Message Passing in Erlang: Class Design Exercise

N. Ramsey and D. Nunez, for COMP 250RTS

November 13, 2017

Introduction

The design and implementation of Erlang's message-passing mechanism seems to be governed by these considerations:

- Each Erlang process should have not only its own stack but also its own heap and its own garbage collector.
- Processes should have the opportunity to run in parallel.
- Memory should be managed with an eye to performance—controlling both the amount of copying and the amount of contention for shared mutable resources.
- If a process needs to acquire a lock, the amount of work done while holding the lock should be bounded above by a small constant.

Given these constraints, how do you implement asynchronous message passing? Until you've wrestled with this question, it's hard to make sense of the answer. So we'll spend a class exploring the design space.

Knowledge

Some things we know:

- Every message is sent to a mailbox, asynchronously.
- Senders may compete with other senders as well as with the owner of the mailbox.
- An owner accesses a mailbox only when evaluating a `receive` expression.
- If an owner blocks on `receive`, it stays blocked until the mailbox changes (ignoring timeouts).
- Only the sender knows the contents of a message.
- A message is an Erlang value and is therefore immutable.
- There is no *a priori* upper bound on the size of a message. In particular, a message might not fit in a machine word.
- Receipt of a message never causes an interrupt—an Erlang process chooses when it is ready to receive information. (It evaluates a `receive` expression.)

Some things that are easily discoverable in the BEAM book:

- As noted above, each Erlang process has its own heap and its own garbage collector.
- The default heap is 233 words—just 1.8KiB. Garbage-collection algorithms and data structures have to be suited to a small heap.
- Heaps can grow. It is not obvious where the memory for heap growth comes from—we conjecture that it comes from a central resource, to which access has to be serialized.

Questions

(1) When a message is sent, *who* can allocate the memory necessary to hold the representation of the message? Who can initialize that memory?

(2) *Where* could the memory for a message be allocated?

(3) The memory used for a message must eventually be reclaimed.

   (a) *When* do you want the memory to be reclaimed?
   (b) *How*?
   (c) What alternatives can you think of?
   (d) What rights and obligations might the garbage collector have with respect to memory that has been allocated to hold a message?

(4) When evaluating a `receive` expression, an Erlang process needs to look at its mailbox. Its view needs to be consistent even if other processes (senders) are contending for the mailbox.

   (a) How do you want to represent a mailbox?
   (b) How will you control concurrent access to or operations on a mailbox?
   (c) Now that we know who allocates the memory for a message and where the memory is allocated, how should send work?
   (d) How should `receive` work?
   (e) Suppose somebody tries to send me a message while I'm collecting garbage. How should that work?

(5) Selective receive is an important programming paradigm, and it would be nice if it worked efficiently even in the presence of large mailboxes. What ideas do you have for making selective receive work efficiently?