Lecture 21

Note Title 4/8/2008

How might parallelism in synchronous comms be achieved?

Toward a parallel implementation of Concurrent ML
John Reppy & Yingqi Xiao - Declarative aspects of Multicore Programming, Jan. 2008

Or

find an implementation of synchronous comms that ~~optimistically~~ benefits when interference does not occur.

Subject of paper: Toward a parallel implementation of Concurrent ML, John Reppy and Yingqi Xiao 2008 (Declarative Aspects of Multicore Programming, 2008)

Addresses a system with only sync for sendEvts and select for receive events only.

Recall the single threaded protocol: (from the project)

Poll the events in the list; if any are enabled, do one of them

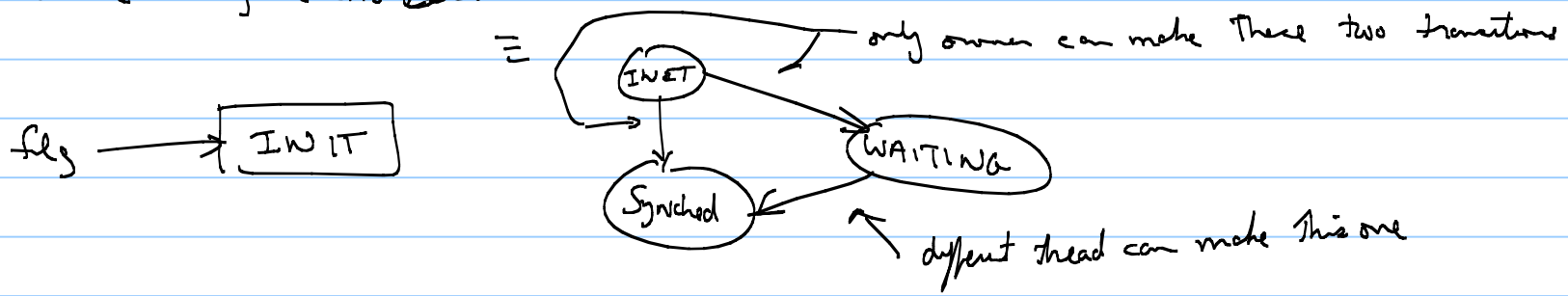
o.w. enqueue events of the list using their block functions.

all done while holding a global lock.

What if we try to use multiple locks — one per channel, say? Deadlock is distinct possibility; Might sort the channels and acquire in order — avoids deadlock on channel locks.

The approach here is optimistic:

try to perform one enabled event (may no longer be enabled) under a lock (channel) block using a CAS call.



if enabled event detected during blocking move to synched else move to waiting

```
claimEvt flg = (case cas (flg, WAITING, SYNCHED)
    of WAITING ⇒ true      — success
     | INIT    ⇒ claimEvt flg — still being INIT'ED
     | SYNCHED ⇒ false     — some body else got there first.
```

RecvEvt — Chi lock, sendq, recvq

poll
```
spinLock (lock)
item = dequeue sendq
unlock (lock)
case item of
    NONE => ()
    SOME (msg, sender) => resume sender; return msg
```

block (flg)
```
spinLock (lock)
item = dequeue sendq
case item of
    SOME (msg, sender) =>
        spinUnlock (lock)
        flg := SYNCHED          (from INIT)
        resume sender; return msg
  | NONE => enqueue (recvq, (flg, curthread)); unlock (lock)
```
block is a list calls block on all events; if none succeed sets the event list flag to WAITING

```
send (Ch (lock, sendq, recvq), msg)
    spin lock (lock)
    item = dequeue recvq
    case item of
        Some (flg, receiver) =>
            if claimed (flg) then unlock lock; transfer msg to receiver and resume it) return;
            else loop
      | NONE =>
            enque (sendq, (msg, curthread))
            unlock lock
            go to sleep
```

Note: use of spin locks is perhaps problematic — assumes a thread holding a spinlock will not be preempted by thread that wants the lock

Note 2: send is not first-class — can't be in a selection list.

Claim: local masking of preemption solves 1st issue: Discuss viability — maybe ok because solving only a performance problem, not synch. problem.

Second problem: have to atomically commit to SYNCHED state in two selection lists

ordered locks for selection lists?

use STM transactional memory technique

unsolved in this paper.