Lecture 4 In-class presentation

Note Title 1/17/2008

Thread Scheduling — Priorities

Def'n (old): no lower priority runs if a higher priority is ready

Three issues:

1) unfairness
2) very hard to implement that policy on a multiprocessor
   - it's bad for programmer's to rely on this property for synchronization
3) deadlock —— priority inversion

T_1 low priority holds lock T_2 — Med. priority CPU bound
T_3 high priority wants lock

Cure for priority inversion — priority inheritance

Thread holding a resource receives highest prior. of threads wanting the resource.

- a bit complicated — necessary in some situations
  - OZ takes a simpler approach

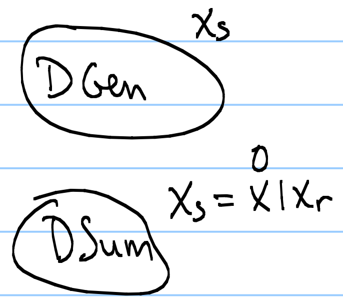
H : 100
M : 10
L : 1

Demand-driven dataflow

```
proc {DGen N Xs} case Xs of
    X|Xr then X=N {DGen N+1 Xr} end
  end
end
declare X
{DGen 0 X}

fun {DSum ?Xs A Lim}
    if Lim>0 then X|Xr=Xs in {DSum Xr A+X Lim-1}
        else A end
end
```

```
local Xs S in
   thread {DGen 0 Xs} end
   thread S = {DSum Xs 0 150000} end
   {Browse S}
end
```



Add'l exercise: write fibonnacci using demand-driven style.

Add'l practice: try wrapping different parts of your eager implementation in Thread... end.

Eager evaluation – produce freely

Lazy evaluation – produce only what's asked for.

poor throughput (and latency)

- parallel processing capability.
- there is overhead assoc. w/ asking for every item to be produced.

Bounded Buffer

```
proc {Buffer N ?Xs Ys}
   fun {Startup N ?Xs}
      if N==0 then Xs else Xr in
         Xs = _|Xr {Startup N-1 Xs} end
   end
   fun {Askloop Ys ?Xs ?End}
      case Ys of Y|Yr then Xr End2 in
         Xs = Y|Xr
         End = _|End2
         {Askloop Yr Xr End2}
      end
```

```
        end
    in
        End = {Startup N Xs}
    end

local Xs Ys S in
   thread {DGenerate 0 Xs} end
   thread {Buffer 4 Xs Ys} end
   thread S = {DSum Ys 0 150 000} end
   {Browse S} {Browse Xs} ...
end
```

Stream Objects:

Threads and Streams together provide a way to program objects.

Object: a value that encapsulates state and behavior

Stream object: gets its behavior commands from an input stream; retains its state in accumulators of its recursive body.

input stream (S1) · state (X1) · output str. (T1)

```
proc {StreamObject S1 X1 T1}
   case S1 of
      M|S2 then N X2 T2 in
         {NextState M X1 N X2}
         T1 = N|T2
         {StreamObject S2 X2 T2}
      [] then T1 = nil end
end
```

output message (N) · next state (X2)

```
declare S0 X0 T0 in
   Thread {StreamObject S0 X0 T0}
end ... )
```

connect S0 to a Stream Source
connect T0 to a Stream Sink
bind X0 to the initial state for
Next State