Deducing a better language from a driver implementation

Maxim Kharchenko, Cloudozer LLP

09/08/2014

1 Overview

Here is the plan.

- Add a module named `membrane` which manages all `pores`.
- The membrane should handle all negotiations about the pores.
- Make pore polymorphic and active, similar to outlets.
- A pore sends a message to its controller process a soon as it has data available
- Add a BIF to query a pore for the number of slots available for the output. The same of abother BIF may receive a request when certain number of output slots are there. The notification about slots delivered as a message.
- An event channels are not exposed by pores. It is possible for a pore to use two (or more) event channels (netmap).

2 Notes

Two drivers strawman and xenstore. Strawman uses xenstore.

Both drivers use shared pages and an event channel.

The combination of a set of shared pages and an event channel abstracted out as a 'pore'.

An event channel is a Xen primitive.

There is a hypercall to 'bind' an event channel. The event is signalled as a bit in the shared info page.

It is possible to sleep until an event channel is signalled.

Xenstore driver have its shared page and event channel appear magically. Strawman uses xenstore driver to communicate the shared pages and event channel between backend and frontends.

The xenstore driver has only the frontend. Its backend leaves in Dom0.

[Consider strawman only and treat xenstore as a Xen primitive?]

3 Event channel

The event is only a bit flip. The event deliver requires either an interrrupt or an ability to wait for an event.

Interrupts are bad because everything is in unknown state during an interrupt. The interrupt handler cannot do much. The ability to wait for event means that we may have suspended execution contexts (green threads). In addition we need a uniform way to wait for all events and react to events selectively. Do we want to react to a conjunction of events?

4 Xenstore

Xenstore is thought as a part of Xen, the lowest level infrastructure.

Xenstore uses strings as representation of keys and values. The language must be able to create a concatenation of strings. Some of the strings may represent integer variables.

Xenstore has a scarce resource - watches. We need to keep track of watches we open and `unwatch` them when they are not needed.

Traditionally, Xenstore is human-readable.

The standard approach to negotiation between front- and backend of a driver is to go through a fixed series of steps: INITIALISING, INIT_WAIT, INITIALISED, CONNECTED, CLOSING, CLOSED.

5 Strawman architecture

A process that listens on xenstore and maintains a list of straws.

TODO

6 Driver scaffolding

We may represent a lot of setup/teardown functionality of a driver using a library (a behaviour?).

7 What is bad about the code?

1. Too much boilerplate, related an general driver not this particular one.
2. Only one event channel per driver (netmap uses two).
3. Shared page is not a builtin data type (pore).
4. Each driver uses a new set of BIFs to talk to its particular type of pore.
5. Input and output buffers are lists or binaries. We have to keep the data size separately. There is no way to cleanly chip from the beginning of the buffer.

8 Appendix

```
1  -module(strawman).
2  -behaviour(gen_server).
3  -define(SERVER, ?MODULE).
4  -export([short_straw/3,short_straw/5]).
```

```
 5
 6  -include("xenstore.hrl").
 7
 8  -define(NUM_STRAW_REFS, 8).
 9
10  %% --------------------------------------------------------------------
11  %% API Function Exports
12  %% --------------------------------------------------------------------
13
14  -export([start_link/0]).
15  -export([open/1]).
16  -export([split/1]).
17
18  %% --------------------------------------------------------------------
19  %% gen_server Function Exports
20  %% --------------------------------------------------------------------
21
22  -export([init/1, handle_call/3, handle_cast/2, handle_info/2,
23           terminate/2, code_change/3]).
24
25  %% --------------------------------------------------------------------
26  %% API Function Definitions
27  %% --------------------------------------------------------------------
28
29  start_link() ->
30      gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).
31
32  open(Domid) ->
33      gen_server:call(?SERVER, {open,Domid}).
34
35  split(Domid) ->
36      gen_server:call(?SERVER, {split,Domid}).
37
38  %% --------------------------------------------------------------------
39  %% gen_server Function Definitions
40  %% --------------------------------------------------------------------
41
42  -record(sm, {top,straws =[]}).
43
44  init(_Args) ->
45      process_flag(trap_exit, true),
46      Me = xenstore:domid(),
47      StrawTop = "data/straw",
48      ok = xenstore:mkdir(StrawTop),
49      ok = xenstore:set_perms(StrawTop, [lc("b", Me)]),
50      WartsTop = "data/warts",
51      ok = xenstore:mkdir(WartsTop),
52      ok = xenstore:set_perms(WartsTop, [lc("r", Me)]),
53      ok = xenstore:watch(StrawTop),
```

```
54      {ok,#sm{top =StrawTop}}.
55
56  handle_call({open,Domid}, _From, St) ->
57      Me = xenstore:domid(),
58      WartsDir = lc(["/local/domain/",Me,"/data/warts/",Domid]),
59      StrawDir = lc(["/local/domain/",Domid,"/data/straw/",Me]),
60      case xenstore:read(WartsDir) of
61          {ok,_}    -> {reply,{error,exists},St};
62          {error,_} ->
63              case xenstore:mkdir(StrawDir) of
64                  ok -> do_open(Domid, WartsDir, StrawDir, St);
65                  _  -> {reply,{error,not_found},St} end end;
66
67  handle_call({split,Domid}, _From, #sm{straws =Straws} =St) ->
68      case lists:keyfind(Domid, 2, Straws) of
69          {_,_,StrawProc,_,_} -> {reply,{ok,StrawProc},St};
70          false -> {reply,{error,not_found},St} end.
71
72  handle_cast(_Msg, State) ->
73      {noreply, State}.
74
75  handle_info({watch,WatchKey}, #sm{top =StrawTop} =St) ->
76      case lists:prefix(StrawTop, WatchKey) of
77          true  -> Suffix = lists:nthtail(length(StrawTop), WatchKey),
78                   case string:tokens(Suffix, "/") of
79                      [X,"warts"] ->
80                          %% peer wants to communicate
81                          {ok,WartsDir} = xenstore:read(WatchKey),
82                          Domid = list_to_integer(X),
83                          knock_knock(Domid, WartsDir, lc([StrawTop,"/",X]), St);
84                      _ -> {noreply,St} end;
85          false -> straw_state(WatchKey, St) end;
86
87  handle_info({'EXIT',_,peer_closed}, St) -> {noreply,St};
88  handle_info(Msg, St) ->
89      io:format("strawman: info ~p\n", [Msg]),
90      {noreply,St}.
91
92  terminate(shutdown, #sm{straws =Straws}) ->
93      ok = close_straws(Straws).
94
95  code_change(_OldVsn, St, _Extra) -> {ok,St}.
96
97  %% --------------------------------------------------------------------
98  %% Internal Function Definitions
99  %% --------------------------------------------------------------------
100
101 do_open(Domid, WartsDir, StrawDir, #sm{straws =Straws} =St) ->
102     %% StrawDir exists, WartsDir does not
```

```
103     {ok,Tid} = xenstore:transaction(),
104     ok = xenstore:mkdir(WartsDir, Tid),
105     ok = xenstore:write(lc(WartsDir, "/straw"), StrawDir, Tid),
106     ok = xenstore:write(lc(WartsDir, "/state"), ?STATE_INIT_WAIT, Tid),
107     ok = xenstore:write(lc(StrawDir, "/warts"), WartsDir, Tid), %% wakes up peer
108     ok = xenstore:commit(Tid),
109     StrawState = lc(StrawDir, "/state"),
110     ok = xenstore:watch(StrawState),
111     case xenstore:wait(StrawState, ?STATE_INITIALISED) of
112         {error,_} =Error -> %% peer gone
113             ok = xenstore:delete(WartsDir),
114             ok = xenstore:unwatch(StrawState),
115             {reply,Error,St};
116         ok ->
117             Refs =
118             lists:map(fun(N) -> {ok,Ref} = xenstore:read_integer(lc([StrawDir,"/ring-ref
119                                 Ref end, lists:seq(1, ?NUM_STRAW_REFS)),
120             {ok,Channel} = xenstore:read_integer(lc(StrawDir, "/event-channel")),
121       Format = select_format(StrawDir, WartsDir),
122             StrawProc = spawn_link(?MODULE, short_straw, [self(),Domid,Refs,Channel,Forma
123             receive {ready,StrawProc} -> ok end,
124             ok = xenstore:write(lc(WartsDir, "/state"), ?STATE_CONNECTED),
125             case xenstore:wait(StrawState, ?STATE_CONNECTED) of
126                 {error,_} =Error ->
127                     ok = xenstore:delete(WartsDir),
128                     ok = xenstore:unwatch(StrawState),
129                     exit(StrawProc, peer_closed),
130                     {reply,Error,St};
131                 ok ->
132                     %% StrawState is being watched
133                     SI = {passive,Domid,StrawProc,StrawState,WartsDir},
134                     {reply,ok,St#sm{straws =[SI|Straws]}} end end.
135
136 knock_knock(Domid, WartsDir, StrawDir, #sm{straws =Straws} =St) ->
137     StrawState = lc(StrawDir, "/state"),
138     ok = xenstore:write(StrawState, ?STATE_INITIALISING),
139     WartsState = lc(WartsDir, "/state"),
140     ok = xenstore:watch(WartsState),
141     case xenstore:wait(WartsState, ?STATE_INIT_WAIT) of
142         {error,_} ->
143             ok = xenstore:delete(StrawDir),
144             ok = xenstore:unwatch(WartsState),
145             {noreply,St};
146         ok ->
147       Format = select_format(StrawDir, WartsDir),
148             StrawProc = spawn_link(?MODULE, short_straw, [self(),Domid,Format]),
149             receive {ready,StrawProc,Refs,Channel} -> ok end,
150             {ok,Tid} = xenstore:transaction(),
151             lists:foreach(fun({N,Ref}) -> ok = xenstore:write(lc([StrawDir,"/ring-ref-",
```

```
152                             lists:zip(lists:seq(1, ?NUM_STRAW_REFS), Refs)),
153             ok = xenstore:write(lc(StrawDir, "/event-channel"), Channel, Tid),
154             ok = xenstore:write(StrawState, ?STATE_INITIALISED, Tid),
155             ok = xenstore:commit(Tid),
156             ok = xenstore:wait(WartsState, ?STATE_CONNECTED),
157             ok = xenstore:write(StrawState, ?STATE_CONNECTED),
158             SI = {active,Domid,StrawProc,WartsState,StrawDir},
159             St1 = St#sm{straws = [SI|Straws]},
160             {noreply,St1} end.
161
162 %%-------------------------------------
163 %% Active                   Passive
164 %% ======                   =======
165 %% state=CLOSING
166 %%                          unmap refs
167 %%                          state=CLOSED
168 %%                          wait=CLOSED
169 %% wait=CLOSED
170 %% end access to refs
171 %% state=CLOSED
172 %%-------------------------------------
173 %% Active                   Passive
174 %% ======                   =======
175 %%                          unmap refs
176 %%                          state=CLOSED
177 %%                          wait=CLOSED
178 %% wait=CLOSED
179 %%-------------------------------------
180
181 straw_state(WatchKey, #sm{straws =Straws} =St) ->
182     SI = lists:keyfind(WatchKey, 4, Straws),
183     straw_state1(SI, St).
184
185 straw_state1(false, St) -> {noreply,St};
186 straw_state1({_,_,_,StatePath,_} =SI, St) ->
187     straw_state1(xenstore:read(StatePath), SI, St).
188
189 straw_state1({ok,?STATE_CONNECTED}, _, St) -> {noreply,St};
190 straw_state1({ok,_}, {active,_,_,_,_}, St) -> {noreply,St}; %% see chart above
191 straw_state1(_, {_,Domid,StrawProc,StatePath,DataDir}, #sm{straws =Straws} =St) ->
192     ok = xenstore:unwatch(StatePath),
193     exit(StrawProc, peer_closed),
194     ok = xenstore:delete(DataDir),
195     io:format("strawman: connection to domain ~w lost\n", [Domid]),
196     Straws1 = lists:keydelete(StrawProc, 3, Straws),
197     {noreply,St#sm{straws =Straws1}}.
198
199 close_straws([]) -> ok;
200 close_straws([{Mode,Domid,StrawProc,StatePath,DataDir}|Straws]) ->
```

```
201     if Mode =:= active ->
202         ok = xenstore:delete(DataDir),
203         xenstore:wait(StatePath, ?STATE_CLOSED);
204             true -> ok end,
205     exit(StrawProc, shutdown),
206     io:format("strawman: connection to domain ~w closed\n", [Domid]),
207     close_straws(Straws).
208
209 short_straw(ReplyTo, Domid, Refs, Channel, Format) ->
210     Pore = pore_straw:open(Domid, Refs, Channel),
211     ReplyTo ! {ready,self()},
212     looper(Pore, Format).
213
214 short_straw(ReplyTo, Domid, Format) ->
215     Pore = pore_straw:open(Domid),
216     {Refs,Channel} = pore_straw:info(Pore),
217     ReplyTo ! {ready,self(),Refs,Channel},
218     looper(Pore, Format).
219
220 looper(Pore, Format) ->
221     {IA,OA} = pore_straw:avail(Pore),
222     looper(Pore, IA, OA, undefined, [], 0, [], 0, Format).
223
224 looper(Pore, _IA, OA, ExpSz, InBuf, InSz, OutBuf, OutSz, Fmt) when OutSz > 0, OA > 0 ->
225     {Chip,OutBuf1,OutSz1} = chip(OA, OutBuf, OutSz),
226     ok = pore_straw:write(Pore, Chip),
227     true = pore:poke(Pore),
228     {IA1,OA1} = pore_straw:avail(Pore),
229     looper(Pore, IA1, OA1, ExpSz, InBuf, InSz, OutBuf1, OutSz1, Fmt);
230
231 looper(Pore, IA, OA, undefined, InBuf, InSz, OutBuf, OutSz, Fmt) when InSz >= 4 ->
232     {<<ExpSz:32>>,InBuf1,InSz1} = chip(4, InBuf, InSz),
233     looper(Pore, IA, OA, ExpSz, InBuf1, InSz1, OutBuf, OutSz, Fmt);
234
235 looper(Pore, IA, OA, ExpSz, InBuf, InSz, OutBuf, OutSz, Fmt) when ExpSz =/= undefined, I
236     {Chip,InBuf1,InSz1} = chip(ExpSz, InBuf, InSz),
237     deliver(Chip, Fmt),
238     looper(Pore, IA, OA, undefined, InBuf1, InSz1, OutBuf, OutSz, Fmt);
239
240 looper(Pore, IA, _OA, ExpSz, InBuf, InSz, OutBuf, OutSz, Fmt) when IA > 0 ->
241     Data = pore_straw:read(Pore),
242     true = pore:poke(Pore),
243     {IA1,OA1} = pore_straw:avail(Pore),
244     looper(Pore, IA1, OA1, ExpSz, [InBuf,Data], InSz+iolist_size(Data), OutBuf, OutSz, F
245
246 looper(Pore, IA, OA, ExpSz, InBuf, InSz, OutBuf, OutSz, Fmt) ->
247     receive
248     {envelope,_,_} =Envelope when Fmt =:= erlang ->
249       EnvBin = term_to_binary(Envelope),
```

```
250       looper_s(Pore, IA, OA, ExpSz, InBuf, InSz, OutBuf, OutSz, Fmt, EnvBin);
251
252     {envelope,Addressee,Message} when Fmt =:= json, is_atom(Addressee) ->
253       try
254         Json = [{<<"addr">>,to_bin(Addressee)},
255                 {<<"msg">>,Message}],
256         EnvBin = jsx:encode(Json),
257         looper_s(Pore, IA, OA, ExpSz, InBuf, InSz, OutBuf, OutSz, Fmt, EnvBin)
258       catch _:_ ->
259         io:format("strawman: malformed JSON: ~s\n", [Message]),
260         looper(Pore, IA, OA, ExpSz, InBuf, InSz, OutBuf, OutSz, Fmt) end;
261
262     {irq,Pore} ->
263       {IA1,OA1} = pore_straw:avail(Pore),
264       looper(Pore, IA1, OA1, ExpSz, InBuf, InSz, OutBuf, OutSz, Fmt) end.
265
266 looper_s(Pore, IA, OA, ExpSz, InBuf, InSz, OutBuf, OutSz, Fmt, EnvBin) ->
267         Sz = byte_size(EnvBin),
268         OutBuf1 = [OutBuf,<<Sz:32>>,EnvBin],
269         OutSz1 = OutSz + 4 + Sz,
270         looper(Pore, IA, OA, ExpSz, InBuf, InSz, OutBuf1, OutSz1, Fmt).
271
272 select_format(Dir1, Dir2) ->
273   select_format1(fmt(Dir1), fmt(Dir2)).
274
275 select_format1(erlang, erlang) -> erlang;
276 select_format1(_, _) -> json.
277
278 fmt(Dir) ->
279   case xenstore:read(lc(Dir, "/format")) of
280     {ok,Fmt}  -> list_to_atom(Fmt);
281     {error,_} -> erlang end.
282
283 deliver(Bin, erlang) ->
284     try
285       {envelope,Addressee,Message} = binary_to_term(Bin),
286       Addressee ! Message
287     catch _:_ ->
288       io:format("strawman: bad message: ~p\n", [Bin]) end;
289
290 deliver(Bin, json) ->
291     try
292         Json = jsx:decode(Bin),
293         {_,AddrBin} = lists:keyfind(<<"addr">>, 1, Json),
294         {_,Message} = lists:keyfind(<<"msg">>, 1, Json),
295         Addressee = list_to_atom(binary_to_list(AddrBin)),
296         Addressee ! {json,Message}
297     catch _:_ ->
298         io:format("strawman: malformed JSON message: ~s\n", [Bin]) end.
```

```
299
300 chip(N, Buf, Sz) when Sz =< N -> {iolist_to_binary(Buf),[],0};
301 chip(N, Buf, Sz) when is_binary(Buf) ->
302     <<Chip:(N)/binary,Buf1/binary>> =Buf,
303     {Chip,Buf1,Sz-N};
304 chip(N, Buf, Sz) -> chip(N, iolist_to_binary(Buf), Sz).
305
306 to_bin(Atom) -> list_to_binary(atom_to_list(Atom)).
307
308 lc(X) -> lists:concat(X).
309 lc(X, Y) -> lists:concat([X,Y]).
```

```
 1 #include ”bif_impl.h”
 2
 3 #define MAX_PORE_DATA   2048
 4
 5 term_t cbif_pore_xs_open0(proc_t *proc, term_t *regs)
 6 {
 7     pore_xs_t *xp = (pore_xs_t *)pore_make_N(A_XENSTORE,
 8             sizeof(pore_xs_t), proc->pid, 0, start_info.store_evtchn);
 9     if (xp == 0)
10         fail(A_NO_MEMORY);
11     xp->intf = mfn_to_virt(start_info.store_mfn);
12     return xp->parent.eid;
13 }
14
15 term_t cbif_pore_xs_write2(proc_t *proc, term_t *regs)
16 {
17     term_t Pore = regs[0];
18     term_t Data = regs[1];
19     if (!is_short_eid(Pore))
20         badarg(Pore);
21     if (!is_list(Data) && !is_boxed_binary(Data))
22         badarg(Data);
23     pore_t *pr = pore_lookup(Pore);
24     if (pr == 0 || pr->tag != A_XENSTORE)
25         badarg(Pore);
26
27     int64_t size = iolist_size(Data);
28     if (size < 0)
29         badarg(Data);
30     uint8_t buf[size];
31     iolist_flatten(Data, buf);
32
33     pore_xs_t *xp = (pore_xs_t *)pr;
34     struct xenstore_domain_interface *intf = xp->intf;
35     uint32_t cons = intf->req_cons;
```

```
    uint32_t prod = intf->req_prod;
    assert(prod +size -cons <= XENSTORE_RING_SIZE);
    mb();
    uint8_t *pd = buf;
    for (uint32_t i = prod; i < prod +size; i++)
        intf->req[MASK_XENSTORE_IDX(i)] = *pd++;
    wmb();
    intf->req_prod += size;

    return A_OK;
}

term_t cbif_pore_xs_read1(proc_t *proc, term_t *regs)
{
    term_t Pore = regs[0];
    if (!is_short_eid(Pore))
        badarg(Pore);
    pore_t *pr = pore_lookup(Pore);
    if (pr == 0 || pr->tag != A_XENSTORE)
        badarg(Pore);

    pore_xs_t *xp = (pore_xs_t *)pr;
    struct xenstore_domain_interface *intf = xp->intf;
    uint32_t cons = intf->rsp_cons;
    uint32_t prod = intf->rsp_prod;
    uint32_t avail = prod - cons;
    assert(avail > 0);
    rmb();
    uint8_t *ptr;
    term_t bin = heap_make_bin(&proc->hp, avail, &ptr);
    for (uint32_t i = cons; i < prod; i++)
        *ptr++ = intf->rsp[MASK_XENSTORE_IDX(i)];
    mb();
    intf->rsp_cons += avail;

    return bin;
}

term_t cbif_pore_xs_avail1(proc_t *proc, term_t *regs)
{
    term_t Pore = regs[0];
    if (!is_short_eid(Pore))
        badarg(Pore);
    pore_t *pr = pore_lookup(Pore);
    if (pr == 0 || pr->tag != A_XENSTORE)
        badarg(Pore);

    pore_xs_t *xp = (pore_xs_t *)pr;
    struct xenstore_domain_interface *intf = xp->intf;
```

```
 85      int qa = XENSTORE_RING_SIZE -intf->req_prod +intf->req_cons;
 86      int ra = intf->rsp_prod -intf->rsp_cons;
 87
 88      return heap_tuple2(&proc->hp, tag_int(qa), tag_int(ra));
 89  }
 90
 91  static void straw_destroy(pore_t *pore)
 92  {
 93      assert(pore->tag == A_STRAW);
 94      pore_straw_t *ps = (pore_straw_t *)pore;
 95      if (ps->active)
 96          for (int i = 0; i < NUM_STRAW_REFS; i++)
 97              grants_end_access(ps->ring_refs[i]);
 98      else
 99          ms_unmap_pages(ps->shared, NUM_STRAW_REFS, ps->map_handles);
100  }
101
102  term_t cbif_pore_straw_open1(proc_t *proc, term_t *regs)
103  {
104      term_t Domid = regs[0];
105      if (!is_int(Domid))
106          badarg(Domid);
107      int peer = int_value(Domid);
108
109      uint32_t evtchn = event_alloc_unbound(peer);
110      assert(sizeof(straw_ring_t) == NUM_STRAW_REFS*PAGE_SIZE);
111      int size = (NUM_STRAW_REFS+1)*PAGE_SIZE -sizeof(memnode_t);
112      pore_straw_t *ps = (pore_straw_t *)pore_make_N(A_STRAW, size, proc->pid, straw_destr
113      if (ps == 0)
114          fail(A_NO_MEMORY);
115
116      straw_ring_t *ring = (straw_ring_t *)((uint8_t *)ps -sizeof(memnode_t) +PAGE_SIZE);
117      assert(((uintptr_t)ring & (PAGE_SIZE-1)) == 0); // page-aligned
118      ps->shared = ring;
119      ps->active = 1;
120      // all other fields are zero
121
122      for (int i = 0; i < NUM_STRAW_REFS; i++)
123      {
124          void *page = (void *)ps->shared + PAGE_SIZE*i;
125          grants_allow_access(&ps->ring_refs[i], peer, virt_to_mfn(page));
126      }
127
128      return ps->parent.eid;
129  }
130
131  term_t cbif_pore_straw_open3(proc_t *proc, term_t *regs)
132  {
133      term_t Domid = regs[0];
```

```
134     term_t Refs = regs[1];
135     term_t Channel = regs[2];
136     if (!is_int(Domid))
137         badarg(Domid);
138     int peer_domid = int_value(Domid);
139     if (!is_int(Channel))
140         badarg(Channel);
141     int peer_port = int_value(Channel);
142     term_t l = Refs;
143     uint32_t refs[NUM_STRAW_REFS];
144     for (int i = 0; i < NUM_STRAW_REFS; i++)
145     {
146         if (!is_cons(l))
147             badarg(Refs);
148         term_t *cons = peel_cons(l);
149         if (!is_int(cons[0]))
150             badarg(Refs);
151         refs[i] = int_value(cons[0]);
152         l = cons[1];
153     }
154     if (l != nil)
155         badarg(Refs);
156 
157     uint32_t evtchn = event_bind_interdomain(peer_domid, peer_port);
158 
159     assert(sizeof(straw_ring_t) == NUM_STRAW_REFS*PAGE_SIZE);
160     pore_straw_t *ps = (pore_straw_t *)pore_make_N(A_STRAW,
161             sizeof(pore_straw_t), proc->pid, straw_destroy, evtchn);
162     if (ps == 0)
163         fail(A_NO_MEMORY);
164 
165     for (int i = 0; i < NUM_STRAW_REFS; i++)
166         ps->ring_refs[i] = refs[i];
167 
168     ps->shared = (straw_ring_t *)ms_map_pages(ps->ring_refs,
169                     NUM_STRAW_REFS, peer_domid, ps->map_handles);
170     // all other fields are zero
171 
172     return ps->parent.eid;
173 }
174 
175 term_t cbif_pore_straw_write2(proc_t *proc, term_t *regs)
176 {
177     term_t Pore = regs[0];
178     term_t Data = regs[1];
179     if (!is_short_eid(Pore))
180         badarg(Pore);
181     if (!is_list(Data) && !is_boxed_binary(Data))
182         badarg(Data);
```

```
    pore_t *pr = pore_lookup(Pore);
    if (pr == 0 || pr->tag != A_STRAW)
        badarg(Pore);

    int64_t size = iolist_size(Data);
    if (size < 0)
        badarg(Data);
    uint8_t buf[size];
    iolist_flatten(Data, buf);

    pore_straw_t *ps = (pore_straw_t *)pr;
    straw_ring_t *ring = ps->shared;
    int prod = (ps->active) ?ring->out_prod :ring->in_prod;
    int cons = (ps->active) ?ring->out_cons :ring->in_cons;
    mb();
    uint8_t *ptr = buf;
    uint8_t *buffer = (ps->active) ?ring->output :ring->input;
    while (size-- > 0)
    {
        buffer[prod++] = *ptr++;
        if (prod == STRAW_RING_SIZE)
            prod = 0;
        assert(prod != cons);   // too long - avoid crash?
    }
    wmb();
    if (ps->active)
        ring->out_prod = prod;
    else
        ring->in_prod = prod;

    return A_OK;
}

term_t cbif_pore_straw_read1(proc_t *proc, term_t *regs)
{
    term_t Pore = regs[0];
    if (!is_short_eid(Pore))
        badarg(Pore);
    pore_t *pr = pore_lookup(Pore);
    if (pr == 0 || pr->tag != A_STRAW)
        badarg(Pore);

    pore_straw_t *ps = (pore_straw_t *)pr;
    straw_ring_t *ring = ps->shared;
    int prod = (ps->active) ?ring->in_prod :ring->out_prod;
    int cons = (ps->active) ?ring->in_cons :ring->out_cons;
    int avail = prod - cons;
    while (avail < 0)
        avail += STRAW_RING_SIZE;
```

```
    assert(avail > 0);
    rmb();
    uint8_t *ptr;
    uint8_t *buffer = (ps->active) ?ring->input :ring->output;
    term_t bin = heap_make_bin(&proc->hp, avail, &ptr);
    while (avail-- > 0)
    {
        *ptr++ = buffer[cons++];
        if (cons >= STRAW_RING_SIZE)
            cons = 0;
    }
    mb();
    if (ps->active)
        ring->in_cons = cons;
    else
        ring->out_cons = cons;

    return bin;
}

term_t cbif_pore_straw_info1(proc_t *proc, term_t *regs)
{
    term_t Pore = regs[0];
    if (!is_short_eid(Pore))
        badarg(Pore);
    pore_t *pr = pore_lookup(Pore);
    if (pr == 0 || pr->tag != A_STRAW)
        badarg(Pore);
    pore_straw_t *ps = (pore_straw_t *)pr;
    term_t refs = nil;
    for (int i = NUM_STRAW_REFS-1; i >= 0; i--)
    {
        int ref = ps->ring_refs[i];
        assert(fits_int(ref));
        refs = heap_cons(&proc->hp, tag_int(ref), refs);
    }

    assert(fits_int((int)pr->evtchn));
    return heap_tuple2(&proc->hp, refs, tag_int(pr->evtchn));
}

term_t cbif_pore_straw_avail1(proc_t *proc, term_t *regs)
{
    term_t Pore = regs[0];
    if (!is_short_eid(Pore))
        badarg(Pore);
    pore_t *pr = pore_lookup(Pore);
    if (pr == 0 || pr->tag != A_STRAW)
        badarg(Pore);
```

```
281
282     pore_straw_t *ps = (pore_straw_t *)pr;
283     straw_ring_t *ring = ps->shared;
284
285     // how much we can read
286     int avail1 = (ps->active) ?ring->in_prod - ring->in_cons
287                               :ring->out_prod - ring->out_cons;
288     while (avail1 < 0)
289         avail1 += STRAW_RING_SIZE;
290
291     // how much we can write
292     int avail2 = (ps->active) ?ring->out_cons - ring->out_prod
293                               :ring->in_cons - ring->in_prod;
294     while (avail2 <= 0)
295         avail2 += STRAW_RING_SIZE;
296     avail2--;   // unused byte
297
298     return heap_tuple2(&proc->hp, tag_int(avail1), tag_int(avail2));
299 }
300
301 term_t cbif_pore_poke1(proc_t *proc, term_t *regs)
302 {
303     term_t Pore = regs[0];
304     if (!is_short_eid(Pore))
305         badarg(Pore);
306     pore_t *pr = pore_lookup(Pore);
307     if (pr == 0)
308         badarg(Pore);
309
310     event_kick(pr->evtchn);
311     return A_TRUE;
312 }
313
314 term_t cbif_pore_close1(proc_t *proc, term_t *regs)
315 {
316     term_t Pore = regs[0];
317     if (!is_short_eid(Pore))
318         badarg(Pore);
319
320     pore_t *pr = pore_lookup(Pore);
321     if (pr == 0)
322         return A_FALSE;
323
324     pore_destroy(pr);
325     return A_TRUE;
326 }
```

```
#include ”pore.h”

#include ”ling_common.h”
#include <string.h>
#include ”event.h”
#include ”scheduler.h”
#include ”atom_defs.h”

static uint32_t next_pore_id = 0;
static pore_t *active_pores = 0;

static void pore_universal_handler(uint32_t evtchn, void *data);

pore_t *pore_make_N(term_t tag,
        uint32_t size, term_t owner, void (*destroy_private)(pore_t *), uint32_t evtchn)
{
    memnode_t *home = nalloc_N(size);
    if (home == 0)
        return 0;
    pore_t *np = (pore_t *)home->starts;
    memset(np, 0, size);

    np->eid = tag_short_eid(next_pore_id++);
    np->tag = tag;
    np->owner = owner;
    np->destroy_private = destroy_private;
    np->home = home;
    np->evtchn = evtchn;

    if (evtchn != NO_EVENT)
        event_bind(evtchn, pore_universal_handler, np);

    if (active_pores != 0)
        active_pores->ref = &np->next;
    np->ref = &active_pores;
    np->next = active_pores;
    active_pores = np;

    return np;
}

static void pore_universal_handler(uint32_t evtchn, void *data)
{
    assert(data != 0);
    pore_t *pore = (pore_t *)data;
    proc_t *proc = scheduler_lookup(pore->owner);
    if (proc == 0)
        return; // drop

```

```
50      // {irq,Pore}
51      uint32_t *p = heap_alloc_N(&proc->hp, 3);
52      if (p == 0)
53          goto no_memory;
54      term_t irq = tag_tuple(p);
55      *p++ = 2;
56      *p++ = A_IRQ;
57      *p++ = pore->eid;
58      heap_set_top(&proc->hp, p);
59
60      if (scheduler_new_local_mail_N(proc, irq) < 0)
61          goto no_memory;
62      return;
63
64  no_memory:
65      scheduler_signal_exit_N(proc, pore->eid, A_NO_MEMORY);
66  }
67
68  pore_t *pore_lookup(term_t eid)
69  {
70      assert(is_short_eid(eid));
71      pore_t *pr = active_pores;
72      while (pr != 0)
73      {
74          if (pr->eid == eid)
75              return pr;
76          pr = pr->next;
77      }
78      return 0;
79  }
80
81  void pore_destroy(pore_t *pore)
82  {
83      if (pore->evtchn != NO_EVENT)
84          event_unbind(pore->evtchn);
85
86      *pore->ref = pore->next;
87      if (pore->next != 0)
88          pore->next->ref = pore->ref;
89
90      if (pore->destroy_private != 0)
91          pore->destroy_private(pore);
92
93      nfree(pore->home);
94  }
95
96  void pore_destroy_owned_by(term_t pid)
97  {
98      pore_t *pr = active_pores;
```

```
 99      while (pr != 0)
100      {
101          if (pr->owner == pid)
102          {
103              pore_t *doomed = pr;
104              pr = pr->next;
105              pore_destroy(doomed);
106          }
107          else
108              pr = pr->next;
109      }
110  }
```