Components of a Synchronization Event

Acquire method
• Acquire right to the synch (enter critical section, go past event

Waiting algorithm
• Wait for synch to become available when it isn’t

Release method
• Enable other processors to acquire right to the synch

Waiting algorithm is independent of type of synchronization

Waiting Algorithms

Blocking
• Waiting processes are descheduled
• High overhead
• Allows processor to do other things

Busy-waiting
• Waiting processes repeatedly test a location until it changes value
• Releasing process sets the location
• Lower overhead, but consumes processor resources
• Can cause network traffic

Busy-waiting better when
• Scheduling overhead is larger than expected wait time
• Processor resources are not needed for other tasks
• Scheduler-based blocking is inappropriate (e.g. in OS kernel)

Hybrid methods: busy-wait a while, then block
Role of System and User

User wants to use high-level synchronization operations
• Locks, barriers...
• Doesn’t care about implementation
System designer: how much hardware support in implementation?
• Speed versus cost and flexibility
• Waiting algorithm difficult in hardware, so provide support for others
Popular trend:
• System provides simple hardware primitives (atomic operations)
• Software libraries implement lock, barrier algorithms using these
• But some propose and implement full-hardware synchronization

Challenges

Same synchronization may have different needs at different times
• Lock accessed with low or high contention
• Different performance requirements: low latency or high throughput
• Different algorithms best for each case, and need different primitives
Multiprogramming can change synchronization behavior and needs
• Process scheduling and other resource interactions
• May need more sophisticated algorithms, not so good in dedicated case
Rich area of software-hardware interactions
• Which primitives available affects what algorithms can be used
• Which algorithms are effective affects what primitives to provide

Need to evaluate using workloads
Mutual Exclusion: Hardware Locks

Separate lock lines on the bus: holder of a lock asserts the line
  • Priority mechanism for multiple requestors

Lock registers (Cray XMP)
  • Set of registers shared among processors

Inflexible, so not popular for general purpose use
  – few locks can be in use at a time (one per lock line)
  – hardwired waiting algorithm

Primarily used to provide atomicity for higher-level software locks

First Attempt at Simple Software Lock

lock:    ld   register, location /* copy location to register */
         cmp location, #0 /* compare with 0 */
         bnz lock /* if not 0, try again */
         st location, #1 /* store 1 to mark it locked */
         ret /* return control to caller */

and

unlock:  st location, #0 /* write 0 to location */
         ret /* return control to caller */

Problem: lock needs atomicity in its own implementation
  • Read (test) and write (set) of lock variable by a process not atomic

Solution: atomic read-modify-write or exchange instructions
  • atomically test value of location and set it to another value, return
    success or failure somehow
### Atomic Exchange Instruction

Specifies a location and register. In atomic operation:

- Value in location read into a register
- Another value (function of value read or not) stored into location

Many variants
- Varying degrees of flexibility in second part

Simple example: test&set
- Value in location read into a specified register
- Constant 1 stored into location
- Successful if value loaded into register is 0
- Other constants could be used instead of 1 and 0

Can be used to build locks

---

### Simple Test&Set Lock

```assembly
lock:    t&s  register, location
bnz    lock    /* if not 0, try again */
ret    /* return control to caller */

unlock: st  location, #0    /* write 0 to location */
ret    /* return control to caller */
```

Other read-modify-write primitives can be used too
- Swap
- Fetch&op
- Compare&swap
  - Three operands: location, register to compare with, register to swap with
  - Not commonly supported by RISC instruction sets

Can be cacheable or uncacheable (we assume cacheable)
T&S Lock Microbenchmark Performance

On SGI Challenge. Code: \texttt{lock; delay(c); unlock;}

Same total no. of lock calls as $p$ increases; measure time per transfer

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Time (\textmu s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>18</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

- Performance degrades because unsuccessful \texttt{test&set}s generate traffic

Enhancements to Simple Lock Algorithm

Reduce frequency of issuing \texttt{test&set}s while waiting

- \textit{Test&set lock with backoff}
  - Don’t back off too much or will be backed off when lock becomes free
  - Exponential backoff works quite well empirically: $i^{th}$ time = $k \cdot c^i$

Busy-wait with read operations rather than \texttt{test&set}

- \textit{Test-and-test&set lock}
  - Keep testing with ordinary load
    - cached lock variable will be invalidated when release occurs
  - When value changes (to 0), try to obtain lock with \texttt{test&set}
    - only one attemptor will succeed; others will fail and start testing again
Performance Criteria (T&S Lock)

Uncontended Latency
- Very low if repeatedly accessed by same processor; indept. of \( p \)

Traffic
- Lots if many processors compete; poor scaling with \( p \)
- Each t&s generates invalidations, and all rush out again to t&s

Storage
- Very small (single variable); independent of \( p \)

Fairness
- Poor, can cause starvation

Test&set with backoff similar, but less traffic
Test-and-test&set: slightly higher latency, much less traffic
But still all rush out to read miss and test&set on release
  - Traffic for \( p \) processors to access once each: \( O(p^2) \)

Luckily, better hardware primitives as well as algorithms exist

Improved Hardware Primitives: LL-SC

Goals:
- Test with reads
- Failed read-modify-write attempts don’t generate invalidations
- Nice if single primitive can implement range of r-m-w operations

Load-Locked (or -linked), Store-Conditional

LL reads variable into register
Follow with arbitrary instructions to manipulate its value
SC tries to store back to location if and only if no one else has written to the variable since this processor’s LL
  - If SC succeeds, means all three steps happened atomically
  - If fails, doesn’t write or generate invalidations (need to retry LL)
  - Success indicated by condition codes; implementation later
**Simple Lock with LL-SC**

```assembly
lock:   ll  reg1, location    /* LL location to reg1 */
sc     location, reg2        /* SC reg2 into location*/
beqz  reg2, lock            /* if failed, start again */
ret
unlock: st  location, #0    /* write 0 to location */
ret
```

Can do more fancy atomic ops by changing what’s between LL & SC
- But keep it small so SC likely to succeed
- Don’t include instructions that would need to be undone (e.g. stores)

SC can fail (without putting transaction on bus) if:
- Detects intervening write even before trying to get bus
- Tries to get bus but another processor’s SC gets bus first

LL, SC are not lock, unlock respectively
- Only guarantee no conflicting write to lock variable between them
- But can use directly to implement simple operations on shared variables

---

**More Efficient SW Locking Algorithms**

Problem with Simple LL-SC lock
- No invals on failure, but read misses by all waiters after both release and successful SC by winner
- No test-and-test&set analog, but can use backoff to reduce burstiness
- Doesn’t reduce traffic to minimum, and not a fair lock

Better SW algorithms for bus (for r-m-w instructions or LL-SC)
- Only one process to try to get lock upon release
  - valuable when using test&set instructions; LL-SC does it already
- Only one process to have read miss upon release
  - valuable with LL-SC too
- *Ticket lock* achieves first
- *Array-based queueing lock* achieves both
- Both are fair (FIFO) locks as well
Ticket Lock

Only one r-m-w (from only one processor) per acquire
Works like waiting line at deli or bank

- Two counters per lock \((\text{next} \_\text{ticket}, \text{now} \_\text{serving})\)
- Acquire: fetch&inc \text{next} \_\text{ticket}; wait for \text{now} \_\text{serving} to equal it
  - atomic op when arrive at lock, not when it’s free (so less contention)
- Release: increment \text{now} \_\text{serving}
- FIFO order, low latency for low-contention if fetch&inc cacheable
- Still \(O(p)\) read misses at release, since all spin on same variable
  - like simple LL-SC lock, but no inval when SC succeeds, and fair
- Can be difficult to find a good amount to delay on backoff
  - exponential backoff not a good idea due to FIFO order
  - backoff proportional to \text{now} \_\text{serving} - \text{next} \_\text{ticket} may work well

Wouldn’t it be nice to poll different locations ...

Array-based Queuing Locks

Waiting processes poll on different locations in an array of size \(p\)

- Acquire
  - fetch&inc to obtain address on which to spin (next array element)
  - ensure that these addresses are in different cache lines or memories
- Release
  - set next location in array, thus waking up process spinning on it
- \(O(1)\) traffic per acquire with coherent caches
- FIFO ordering, as in ticket lock
- But, \(O(p)\) space per lock
- Good performance for bus-based machines
- Not so great for non-cache-coherent machines with distributed memory
  - array location I spin on not necessarily in my local memory (solution later)
Lock Performance on SGI Challenge

Loop: lock; delay(c); unlock; delay(d);

- Simple LL-SC lock does best at small p due to unfairness
  - Not so with delay between unlock and next lock
  - Need to be careful with backoff
- Ticket lock with proportional backoff scales well, as does array lock
- Methodologically challenging, and need to look at real workloads

Point to Point Event Synchronization

Software methods:
- Interrupts
- Busy-waiting: use ordinary variables as flags
- Blocking: use semaphores

Full hardware support: full-empty bit with each word in memory
- Set when word is “full” with newly produced data (i.e. when written)
- Unset when word is “empty” due to being consumed (i.e. when read)
- Natural for word-level producer-consumer synchronization
  - producer: write if empty, set to full; consumer: read if full; set to empty
- Hardware preserves atomicity of bit manipulation with read or write
- Problem: flexibility
  - multiple consumers, or multiple writes before consumer reads?
  - needs language support to specify when to use
  - composite data structures?
Barsriers

Software algorithms implemented using locks, flags, counters

Hardware barriers

- Wired-AND line separate from address/data bus
- Set input high when arrive, wait for output to be high to leave
- In practice, multiple wires to allow reuse
- Useful when barriers are global and very frequent
- Difficult to support arbitrary subset of processors
  - even harder with multiple processes per processor
- Difficult to dynamically change number and identity of participants
  - e.g. latter due to process migration
- Not common today on bus-based machines

Let’s look at software algorithms with simple hardware primitives

A Simple Centralized Barrier

Shared counter maintains number of processes that have arrived
- increment when arrive (lock), check until reaches numprocs

```c
struct bar_type {int counter; struct lock_type lock; int flag = 0;} bar_name;

BARRIER (bar_name, p) {
    LOCK(bar_name.lock);
    if (bar_name.counter == 0) {
        bar_name.flag = 0; /* reset flag if first to reach */
        mycount = bar_name.counter++; /* mycount is private */
        UNLOCK(bar_name.lock);
    } else if (mycount == p) {
        bar_name.flag = 1; /* release waiters */
        bar_name.counter = 0; /* last to arrive */
    } else while (bar_name.flag == 0) {}; /* busy wait for release */
}
```

• Problem?
A Working Centralized Barrier

Consecutively entering the same barrier doesn’t work
  • Must prevent process from entering until all have left previous instance
  • Could use another counter, but increases latency and contention

Sense reversal: wait for flag to take different value consecutive times
  • Toggle this value only when all processes reach

BARRIER (bar_name, p) {
  local_sense = !(local_sense); /* toggle private sense variable */
  LOCK(bar_name.lock);
  mycount = bar_name.counter++; /* mycount is private */
  if (bar_name.counter == p)
    UNLOCK(bar_name.lock);
    bar_name.flag = local_sense; /* release waiters*/
  else
    { UNLOCK(bar_name.lock);
      while (bar_name.flag != local_sense) {}; }
}

Centralized Barrier Performance

Latency
  • Want short critical path in barrier
  • Centralized has critical path length at least proportional to \( p \)

Traffic
  • Barriers likely to be highly contended, so want traffic to scale well
  • About \( 3p \) bus transactions in centralized

Storage Cost
  • Very low: centralized counter and flag

Fairness
  • Same processor should not always be last to exit barrier
  • No such bias in centralized

Key problems for centralized barrier are latency and traffic
  • Especially with distributed memory, traffic goes to same node
**Improved Barrier Algorithms for a Bus**

Software combining tree

- Only $k$ processors access the same location, where $k$ is degree of tree

- Separate arrival and exit trees, and use sense reversal
- Valuable in distributed network: communicate along different paths
- On bus, all traffic goes on same bus, and no less total traffic
- Higher latency ($\log p$ steps of work, and $O(p)$ serialized bus xactions)
- Advantage on bus is use of ordinary reads/writes instead of locks

**Barrier Performance on SGI Challenge**

- Centralized does quite well
  - Will discuss fancier barrier algorithms for distributed machines
- Helpful hardware support: piggybacking of reads misses on bus
  - Also for spinning on highly contended locks
**Synchronization Summary**

Rich interaction of hardware-software tradeoffs

Must evaluate hardware primitives and software algorithms together
  - primitives determine which algorithms perform well

Evaluation methodology is challenging
  - Use of delays, microbenchmarks
  - Should use both microbenchmarks and real workloads

Simple software algorithms with common hardware primitives do well on bus
  - Will see more sophisticated techniques for distributed machines
  - Hardware support still subject of debate

Theoretical research argues for swap or compare&swap, not fetch&op
  - Algorithms that ensure constant-time access, but complex