SELECTIVE RECEIVES

This 'flushing' concept makes it possible to implement a *selective receive* which can give a priority to the messages you receive by nesting calls:

```
important() ->
receive
{Priority, Message} when Priority > 10 ->
[Message | important()]
after 0 ->
normal()
end.

normal() ->
receive
{_, Message} ->
[Message | normal()]
after 0 ->
[]
end.
```

This function will build a list of all messages with those with a priority above 10 coming first:

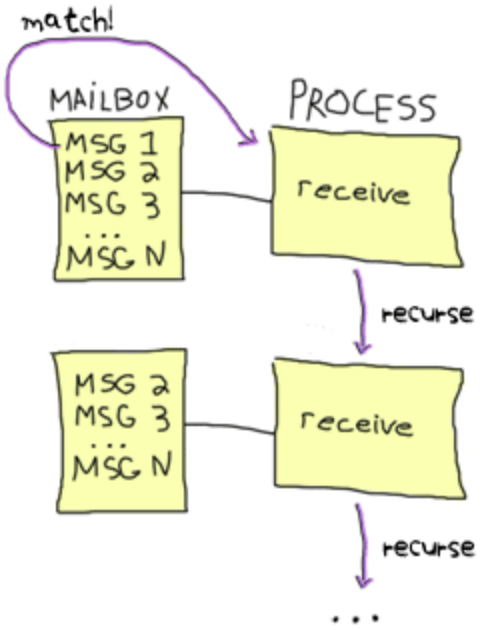
```
1> c(multiproc).
{ok,multiproc}
2> self() ! {15, high}, self() ! {7, low}, self() ! {1, low}, self() ! {17,high}.
{17,high}
3> multiproc:important().
[high,high,low,low]
```

Because I used the `after 0` bit, every message will be obtained until none is left, but the process will try to grab all those with a priority above 10 before even considering the other messages, which are accumulated in the `normal/0`call.

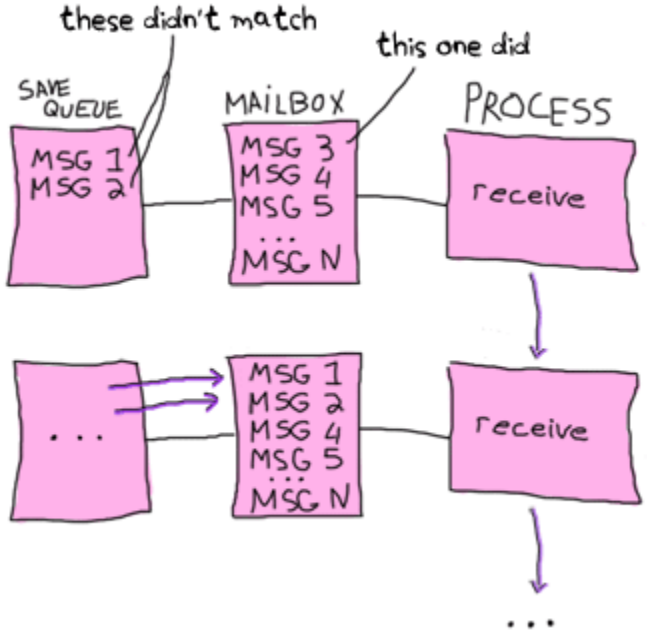
If this practice looks interesting, be aware that is is sometimes unsafe due to the way selective receives work in Erlang.

When messages are sent to a process, they're stored in the mailbox until the process reads them and they match a pattern there. As said in the previous chapter, the messages are stored in the order they were received. This means every time you match a message, it begins by the oldest one.

That oldest message is then tried against every pattern of the `receive` until one of them matches. When it does, the message is removed from the mailbox and the code for the process executes normally until the next `receive`. When this next `receive` is evaluated, the VM will look for the oldest message currently in the mailbox (the one after the one we removed), and so on.



When there is no way to match a given message, it is put in a *save queue* and the next message is tried. If the second message matches, the first message is put back on top of the mailbox to be retried later.



This lets you only care about the messages that are useful. Ignoring some messages to handle them later in the manner described above is the essence of *selective receives*. While they're useful, the problem with them is that if your process has a lot of messages you never care about, reading useful messages will actually take longer and longer (and the processes will grow in size too).

In the drawing above, imagine we want the 367th message, but the first 366 are junk ignored by our code. To get the 367th message, the process needs to try to match the 366 first ones. Once it's done and they've all

been put in the queue, the 367th message is taken out and the first 366 are put back on top of the mailbox. The next useful message could be burrowed much deeper and take even longer to be found.

This kind of receive is a frequent cause of performance problems in Erlang. If your application is running slow and you know there are lots of messages going around, this could be the cause.

If such selective receives are effectively causing a massive slowdown in your code, the first thing to do is to ask yourself is why you are getting messages you do not want. Are the messages sent to the right processes? Are the patterns correct? Are the messages formatted incorrectly? Are you using one process where there should be many? Answering one or many of these questions could solve your problem.

Because of the risks of having useless messages polluting a process' mailbox, Erlang programmers sometimes take a defensive measure against such events. A standard way to do it might look like this:

```
receive
Pattern1 -> Expression1;
Pattern2 -> Expression2;
Pattern3 -> Expression3;
...
PatternN -> ExpressionN;
Unexpected ->
io:format("unexpected message ~p~n", [Unexpected])
end.
```

What this does is make sure any message will match at least one clause. The *`Unexpected`* variable will match anything, take the unexpected message out of the mailbox and show a warning. Depending on your application, you might want to store the message into some kind of logging facility where you will be able to find information about it later on: if the messages are going to the wrong process, it'd be a shame to lose them for good and have a hard time finding why that other process doesn't receive what it should.

In the case you do need to work with a priority in your messages and can't use such a catch-all clause, a smarter way to do it would be to implement a min-heap or use the `gb_trees` module and dump every received message in it (make sure to put the priority number first in the key so it gets used for sorting the messages). Then you can just search for the `smallest` or `largest` element in the data structure according to your needs.

In most cases, this technique should let you receive messages with a priority more efficiently than selective receives. However, it could slow you down if most messages you receive have the highest priority possible. As usual, the trick is to profile and measure before optimizing.

Note: Since R14A, a new optimization has been added to Erlang's compiler. It simplifies selective receives in very specific cases of back-and-forth communications between processes. An example of such a function is `optimized/1` in multiproc.erl.
To make it work, a reference (`make_ref()`) has to be created in a function and then sent in a message. In the same function, a selective receive is then made. If no message can match unless it contains the same reference, the compiler automatically makes sure the VM will skip messages received before the creation of that reference.
Note that you shouldn't try to coerce your code to fit such optimizations. The Erlang developers only look for patterns that are frequently used and then make them faster. If you write idiomatic code, optimizations should come to you. Not the other way around.

Source : http://learnyousomeerlang.com/more-on-multiprocessing