vlocks for Bare-Metal Mutual Exclusion

Voting Locks, or "vlocks" provide a simple low-level mutual exclusion mechanism, with reasonable but minimal requirements on the memory system.

These are intended to be used to coordinate critical activity among CPUs which are otherwise non-coherent, in situations where the hardware provides no other mechanism to support this and ordinary spinlocks cannot be used.

vlocks make use of the atomicity provided by the memory system for writes to a single memory location. To arbitrate, every CPU "votes for itself", by storing a unique number to a common memory location. The final value seen in that memory location when all the votes have been cast identifies the winner.

In order to make sure that the election produces an unambiguous result in finite time, a CPU will only enter the election in the first place if no winner has been chosen and the election does not appear to have started yet.

Algorithm

The easiest way to explain the vlocks algorithm is with some pseudo-code:

```
int currently_voting[NR_CPUS] = { 0, };
int last_vote = -1; /* no votes yet */

bool vlock_trylock(int this_cpu)
{
        /* signal our desire to vote */
        currently_voting[this_cpu] = 1;
        if (last_vote != -1) {
                /* someone already volunteered himself */
                currently_voting[this_cpu] = 0;
                return false; /* not ourself */
        }

        /* let's suggest ourself */
        last_vote = this_cpu;
        currently_voting[this_cpu] = 0;

        /* then wait until everyone else is done voting */
        for_each_cpu(i) {
                while (currently_voting[i] != 0)
                        /* wait */;
        }

        /* result */
        if (last_vote == this_cpu)
                return true; /* we won */
        return false;
}

bool vlock_unlock(void)
{
        last_vote = -1;
}
```

The currently_voting[] array provides a way for the CPUs to determine whether an election is in progress, and plays a role analogous to the "entering" array in Lamport's bakery algorithm [1].

However, once the election has started, the underlying memory system atomicity is used to pick the winner. This avoids the need for a static priority rule to act as a tie-breaker, or any counters which could overflow.

As long as the last_vote variable is globally visible to all CPUs, it will contain only one value that won't change once every CPU has cleared its currently_voting flag.

Features and limitations

- vlocks are not intended to be fair. In the contended case, it is the _last_ CPU which attempts to get the lock which will be most likely to win.

  vlocks are therefore best suited to situations where it is necessary to pick a unique winner, but it does not matter which CPU actually wins.

- Like other similar mechanisms, vlocks will not scale well to a large number of CPUs.

  vlocks can be cascaded in a voting hierarchy to permit better scaling if necessary, as in the following hypothetical example for 4096 CPUs:

```
/* first level: local election */
my_town = towns[(this_cpu >> 4) & 0xf];
I_won = vlock_trylock(my_town, this_cpu & 0xf);
if (I_won) {
        /* we won the town election, let's go for the state */
        my_state = states[(this_cpu >> 8) & 0xf];
        I_won = vlock_lock(my_state, this_cpu & 0xf));
        if (I_won) {
                /* and so on */
                I_won = vlock_lock(the_whole_country, this_cpu & 0xf];
                if (I_won) {
                        /* ... */
                }
                vlock_unlock(the_whole_country);
        }
        vlock_unlock(my_state);
}
vlock_unlock(my_town);
```

ARM implementation

The current ARM implementation [2] contains some optimisations beyond the basic algorithm:

- By packing the members of the currently_voting array close together, we can read the whole array in one transaction (providing the number of CPUs potentially contending the lock is small enough). This reduces the number of round-trips required to external memory.

  In the ARM implementation, this means that we can use a single load and comparison:

  ```
  LDR     Rt, [Rn]
  CMP     Rt, #0
  ```

  ...in place of code equivalent to:

  ```
  LDRB    Rt, [Rn]
  CMP     Rt, #0
  LDRBEQ  Rt, [Rn, #1]
  CMPEQ   Rt, #0
  LDRBEQ  Rt, [Rn, #2]
  CMPEQ   Rt, #0
  LDRBEQ  Rt, [Rn, #3]
  CMPEQ   Rt, #0
  ```

  This cuts down on the fast-path latency, as well as potentially reducing bus contention in contended cases.

  The optimisation relies on the fact that the ARM memory system guarantees coherency between overlapping memory accesses of different sizes, similarly to many other architectures. Note that we do not care which element of currently_voting appears in which bits of Rt, so there is no need to worry about endianness in this optimisation.

  If there are too many CPUs to read the currently_voting array in one transaction then multiple transations are still required. The implementation uses a simple loop of word-sized loads for this case. The number of transactions is still fewer than would be required if bytes were loaded individually.

  In principle, we could aggregate further by using LDRD or LDM, but to keep the code simple this was not attempted in the initial implementation.

- vlocks are currently only used to coordinate between CPUs which are unable to enable their caches yet. This means that the implementation removes many of the barriers which would be required when executing the algorithm in cached memory.

  packing of the currently_voting array does not work with cached memory unless all CPUs contending the lock are cache-coherent, due to cache writebacks from one CPU clobbering values written by other CPUs. (Though if all the CPUs are cache-coherent, you should be probably be using proper spinlocks instead anyway).

- The "no votes yet" value used for the last_vote variable is 0 (not -1 as in the pseudocode). This allows statically-allocated vlocks to be implicitly initialised to an unlocked state simply by putting them in .bss.

  An offset is added to each CPU's ID for the purpose of setting this variable, so that no CPU uses the value 0 for its ID.

Colophon

Originally created and documented by Dave Martin for Linaro Limited, for use in ARM-based big.LITTLE platforms, with review and input gratefully received from Nicolas Pitre and Achin Gupta. Thanks to Nicolas for grabbing most of this text out of the relevant mail thread and writing up the pseudocode.

linux/COPYING.

References

[1] Lamport, L. "A New Solution of Dijkstra's Concurrent Programming Problem", Communications of the ACM 17, 8 (August 1974), 453-455.
https://en.wikipedia.org/wiki/Lamport%27s_bakery_algorithm

[2] linux/arch/arm/common/vlock.S, www.kernel.org.