CMPXCHG

CS 520

Dept. of Computer Science

Univ. of New Hampshire

How is *locking* implemented?

```
lock l
load sum
add value
store sum
unlock l
```

only one thread can hold lock at a time

but *how* is this done?

what is a lock?

object that contains:

- thread id for owner of lock
- queue of waiting threads

but it is also shared by the threads!

- need to lock before we update it

so chicken-and-egg problem!

- need a lock to implement a lock

VM520 instructions

jmp

Operation

Unconditional jump.

Format

jmp addr

Encoding

jmp = 20 (0x14)

Description

pc += *addr*.

---

cmpxchg

← hardware support for locking

Operation

Compare and exchange.

Format

cmpxchg reg1,reg2,addr

Encoding

cmpxchg = 21 (0x15)

Description

if *reg1* == *(pc + *addr*) then *(pc + *addr*) = *reg2* else *reg1* = *(pc + *addr*).
Note: this is done atomically by locking the memory bus for the duration of the instruction. If the effective address is out of range of the available memory, then the executing processor halts with an error. The comparison is an integer comparison.

ie two memory operations done atomically pi.asm

```
#
# implementation of the parallel pi computation in vm520 assembler
#
# number of intervals must be evenly divisible by number of processors
#
# machine only supports single precision so cannot get very accurate value
# for pi.

##########################################################################
# computational core
#

# figure out how much work (number of intervals) to do
  getpn r0
  load  r2, intervals
  divi  r2, r0
  store r2, chunk          # chunk = number of intervals to process

# figure out where to start on the x-axis
#   this is a pain because there is no instruction to convert int to float
#   so: start_x = (chunk * processor_id + 0.5) * width
#   re-write to: start_x = (chunk * processor_id * width) + (0.5 * width)
#   and implement (chunk * processor_id * width) by repeated addition
#
  getpid r0
```

Actually this is the instruction to get the processor ID.

```
  load   r1, chunk
  muli   r0, r1            # r0 = chunk * processor_id
  ldimm  r1, 0             # r1 = 0
  load   r2, width         # r2 = width
  ldimm  r3, 0             # r3 is the loop bound
  ldimm  r4, 1             # r4 used to decrement loop index
lab1:                      # while (r0 > 0)
  beq  r0, r3, lab2        #
  addf r1, r2              # r1 += width
  subi r0, r4              # r0 -= 1
  jmp  lab1                # end while
```

getpn gets the number of processors.

```
lab2:                      # now r1 = chunk * processor_id * width
  load r3, oneHalf         # r3 = 0.5
  mulf r2, r3              # r2 = (width * 0.5)
  addf r1, r2              # r1 = (chunk * processor_id * width) + (0.5 * width)

# iterate over the chunk of intervals summing f(x) = 4.0/(1.0 + x^2)
                           # r1 contains the initial x value
  load  r0, chunk          # r0 = chunk (ie loop index)
  ldimm r2, 0              # r2 = 0 (loop bound)
  ldimm r5, 0              # r5 = 0 (sum)
  load  r6, width          # r6 = width
  ldimm r7, 1              # r7 = 1 (to decrement loop index)
lab3:                      # while (r0 > r2)
  beq r0, r2, lab4
  ldimm r3, 0              # r3 = x
  addf r3, r1
  mulf r3, r1              # r3 = x * x
  load r4, one             # r4 = 1.0
  addf r4, r3              # r4 = 1.0 + (x * x)
  load r3, four            # r3 = 4.0
  divf r3, r4              # r3 = 4.0 / (1 + (x * x))
  addf r5, r3              # sum += 4.0 / (1 + (x * x))
  addf r1, r6              # x += width
  subi r0, r7              # r0 -= 1
  jmp lab3                 # end while
lab4:

# multiply the local sum by width
  load r0, width
  mulf r5, r0
```

```
# sum local answer into the answer word
#   use a lock to protect this update when multiple processors
#   busy wait on the lock
  ldimm    r1, 1                 # want to set the lock to 1
tryAgain:
  ldimm    r2, 0                 # need to wait until lock is 0
  cmpxchg r2, r1, lock           # if lock is 0 then lock it
  beq      r2, r1, tryAgain      # else r2 will be set to 1, and if so repeat
  load     r0, answer            # now have exclusive control
  addf     r0, r5                # so safe to add in the local answer
  store    r0, answer
  ldimm    r2, 0                 # unlock
  store    r2, lock

# all done!
  halt

##############################################################################
# variables

# number of intervals per processor
chunk:
  word 0

# lock for exclusive access to the word to contain the answer
#   0 means lock is available
#   1 means it is locked
lock:
  word 0

##############################################################################
# constants and variables to be initialized via putWord

# 4.0
export four
four:
  word 0

# 1.0
export one
one:
  word 0

# 0.5
export oneHalf
oneHalf:
  word 0

# number of intervals to divide the x-axis on [0,1]
export intervals
intervals:
  word 0

# 1.0/intervals
export width
width:
  word 0

export answer
answer:
  word 0
```

lock

unlock

if lock == ∅ then lock = 1
else r2 = lock (1)

This approach uses "spin/busy waiting".

ie repeatedly try to lock until it works

CMPXCHG can be used to do locking because its two memory acasses are atomic

memory bus is locked for the duration of the instruction

ie no other processor can execute a memory operation until the CMPXCHG completes

vm520 implementation

memory treated as shared data

a processor must lock a mutex before
it accesses memory

Intel CMPXCHG

CMPXCHG r32, r/m32

Compare eax with r/m32.

If equal, set ZF and r32 is ~~loaded~~ stored into r/m32.

Else, clear ZF and load r/m32 into eax.

LOCK prefix can be used with CMPXCHG to make it execute atomically.

Note that CMPXCHG r32, m32 has two memory accesses:

1. read m32
2. write m32 if eax == m32

To do locking, we need these two steps to be atomic.

implementing locks on Intel

To lock:

```
        tryAgain:
            movl $0, %eax
            movl $1, %edx
            lock
            cmpxchg %edx X
            beq gotLock
            call yield
            b tryAgain
        gotLock:
```

name of lock
just word in memory
value of 0 indicates
that the lock is free

branch if
ZF
set

```
if X == 0 (eax)
    then X = 1 (edx)
        set ZF
else    clear ZF
        eax = X
```

To unlock:

```
        movl $0, X
```

yield: allow another thread to run

CMPXCHG is used to obtain internal lock

only held for short amount of time in
order to update lock object