

cl-muproc

Erlang-inspired multi-processing in Common LISP

overview

- background
- muproc goals, non-goals
- muproc fundamentals
 - concepts, abstractions, operators
- erlang & muproc side-by-side
- generic behaviors
 - generic server, generic supervisor
- conclusion, reflections
- questions, discussion

background

- building on-line betting platform
- lots of concurrent activities
- high reliability, resiliency required
- we want scalability
- we like Erlang's model of concurrency
- we believe CL is better suited overall

So we'll borrow what we need from Erlang!

Goals

- language embedded in CL which has/does:
 - message-passing
 - pattern matching on incoming messages
 - process linking, monitoring
 - good scalability
 - expandable to distributed operation

 - generic servers, supervisors

Non-Goals

- erlang features we're **NOT** aiming for include
 - immutable message objects
 - we're passing Lisp objects around – very efficient
 - treating them as immutable is good policy
 - very large number of threads
 - We rely on the implementation's process system
 - code upgrades on live systems
 - We do not need that (right now)

...and lots of other stuff

Erlang is a mature, full language – we're focusing on what we need to do the job!

Erlang at a glance

cf. http://www.erlang.org/white_paper.html

- concurrency
 - LOTS of lightweight processes
- distribution
- robustness
 - fail-and-restart
- soft real-time
- upgrades to running systems
- shared-nothing, message-passing
- functional

muproc fundamentals

- **muproc**

- a special LISP process (of the underlying implementation), extended with special run-time context:

- unique muproc name (distinct from LISP process name)
- single input port for incoming messages
- errorstream
- may be linked to one or more other muproc
- may or may not trap exits
- set of initial special bindings
- set of registered port names
- managed by the muproc run-time system

muproc fundamentals

- linking
 - core concept
 - two-way relationship between muproc
 - If one terminates, the other is notified
 - default: terminate
 - When receiving notification of termination of linked muproc
 - trapping
 - Get special message when linked muproc terminates
 - Message: Which muproc terminated, Reason
- monitoring
 - like linking, but one-way

muproc fundamentals

- **mumsg**
 - container object
 - used with **mumsg-send**, **mumsg-receive**
 - basis for pattern matching on messages
 - consists of a set named values

```
(mumsg :quantity 3 :price 42)
```

muproc fundamentals

- **mumsg-send dest &rest plist**
 - **dest** is muproc or named port
 - **plist** is used to build a **mumsg**

```
(mumsg-send :logger :msg "Error.")
```

muproc fundamentals

- **mumsg-receive (from) clause***
 - will look for message on input port matching **clause***
 - If match, binds **from** sender of message
 - a clause consists of
 - a list of names which must be present in the message
 - a guard predicate which must evaluate to a non-NIL value to match a message
 - a list of forms which will be evaluated iff the message contains the name list and the guard evaluated to a non-NIL value

muproc-receive

- **mumsg-receive (cont' d)**
 - messages are matched in order
 - clauses are matched in order
 - blocks if there are no matching messages
 - consumes matched message
 - you don't want side-effects in guards!

```
(mumsg-receive (from)
  ((counter) (and (integerp counter)
                  (oddp counter))
   (printout "Counter is an odd integer."))
  ((counter) t
   (printout "Counter is not an odd integer."))
  (()) t
  (printout "Unexpected message."))
```

muproc fundamentals

- other major muproc operators
 - `muproc-spawn`
 - `muproc-link`, `muproc-monitor`
 - `muproc-with-registered-port`
 - `muproc-schedule`
 - `muproc-interrupt`
 - `muproc-set-trap-exits`
 - `muproc-with-timeout`
 - `muproc-with-message-tag`
 - `muproc-exit`, `muproc-kill`
 - `muproc``n`

examples

erlang & muproc side-by-side

cf. http://www.erlang.org/white_paper.html

example 6: Area Server, erlang

```
-module(area_server).  
-export([start/1, loop/1]).  
  
start() ->  
    spawn(area_server, loop, [0]).  
  
loop(Tot) ->  
    receive  
        {Client, {square, X}} ->  
            Client ! X*X,  
            loop(Tot + X*X);  
        {Client, {rectangle, X, Y}} ->  
            Client ! X*Y,  
            loop(Tot + X*Y);  
        {Client, areas} ->  
            Client ! Tot,  
            loop(Tot)  
    end.
```

example 6 - Area Server, muproc

```
(defun area-server ()
  (let ((total 0))
    (loop
      (mumsg-receive (from)
        ((cmd x) (eq cmd :square)
          (mumsg-send from :answer (incf total (* x x))))
        ((cmd x y) (eq cmd :rectangle)
          (mumsg-send from :answer (incf total (* x y))))
        ((cmd) (eq cmd :areas)
          (mumsg-send from :answer total))))))

(defvar *area-server*)

(defun start-area-server ()
  (setf *area-server*
    (muproc-spawn 'area-server #'area-server ()
      :errorstream *trace-output*)))
```

example 7: Area Client

erlang

```
Server ! {self(), {square, 10}},  
        receive  
            Area ->  
                Area  
        end
```

muproc

```
(progn  
  (mumsg-send *area-server* :cmd :square :x 10)  
  (mumsg-receive (from)  
    ((answer) t answer)))
```

example 8: Named Server

erlang

```
Pid = spawn(Module, Fun, Args),  
register(area_server, Pid)
```

Use: `area_server ! SomeMessage`

muproc

```
(muproc-register-port-name :area-server)  
...  
(muproc-unregister-port-name :area-server))
```

-or-

```
(muproc-with-registered-port (:area-server)  
...)
```

Use: `(mumsg-send :area-server :cmd :areas)`

example 9: Spawn on Remote Node

```
-module(ex9).
  -export([start/0, world/0]).

start() ->
  Pid = spawn('b@host2', ex9, world, []),
  Pid ! {self(), 'hello world!'},
  receive
    Msg ->
      io:format('~w~n', [Msg])
  end.

world() ->
  receive
    {Pid, 'hello world!'} ->
      io:format(user, 'hello world!~n', []),
      Pid ! 'hello, little thread'
  end.
```

This cannot be done in current muproc

...but the design allows for it to be implemented later

example 12: Links, Trapping Exits

```
erlang process_flag(trap_exit, true),
Pid = spawn_link(Mod, Fun, Args),
receive
    {'EXIT', Pid, Why} ->
        handle_process_crash(Why, ...);
    ...
end
```

```
muproc (muproc-set-trap-exits t)
(muproc-spawn ...
            :link t)
...
(mumsg-receive (from)
    ((terminated reason) t
    ...))
```

muproc linking & trapping I

```
(defun child (max)
  (unwind-protect
    (let ((period (random max)))
      (sleep period)
      (muproc-exit (cons :finished-after period)))
    (out "~s exiting." (muproc-current-process))))

(defun parent (count max)
  (loop repeat count
    do (muproc-spawn (gensym) #'child (list max)
                   :link t))

  (loop repeat count
    do (mumsg-receive (from)
        ((terminated reason) t
         (out "PARENT -- ~a terminated with reason ~a."
              terminated reason))))
  (out "PARENT -- terminating."))

(defun start-parent (count max)
  (muproc-spawn: 'parent #'parent (list count max)
                :trap-exits t
                :errorstream *trace-output*))
```

muproc linking & trapping I – output

```
DKCLUG-MUPROC> (start-parent 3 10)
#<MP:PROCESS Name "muproc-4063[G13907]" Priority 0 State "Running"> exiting.
PARENT -- #<MP:PROCESS Name :DEAD-PROCESS Priority 0 State "Dead">
          terminated with reason (FINISHED-AFTER . 2).
#<MP:PROCESS Name "muproc-4065[G13909]" Priority 0 State "Running"> exiting.
PARENT -- #<MP:PROCESS Name :DEAD-PROCESS Priority 0 State "Dead">
          terminated with reason (FINISHED-AFTER . 7).
#<MP:PROCESS Name "muproc-4064[G13908]" Priority 0 State "Running"> exiting.
PARENT -- #<MP:PROCESS Name :DEAD-PROCESS Priority 0 State "Dead">
          terminated with reason (FINISHED-AFTER . 8).
PARENT -- terminating.
DKCLUG-MUPROC>
```

muproc linking & trapping II

```
(defun parent2 (count max)
  (loop repeat count
    do (muproc-spawn (gensym) #'child (list max)
                   :link t))
  ;; No loop - exit on first 'child' termination
  (mumsg-receive (from)
    ((terminated reason) t
     (out "PARENT -- ~a terminated with reason ~a."
          terminated reason)))
  (out "PARENT -- terminating."))

(defun start-parent2 (count max)
  (muproc-spawn 'parent2 #'parent2 (list count max)
               :trap-exits t
               :errorstream *trace-output*))
```

muproc linking & trapping II – output

```
DKCLUG-MUPROC> (start-parent2 3 10)
#<MP:PROCESS Name "muproc-4070[PARENT2]" Priority 850000 State "Running">
#<MP:PROCESS Name "muproc-4071[G14139]" Priority 0 State "Running"> exiting.
PARENT -- #<MP:PROCESS Name :DEAD-PROCESS Priority 0 State "Dead">
           terminated with reason (FINISHED-AFTER . 3).
PARENT -- terminating.
#<MP:PROCESS Name "muproc-4073[G14141]" Priority 0 State "Running"> exiting.
#<MP:PROCESS Name "muproc-4072[G14140]" Priority 0 State "Running"> exiting.
DKCLUG-MUPROC>
```

Linked muproc terminate together

example 13: Process Supervision

```
-module(ex13).  
-export([start/0, loop/0]).  
  
start() ->  
    process_flag(trap_exit, true),  
    parent(Child, 5),  
    Child = spawn(ex13, child, []).  
  
parent(Child, K) ->  
    receive  
        {'EXIT', Child, Why} when K > 0 ->  
            io:format('child died with reason ~p~n', [Why]),  
            NewChild = spawn_link(ex13, child, []),  
            parent(NewChild, K-1);  
        {'EXIT', Child, _} ->  
            io:format('too many restarts, bye~n', [])  
    end.  
  
child() -> exit(died).
```

example 13: Process Supervision

```
(defun supervision-child ()
  (muproc-exit :done))

(defun supervisor-parent (count)
  (flet ((spawn-one ()
          (muproc-spawn 'child-one #'supervision-child ()
                        :link t)))
    (spawn-one)
    (loop
     (mumsg-receive (from)
      ((terminated reason) (> count 0)
       ;; side-effects in a guard is a bad idea
       (decf count)
       (out "SUPERVISOR-PARENT -- ~a terminated." terminated)
       (spawn-one))
      ((terminated reason) t
       (out "SUPERVISOR-PARENT -- Too many restarts.")
       (muproc-exit :too-many-restarts))))))

(defun supervisor-start (count)
  (muproc-spawn 'supervisor #'supervisor-parent (list count)
               :trap-exits t
               :errorstream *trace-output*))
```

'ping-pong', erlang

```
-module(tut15).
-export([start/0, ping/2, pong/0]).

ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished~n", []);

ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, Pong_PID).

pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            io:format("Pong received ping~n", []),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    Pong_PID = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_PID]).
```

```
1> tut15: start().
<0.36.0>
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
ping finished
Pong finished
2>
```

'ping-pong', muproc

```
(defun pong ()
  (loop
    (mumsg-receive (from)
      ((ping) t
        (out "PONG -- Got ping: ~a." ping)
        (mumsg-send from :pong (* 100 ping)))
      ((done) t
        (out "PONG -- Got done.")
        (muproc-exit :done))))))

(defun ping (pong count)
  (loop for iter from 1 to count
    do (mumsg-send pong :ping iter)
    do (mumsg-receive (from)
      ((pong) t
        (out "PING -- Got pong: ~a." pong))))
  (mumsg-send pong :done :dummy))

(defun ping-pong (count)
  (let ((pong (muproc-spawn 'pong #'pong ()
    :errorstream *trace-output*)))
    (muproc-spawn 'ping #'ping (list pong count)
      :errorstream *trace-output*)))
```

```
DKCLUG-MUPROC> (ping-pong 3)
PONG -- Got ping: 1.
PING -- Got pong: 100.
PONG -- Got ping: 2.
PING -- Got pong: 200.
PONG -- Got ping: 3.
PING -- Got pong: 300.
PONG -- Got done.
DKCLUG-MUPROC>
```

Status - Erlang in 14 Examples

#	Example	Comment	CL+muproc
1	Factorial	Basic Erlang, CL equivalents exist	√
2	Last	Ditto	√
3	Append	Ditto	√
4	Sort	Ditto	√
5	Adder, Binary tree	Erlang higher-order functions, ditto	√
6	Area Server	Shown	√
7	Area Client	Shown	√
8	Named Server	Shown	√
9	Spawn on Remote Node	Not implemented in muproc	⊗
10	Catch	Basic Erlang, CL equivalents exist	√
11	Catch and Throw	Ditto	√
12	Links and Trapping Exits	Shown	√
13	Process Supervision	Shown	√
14	Hot code upgrade	Not implemented in muproc	⊗

The above comparison is *not* meant to suggest that cl-muproc and Erlang are comparable in general.
Examples from http://www.erlang.org/white_paper.html

a step further – generic behaviours

generic server
generic supervisor

early muproc idiom

```
(defun %handle-msg-type1 (from arg1 arg2 arg3 tag)
  ...
  (mumsg-send from ... :tag tag))

(defun %handle-msg-type2 (from arg1 arg2 tag)
  ...
  (mumsg-send from ... :tag tag))

(defun some-server (...)
  ...
  (loop
   ...
   (muproc-receive (from)
    ((msgtype arg1 arg2 arg3 tag) (and (eq msgtype :type1) ...))
    (%handle-msg-type1 from arg1 arg2 arg3 tag))
    ((msgtype arg1 arg2 tag) (and (eq msgtype :type2) ...))
    (%handle-msg-type2 from arg1 arg2 tag))
    (...))
  ...))
```

Cumbersome in the long run!

generic server

- generate message handling code from spec
 - concept / idea from Erlang/OTP
- 'synchronous' –vs- 'asynchronous'
 - **muproc-define-CALL-handler**
 - sync, caller blocks until return value message received
 - **muproc-define-CAST-handler**
 - async, no return value, caller proceeds immediately
- multiple instances of same type of server
 - **muproc-default-server-name**
 - two I/F functions – selecting default server or not
- defined server life cycle
 - explicit server state, managed by generic code
 - **generic-server-start**
 - **initialize, terminate** call-backs
 - **muproc-exit**
 - works as expected
 - **muproc-exit-after-handler**
 - finish handling this request, then terminate

generic server

```
(in-package :dkclug-muproc)

(muproc-default-server-name generic-area-server)

(defclass area-server-state ()
  ((count :initform 0
          :accessor area-server-count)))

(defun initialize ()
  (muproc-log-errorstream "GENERIC AREA SERVER initializing.")
  (make-instance 'area-server-state))

(defun terminate (state)
  (muproc-log-errorstream
   "GENERIC AREA SERVER terminating (count=~a)."
   (area-server-count state)))

(muproc-define-CALL-handler generic-add-square (state from x)
  (incf (area-server-count state) (* x x)))

(muproc-define-CALL-handler generic-add-rectangle (state from x y)
  (incf (area-server-count state) (* x y)))

(muproc-define-CALL-handler generic-get-areas (state from)
  (area-server-count state))

(defun start-generic-area-server ()
  (muproc ()
    (muproc-spawn 'generic-area-server
                  (lambda () (muproc-generic-start))
                  nil)))
```

using generic-area-server

```
DKCLUG-MUPROC> (start-generic-area-server)
12:22:18.072 GENERIC-AREA-SERVER GENERIC AREA SERVER initializing.
#<MP:PROCESS Name "muproc-94[GENERIC-AREA-SERVER]" Priority 0 State :TIMEOUT>
DKCLUG-MUPROC> (muproc ()
                 (generic-add-square 10))
100
DKCLUG-MUPROC> (muproc ()
                 (generic-get-areas))
100
DKCLUG-MUPROC> (muproc ()
                 (generic-get-areas* 'generic-area-server))
100
DKCLUG-MUPROC> (muproc ()
                 (generic-add-rectangle 10 30))
400
DKCLUG-MUPROC>
```

Note

- `muproc` is muproc's version of `progn`
- it creates a muproc and evaluates forms in it
- it is necessary here because our client needs a complete muproc context to talk to the generic server
- under 'normal' circumstances, everything is in muproc, so `muproc` is not needed
- `muproc` is used a lot when *developing* muproc code, to interactively talk to the muproc

generic server

```
(muproc-define-CALL-handler generic-add-square (state from x)
  (incf (area-server-count state) (* x x)))

(PROGN
  (EXPORT 'GENERIC-ADD-SQUARE)
  (EXPORT 'GENERIC-ADD-SQUARE*)
  (DEFUN GENERIC-ADD-SQUARE (X)
    (MUPROC.GENERIC-SERVER::DBGFORMAT 4
      "Calling ~s, ref=~s"
      'GENERIC-ADD-SQUARE
      *DEFAULT-SERVER-NAME*)
    (MUPROC-GENERIC-CALL *DEFAULT-SERVER-NAME*
      'CALL-HANDLER/GENERIC-ADD-SQUARE
      X))
  (DEFUN GENERIC-ADD-SQUARE* (MUPROC.GENERIC-SERVER::SERVER-REF X)
    (MUPROC.GENERIC-SERVER::DBGFORMAT 4
      "Calling ~s, ref=~s"
      'GENERIC-ADD-SQUARE
      MUPROC.GENERIC-SERVER::SERVER-REF)
    (MUPROC-GENERIC-CALL MUPROC.GENERIC-SERVER::SERVER-REF
      'CALL-HANDLER/GENERIC-ADD-SQUARE
      X))
  (DEFUN CALL-HANDLER/GENERIC-ADD-SQUARE (STATE FROM X)
    (DECLARE (IGNORABLE STATE FROM))
    (INCF (AREA-SERVER-COUNT STATE) (* X X))))
```

front-
end

back-
end

This goes beyond what is available in Erlang.

generic supervisor

- monitor muproc and restart them if they fail
 - idea from Erlang/OTP
- supervisors link to their children
 - when a child terminates, its supervisor may restart it
- child specifications
 - the supervisor needs to know *how* to (re-)start children
- restart strategies
 - all-for-one, one-for-one
 - maximum restart frequency
 - permanent, transient, temporary
- termination protocol
 - `:kill`, `:wait-forever`, *period-to-wait*

erlang supervisor example

```
-module(simple_sup).
-behaviour(supervisor).

-export([start/0, init/1]).

start() ->
    supervisor:start_link({local, simple_supervisor},
                          ?MODULE, nil).

init(_) ->
    {ok,
     {{one_for_one, 5, 1000},
      [
        {packet,
         {packet_assembler, start, []},
         permanent, 500, worker, [packet_assembler]}},
        {server,
         {kv, start, []},
         permanent, 500, worker, [kv]}},
        {logger,
         {simple_logger, start, []},
         permanent, 500, worker, [simple_logger]}}}}.
```

Source – Joe Armstrong: *"Making reliable distributed systems in the presence of software errors"*, p.148

muproc supervisor example I

```
(defun proc1 ()
  (muproc-log-errorstream "PROC1 starting")
  (unwind-protect
    (loop
      (sleep 10)
      (muproc-log-errorstream "PROC1 HERE"))
    (muproc-log-errorstream "PROC1 terminating"))))

(defun proc2 ()
  (muproc-log-errorstream "PROC2 starting")
  (unwind-protect
    (loop
      (sleep 8)
      (muproc-log-errorstream "PROC2 HERE"))
    (muproc-log-errorstream "PROC2 terminating"))))

(defun proc31 ()
  (muproc-log-errorstream "PROC31 starting")
  (unwind-protect
    (loop
      (sleep 6)
      (muproc-log-errorstream "PROC31 HERE"))
    (muproc-log-errorstream "PROC31 terminating"))))
```

These just periodically print stuff.

muproc supervisor example II

```
(defun proc32 ()
  (muproc-set-trap-exits t)
  (muproc-log-errorstream "PROC32 starting")
  (muproc-schedule-timer-relative
   (muproc-make-interrupt-timer (lambda ()
                                   (mumsg-send *muproc-inport* :wakeup nil)))
   0 4)
  (muproc-log-errorstream "PROC32 HERE")
  (unwind-protect
   (loop
    (mumsg-receive (from)
     ((terminated reason) t
      (muproc-log-errorstream "Got SHUTDOWN")
      (sleep 3)
      (muproc-exit :shutting-down-myself))
     (wakeup) t
      (muproc-log-errorstream "Got WAKEUP"))
    (()) t
      (muproc-log-errorstream "Got unknown package ~s."
                              *muproc-packet*))))
  (muproc-log-errorstream "PROC32 terminating"))
```

PROC32 traps exits and takes a while to terminate

muproc supervisor example III

```
(defun start-super1 (&optional (errorstream *muproc-errorstream*))
  (supervisor-start
    (supervisor
      :all-for-one 2 5
      (:permanent :kill (supervised-muproc proc1 #'proc1 ()))
      (:permanent 10 (supervised-muproc proc2 #'proc2 ()))
      (:permanent 5
        (supervisor
          :all-for-one 2 5
          (:permanent :kill (supervised-muproc proc31 #'proc31 ()))
          (:permanent 5 (supervised-muproc proc32 #'proc32 ())))
        ))
      :errorstream errorstream))
```

Supervisor specification:

- no more than 2 restarts per 5 seconds
- restart children if one child terminates
- both kill and waiting termination methods

supervisor transcript I

```
DKCLUG-MUPROC> (start-super1 *trace-output*)
13:45:43.663 PROC1 PROC1 starting
13:45:43.664 PROC2 PROC2 starting
13:45:43.669 PROC31 PROC31 starting
13:45:43.670 PROC32 PROC32 starting
13:45:43.671 PROC32 PROC32 HERE
13:45:43.672 PROC32 Got WAKEUP
13:45:47.671 PROC32 Got WAKEUP
13:45:49.671 PROC31 PROC31 HERE
13:45:51.665 PROC2 PROC2 HERE
...
```

supervisor transcript II

```
DKCLUG-MUPROC> (muproc-exit :adieu (muproc-find 'PROC1))
13:45:57.131 PROC1 PROC1 terminating
13:45:57.132 PROC2 PROC2 terminating
13:45:57.149 PROC31 PROC31 terminating
13:45:57.150 PROC31 PROC31 was terminated due to #<EXIT PROC31 :SUPERVISOR-KILL>.
13:45:57.153 PROC32 Got SHUTDOWN
13:46:00.154 PROC32 PROC32 terminating
13:46:00.155 {PROC31*PROC32} SUPERVISOR #<SUPERVISOR {PROC31*PROC32} ALL-FOR-ONE:2:5
      DOWN:DOWN STABLE> terminating.
13:46:00.158 PROC1 PROC1 starting
13:46:00.158 PROC2 PROC2 starting
13:46:00.166 PROC31 PROC31 starting
13:46:00.167 PROC32 PROC32 starting
13:46:00.167 PROC32 PROC32 HERE
13:46:00.168 PROC32 Got WAKEUP
13:46:04.168 PROC32 Got WAKEUP
13:46:06.166 PROC31 PROC31 HERE
13:46:08.165 PROC2 PROC2 HERE
13:46:08.168 PROC32 Got WAKEUP
13:46:10.158 PROC1 PROC1 HERE
```

supervisor transcript III

```
DKCLUG-MUPROC> (muproc-exit :adieu (muproc-find 'PROC32))
13:46:23.074 PROC32 Got SHUTDOWN
13:46:24.167 PROC2 PROC2 HERE
13:46:26.074 PROC32 PROC32 terminating
13:46:26.076 PROC31 PROC31 terminating
13:46:26.077 PROC31 PROC31 was terminated due to #<EXIT PROC31 :SUPERVISOR-KILL>.
13:46:26.079 PROC31 PROC31 starting
13:46:26.079 PROC32 PROC32 starting
13:46:26.080 PROC32 PROC32 HERE
13:46:26.080 PROC32 Got WAKEUP
...
```

supervisor transcript IV

```
DKCLUG-MUPROC> (mapcar (lambda (p) (muproc-exit :adieu p)) (muproc-all-processes))
13:46:27.218 PROC1 PROC1 terminating
13:46:27.239 PROC2 PROC2 terminating
13:46:27.252 PROC31 PROC31 terminating
13:46:27.254 PROC32 Got SHUTDOWN
13:46:30.257 PROC32 PROC32 terminating
13:46:30.261 {PROC31*PROC32} SUPERVISOR #<SUPERVISOR {PROC31*PROC32}
      ALL-FOR-ONE:2:5 DOWN:DOWN STABLE> terminating.
13:46:30.262 {PROC1*PROC2*{PROC31*PROC32}} SUPERVISOR #<SUPERVISOR
      {PROC1*PROC2*{PROC31*PROC32}} ALL-FOR-ONE:2:5 DOWN:DOWN STABLE> terminating.
(#<MP:PROCESS Name "muproc-127[PROC1]" Priority 850000 State "Sleeping on mailbox">
 #<MP:PROCESS Name "muproc-128[PROC2]" Priority 850000 State "Sleeping on mailbox">
 #<MP:PROCESS Name "muproc-129[{PROC31*PROC32}]" Priority 850000 State :TIMEOUT>
 #<MP:PROCESS Name "muproc-134[PROC31]" Priority 850000 State "Sleeping on mailbox">
 #<MP:PROCESS Name "muproc-135[PROC32]" Priority 850000 State :TIMEOUT>
 #<MP:PROCESS Name "muproc-121[{PROC1*PROC2*{PROC31*PROC32}}]"
      Priority 850000 State :TIMEOUT>)
DKCLUG-MUPROC>
```

experiences so far

- muproc used in on-line betting project at Mu
 - 14 muprocs total
 - 9 are derived from Generic Server
 - 4 are Supervisors
 - supervision tree 3 levels deep
 - 3 'all-for-one', 1 'one-for-one'
 - 1 'basic' muproc
 - generic servers are a huge win
 - supervisors introduced recently
- muprocs have performed well in project
 - clear way to structure a system
 - strong separation of concerns
 - you can contribute w/o global understanding of the system

winding down

- feature 'integration'
 - erlang is a 'whole', an integrated set of features
 - muproc must acknowledge its own feature set, mesh well with CL
 - we cannot / should not simply adopt erlang idioms *en bloc*
 - eg. when to spawn a task to do accomplish a task?
- performance
 - should perform really well on multi-cpu/multi-core HW
 - no locking except at muproc creation & termination
 - process count ('few hundreds' (?)), creation time ('few ms')
 - message switching speed (circa 200K/s)
 - could probably be improved significantly, 'no optimizations'
 - msg switching speed is not a limiting factor in our current use of muproc
 - distribution
 - print/read ? efficiency, performance, correctness
- essentially we haven't really explored muproc yet