Linux Timers(/devblog/2014/05/linux-timers/)

May 16, 2014

A short introduction to measuring time and a selection of timers present on modern Linux systems.

Introduction

Imagine you are given a task, let's say to carry bricks from point A to point B. Furthermore, assume that you are required not only to perform the task, but also to measure the time it took you to do so.

How would you do it?

If you only have to carry one single brick from point A to point B, you might look at your wall clock before you do it and after you do it. Then you can just take the difference of both of these times and take that as the time it took you to complete the task.

However, if you have a lot of bricks, which means going back and forth a lot to transport all of them, you might not be working continuously. If you take breaks, or even split the work over multiple days, you only want to know the time you actually spent carrying bricks. You could look at the wall clock each time you start and stop working and accumulate all the differences, but that's a very tedious way to do it.

So you might rather carry a timer with you that you start when you begin working and pause when you interrupt your work. At the end you just have to read the value to get an accurate result.

Similarily to this example, execution of a program on a computer is not continuous most of the time. Therefore, the POSIX-standard contains lots of different timers. We will be examining the following five in this article:

- clock()
- CLOCK_REALTIME
- CLOCK_MONOTONIC
- CLOCK_PROCESS_CPUTIME_ID
- CLOCK_THREAD_CPUTIME_ID

Furthermore, there are two coarse variants specified in the Linux kernel starting with version 2.6.32 and 2.6.28, respectively: CLOCK_REALTIME_COARSE and CLOCK_MONOTONIC_COARSE

Timers

I will give a few performance results about the resolution and query speed for these timers directly in the text. Please keep in mind that these results are specific to my system (with an Intel i5 mobile processor), and may vary on yours, especially if you use a different architecture. We have collected results on a few different systems and I compiled them in a table at the end of the article. If you want to test the timers on your system, you can get the source code here(https://gist.github.com/MarcusVoelker/9cdde98d57a0475ab84b#file-main-cpp). If you get results that are different enough from ours to be interesting, feel free to post them in the comments along with a short description of your system, and I will add them to the article.

clock()

clock()(http://linux.die.net/man/3/clock) approximates the number of clock cycles that have passed since the program started. Notably, this value is not necessarily initialized with zero, so if you want to measure the time your whole program took, you should store the result of clock() once at the start of your program, and compare with that value at the end. Also, you will have to divide the result by CLOCKS_PER_SEC in order to get the time in seconds.

On my system, clock() is not particularily fast, taking about 190 ns to query.

The other four timers are invoked (and possibly set) via the clock_gettime and clock_settime(http://linux.die.net/man/2/clock_gettime) functions.

CLOCK_REALTIME

CLOCK_REALTIME is a wall clock. Querying it results in getting the current system time. While there is relatively little overhead to reading this timer and it is rather exact (~25 ns per query, 1 ns accuracy on my system), it is not only unsuitable to measure task time in a multithreaded environment, but it is also subject to changes to the system clock. So if you take your starting and your finishing time, but the administrator changed the clock in the meantime, you will get wildly inaccurate results.

If you are really concerned about the speed, but not so much about accuracy, you can use CLOCK_REALTIME_COARSE, which is less exact (clock_getres gives me a resolution of 1 ms), but can be read in ~8 ns on my system.

CLOCK_MONOTONIC

To prevent being influenced by a changing system clock, one can use CLOCK_MONOTONIC instead. The monotonic clock is set to an unspecified starting value, but is guaranteed to only increase its value between two successive measurements, except for adjustments made by adjtime(http://linux.die.net/man/3/adjtime) or NTP(https://en.wikipedia.org/wiki/Network_Time_Protocol). However, these adjustments are incremental and should therefore not affect the results too much in common use cases. Note that to ensure monotonicity, you cannot set this clock with the clock_settime function.

CLOCK_MONOTONIC is better suited to measuring a time difference (which is our prime application in this article), but worthless if you need the absolute time for your use-case. Also, it is as insufficient for a multithreaded environment as CLOCK_REALTIME. Finally, there also is a coarse variant CLOCK_MONOTONIC_COARSE, which is about as fast and exact as CLOCK_REALTIME_COARSE.

CLOCK_PROCESS_CPUTIME_ID

While the previous two timers are fast, they do not account for the fact that a system usually executes more than one process at once, and processes usually have to share CPU time and are not allotted all CPU cycles that occur between start and finish. CLOCK_PROCESS_CPUTIME_ID allows to measure not the elapsed system time, but the time taken by the process itself. This is paid for by a slower timer evaluation (~160 ns on my system).

Furthermore, if the process is moved between CPUs, depending on the hardware used the results may be useless. This is due to some hardware having a separate timer on each CPU, and these do not have to align, resulting in completely disparate start and finish times.

CLOCK_THREAD_CPUTIME_ID

CLOCK_THREAD_CPUTIME_ID has the same advantages and disadvantages as CLOCK_PROCESS_CPUTIME_ID (it is marginally faster on my system at ~140 ns per query), but it counts thread time instead of process time. If your programm uses multiple threads, you will likely want to use this over CLOCK_PROCESS_CPUTIME_ID.

Similar to CLOCK_MONOTONIC, the two CPUTIME clocks cannot be set, and have a resolution of 1 ns on my system.

Experiment Results

Results were collected with this C++ program(https://gist.github.com/MarcusVoelker/9cdde98d57a0475ab84b#file-main-cpp). If you share interesting, different results in the comments, I will add them to this collection.

Gentoo Kernel 3.12.13, Intel i5-2520M

Timer	Timer speed	Timer resolution	Coarse Timer speed	Coarse Timer resolution
clock()	**190.0 ns**	**---**	**---**	**---**
CLOCK_REALTIME	**25.44 ns**	**1 ns**	**8.521 ns**	**1 ms**

CLOCK_MONOTONIC	**24.64 ns**	**1 ns**	**7.349 ns**	**1 ms**
CLOCK_PROCESS_CPUTIME_ID	**163.2 ns**	**1 ns**	**---**	**---**
CLOCK_THREAD_CPUTIME_ID	**143.8 ns**	**1 ns**	**---**	**---**

Raspbian Kernel 3.10.36, ARM1176JZF-S

Timer	**Timer speed**	**Timer resolution**	**Coarse Timer speed**	**Coarse Timer resolution**
clock()	**1765 ns**	**---**	**---**	**---**
CLOCK_REALTIME	**1153 ns**	**1 ns**	**971.4 ns**	**10 ms**
CLOCK_MONOTONIC	**1169 ns**	**1 ns**	**1072 ns**	**10 ms**
CLOCK_PROCESS_CPUTIME_ID	**1702 ns**	**1 ns**	**---**	**---**
CLOCK_THREAD_CPUTIME_ID	**1520 ns**	**1 ns**	**---**	**---**

Linux Kernel 3.8.0-35, Intel i7-3630QM

Timer	**Timer speed**	**Timer resolution**	**Coarse Timer speed**	**Coarse Timer resolution**
clock()	**158.9 ns**	**---**	**---**	**---**
CLOCK_REALTIME	**18.60 ns**	**1 ns**	**6.285 ns**	**4 ms**

CLOCK_MONOTONIC	**18.21 ns**	**1 ns**	**6.594 ns**	**4 ms**
CLOCK_PROCESS_CPUTIME_ID	**131.1 ns**	**1 ns**	**---**	**---**
CLOCK_THREAD_CPUTIME_ID	**115.8 ns**	**1 ns**	**---**	**---**

Conclusion

clock() is slow and returns CPU cycles rather than time, so it is not very nice to use. As long as you don't care about your process being preempted, or want a fast timer, you should use CLOCK_MONOTONIC. If you need the actual timestamp, you want CLOCK_REALTIME. If you need to know exactly how long your code took independently of other tasks on the system, can spare the time for a slower query, and can guarantee that your process will not switch CPUs during your execution, you want one of the CPUTIME clocks, depending on whether you need the time for the whole process or just a single thread.

If you are on a slow system, however, the performance differences between the timer types become less pronounced compared to the absolute time it takes to query any timer, and I dare say the performance gain of the COARSE timers is not worth the loss in accuracy for most use cases.

(https://plus.google.com/share?url=https://upvoid.com/devblog/2014/05/linux-timers/) (http://twitter.com/share?url=https://upvoid.com/devblog/2014/05/linux-timers/&text=Linux%20Timers) (https://www.facebook.com/sharer.php?u=https://upvoid.com/devblog/2014/05/linux-timers/)

Marcus Völker(/about/marcus/) // May 16, 2014 // devblog(/blog/tags/devblog)

Neels
a year ago

model name : Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
Kernel version : 3.2.0-64-generic

Test results:

```
14310503 executions of clock() in 2 seconds; 139.757 ns per execution

110215450 executions of CLOCK_REALTIME in 2 seconds; 18.1463 ns per
execution
Resolution is 1

276525195 executions of CLOCK_REALTIME_COARSE in 2 seconds; 7.23262
ns per execution
Resolution is 4000250

110809330 executions of CLOCK_MONOTONIC in 2 seconds; 18.049 ns per
execution
Resolution is 1

267047068 executions of CLOCK_MONOTONIC_COARSE in 2 seconds;
```

see more

Reply