Computer Timing

Table of Contents

Network Time Protocol (NTP)

The **Network Time Protocol (NTP)** is a networking protocol for clock synchronization between computer systems over packet-switched, variable-latency data networks.

NTP is intended to synchronize all participating computers to within a few milliseconds of *Coordinated Universal Time (UTC)*. It uses the intersection algorithm, a modified version of *Marzullo's algorithm*, to select accurate time servers and is designed to mitigate the effects of variable network latency. **NTP** can usually maintain time to within tens of milliseconds over the public Internet, and can achieve better than one millisecond accuracy in local area networks under ideal conditions.

Implementations send and receive timestamps using the *User Datagram Protocol (UDP)* on port number 123. They can also use broadcasting or multicasting, where clients passively listen to time updates after an initial round-trip calibrating exchange.

Process Time Statistics

- **Real Time:** It is *wall clock time*, time from start to finish of the call. This is all *elapsed time* including time slices used by other processes and time the process spends blocked (for example if it is waiting for I/O to complete).

- **User Time:** It is the amount of *CPU time* spent in user-mode code (outside the kernel) within the process. This is only actual *CPU time* used in executing the process. Other processes and time the process spends blocked do not count towards this figure.

- **Sys Time:** It is the amount of *CPU time* spent in kernel-mode within the process. Like *User Time*, this is only *CPU time* used by the process.

*`User+Sys` will tell you how much actual CPU time **your process** used. Note that this is across **all CPUs**, so if the process has multiple threads and this process is running on a computer with more than one processor, it could potentially exceed the wall clock time reported by Real (which usually occurs).*

On a multi-processor machine, a multi-threaded process or a process forking children could have an elapsed time smaller than the total CPU time - as different threads or processes may run in parallel.

Kernel vs User Mode

On Unix, or any protected-memory operating system, **Kernel** or **Supervisor mode** refers to a privileged mode that the CPU can operate in. Certain privileged actions that could affect security or stability can only be done when the CPU is operating in this mode. These actions are not available to application code. An example of such an action might be manipulation of the *MMU* to gain access to the address space of another process. Normally, *user-mode* code cannot do this, although it can request *shared memory* from the *kernel*, which could be read or written by more than one process. In this case, the *shared memory* is explicitly requested from the *kernel* through a secure mechanism and both processes have to explicitly attach to it in order to use it.

In order to switch to **kernel mode** you have to issue a specific instruction (often called a *trap*) that switches the CPU to running in **kernel mode** and runs code from a specific location held in a jump table. For security reasons, you cannot switch to **kernel mode** and execute arbitrary code - the *traps* are managed through a table of addresses that cannot be written to unless the CPU is running in **supervisor mode**. You trap with an explicit trap number and the address is looked up in the jump table; the kernel has a finite number of controlled entry points.

There are things that your code cannot do from **user mode** - things like allocating memory or accessing hardware (HDD, network, etc). These are under the supervision of the *kernel*, and it alone can do them. Some operations that you do like *malloc* or *fread/fwrite* will invoke these *Kernel* functions and that then will count as **sys time**. Unfortunately it's not as simple as "every call to *malloc* will be counted in **sys time**". The call to *malloc* will do some processing of its own still counted in **user time** and then somewhere along the way it may call the function in *kernel* counted in **sys time**. After returning from the *kernel* call, there will be some more time in *user* and then *malloc* will return to your code. As for when the switch happens, and how much of it is spent in kernel mode; you cannot say. It depends on the implementation of the library.

How to measure wall-clock time in C

To measure the elapsed wall-clock time in C, we can use the following function:

`int gettimeofday(struct timeval *tv, struct timezone *tz);`

```
#include <sys/time.h>

double cpuSecond() {
    struct timeval tp;
    gettimeofday(&tp,NULL);
    return ((double)tp.tv_sec + (double)tp.tv_usec*1.e-6);
}
```

Potential Problem

As we said before, `gettimeofday()` returns the current wall clock time and timezone. Therefore, it seems logical to use it in order to measure the passage of time. After all, in real life we measure by checking our watch before and after an operation. Unfortunately, there are differences:

1. You usually aren't running NTP on your wristwatch, so it probably won't jump a second or two (or 15 minutes) in a random direction because it happened to sync up against a proper clock at that point.

   Good NTP implementations try to not make the time jump like this. They instead make the clock go faster or slower so that it will drift to the correct time. But while it's drifting you either have a clock that's going too fast or too slow. It's not measuring the passage of time properly.

2. In real life you are not expected to measure sub-second times. You can't measure the difference between 1.08 seconds and 1.03 seconds. This problem is mostly (but far from entirely) in the small scale.

What to use instead

The most portable way to measure time correctly seems to be `clock_gettime()`. It's not stable across reboots, but we don't care about that. We just want a timer that goes up by one second for each second that passes in the physical world.

`int clock_gettime(clockid_t clk_id, struct timespec *tp)`

```
struct timespec {
        time_t   tv_sec;        /* seconds */
        long     tv_nsec;       /* nanoseconds */
    };
```

- **clk_id**: It is the identifier of the particular clock on which to act. A clock may be system-wide and hence visible for all processes, or per-process if it measures time only within a single process. Some often predefined clocks are:
  - **CLOCK_REALTIME:** represents the machine's best-guess as to the current wall-clock, time-of-day time.This means that *CLOCK_REALTIME* can jump forwards and backwards as the system time-of-day clock is changed, including by *NTP*.
  - **CLOCK_MONOTONIC:** represents the absolute elapsed wall-clock time since some arbitrary, fixed point in the past. It isn't affected by changes in the system time-of-day clock. The important aspect of a monotonic time source is NOT the current value, but the guarantee that the time source is strictly linearly increasing, and thus useful for calculating the difference in time between two samplings.

Usage of `clock_gettime()`

```
#include <time.h>

struct timespec ts_start;
struct timespec ts_end;

clock_gettime(CLOCK_MONOTONIC, &ts_start);
```

```
foo();
clock_gettime(CLOCK_MONOTONIC, &ts_end);

time_t secs = ts_end.tv_sec - ts_start.tv_sec;
long nsecs = ts_end.tv_nsec - ts_start.tv_nsec;
```

Summary of different C Linux Timing functions

- **time():** returns the wall-clock time from the OS, with precision in seconds.
- **clock():** seems to return the sum of user and system time. At one time this was supposed to be the CPU time in cycles, but modern standards like POSIX require *CLOCKS_PER_SEC* to be 1000000, giving a maximum possible precision of 1 µs. Since *glibc 2.18* it is implemented with *clock_gettime(CLOCK_PROCESS_CPUTIME_ID, ...)* in Linux.
- **clock_gettime(CLOCK_MONOTONIC, ...):** provides nanosecond resolution and is monotonic. The 'seconds' and 'nanoseconds' are stored separately, each in 32-bit counters. Thus, any wrap-around would occur after many dozen years of uptime. This looks like a very good clock.
- **getrusage():** It reports the user and system times separately and does not wrap around.
- **gettimeofday():** returns the wall-clock time with (nominally) µs precision but the resolution of the system clock is hardware dependent. Applications should use the *clock_gettime()* function instead of the obsolescent *gettimeofday()* function.