Mutual Exclusion
Locks, Semaphores and Monitors

Programming II - Elixir Version

Johan Montelius

Spring Term 2018

Getting started

In this assignment you will learn about the concept of *mutual exclusion* and how locks, semaphores and monitors can give us what we want. These concepts are not frequently used in Elixir programming but you should know about them and understand why they are not often explicitly used in Elixir.

The main idea with mutual exclusion is that we want to limit the concurrency to at most one executing process in a *critical section*. A critical section could be a section in the program where we modify some data structure and we do not want any other process to see what we have done until we are completely done. Since we do not have any mutable data structures in Elixir the need for locks is limited but think about updating a set of files where you want to do all the changes before you let another process see what you have done or do their modifications.

1 Let's implement a lock

We will start by implementing a *lock process* (or rather try to implement it). A lock is something that can only be held by one process, the process that *takes the lock* knows that it is the sole owner of the lock and can proceed to a critical section.

Our first attempt to implement a lock process is quite straight forward - we will implement the lock as a process that only holds one value and accepts two messages: **set** and **get**.

```
defmodule Cell do

  def new(), do: spawn_link(fn -> cell(:open) end)

  defp cell(state) do
```

```
    receive do
      {:get, from} ->
        send(from, {:ok, state})
        cell(state)

      {:set, value, from} ->
        send(from, :ok)
        cell(value)
    end
  end

end
```

To make it easier to use the lock we also provide two functions that hide the fact that we do asynchronous communication.

```
def get(cell) do
  send(cell, {:get, self()})
  receive do
    {:ok, value} -> value
  end
end

def set(cell, value) do
  send(cell, {:set, value, self()})
  receive do
    :ok -> :ok
  end
end
```

If we have created a cell we could use it to protect a critical operation using the codes that follows (not true so don't stop reading here).

```
def do_it(thing, lock) do
  case Cell.get(lock) do
    :taken ->
      do_it(thing, lock)
    :open ->
      Cell.set(lock, :taken)
      do_ya_critical_thing(thing)
      Cell.set(lock, :open)
  end
end
```

Perfect, case closed ... ehhh, there is something wrong - what happens if...? Before you continue think about what would happen if two processes called the `do_it/2` procedure with the same lock; do we guarantee that the two processes will never ever execute the critical section at the same time?

2 The atomic swap

There are two solutions to the problem of the lock in the first section: *atomic swap* and *Peterson's algorithm*. Using atomic swap we implement a new message that will read and write to the cell in the same operation. This feature is often found in hardware and programs written in C or C++ can often make direct use of this. In our implementation we will implement it ourselves with a small extension to the lock.

```
defp cell(state) do
  receive do
    {:swap, value, from} ->
      send(from, {:ok, state})
      cell(value)
    {:set, value, from} ->
      send(from, :ok)
      cell(value)
  end
end
```

Assuming we also provide a functional interface we could now use the lock as follows:

```
def do_it(thing, lock) do
  case Cell.swap(lock, :taken) do
    :taken ->
      do_it(thing, lock)
    :open ->
      do_ya_critical_thing(thing)
      Cell.set(lock, :open)
  end
end
```

In this version it does not matter if two processes calls the procedure at the same time; both of them will swap in the value `:taken` but only one of them would receive the `:open` value in return. The process that loses the race will have to retry to take the lock and will succeed once the holding process sets the lock to `:open`.

3 Peterson's algorithm

You might wonder if there is a way to implement a lock without an atomic swap operation. If not, we are sure lucky that the hardware people have implemented it. It turns out that there is and the algorithm is fairly simple once you understand why it works.

3.1 the algorithm

Assume we have our original cell with only the `:get` and `:set` operations. We also assume that there are only two processes that will compete for the lock. We will now use three cells: `p1`, `p2` and `q`. In the cell `p1` the first process will declare its interest in moving into the critical section. The second process will declare its interest using `p2`. The `q` cell will be used to determine the winner if we have a draw.

The two processes will execute slightly different code when trying to enter the critical section; or rather, the code is the same but the parameters are shifted. The first process will call the procedure `lock(0, p1, p2, q)` where as the second process will call `lock(1, p2, p1, q)`.

```
def lock(id, m, p, q) do
  Cell.set(m, true)
  other = rem(id + 1, 2)
  Cell.set(q, other)

  case Cell.get(p) do
    false ->
      :locked
    true ->
      case Cell.get(q) do
        ^id ->
          :locked
        ^other ->
          lock(id, m, p, q)
      end
  end
end

def unlock(_id, m, _p, _q) do
  Cell.set(m, false)
end
```

The intuition is that each process begins to declare that they are interested in taking the lock. They then set the common cell `q` to the second process's identifier as a signal to the second process to go ahead if they are both interested of the lock. If a process however sees that the second process is not interested it holds the lock and proceed into the critical section.

3.2 prove it

To understand how Peterson's algorithm works is not easy; to prove that it ensures that the two processes do not believe to hold the lock at the same time is more difficult.

If you want to prove that Peterson's algorithm works, you can draw a finite state machine diagram with the state of the variables: *p1*, *p2*, *q* and two variable *l1* and *l2* that describes if a process is in the critical section. One alternative method is to use so-called *temporal logic* where the rules can prove that it is never so that *l1* and *l2* are both true.

An interesting question is whether one can make use of Peterson's algorithm in a computer that has as much as possible in the cache, or in a distributed system where clients have local copies. A prerequisite for the algorithm to work is that if a process sets *p1* to *true* and then reads that *p2* is *false* then it can not be that the second process manages to set *p2* to *true* and then read that *p1* is *false*. Describe a scenario using cached local copies that will violate this prerequisite.

3.3 the bakery algorithm

When I lived in Barcelona I learned a wonderful algorithm for keeping track of who is next to be served in a bakery. When you entered a bakery (or butchery) you simply greeted every one with the phrase "¿Buenos dias, ultimo?". The person who was the last person in line would reply "Si" and then you would know who was the person just in front of you. If someone else entered the store you would be the one to reply "Si" and that was it. The system works perfectly and you avoid the hassle of finding a machine to give you a ticket. If Leslie Lamport had lived in Barcelona he would probably never have named his algorithm *the bakery algorithm* since it is based on a numbering system where entering processes picks a number that is higher than any other number in the queue.

The algorithms uses a shared array with one index per process. If the index of a process is set to 0 it means that the process is not interested in entering the critical section. When a process wishes to enter the critical section it will scan the array and find the highest ticket number and then set its own index to the number plus one. The intuition is that all processes that entered the store before it should have precedence. It could of course happen that two processes enters at the same time and chooses an identical ticket but this is solved by giving precedence to the process with the lowest id.

When a process has selected a ticket number it will again scan the array from the beginning and wait until all indexes before its own, are either set to 0 or have a ticket number that is higher than its own and, all indexes after its own are set to 0 or have ticket numbers higher or equal to its own. When the process has scanned the array it is allowed into the critical section and will, when it is done, set its own index to 0.

The scanning of the array can proceed one step at a time, if a index has the value 0 it could of course be set by the owner of the index but then it will be set to a value equal or higher to the ticket of the scanning process. An index that holds a value that is lower than then ticket of the scanner will eventually be set to 0 once the processed has completed its critical section.

4 The semaphore

In the previous section we implemented the locks using a *busy waiting* or *spin-lock* strategy. A process that would not immediately require the lock would try and try again until the lock was acquired. This is a very aggressive strategy that in the worst case means that a process will spend a lot of resources just reading from a memory location, or as in our case send thousands of messages to a cell process and request its state.

A better strategy (not always better) could be to suspend the execution if the lock is not taken and only continue to execute once the lock has been acquired. The semaphore concept is also often described as being more general compared to a binary lock. We will describe a semaphore that will allow at most n processes to enter the critical section. If n is 1 then it called a *binary semaphore*.

4.1 was this it

In Elixir this is expressed so easily that you hardly realize that you have implemented a semaphore. Look at the code below:

```
def semaphore(0) do
  receive do
    :release ->
      semaphore(1)
  end
end

def semaphore(n) do
  receive do
    {:request, from} ->
      send(from, :granted)
      semaphore(n - 1)
    :release ->
      semaphore(n + 1)
  end
end
```

If we create a semaphore with the initial value 4 then at most four processes will be granted access to the critical section. The code for requesting entrance would of course look like follows:

```
def request(semaphore) do
  send(semaphore, {:request, self()})
  receive do
    :granted ->
      :ok
  end
end
```

The difference from the locks we implemented before is that the requesting process will now be suspended waiting in a receive statement until the semaphore responds with a `:granted` message.

4.2 I've heard it was tricky

In the classical description of a semaphore one must explain what happens when a process request access and there are no resources left. One will then describe how this process is added to a queue of waiting processes and how it then yields the execution. When a process leaves the critical section, the first process in the queue of suspended processes will be selected and added to the set of runnable processes (often by sending it a signal to wake up).

In our Elixir implementation all this is hidden in the message queue. A process that sends a `:request` message will of course have its message inserted as the last message in the message queue. If there are no resources left (the first clause), then request messages will simply not be handled. Only when resources are available will the semaphore handle requests and then it will of course handle them in the order they have arrived in the message queue.

As an exercise you can re-write the semaphore so that it only has one clause and always accepts requests. If there are no resources available the requesting process must be held on hold in a list of waiting processes. When a release message is received and the resource is incremented from zero the first process, if any, in the lists of waiting processes should be granted access.

5 The monitor

The semaphore gave us a solution to the mutual exclusion problem but is does of course require that the processes do respect the rules. No process is allowed to enter the critical section if it has not being granted access by the semaphore. It must also release its access when it leaves the critical section. A process that not play by the rules could of course ruin the whole system.

A better strategy is to encapsulate the critical section inside a semaphore so that no process can enter the critical section without using the semaphore. This concept is called *a monitor* and is how things are done in Elixir as well as Java. In Java one would declare a method of an object to be *synchronized* thereby preventing more than one thread at a time to execute the method. In Elixir the same thing is of course handled by messages.

```
def monitor(state) do
  receive do
    {:request, from} ->
      updated = critical(state)
      send(from, :ok)
      monitor(updated)
  end
end
```

The Actors model, that Elixir is built on, automatically gives us the properties of a monitor. From the implementation of `monitor/1` we easily see that request to execute the critical section are of course handled one by one and will even be done in a fair order.

You could easily implement more advanced constructs where a request could contain a lambda expression that should be applied to the monitor state. In this way the requesting process has more freedom to control what should be done.

```
def monitor(state) do
  receive do
    {:request, fun, from} ->
      updated = fun.(state)
      send(from, :ok)
      monitor(updated)
  end
end
```

In one way, you can view every Elixir process as a monitor that protects its state. It will only handle one message at a time and it is the only process that has access to the state.

6 Deadlocks

The locks, semaphores and monitors solves the problem of data corruption i.e. by only allowing one transformation at a time. As you have seen the Actors model gives us this almost for free but this is only half of the problem introduced by concurrency. A equally important problem is the problem of *deadlock* i.e. a situation were no process can proceed since they are all waiting for someone else to take the first step.

6.1 shades of hell

A complete deadlock is of course the worst thing that could happen but there are other related problems that are almost as bad:

- deadlock: Nothing moves.
- livelock: Things move but we're not making progress.
- starvation: We make progress but at least one process is prevented from progressing.
- non-fair scheduling: All processes make progress but some processes do not get a fair share of the resources.

It is not necessarily so that every system you implement must implement a fair scheduling algorithm that guarantees that all processes should have a equal chance in acquiring the resources of the system. It could be enough that it is starvation free or proved to never go into a livelock. Its important to understand what is required and then choose algorithms to meet these requirements. If you always go for an implementation that guarantees fair scheduling then you might pay more than anyone asked for.

6.2 locks, semaphores and monitors

Go back to the locks, semaphores and monitors in the previous section and ask yourself what the implemented solutions actually provided. They were hopefully dead lock free but were they starvation free? Did they provide a fair scheduling i.e. if a process has requested a lock or access to a critical section, will the process be granted access before any process that issues a request at a later moment?

6.3 detecting a deadlock

Even if the locks, semaphores and monitors that we have discussed have algorithms that will not deadlock, we can easily create a deadlock if we have two or more locks. If process *P1* is granted a lock *A*, process *P2* is granted a lock *B* and requests lock *A* it will of course have to wait. If process *P1* now requests lock *B* we have a circular dependency that has caused a deadlock.

We can create a similar circular dependency with monitors if one monitor will, as part of its critical section, request the service of another monitor. This monitor might in turn request a third resource that request the service of the first monitor.

One way to break the deadlock is to give up waiting for a lock, release some of the locks held, do something else for a while and then retry to take the locks. We can implement this using a timeout in the receive statement. The implementation will however be more complicated than you might first think.

6.4 master should be resting

Assume that we have implemented a semaphore and use the following function to acquire access.

```
def request(semaphore) do
  send(semaphore, {:request, self()})
  receive do
    :granted ->
      :ok
  after
```

```
    1000 ->
      :abort
  end
end
```

A process that calls the procedure `request/1` will then be given the result `:ok` or `:aborted`. If everything went fine it will continue to execute but if it receives `:aborted` it knows that we could in the worst case be in a deadlock. If it is not holding any other lock it is not much it can do but if it holds another request a good strategy could be to release this resource and then ponder π for a while.

> This all sounds fine but the above explanation could have been given by Gollum.

If you implement it like this your in for a surprise. The message that you sent to the semaphore is not lost but simply waiting in the message queue of the semaphore. Sooner or later the semaphore will handle this request and send a `:granted` message to you. Now you have the resource even if you don't realize it. If you at a later point in time return to the semaphore and again request the resource you will find this granted message that has been there all the time.

```
def request(semaphore) do
  ref = make_ref()
  send(semaphore, {:request, ref, self()})
  wait(semaphore, ref)
end

def wait(semaphore, ref) do
  receive do
    {:granted, ^ref} ->
      :ok
    {:granted, _} ->
      wait(semaphore, ref)
  after
    1000 ->
      send(semaphore, :release)
      :abort
  end
end
```

We should of course also change the implementation of the semaphore so that it accepts requests on the form {`:request, ref, from`} and reply with a {`:granted, ref`} but then we are on the safe side.

The reason we can send a `:release` before we actually have been granted the request is that we know that our request is in the message queue. We also make sure, by using the unique references, that we will not mistake an old granted message as a reply to a new request.

Better than trying to resolve a deadlock situation is of course to avoid it all together and there is a simple strategy that will always work.

6.5 avoiding deadlock

The problem with a deadlock is of course that we have a circular structure where everyone is waiting for someone else. If we can avoid building a circular structure we could avoid deadlocks all together.

Assume all resources (locks, semaphores or monitors) are ordered and you're never allowed to take higher resource before a lower resource. If you find that you're waiting for a resource the resource is of course held by someone. This someone is either working, in which case everything is fine and you will be given access sooner or later, or suspended waiting for a higher resource. It can not be suspended waiting for a lower resource since it is not allowed to request a lower resource if it is holding the resource that you're waiting for.

You might ask, what happens if the process is suspended waiting for a higher resource but then it is in the same situation as you are. We have a chain of processes, all waiting. Since we only have a finite set of resources the process in the end of the chain is not suspended but working. Sooner or later it will let go of its resources and allow the next process in line to continue its execution.

The only problem with this strategy is that it sometimes is hard to order the resources so that everyone knows the order. Nor is it always easy to determine beforehand which resources that will be needed - if you start by grabbing a resource that you know you need you're not allowed to grab a resource with a lower rank.

7 Summary

The Actors model of handling concurrency saves us from most of the problems of protecting critical sections. The process is in a sense a monitor that protects a critical section and the message handling will give all processes fair access to the resource.

Deadlock is a problem but can often be avoided by ordering the resources and always request the resources in this order. Detecting a deadlock situation and resolve the situation might be trickier than think.