ACM Operating Systems Review, Vol. 29, No. 3, July 1995, pp 31-34.

A Short Note on Cheap Fine-grained Time Measurement

Jochen Liedtke
GMD — German National Research Center for Information Technology *
jochen.liedtke@gmd.de

Abstract

High resolution processor time bookkeeping for user and kernel threads can be done without degrading kernel performance. On a PC, threads with active intervals of 2 μs are measured sufficiently precisely. The overhead can be reduced down to 0.1% of overall processor time.

1 Motivation

This paper was triggered by a discussion with Jon Inouye at 15^{th} SOSP [Inouye 1995]. His thesis was that for some operating systems, system-call cost is high because a user/kernel cpu-clock update is required per kernel entry/exit.

1.1 Strict Clock Updates

Strict clock updates require

kernel entry:
    stop := clock ;
    $t_{\text{current thread}}$ INCR (stop − start) .

kernel exit:
    start := clock ;
    t_{kernel} INCR (start − stop) .

The overhead can be ignored as long as system calls are implemented inefficiently, 900 cycles or more. On the other hand, we know that a properly designed kernel can reduce the system call cost basically to the bare processor overhead for switching between user and kernel mode. Depending on the processor, this is 100 cycles or even less. In this context, the clock update overhead may be relevant.

On most processors, the update cost is dominated by reading the cpu clock. On a 486 PC, the cpu clock is an external device. Reading its registers costs about 140 cycles, i.e. we need approximately additional 300 cycles per system call (which itself costs only 107 cycles). On a Pentium, we could instead use an internal 64-bit register which counts processor cycles. But even this costs at least 24 cycles per system call which itself costs 75 cycles on this processor. The additional costs are not negligible.

1.2 Periodic Clock Updates

A standard technique to avoid the mentioned system-call and interrupt overhead is updating the clocks per incoming clock interrupt:

periodic clock interrupt:
    if in user mode
        then $t_{\text{current thread}}$ INCR Δ
        else t_{kernel} INCR Δ
    fi .

Here, Δ is the constant time between two adjacent clock interrupt pulses. It is determined by the hardware facilities and by the required scheduling granularity. Usual tick intervals range from 1 ms to 10 ms. The clock-update cost per per interrupt is less than 20 cycles and can thus be ignored.

*GMD SET-RS, Schlo Birlinghoven, 53757 Sankt Augustin, Germany

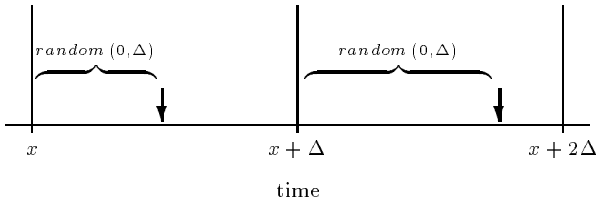
We define the *active interval* as the period that a process would use the processor continuously, provided that the timeslice is sufficiently long and no interrupt occurs. Obviously, the measurements are precise as long as the active intervals of the running processes are substantially larger than the tick interval.

However, even for arbitrary small active intervals, the method remains probabilistically precise *as long as the intervals are randomly distributed over time.* Therefore, asynchronuous device interrupts and high RPC frequencies can be tolerated without reducing the long-term precision.

Unfortunately, measurements become arbitrarily imprecise when sufficiently short active intervals are no longer randomly distributed. This happens for example when short active intervals are timer-scheduled, i.e. start always directly after a clock interrupt and terminate before the next one. Similar situations may occur in a systematically priority-scheduled environment. Therefore, periodic clock updating is certainly no adequate technique for real-time systems.

2 Randomized Clock Updates

All disadvantages of the perodic-clock-update technique are due to the insufficient randomness of the distribution of active intervals. This can be completely overcome by *randomizing the clock update process.* Incrementing the clocks is no longer done at the periodic tick point but at a randomly chosen point within each Δ-interval:



Scheduling remains the job of the periodic clock handler. Clock updates happen with the same frequency but are decoupled from the periodic interrupt. For implementation, we use two hardware timers, one periodic clock and a programmable timer:

periodic clock interrupt:
  set next random intr (random $(0, \Delta)$) .

random clock interrupt:
  if in user mode
    then $t_{\text{current thread}}$ INCR Δ
    else t_{kernel} INCR Δ
  fi .

2.1 Precision

Assume a process which consumes processor time t in real time T. t is the sum of all of its active intervals t_i. If $k\,\Delta \le t_i < (k+1)\,\Delta$, k clock pulses are measured precisely for this active interval. Only the remaining time $t_i - k\,\Delta$ is measured stochastically. For a worst case estimation of the inaccuracy, we therefore assume that all activation intervals are less than one periodic interval, $t_i \le \Delta$.

Then the probability that the measured process is active at a random tick is $p = t/T$. $n = T/\Delta$ random ticks occur so that the expectation value for our measured time is $np\Delta = t$. That is fine.

However, the critical point is the error to be expected. For large n, the standard deviation is $\sigma = \Delta\sqrt{np} = \sqrt{t\,\Delta}$. The corresponding error relative to the time t is

$$e = \frac{\sigma}{t} = \sqrt{\frac{\Delta}{t}} \quad .$$

In other words, for a time measurement with an expected error of e or less, our process must consume processor time

$$t \ge \frac{\Delta}{e^2} \quad .$$

For periodic clock intervals of 1 ms and an expected error of 1%, we get $t \ge 10$ s. If our process is scheduled once per 1 ms interval, we have to measure over at least real time T:

active interval	2 μs	10 μs	50 μs	250 μs
T	1 h 23 m	17 m	200 s	40 s

2.2 Usability

Time measurement is typically used for three purposes:

1. Benchmarks.
   To get sufficiently precise results, let the benchmark run long enough (see above). The periodic clock granularity does not limitate measurements.

2. μ-measurements.
   This technique is inadequate for non-repeated μ-measurements. Instead, code instrumentation, sometimes even kernel instrumentation, is required.

3. Accounting.
   The more cpu time your customers consume, the more precise your measurements will be. If you have a customer who consumes only 1 second per month, do not bill him per month but per year. Then even his bill will be 1%-accurate.
   A further nice feature: although the precision per bill remains constant, the precision of the totally billed cost per customer increases with time.
   Some people will not tolerate bills that are only precise with probability, even if the probability of an error is only 10^{-14}. The counter-argument: even when using strict clock updating, the probabilty for obtaining incorrect results due to hardware or software bugs is much larger.

2.3 Limitations

To get precise measurements, the active intervals are not limited by Δ and not by the random tick frequency. However, the random *granularity* imposes a lower bound on the active-interval size. On a PC, a 1.193-MHz-driven timer can be used for random ticks. As a consequence, active intervals below 1 μs cannot be measured properly. A timer-scheduled active interval below 1 μs per Δ will be measured as 1 μs per Δ. In contrast to the periodic update method, random updating delivers in this case a cpu time larger than the really consumed time.

2.4 Cost

The required cost for the random clock interrupt is basically determined by the basic interrupt service time, calculating the new random time and reloading the timer circuit. On a 50 MHz PC, this costs about 6 μs, i.e. approximately 0.6 % of the processor time. (Half of the cost is due to the slow timer ports.)

This overhead can be arbitrarily reduced by decreasing the random-tick frequency. Do not execute one random tick per Δ but one per $k\Delta$. The cost decreases to $0.6/k$ %. However, processes have to consume at least $10k$ s instead of 10 s before the 1%-precision is achieved. This is probably not a problem for accounting but perhaps makes benchmarking inconvenient. So make the random rate changeable at runtime (note that it does not influence scheduling) and gain another 0.5% of processor time when not running benchmarks.

3 Conclusion

Time measurement is no reason for inefficient system-call implementations. (Benchmark support is no reason for inefficient kernels.) Efficient system-call implementation is no reason for coarse-grained time measurements.

References

INOUYE, J. 1995. private communication.