Synchronization

15-740
Oct. 20, 2014

Topics

• Locks
• Barriers
• Hardware primitives
Types of Synchronization

Mutual Exclusion

- Locks

Event Synchronization

- Global or group-based (barriers)
- Point-to-point (producer-Consumer)
Simple Producer-Consumer Example

Initially $\text{flag}=0$

- $\text{sd } xdata, (xdatap)$
- $\text{li } xflag, 1$
- $\text{sd } xflag, (xflagp)$

$\text{spin: ld } xflag, (xflagp)$
$\text{beqz } xflag, \text{spin}$
$\text{ld } xdata, (xdatap)$

Is this correct?
Memory Model

Sequential ISA only specifies that each processor sees its own memory operations in program order

Memory model describes what values can be returned by load instructions across multiple threads
Simple Producer-Consumer Example

Initially flag=0

Producer

flag

data

Consumer

Initially flag=0

sd xdata, (xdatap)
li xflag, 1
sd xflag, (xflagp)

spin: ld xflag, (xflagp)
beqz xflag, spin
ld xdata, (xdatap)

Can consumer read flag=1 before data written by producer?
Sequential Consistency
A Memory Model

“A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program”

Leslie Lamport

Sequential Consistency = arbitrary order-preserving interleaving of memory references of sequential programs
Simple Producer-Consumer Example

Initially flag = 0

producer

Consumer

flag
data

sd xdata, (xdatap)
li xflag, 1
sd xflag, (xflagp)

spin: ld xflag, (xflagp)
beqz xflag, spin
ld xdata, (xdatap)

Dependencies from sequential ISA

Dependencies added by sequentially consistent memory model
Implementing SC in hardware

Only a few commercial systems implemented SC
  • Neither x86 nor ARM are SC

Requires either severe performance penalty
  • Wait for stores to complete before issuing new store

Or, complex hardware
  • Speculatively issue loads but squash if memory inconsistency with later-issued store discovered (MIPS R10K)
Software reorders too!

//Producer code
*datap = x/y;
*flagp = 1;

//Consumer code
while (!*flagp) {
    d = *datap;
}

- Compiler can reorder/remove memory operations unless made aware of memory model
  - Instruction scheduling, move loads before stores if to different address
  - Register allocation, cache load value in register, don't check memory

- Prohibiting these optimizations would result in very poor performance
Relaxed Memory Models

• Not all dependencies assumed by SC are supported, and software has to explicitly insert additional dependencies were needed

• Which dependencies are dropped depends on the particular memory model

• IBM370, TSO, PSO, WO, PC, Alpha, RMO, ...

• How to introduce needed dependencies varies by system
  - Explicit FENCE instructions (sometimes called sync or memory barrier instructions)
  - Implicit effects of atomic memory instructions

• Programmers supposed to work with this???
Fences in Producer-Consumer Ex

Initially flag = 0

Producer → \text{flag} → \text{data} → Consumer

\text{sd xdata, (xdatap)}
\text{li xflag, 1}
\text{fence.w.w //Write-write fence}
\text{sd xflag, (xflagp)}

\text{spin: ld xflag, (xflagp)}
\text{beqz xflag, spin}
\text{fence.r.r //Read-read fence}
\text{ld xdata, (xdatap)}
// Both threads execute:
ld xdata, (xdatap)
add xdata, 1
sd xdata, (xdatap)

Is this correct?
A protocol based on two shared variables c1 and c2. Initially, both c1 and c2 are 0 (not busy)

**Process 1**

```
...  
c1=1;  
L:  if c2=1 then go to L  
    < critical section>  
c1=0;
```

**Process 2**

```
...  
c2=1;  
L:  if c1=1 then go to L  
    < critical section>  
c2=0;
```

What is wrong? Deadlock!
Mutual Exclusion: second attempt

To avoid *deadlock*, let a process give up the reservation (i.e. Process 1 sets c1 to 0) while waiting.

- Deadlock is not possible but with a low probability a *livelock* may occur.

- An unlucky process may never get to enter the critical section ⇒ *starvation*

```
Process 1
...
L:  c1=1;
   if c2=1 then
      { c1=0; go to L}
   < critical section>
   c1=0

Process 2
...
L:  c2=1;
   if c1=1 then
      { c2=0; go to L}
   < critical section>
   c2=0
```
A protocol based on 3 shared variables c1, c2 and turn. Initially, both c1 and c2 are 0 (not busy)

• turn = i ensures that only process i can wait
• variables c1 and c2 ensure mutual exclusion

Solution for n processes was given by Dijkstra and is quite tricky!
Busy Waiting vs. Blocking

- Above algorithms don’t require special ops, but
  - May need fences for weaker models
  - Don’t scale
  - Complex
- They Busy-wait. Is this ok?
- Busy-waiting is preferable when:
  - scheduling overhead is larger than expected wait time
  - processor resources are not needed for other tasks
  - schedule-based blocking is inappropriate
    - e.g., in OS kernel
Need Atomic Primitive!

- Test&Set
- Swap
- Fetch&Op
  - Fetch&Incr, Fetch&Decr, ...
- Compare&Swap
- Load-linked/Store-Conditional (LL/SC)
  - LL: return value of an adr
  - SC: if value of adr unchanged, store value -> return 1
    else, nop -> return 0
Lock for Mutual-Exclusion Example

// Both threads execute:
li xone, 1
spin:  amoswap xlock, xone, (xlockp)
bnez xlock, spin
ld xdata, (xdatap)
add xdata, 1
sd xdata, (xdatap)
ds x0, (xlockp)

Assumes SC memory model
Mutual-Exclusion with Relaxed MM

// Both threads execute:
li xone, 1

spin:
amoswap xlock, xone, (xlockp)
bnez xlock, spin
fence.r.r

ld xdata, (xdatap)
add xdata, 1
sd xdata, (xdatap)
fence.w.w
sd x0, (xlockp)

Critical Section

Acquire Lock

Release Lock
Test&Set based lock

lock:
   t&s register, location
   bnz lock
   ret

unlock:
   st location, #0
   ret
How to Evaluate?

- Scalability
- Network load
- Single-processor latency
- Space Requirements
- Fairness
- Required atomic operations
- Sensitivity to co-scheduling
Evaluation of Test&Set based lock

lock: t&s register, location
bnz lock
ret

unlock: st location, #0
ret

- Scalability: poor
- Network load: large
- Single-processor latency: good
- Space Requirements: good
- Fairness: poor
- Required atomic operations: T&S
- Sensitivity to co-scheduling: good?
T&S Lock Performance

Code: \texttt{lock; delay(c); unlock;}
Same total no. of lock calls as $p$ increases; measure time per transfer

![Graph showing performance of T&S lock under different conditions]

- Test&set, $c = 0$
- Test&set, exponential backoff, $c = 3.64$
- Test&set, exponential backoff, $c = 0$
- Ideal

Number of processors vs. Time (µs)
Test and Test and Set

A: while (lock != free);
    if (test&set(lock) == free) {
        critical section;
    }
    else goto A;

(+) spinning happens in cache
(-) can still generate a lot of traffic when many processors go to do test&set
Test and Set with Backoff

Upon failure, delay for a while before retrying
  • either constant delay or exponential backoff

Tradeoffs:
  (+) much less network traffic
  (-) exponential backoff can cause starvation for high-contention locks
    - new requestors back off for shorter times

But exponential found to work best in practice
T&S Lock Performance

Code: `lock; delay(c); unlock;`

Same total no. of lock calls as $p$ increases; measure time per transfer
Test and Set with Update

Test and Set sends updates to processors that cache the lock

Tradeoffs:
- (+) good for bus-based machines
- (-) still lots of traffic on distributed networks

Main problem with test&set-based schemes:
- a lock release causes all waiters to try to get the lock, using a test&set to try to get it.
Ticket Lock (fetch&incr based)

Two counters:
- `next_ticket` (number of requestors)
- `now_serving` (number of releases that have happened)

Algorithm:

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><code>ticket = F&amp;I(next_ticket)</code></td>
</tr>
<tr>
<td>2.</td>
<td><code>while (ticket != now_serving) delay(x)</code></td>
</tr>
<tr>
<td>3.</td>
<td><code>// I have lock</code></td>
</tr>
<tr>
<td>4.</td>
<td><code>now_serving++</code></td>
</tr>
</tbody>
</table>
**Ticket Lock (fetch&incr based)**

**Two counters:**
- `next_ticket` (number of requestors)
- `now_serving` (number of releases that have happened)

**Algorithm:**
- `ticket = F&I(next_ticket)`
- `while (ticket != now_serving) delay(x);`
- `// I have lock`
- `now_serving++;`

**What delay to use?**
- **Not exponential!**
- **Can use `now_serving - next_ticket`**.
Ticket Lock (fetch&incr based)

Two counters:
- `next_ticket` (number of requestors)
- `now_serving` (number of releases that have happened)

Algorithm:
- `ticket = F&I(next_ticket)`
- `while (ticket != now_serving) delay(x);`
- `// I have lock`
- `now_serving++;`

(+) guaranteed FIFO order; no starvation possible
(+) latency can be low if fetch&incr is cacheable
(+) traffic can be quite low, but contention on polling
(-) but traffic is not guaranteed to be $O(1)$ per lock acquire
Array-Based Queueing Locks

Every process spins on a unique location, rather than on a single now_serving counter

```
next-slot  Lock  Wait  Wait  Wait  Wait
```

```
my-slot = F&l(next-slot)
my-slot = my-slot % num_procs
while (slots[my-slot] == Wait);
slots[my-slot] = Wait;
critical section;
slots[(my-slot+1)%num_procs] = Lock;
```

Acquire Lock

Critical Section

Release Lock
List-Base Queueing Locks (MCS)

All other good things + \(O(1)\) traffic even without coherent caches (spin locally)

Uses compare\&swap to build linked lists in software

Locally-allocated flag per list node to spin on

Can work with fetch\&store, but loses FIFO guarantee

Tradeoffs:

\(+\) less storage than array-based locks
\(+\) \(O(1)\) traffic even without coherent caches
\(-\) compare\&swap not easy to implement
Barriers

We will discuss five barriers:

• centralized
• software combining tree
• dissemination barrier
• tournament barrier
• MCS tree-based barrier
Centralized Barrier

Basic idea:
- notify a single shared counter when you arrive
- poll that shared location until all have arrived

Simple version require polling/spinning twice:
- first to ensure that all procs have left previous barrier
- second to ensure that all procs have arrived at current barrier

Solution to get one spin: sense reversal
Software Combining Tree Barrier

- Writes into one tree for barrier arrival
- Reads from another tree to allow procs to continue
- Sense reversal to distinguish consecutive barriers
Dissemination Barrier

$\log P$ rounds of synchronization

In round $k$, proc $i$ synchronizes with proc $(i+2^k) \mod P$

Advantage:
- Can statically allocate flags to avoid remote spinning
Minimum Barrier Traffic

What is the minimum number of messages needed to implement a barrier with $N$ processors?
Tournament Barrier

Binary combining tree

Representative processor at a node is \textit{statically chosen}:
\begin{itemize}
  \item no fetch\&op needed
\end{itemize}

In round $k$, proc $i=2^k$ sets a flag for proc $j=i-2^k$:
\begin{itemize}
  \item $i$ then drops out of tournament and $j$ proceeds in next round
  \item $i$ waits for global flag signalling completion of barrier to be set
    \begin{itemize}
      \item could use combining wakeup tree
    \end{itemize}
\end{itemize}
MCS Software Barrier

Modifies tournament barrier to allow static allocation in wakeup tree, and to use sense reversal

Every processor is a node in two P-node trees:

• has pointers to its parent building a fanin-4 arrival tree
• has pointers to its children to build a fanout-2 wakeup tree
Barrier Recommendations

Criteria:

• length of critical path
• number of network transactions
• space requirements
• atomic operation requirements
Space Requirements

Centralized:
  • constant

MCS, combining tree:
  • $O(P)$

Dissemination, Tournament:
  • $O(P \log P)$
Network Transactions

Centralized, combining tree:
  • $O(P)$ if broadcast and coherent caches;
  • unbounded otherwise

Dissemination:
  • $O(P \log P)$

Tournament, MCS:
  • $O(P)$
Critical Path Length

If independent parallel network paths available:
  • all are $O(\log P)$ except centralized, which is $O(P)$

Otherwise (e.g., shared bus):
  • linear factors dominate
Primitives Needed

Centralized and combining tree:
  • atomic increment
  • atomic decrement

Others:
  • atomic read
  • atomic write
Barrier Recommendations

Without broadcast on distributed memory:
- **Dissemination**
  - MCS is good, only critical path length is about 1.5X longer
  - MCS has somewhat better network load and space requirements

Cache coherence with broadcast (e.g., a bus):
- **MCS with flag wakeup**
  - centralized is best for modest numbers of processors

**Big advantage of centralized barrier:**
- adapts to changing number of processors across barrier calls
Synchronization

- Required for concurrent programs
  - mutual exclusion
  - producer-consumer
  - barrier

- Hardware support
  - ISA
  - Cache
  - memory

- Complex interactions
  - Scalability, Efficiency, Indirect effects

- What about message passing?