj Searcher("SearcherC")
      search(67,100)
         printer.add(current)
   printer: obj Process("Printer")
      add(P: ref Person): entry
         console.print("Found: " + P.name + ",age: " + P.age + "\n")
   searcherA.start
   searcherB.start
   searcherC.start
   printer.start
```

Note that the class `Process` used as a superclass in this example is not the same as the one in section , which is defined as a local class of `BasicSystem` whereas the one used here is a local class of `SimpleAsynchSystem`.

A `Process`-object in this system has a queue of waiting method objects. When a `searcher`-object executes `printer.add(current)`, the `add`-object is inserted in the queue of the `printer`. The `printer` repeatedly checks if there is a method object in the queue, and if one is found, it is executed. The checking of the queue is defined in the `Process` class of which `printer` is subclassed.

As mentioned, there are many proposals for high-level communication and synchronisation abstractions and they have many variants. This may be due to the fact that different problems require different mechanisms and no mechanisms have turned out to be dominant.

Low-level synchronization

The monitor and asynchronous method passing are examples of a high-level synchronisation mechanisms. Most mainstream language include what we characterise as low-level synchronization mechanisms. In this section, we describe some of these.

Lock

One example of a is a *lock*. The idea is that a parallel object has to acquire a given lock before entering a critical region. In the example below, we use a lock to ensure mutual access to the global integer variable `N` from the above example:

```
raceSystem: obj BasicSystem
   mutex: obj Lock  -- declaration of a lock
   N: var integer   -- shared data
   P1: obj BasicProcess
      ...
      mutex.wait    --- wait until the lock is free
      N := N + 1    --- critical region
      mutex.signal  --- signal that the lock is released
      ...
   P2: obj BasicProcess
      ...
      mutex.wait
      N := N + 2    --- critical region
      mutex.signal
      ...
   N := 10
   P1.start
   P2.start
```

We have extended the example with a lock declared by `mutex: obj Lock`. Each parallel object `P1`, and `P2`, executes a `mutex.wait` before incrementing `N`. If the `Lock`, `mutex` is free, it becomes locked and the process may enter the critical regions and modify `N`. After this, execution of `mutex.signal` releases the `Lock` and if some other process is waiting for the `Lock`, it may obtain it and enter the critical region.

If more than one process is waiting, they may obtain the `Lock` in the order of which they tried to acquire it or one is picked randomly. This depends on how the `Lock` is implemented.

Semaphore

Another example of a low-level synchronisation mechanism is the *semaphore*. A semaphore has an associated integer variable. The variable is initialised with a (positive) integer value. Like a `Lock`, it has methods `wait` and `signal` The `wait`-method decrements the value and if it becomes negative, the executing process is blocked until the value becomes zero. The `signal` method increments the value.

There are two variants of a semaphore, a *binary semaphore* and a *counting semaphore*. For a binary semaphore, the value may be either 0 or 1. A binary semaphore is thus similar to a lock.

A counting semaphore may be used to manage a pool of shared resources. Suppose you live in community that have 10 bicycles to be shared between people in the community. You may synchronise access to these bicycles using a counting semaphore:

```
myCommunity: obj BasisSystem
   bicyclePool: obj CountingSemaphore(10)
   P1: obj BasicProcess
       ...
       bicyclePool.wait     -- wait for a free bicycle
       -- take a free bicycle
       ...
       bicyclePool.signal   -- return the bicycle
   P2: obj BasicProcess
       ...
```

The `myCommunity` object has a `CountingSemaphore` object, `bicyclePool`, initialized to the value 10 representing the availability of 10 bicycles. A parallel object like `P1` that wants to obtain a bicycle executes the method `bicyclePool.wait`. For each such invocation, the associated integer is decremented by one and it if becomes zero, the process has to wait. In this way up to 10 processes can obtain a bicycle. When a process has finished using a bicycle it must return it and execute a `bicyclePool.signal,` which increments the associated integer.

We leave it as an exercise to the reader to rewrite the `search`-example to use locks and/or semaphores.

It is in general more safe to use a high-level synchronisation mechanism like a monitor than the above low-level mechanisms. Using locks and semaphores, a programmer may forget to invoke a `wait`– or `signal`-method, in which case race-conditions may appear. There is no way the language and compiler can guarantee that no such errors are made.

Common challenges in parallel programming

In this section, we mention some of the problems that often arise in parallel programming.

Deadlock

*A **deadlock*** is a situation where a set of parallel objects are blocked because each object is holding a resource and waiting for another resource acquired by some other object.

As a real-life example, a deadlock may arise if two cars crossing a single-lane bridge from opposite directions and as long as none of the cars is willing to back, none of them can proceed.

As a programming example, we may use the bank system. If multiple clerks can make transactions on the accounts, one needs to synchronize acces to accounts. This can be done by introducing a lock for each account:

```
JohnSmithsAccount: obj Account("John Smith")
JohnSmitLock: obj Lock
lizaJonesAccount: obj Account("John Smith")
LizaJonesLock: obj Lock
```

A clerk, `clerkA`, may need to transfer money from `JohnSmithsAccount` to `LizaJonesAccount`, and to do this he/she needs to acquire the locks for these accounts:

```
clerkA: obj BasicProcess
   JohnSmithLock.wait
   LizaJonesLock.wait
   transfer(500,JohnSmithsAccount, lizaJonesAccount)
   JohnSmithLock.signal
   LizaJonesLock.signal
```

Another clerk, `clerkB`, may decide to transfer money from `LizaJonesAccount` to `JohnSmithsAccount` and thus acquire the locks:

```
clerkB: obj BasicProcess
   LizaJonesLock.wait
   JohnSmithLock.wait
   transfer(500,lizaJonesAccount,JohnSmithsAccount)
   LizaJonesLock.signal
```

```
JohnSmithLock.signal
```

A situation may thus arise where `clerkA` has acquired JohnSmithLock and is waiting for `LizaJonesLock` while at the same time, `clerkB` has acquired `LizaJonesLock` and is waiting for `JohnSmithLock`. The implication of this is that the two clerks are waiting for each other to release a lock, but this cannot happen.

Starvation

Starvation or resource starvation is a problem encountered where a parallel object is constantly denied the necessary resource to carry out its work. Starvation can be caused by errors in the algorithms scheduling the parallel objects and/or the synchronisation mechanism like lock, semaphore, monitor, etc.

Termination

For an algorithm in general, it is an important property that it can be validated that the algorithm actually terminates – i.e. finishes its computation. For sequential algorithms, this can be more or less difficult. For parallel algorithms it may be even more difficult since it may involve a more or less complicated protocol where the various parallel objects involved in the algorithm communicate to each other that the algorithm should terminate.

Overhead

A system of parallel objects involves communication and synchronization between the objects and this gives an overhead compared to the core of the algorithm being computed. It is therefore necessary to be aware of the possible overhead when designing parallel systems.

Further reading/courses

We have previously said that in order to acquire the necessary skills to become a software developer, the reader must take a course in algorithms and data structures. To be able to develop parallel systems it is also necessary to take a course on parallel algorithms and data structures. This includes techniques to handle the above mentioned problems with deadlock, starvation, termination and overhead.