Atomic increments

Enrique Soriano–Salvador
Laboratorio de Sistemas
Universidad Rey Juan Carlos
esoriano@lsub.org

Gorka Guardiola Múzquiz
Laboratorio de Sistemas
Universidad Rey Juan Carlos
paurea@lsub.org
11/8/2012

ABSTRACT

Following a discussion in the NIX mailing list, we have measured the cost of different implementations of atomic increment using a benchmark. These implementations are (i) the standard assembler routine using a hardware protected addition instruction; (ii) a lock–free assembler routine using compare–and–exchange; and (iii) a spin lock protected counter. In this paper we present the implementations, the results of running a benchmark against them and our conclusions about which one we should keep. Quite surprisingly, spin locks are better than the other implementations for a high contention scenario in a 32 cores multiprocessor.

Introduction

While implementing various synchronization mechanisms in NIX [2], we started a discussion about the cost of different atomic operations. In particular, one of the simplest operations we can imagine is incrementing or decrementing an atomic integer (aInc). Several mechanisms have been implemented in the C library and in the kernel while trying to optimize this operations. Specifically, we found two assembler implementations of aInc and a C implementation using spin locks [1].

AInc implementations

The first assembly routine we found uses a hardware protected addition instruction (Intel’s LOCK prefix). As the AMD64 manual states [3]:

The LOCK prefix causes certain kinds of memory read–modify–write instructions to occur atomically. The mechanism for doing so is implementation–dependent (for example, the mechanism may involve bus signaling or packet messaging between the processor and a memory controller). The prefix is intended to give the processor exclusive use of shared memory in a multiprocessor system.
The routine for AMD64 is:

```
TEXT lainc(SB), 1, $-4
  MOVL $1, AX
  LOCK; XADDL AX, (RARG)
  ADDL $1, AX
  RET
```

The routine for 386 is:

```
TEXT lainc(SB), $0
  MOVL 1+0(FP), AX
  LOCK; INCL 0(AX)
  RET
```

The version of ainc we show next is also an assembly routine, but it is subtler. This implementation is *lock free* [4]. It copies the value of the counter, updates it and tries to put it back into the counter with compare-and-exchange. If the value of the counter changed in the window, the attempt fails, and the operation is retried.

The routine for AMD64 is:

```
TEXT lfainc(SB), 1, $0
ainclp:
  MOVL (RARG), AX /* exp */
  MOVL AX, BX
  INCL BX /* new */
  LOCK; CMPXCHGL BX, (RARG)
  JNZ ainclp
  MOVL BX, AX
  RET
```

The routine for 386 is:

```
TEXT lfainc(SB), $0 /* int ainc(int *); */
ainclp:
  MOVL (BX), AX
  MOVL AX, CX
  INCL CX
  LOCK
  BYTE $0x0F; BYTE $0xB1; BYTE $0x0B /* CMPXCHGL CX, (BX) */
  JNZ ainclp
  MOVL CX, AX
  RET
```

The third version uses the standard Plan 9’s C library spin lock implemented with test-and-set (which is just the XCHGL instruction which acts as if having and implicit lock prefix) and back off:
void
lock(Lock *lk)
{
    int i;

    /* once fast */
    if(!_tas(&lk->val))
        return;

    /* a thousand times pretty fast */
    for(i=0; i<1000; i++){
        if(!_tas(&lk->val))
            return;
        sleep(0);
    }

    /* now nice and slow */
    for(i=0; i<1000; i++){
        if(!_tas(&lk->val))
            return;
        sleep(100);
    }

    /* take your time */
    while(_tas(&lk->val))
        sleep(1000);
}

Test-and-set for AMD64 is:

    MOVL    $0xdeaddead,AX
    XCHGL   AX,(RARG)
    RET

Test-and-set for 386 is mostly the same:

    TEXT    _tas(SB),$0

    MOVL    $0xdeaddead,AX
    MOVL    1+F(FP),BX
    XCHGL   AX,(BX)
    RET

Finally, this version of the atomic increment is:

    lock(&l);
    counter++;
    unlock(&l);

Clearly, having three different implementations for such a simple operation is an overkill. The question is which one should we keep. We have measured them in various scenarios to try to answer to this question.

Methodology
The machine(s) in which we took the measurements are a 32-core AMD K10 Opteron 6128 running NIX and a 2594MHz Pentium IV running Plan 9. For the AMD multiprocessor, we have taken measurements using 1 and 32 cores. NIX uses an AMP scheduler when running on multiple processors, and a SMP scheduler for one processor. We measured the cost for the three implementations of ainfc showed in the previous section.
In AMD64 (only for the implementation based on the LOCK prefix) we have taken the measurements for 64 byte aligned values and unaligned (i.e. 32 byte aligned) values. The \( \mu \)benchmark measures the time taken to increment the value with different levels of contention. The program spawns a number of processes that try to increment the counter in a closed loop. The contention depends on two factors:

- The number of processes spawned. The \( \mu \)benchmark has been executed for 1, 2, 5, 10 and 20 processes. In addition, the multiprocessor test has been executed for 50 and 100 processes.
- The amount of time wasted between increments in the loop. The \( \mu \)benchmark has been executed for 0, 1, 50, 100, 200 and 500 ns. This time, of course, is not taken into account when measuring the increment. The rationale behind adding this parameter is twofold. First, small values of this delay (\( \text{waste} \)) try to compensate the effect on contention of the time taken to perform the \( \text{a} \text{inc} \). A faster \( \text{a} \text{inc} \) would have a higher rate of operations executed, and thus, higher contention. When \( \text{waste} \) is much bigger than the time taken to perform an \( \text{a} \text{inc} \), the contention only depends on the number of processes and size of \( \text{waste} \), and not on the time to perform of the operation itself. Second, the delay allows us to measure the operation in different contention scenarios for the same number of processes and processors.

The core of the program is (in pseudocode):

```plaintext
poll until all proc ready
for i in 1..1000
  t1 = taketime()
  ainc(v)
  t2 = taketime()
  times[i] = t2 - t1
  do
    wastesometime()
    t3 = taketime()
    while t3-t1 < Waste
endloop
```

We take the time using the time stamp counter, by using the RDTSC instruction. Notice that in the multiprocessor case, the hardware counters are synchronized on boot. We booted the machine for each execution of the \( \mu \)benchmark. The drift of the counters for the time taken to perform the measurements is negligible.

**Results**

What follows depicts Tukey diagrams of the measurements. The labels shown in the graphs are:

- \( \text{a} \text{inc} \text{–align} \): the \( \text{a} \text{inc} \) implementation based on the LOCK prefix, value aligned to 64 byte.
- \( \text{a} \text{inc} \text{–noalign} \): the \( \text{a} \text{inc} \) implementation based on the LOCK prefix, value unaligned to 64 byte, aligned to 32 byte.
- \( \text{lockfree} \text{a} \text{inc} \): the lock-free \( \text{a} \text{inc} \) implementation based on compare-and-swap.
- \( \text{lock} \): the \( \text{a} \text{inc} \) implementation using spin locks.

The most surprising result is that for the multiprocessor case, the spin lock version is much faster than the other implementations. This is not what we expected. All the implementations end up using a LOCK prefix instruction (note that there is an implicit LOCK prefix in the XCHGL instruction). In addition, the spin lock implementation
Figure 1: NIX on AMD64, 32 cores, (A) waste = 0 ns, (B) waste = 50 ns, (C) waste = 200 ns, (D) waste = 500 ns requires to execute more instructions and an extra function call (unlock).

It may well be that the backoff of the spin locks is responsible for their efficiency in the multicore case. Nevertheless, note that in Figure 2 (A), for 1 process (i.e. no contention), the spin locks are better than the other implementations. This may be due to the effect of extreme flooding in the coherency fabric. This kind of effect is what we tried to correct by using different waste values. There seems to be no effect associated to the alignment of the values.
Figure 2: NIX on AMD64, 1 core, (A) waste = 0 ns, (B) waste = 50 ns, (C) waste = 200 ns, (D) waste = 500 ns

Notice also that, except for one case (Figure 2 (C)), the lock-free version seems to perform equal or worse than the other implementations.

In the 386 experiments, the results show more predictable results. The interesting result in these experiments is that the lock-free implementation is still worse than the others. This is easy to understand if the CMPXCHGL is as expensive as the XADDL with the LOCK prefix, because the lock-free implementation requires more instructions and may need several attempts to perform the operation.
Conclusions and Future Work

We can conclude that the lock-free implementation is not the best choice in any case, at least with the current implementation of the instructions for these architectures. Between the two other implementations, it is difficult to conclude which one is better. For the 386 experiment there is a 10% gain when using the lock prefix implementation. On the other hand, in most cases for the multiprocessor it seems to be better to use the spin lock implementation. When using only 1 core, the results are (more or less)
balanced. Nevertheless, when using 32 cores, the spin lock version is much better. Another argument in favor of the spin lock version is that spin locks are already needed as a general purpose synchronization primitive.

Last, note that the AMP scheduler of NIX is new, so there may be lurking bugs affecting the results of the experiments.

References


