Locks, semaphores and monitors

Johan Montelius

July 7, 2017

Getting started

In this assignment you will learn about locks, semaphores and monitors. These concepts are not frequently used in Erlang programming but you should know about them and understand why they are not often explicitly used in Erlang.

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* where we can only have one process executing at a time.

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 updatable data structures in Erlang 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.

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.

```
-module(cell).
-export([new/0]).

new() -> spawn_link(fun() -> cell(open) end).

cell(State) ->
  receive
     {get, From} ->
         From ! {ok, State},
         cell(State);
     {set, Value, From} ->
         From ! ok,
         cell(Value)
  end.
```

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

```
get(Cell) ->
   Cell ! {get, self()},
   receive
       {ok, Value} -> Value
   end.

set(Cell, Value) ->
   Cell ! {set, Value, self()},
   receive
       ok -> ok
   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's stop reading here).

```
do_it(Thing, Lock) ->
   case cell:get(Lock) of
     taken ->
        do_it(Thing, Lock);
     open ->
        cell:set(Lock, taken),
        do_ya_critical_ting(Thning),
        cell:set(Lock, open)
   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 cell.

```
cell(State) ->
  receive
```

```
    {swap, Value, From} ->
        From ! {ok, State},
        cell(Value);
    {set, Value, From} ->
        From ! ok,
        cell(Value)
  end.
```

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

```
do_it(Thing, Lock) ->
   case cell:swap(Lock, taken) of
     taken ->
        do_it(Thing, Lock);
     open ->
        do_ya_critical_thing(Thing),
        cell:set(Lock, open)
   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 looses 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 operations. 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)`.

```
lock(Id, M, P, Q) ->
   cell:set(M, interested),
   Other = (Id+1) rem 2,
   cell:set(Q, Other),
   case cell:get(P) of
      false ->
         locked;
      true ->
         case cell:get(Q) of
            Id ->
               locked;
            Other ->
                lock(Id, M, P, Q)
          end
   end
end.

unlock(_Id, M, _P, _Q) ->
   cell:set(M, false).
```

The intuition is that each process begins to declare that they are interested in taking the lock. They then set then 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 is 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 believed 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 sector. One alternative method is to use so-called *temporal logic* where the rules can prove that it is never so that *l1* and em 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 "¿Buenas diaz, 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 processes 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 why not in Erlang

Atomic swap, Peterson's or the bakery algorithm are hardly ever used in a Erlang program, nor in any higher level program. The reason is that the mutual exclusion problem is solved by language constructs that hides the details of how things are implemented.