Message-Passing Concurrency

Annalise Tarhan

June 12, 2020

1 Port Objects That Share One Thread

1.1 The Ping-Pong program has two port objects that execute a method and then asynchronously call the other. One initial message causes an infinite ping-pong of messages between them. What happens if two initial messages are inserted? For example, what happens if the following two initial calls are done? How will object executions be interleaved?

```
{Call Ping ping(0)}
{Call Pong pong(10000000)}
```

Because both port objects are operating on the same thread, they are interleaved in perfect order. First, the two initial calls are added to the stack. Then, as each call executes, the objects display their InfoMsg and add their call to the other object to the stack. The calls are executed FIFO, so the interleaving is in lockstep.

2 Lift Control System

2.1 The current lift control system design has one controller object per lift. Is it a good idea to change it so there is only one controller for the whole system, with which each lift communicates? How does it change the behavior of the lift control system?

It would be a terrible idea. The main problem with it is that it worsens the modularity of the program, making it harder to reason about and work with. The controller's state would be unwieldy to the point of unusable, since it would have to include stopped/running, the floor number, and the lift ID for each of its lifts. Parsing messages would be commensurately more complex. It would

also have to pass the lift ID every time it sent or received messages from the timer.

2.2 Change the lift and controller objects to stop only at requested floors, not all floors that it passes.

```
elseif F<Dest then
      Floors=Dest-F
in
      {Send Tid starttimer(5000*Floors Cid)}
      state(running Dest Lid)
else
      Floors=F-Dest
in
      {Send Tid starttimer(5000*Floors Cid)}
      state(running Dest Lid)
end
```

Changing the controller object simply requires calculating the total number of seconds it will take to reach the destination and sending the timer that total instead of the time for a single floor.

Changing the lift object is even simpler. Since the controller will send it directly to its destination floor instead of stopping at each floor, it is no longer necessary to check if its *NewPos* is equal to the destination or to send another call to the controller. The lines below are the ones to delete.

```
if NewPos==S then
...
else
      {Send Cid step(S)}
      state(NewPos Sched Moving)
end
```

3 Fault Tolerance for the Lift Control System

3.1 Skipped

4 Termination Detection

4.1 Replace the definition of SubThread with the following. Explain why the result is not correct. Give an execution such that there exists a point where the sum of the elements on the port's stream is zero, yet all threads have not terminated.

```
proc {SubThread P}
     thread
          {Send Pt 1} {P} {Send Pt ~1}
     end
end
```

The difference between this version and the original is that the original sends 1 to the port before launching the new thread, which ensures that the sum will not reach zero before the new thread is executed. In this version, 1 is sent within the thread, so the sum is not updated until after the new thread is launched.

Specifically, if {SubThread P} and {ZeroExit 0 Is} are added to the stack, SubThread will execute first, creating a new thread. At that point, it is unknown if the scheduler will continue execution in the same thread, calling ZeroExit first, or switch to the new thread, calling {Send Pt 1} first. If ZeroExit is called first, the sum of the elements on the port's stream will be zero, but the new thread will not have executed or terminated.

5 Concurrent Filter

5.1 What happens when the following is executed? How many elements are displayed by the Show? What is the order of the displayed elements? Is the execution of ConcFilter deterministic? Why or why not?

```
declare Out
{ConcFilter [5 1 2 4 0] fun {$ X} X>2 end Out}
{Show Out}
```

ConcFilter uses Barrier to filter the list to include only those that are greater than 2. Show will display two elements, 5 and 4, either as [4 5] or [5 4]. Since there are multiple possible results, ConcFilter is not deterministic.

5.2 What happens when the following is executed? What is displayed by Show?

```
declare Out
{ConcFilter [5 1 2 4 0] fun {$ X} X>2 end Out}
{Delay 1000}
{Show Out}
```

ConcFilter doesn't return until its result is bound, so the delay doesn't occur until it is finished. The results are the same, but displaying them is delayed by one second. It might show either [4 5] or [5 4].

5.3 What happens when the following is executed? What is displayed by Show? What is the order of the displayed elements? If, after the above, A is bound to 3, then what happens to the list Out?

```
declare Out A
{ConcFilter [5 1 A 4 0] fun {$ X} X>2 end Out}
{Delay 1000}
{Show Out}
```

Before A is bound, nothing is displayed. ConcFilter waits for all of its input to be filtered, which can't happen until they are all bound. Once A is bound, it will be added to the end of Out, returned, and displayed by Show.

5.4 If the input list has n elements, what is the complexity of the number of operations in ConcFilter? Discuss the difference in execution time between Filter and ConcFilter.

The complexity of ConcFilter is proportional to n. The efficiency benefit of ConcFilter is realized when not all of the elements of the input list have been bound. In that case, the rest of the elements can be evaluated immediately. If all elements have already been bound, ConcFilter is less efficient due to the overhead of the extra threads.

6 Semantics of Erlang's Receive

6.1 Verify that the second form reduces to the third form when the time-out delay is zero.

When WaitTwo is called with a time-out delay of zero it behaves identically to IsDet. (This assumes that it has been modified to favor its first argument, as the text indicates. Otherwise its results are unpredictable.)

In the case of a time-out delay of zero, Loop will be called at most once. The second argument, T#T where T is unbound, represents an empty list. If a message is matched, the second list in the difference list, referred to as E inside Loop, is bound to S1. This also binds T to S1 since T and E refer to the same variable in the first iteration of Loop. The third form is simpler because Body is called with S1 directly instead of binding it to a difference list first. The reason the variable binding in the else statement is redundant with a time-out delay of zero is similar.

Without the redundant difference list bindings and with IsDet substituted for WaitTwo, the second form reduces to the third form.

6.2 Verify that the second form reduces to the first form when the time-out delay approaches infinity.

The difference between the first and second form is that the second sets a time-out and waits for either Sin to be bound or the time-out to occur. With a time-out delay approaching infinity, setting the time-out is unnecessary, and the loop should simply wait for Sin to be bound. This occurs implicitly in the first version, since the case statement blocks until S is bound. There is also no need for the else statement after the if statement, since that is only called when the time-out occurs.

Without the line where the time-out is set and the two lines for the if/else statement, the second form is identical to the first. Since those three lines are redundant with a near-infinite delay, the second form reduces to the first.

6.3 Another way to translate the second form would be to insert a unique message (using a name) after n ms. Write another translation of the second form that uses this technique. What are the advantages and disadvantages of this translation with respect to the one in the book?

```
local
      Name={NewTimer}
      {Send Name msg(T(self()) T(Expr))}
      fun {Loop S T#E Sout}
            case S of M|S1 then
                  case M
                  of T(Pattern1) then E=S1 T(Body1 T Sout)
                  ...
                  [] T(PatternN) then E=S1 T(BodyN T Sout)
                  [] Name then  E=S1 T(BodyT T Sout)
                  else E1 in E=M|E1 {Loop S1 T#E1 Sout} end
            end
      end T
in
      {Loop Sin T#T Sout}
end
```

This version uses a new process, NewTimer, which accepts a process identifier and an integer representing a number of milliseconds as arguments. The first thing this version of receive does is start the NewTimer server and note its proecess identifier. Then, it sends a message to that server with its own process identifier and the time-out value (the evaluation of each argument is left in Erlang syntax, as in the original version.) When the time is up, the NewTimer server sends a message containing only its own process identifier. When receive matches the message with Name, it executes the code that was originally contained in the else statement in the original corresponding with the cancel alarm.

Since the behavior of the old and new versions are so similar, the advantages and disadvantages are mainly the readability of each. The new version does start an additional server, but it isn't clear how the cost of an additional Erlang server compares to the cost of the Oz Alarm and WaitTwo functions. The new version hews more closely to Erlang style and keeps the if/else nesting minimal. It also relies on the implicit blocking of the case statement until S is determined instead of using WaitTwo or IsDet explicitly.

7 Erlang's Receive as a Control Abstraction

7.1 Implement the Erlang Receive operation as a control abstraction with the given requirements.

```
fun {Mailbox.receive C [P1#E1 P2#E2 ... Pn#En] D}
      Timeout
      Timesup
in
      case D of T#E then Timeout={Alarm T} Timesup=E end
      if {WaitTwo C Timeout}==1 then
            case C of M|C1 then
                  if {P1 M} then {E1 M}
                  elseif {P2 M} then {E2 M}
                        ...
                  elseif {Pn M} then {En M}
                  end
            end
      else {Timesup C}
      end
end
```

8 Limitations of Stream Communication

8.1 Is the given definition of NameServer, which stores pairs in an internal list referencing the names and input streams of various servers, a practical solution?

No, it is not. Besides the given reason, that it is not possible to name the name server, any call to the name server to register a new server requires the call to also specify the "internal" list of previously registered servers. Since it is the job of the name server to store the server list internally, the fact that everything interacting with it also needs to store the list makes the whole thing redundant.