Concurrent Programming in C using FairThreads

Frédéric Boussinot

MIMOSA Project

frederic.boussinot@sophia.inria.fr

http://www-sop.inria.fr/mimosa/rp

FairThreads in Scheme:

- how to conciliate threads with reactive programming? Response: cooperative control of pthreads + service threads for IOs

FairThreads in C:

1. how to conciliate multi-processing with thread-based reactive prog.? Cooperativeness vs. multi-processors. Synchrony vs. asynchrony
2. how to use thread-based reactive prog. for embedded systems with limited resources (without pthreads and even without OS)?

Proposal: thread-based reactive extension of C able to run on various platforms ranging from very small ones to multiprocessors machines.

 Contents

3/19 – **Posix Threads**

POSIX threads (or Java threads) drawbacks:

- too complex:
  - semantics depends on the scheduling strategy
  - priorities, cancellation points, ...
- too many constraints:
  - limited number of native threads (< 1000)
  - allocation of a stack mandatory
- too low-level:
  - data need to be protected (locks)
  - threads do not mix well with objects

Cannot be considered as basic units for modular programming

Difficult to use with limited resources (embedded systems)

 Objective

So, why continuing to consider threads?

- Because it is **standard** (first class objects in Java)
- Because **multiprocessing** is possible (SMP)

Objective: **alternative to POSIX threads**

- Simple and as much deterministic as possible
- Needing few resources (for embedded systems, or when large number of components are needed)
- With efficient implementation (compared to sequential programming)
- Allowing multiprocessing

Proposal:

FairThreads + **LOFT** (*Langage Over Fair Threads*)

 FairThreads Model

Concurrency Units: **threads**, linked to a scheduler or unlinked

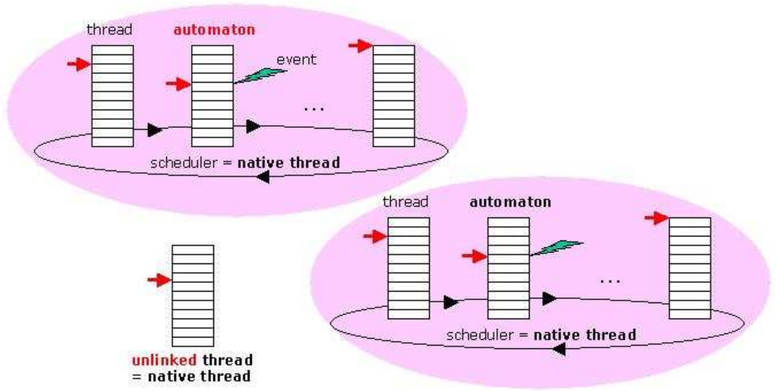
Linked Threads: common **instants**, broadcast events

Schedulers: **cooperative scheduling**, round-robin, determinism

Unlinked Threads: standard pthreads

System structure: dynamic link/unlink operations, **reactive areas**

Automata: always linked (no private stack needed)



6/19 – FairThreads API in C

```
void run (ft_event_t e, ft_thread_t t) {
   ft_thread_await (e);
   ft_scheduler_stop (t); // kill the other thread
   ft_thread_cooperate ();// explicit cooperation
   < body >
}
....
  ft_thread_t t1, t2;
  ft_scheduler_t sched = ft_scheduler_create () ;
  ft_event_t e1 = ft_event_create (sched)
  ft_event_t e2 = ft_event_create (sched);
  ft_scheduler_start (sched);// scheduler run by a native thread
  t1 = ft_thread_create (sched, run, e1, t2);
  t2 = ft_thread_create (sched, run, e2, t1);
....
```

What happens when e1 and e2 are simultaneous?

Automata

```
DEFINE_AUTOMATON (run)
{
   ft_event_t e = args[0]
   ft_thread_t t = args[1]
   BEGIN_AUTOMATON
      STATE_AWAIT (0,e) { ft_scheduler_stop (t); GOTO (1); }
      STATE (1) { ... }
      ...
   END_AUTOMATON
}
...
   t1 = ft_automaton_create (sched, run, e1, t2);
   t2 = ft_automaton_create (sched, run, e2, t1);
```

Objectives:

- Compiled language which extends C
- Basic implementation in FairThreads, where schedulers and fair threads, except automata, are mapped to native threads
- Automaton <-> native thread very simple (automata should be the default)
- Special implementation of the cooperative subset for embedded systems
- Use of efficient techniques for execution (**direct access to the next thread** to execute)
- **Formal semantics** for the cooperative subset of the language.

 Prime Numbers

Recursively defined sieve of Eratosthenes:

```
module producer (event_t out)
local int i;
{local(i) = 1;}
while (1) do
  {local(i) += 2;}
  generate_value (
        local(out),
        (void**)local(i));
  cooperate;
end
end module
```

```
module filter_prime (event_t me)
local int prime, int res, event_t next;
  get_first_val(me,prime);
  {
    output (local(prime));
    filter_prime_create (
          local(next = event_create ()));
  }
  while (1) do
    cooperate;
    get_first_val(me,res);
    {
      if (local(res) % local(prime))
          generate_value (local(next),
                          (void**)local(res));
    }
  end
end module
```

10/19 – **Prime Numbers (2)**

```
module main ()
{
   event_t start = event_create ();
   producer_create (start);
   filter_prime_create (start);
}
end module
```

- First 2000 prime numbers:

Large Number of Threads & Events

Matrix of **pixels** that are threads. Pixels appear at screen during several instants (and color changes) when clicked in

```
module pixel (int x, int y, event_t e)
local unsigned i;// local variable
  while (1) do
    await (local(e));
    {local(i) = 0;};// atomic action
    repeat (8) do
      {local(i) = (local(i)+1)%8;}
      repeat (delay) do
         draw_nice_rectangle(local( x ),local( y ),colors[local(i)]);
         cooperate;
      end
    end
  end
end module
```

12/19 – **Large Number of Threads & Events (2)**

```
pix_event = (event_t*)malloc (resx * resy * sizeof (event_t));
for ( y = 0; y < resy; y++ )
  for ( x = 0; x < resx; x++ ) {
     event_t evt = event_create ();
     pix_event[ y * resx + x ] = evt;
     pixel_create (x*zoom,y*zoom,evt);// creation of a new thread
  }
```

One considers 150x150 threads (implemented as automata):

Other demo with dynamic creation of threads:

These demos have been designed by Olivier Parra

13/19 – **Embedded Systems**

GameBoy Advance of Nintendo: 32 bits ARM, 32kb+96kb VRAM (inside CPU), 2.9” TFT screen

Preys: 5 fair threads: inertia + bounce on borders + collision + synchro + prey behavior

Predators: 9 fair threads: inertia (initialy suspended) + bounce on borders + collision + synchro + predator behavior + 4 directions

Automatic Preys Creation: 1 fair thread

Key Handler: 3 fair threads

Run on a GBA emulator:

14/19 – **Embedded Systems (2)**

```
module prey_sprite ()
local sprite_t s;
{
    sprite_t s = new_sprite (0,PACMAN);
    alloc_behaviors (s,5);
    s->behaviors[0] = inertia_create (s);
    s->behaviors[1] = bounce_on_borders_create (s);
    s->behaviors[2] =
                prey_create (s,pred_evt,prey_evt);
    s->behaviors[3] = sync_sprite_create (s);
    s->behaviors[4] = collide_create (s,prey_evt);
    local(s) = s;
}
await (local(s)->destroy);
run destroy_sprite (local(s));
thread_deallocate (self ());
end module
```

```
module inertia (sprite_t s)
while (1) do
   {
      sprite_t s = local(s);
      s->x += s->sx/INERTIA_FACTOR;
      s->y += s->sy/INERTIA_FACTOR;
   }
   cooperate;
end
end module
```

Multiprocessing

Buffer shared by 2 threads:

```
module process ()
local int v;
while (1) do
   while (buffer->length == 0) do
      await (new_input);
      if (buffer->length == 0) then cooperate; end
   end
   {local (v) = get (buffer);}
   if (local(v) == 0) then return; end
   run process_value (local(v));
   {local(v)--;}
   put (buffer,local(v));
   generate (new_input);
end
end module
```

```
module native process_value (int v)
   unlink
   {
      int i,j
      for (i=0,j=0; i<CYCLES; i++)
         j++;
   }
   link (implicit_scheduler ())
end module
```

- CYCLES=1000,000; BUFF_SIZE=20; INIT_VALUE=100
- 17.5s on a monoprocessor 850Mhz/8.2s on a biprocessor 2x933Mhz

Prime numbers:

1 <u>3</u> 5 7 **9** 11 13 **15** 17 19 **21** 23 25 **27** 29 31 **33** 35 37 **39** 41 43 ...

1 3 <u>5</u> 7 9 11 13 **15** 17 19 21 23 **25** 27 29 31 33 **35** 37 39 41 43 ...

Lucky numbers (start from beginning, do not count red numbers):

1 <u>3</u> **5** 7 9 **11** 13 15 **17** 19 21 **23** 25 27 **29** 31 33 **35** 37 39 **41** 43...

1 3 5 <u>7</u> 9 11 13 15 17 **19** 21 23 25 27 29 31 33 35 37 **39** 41 43...

1 3 5 7 <u>9</u> 11 13 15 17 19 21 23 25 **27** 29 31 33 35 37 39 41 43...

Lucky-prime numbers?

- Reuse previous sieve for primes
- Two schedulers: one for primes, one for luckys
- A `compare` thread that goes between the 2 schedulers

Multiprocessing (2)

```
module compare ()
local int hold, int sw;
while (1) do
   link (lucky_sched);
   COMP(lucky_result,lucky_go);
   link (prime_sched);
   COMP(prime_result,prime_go);
end
end module
```

First 300 lucky-primes, Linux 2.20 SMP 2xPIII 933Mhz, 512Mb memory:

- Using the 2 processors: 10s
- Using only 1 processor: 16s (lucky_sched and prime_sched mapped to the same scheduler)

 Simulations

LOFT can be of valuable use in the **context of simulations**

- Array of N*N schedulers, M boucing balls.
- Each ball dynamically links to the appropriate scheduler and generates a collision event in it.
- Each ball processes all the collisions in the scheduler where it is linked to.
- A supplementary scheduler is in charge of the graphics. Other schedulers are synchronized with it.

N = 5, M = 2000

 – **Conclusion**

Fair threads:

- define basic units for modular concurrent programming
- can be used with limited resources (embedded systems)
- are efficiently implemented (large number of threads and events)
- can benefit from multiprocessor machines

Future work:

- **Migration** of fair threads over the network
- Mixing **reactive intructions** and fair threads

`http://www-sop.inria.fr/mimosa/rp/FairThreads`

`http://www-sop.inria.fr/mimosa/rp/LOFT`