Difference between CLOCK_REALTIME and CLOCK_MONOTONIC?

Asked 14 years, 4 months ago Modified 1 year, 10 months ago

Viewed 284k times

283

Could you explain the difference between `CLOCK_REALTIME` and `CLOCK_MONOTONIC` clocks returned by `clock_gettime()` on Linux?

Which is a better choice if I need to compute elapsed time between timestamps produced by an external source and the current time?

Lastly, if I have an NTP daemon periodically adjusting system time, how do these adjustments interact with each of `CLOCK_REALTIME` and `CLOCK_MONOTONIC`?

linux time clock

Share

Improve this question

Follow

edited Feb 11, 2023 at 13:42

FObersteiner

25.4k 8 57 90

asked Aug 19, 2010 at 15:34

NPE

499k 114 966 1k

8 Answers

Sorted by: Highest score (default)

339

`CLOCK_REALTIME` represents the machine's best-guess as to the current wall-clock, time-of-day time. As Ignacio and MarkR say, this means that `CLOCK_REALTIME` can jump forwards and backwards as the system time-of-day clock is changed, including by NTP.

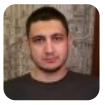
`CLOCK_MONOTONIC` represents the absolute elapsed wall-clock time since some arbitrary, fixed point in the past. It isn't affected by changes in the system time-of-day clock.

If you want to compute the elapsed time between two events observed on the one machine without an intervening reboot, `CLOCK_MONOTONIC` is the best option.

Note that on Linux, `CLOCK_MONOTONIC` does not measure time spent in suspend, although by the POSIX definition it should. You can use the Linux-specific `CLOCK_BOOTTIME` for a monotonic clock that keeps running during suspend.

Share Improve this answer Follow

edited Nov 27, 2019 at 11:49



Ruslan Osmanov
21.5k 8 51 62

answered Aug 20, 2010 at 1:45

caf
239k 41 335 474

18 Note that on newer kernels, CLOCK_MONOTONIC_RAW is available which is even better (no NTP adjustments).

– Joseph Garvin Aug 20, 2012 at 13:59

20 @JosephGarvin for some value of "better", perhaps — CLOCK_MONOTONIC_RAW may run fast or slow of real time by several (or several hundred) parts per million, and its rate might vary due to environmental conditions like temperature or voltage (or steal time on virtual machines). On a properly-working machine, NTP does its best to mitigate all of those factors and so CLOCK_MONOTONIC more closely reflects *true* elapsed time. – hobbs Dec 29, 2012 at 6:37

27 Granted, it might be interesting to have a CLOCK_MONOTONIC_PARBOILED that was affected by NTP's efforts to correct frequency errors, but unaffected by its efforts to correct phase errors, but that's a lot of complexity for a dubious gain :) – hobbs Dec 29, 2012 at 6:40

1 I like the point that @hobbs brings up. What if you're concerned about programs that can be affected by clock drift? Would `CLOCK_MONOTONIC` be the best choice in that scenario? e.g. Patriot Missile System – sjagr Jan 14, 2014 at 18:43

3 I think it's also important to mention that CLOCK_REALTIME is affected by leap seconds. This means that it *will* produce double timestamps every time a leap second is inserted. Last time this happened in June 30, 2012 and quite a lot of software ran into trouble. – user1202136 Apr 2, 2014 at 7:45

45

Robert Love's book **LINUX System Programming 2nd Edition**, specifically addresses your question at the beginning of Chapter 11, pg 363:

> 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

That said, I believe he is assuming the processes are running on the same instance of an OS, so you might want to have a periodic calibration running to be able to estimate drift.

Share Improve this answer Follow

edited Jan 14, 2014 at 18:28

Aliaksandr Belik
12.9k 6 68 92

answered Jul 15, 2013 at 20:30

user2548100
4,661 1 19 18

35

`CLOCK_REALTIME` is affected by NTP, and can move forwards and backwards. `CLOCK_MONOTONIC` is not, and advances at one tick per tick.

Share Improve this answer Follow

answered Aug 19, 2010 at 15:37

Ignacio Vazquez-Abrams

797k 160 1.4k 1.4k

26 CLOCK_MONOTONIC is affected by NTP's time adjustment (time slewing). It won't jump, however. – derobert Aug 23, 2011 at 20:19

5 But on newer kernels there is CLOCK_MONOTONIC_RAW, which really isn't affected by NTP. – Joseph Garvin Aug 20, 2012 at 14:00

2 "tick" -- any rough idea how big/long/CPU instructions is a tick on Linux/amd64? Or where I can get docs on any of this? – kevinarpe Dec 22, 2014 at 3:58

@kevinarpe Not sure but I think a tick is defined as a fraction of time, not a number of CPU cycle, often it's 1/100 second. – Stéphane Mar 2, 2018 at 0:36

@Stéphane: I surely must be tighter than 10ms. I think Java's `System.nanoTime()` uses `CLOCK_MONOTONIC` and can measure *durations* of 1000ns or less. Maybe you are thinking about system time, which is sometimes limited to milliseconds? – kevinarpe Mar 2, 2018 at 6:35

27

POSIX 7 quotes

POSIX 7 specifies both at http://pubs.opengroup.org/onlinepubs/9699919799/functions/clock_getres.html:

`CLOCK_REALTIME`:

> This clock represents the clock measuring real time for the system. For this clock, the values

> returned by clock_gettime() and specified by clock_settime() represent the amount of time (in seconds and nanoseconds) since the Epoch.

`CLOCK_MONOTONIC` (optional feature):

> For this clock, the value returned by clock_gettime() represents the amount of time (in seconds and nanoseconds) since an unspecified point in the past (for example, system start-up time, or the Epoch). This point does not change after system start-up time. The value of the CLOCK_MONOTONIC clock cannot be set via clock_settime().

`clock_settime()` gives an important hint: POSIX systems are able to arbitrarily change `CLOCK_REALITME` with it, so don't rely on it flowing neither continuously nor forward. NTP could be implemented using `clock_settime()`, and could only affect `CLOCK_REALTIME`.

The Linux kernel implementation seems to take boot time as the epoch for `CLOCK_MONOTONIC`: Starting point for CLOCK_MONOTONIC

Share Improve this answer Follow

edited Jan 11, 2022 at 17:06

Andy
4,107 2 22 39

answered Dec 17, 2016 at 22:11

Ciro Santilli
OurBigBook.com
380k • 117 • 1.3k • 1.1k

1 That's what I really wanted to know! Thank you!
– Timur Fayzrakhmanov Sep 12, 2020 at 15:37

23

In addition to Ignacio's answer, `CLOCK_REALTIME` can go up forward in leaps, and occasionally backwards. `CLOCK_MONOTONIC` does neither; it just keeps going forwards (although it probably resets at reboot).

A robust app needs to be able to tolerate `CLOCK_REALTIME` leaping forwards occasionally (and perhaps backwards very slightly very occasionally, although that is more of an edge-case).

Imagine what happens when you suspend your laptop - `CLOCK_REALTIME` jumps forwards following the resume, `CLOCK_MONOTONIC` does not. Try it on a VM.

Share Improve this answer Follow

edited May 23, 2017 at 11:54
Community Bot
1 • 1

answered Aug 19, 2010 at 16:20
MarkR
63.5k • 15 • 119 • 154

5 CLOCK_MONOTONIC starts at 0 when the program starts; it is not for interprocess use. – Benubird Feb 9, 2011 at 10:31

21 @Benubird: It does not start at 0 when the program starts. That's `CLOCK_PROCESS_CPUTIME_ID`. Quick test: `$ perl -w -MTime::HiRes=clock_gettime,CLOCK_MONOTONIC -E 'say clock_gettime(CLOCK_MONOTONIC)'` --> 706724.117565279. That number matches system uptime on Linux, but the standard says its arbitrary. – derobert Aug 23, 2011 at 20:16

6 As an aside, I do not believe that the Linux behaviour where `CLOCK_MONOTONIC` stops over a suspend/resume is POSIX-conforming. It's supposed to be the time since a fixed point in the past, but stopping the clock over suspend/resume breaks that. – caf Jul 16, 2013 at 10:31

11

Sorry, no reputation to add this as a comment. So it goes as an complementary answer.

Depending on how often you will call `clock_gettime()`, you should keep in mind that only *some* of the "clocks" are provided by Linux in the VDSO (i.e. do not require a syscall with all the overhead of one -- which only got worse when Linux added the defenses to protect against Spectre-like attacks).

While `clock_gettime(CLOCK_MONOTONIC,...)`, `clock_gettime(CLOCK_REALTIME,...)`, and `gettimeofday()` are always going to be extremely fast (accelerated by the VDSO), this is *not* true for, e.g. CLOCK_MONOTONIC_RAW or any of the other POSIX clocks.

This can change with kernel version, and architecture.

Although most programs don't need to pay attention to this, there can be latency spikes in clocks accelerated by the VDSO: if you hit them right when the kernel is updating the shared memory area with the clock counters, it has to wait for the kernel to finish.

Here's the "proof" (GitHub, to keep bots away from kernel.org): https://github.com/torvalds/linux/commit/2aae950b21e4bc789d1fc6668faf67e8748300b7

Share Improve this answer Follow

edited Jun 25, 2020 at 14:14

SCouto
7,928 5 34 51

answered Jun 25, 2020 at 12:57

anonymous
131 1 4

1

There's one big difference between CLOCK_REALTIME and MONOTONIC. CLOCK_REALTIME can jump forward or backward according to NTP. By default, NTP allows the clock rate to be speeded up or slowed down by up to 0.05%, but NTP cannot cause the monotonic clock to jump forward or backward.

Share Improve this answer Follow

answered Jan 24, 2021 at 20:38

selcukakarin
48 6

1

I'd like to clarify what "the system is suspended" means under this context.

I am reading `timefd_create` and from the manpage, https://man7.org/linux/man-pages/man2/timerfd_create.2.html

> ```
>  CLOCK_BOOTTIME (Since Linux 3.15)
>  Like CLOCK_MONOTONIC, this is a monotonically increasing
>  clock.  However, whereas the CLOCK_MONOTONIC clock does
>  not measure the time while a system is suspended, the
>  CLOCK_BOOTTIME clock does include the time during which
>  the system is suspended.  This is useful for applications
>  that need to be suspend-aware. CLOCK_REALTIME is not
>  suitable for such applications, since that clock is
>  affected by discontinuous changes to the system clock.
> ```

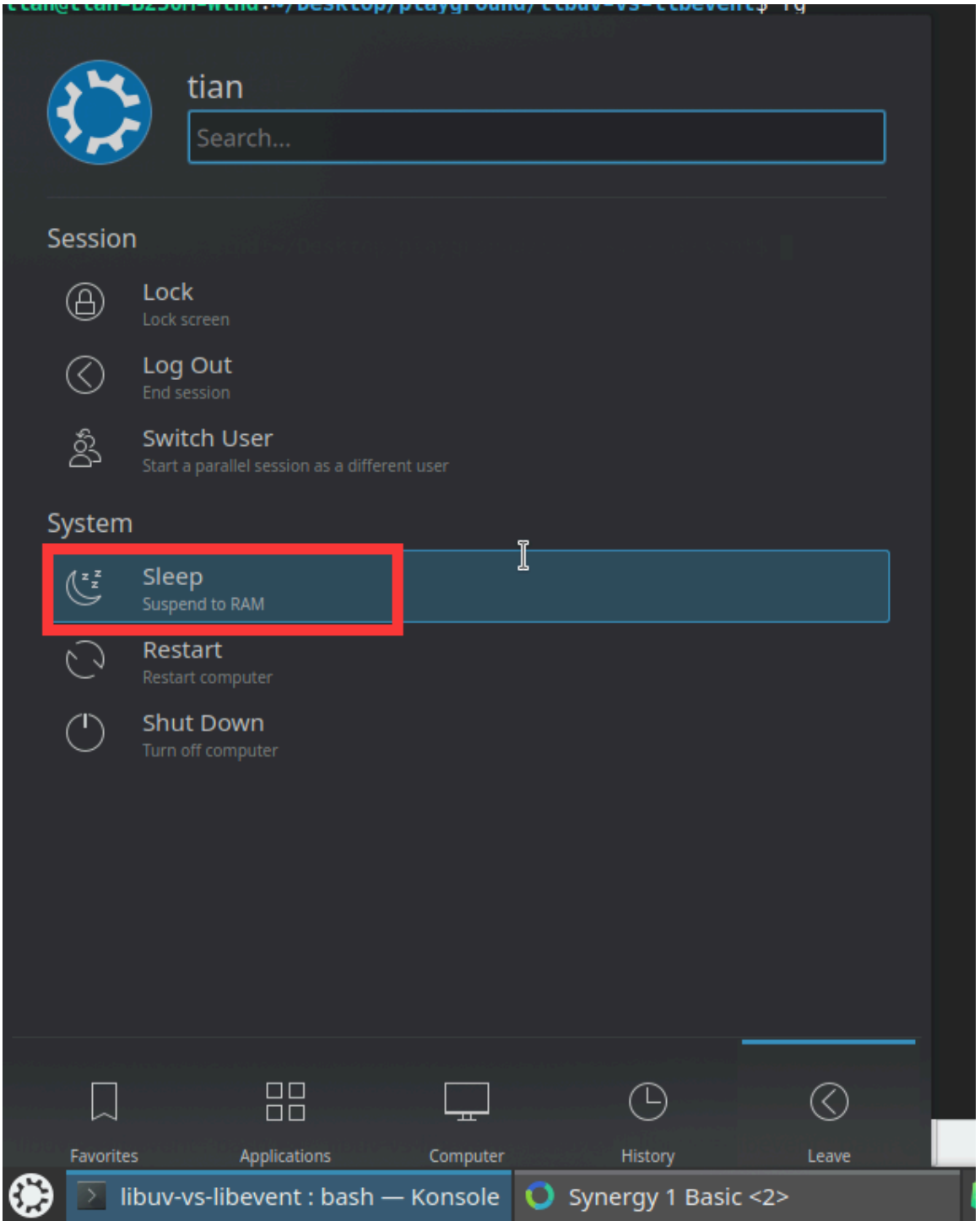
Based on the above description, we can indicate that `CLOCK_REALTIME` and `CLOCK_BOOTTIME` still count time when the system is suspended, while `CLOCK_MONOTONIC` doesn't.

I was confused about what "the system is suspended" mean exactly. At first I was thinking it means when we send `Ctrl + Z` from the terminal, making the process suspended. But it's not.

@MarkR's answer inspired me:

> Imagine what happens when you **suspend your laptop** - .... **Try it on a VM**.

So literally "the system is suspended" means you put your computer into sleep mode.



That said, `CLOCK_REALTIME` counts the time when the computer is asleep.

Compare the output of these 2 pieces of code

timefd_create_realtime_clock.c

copy from `man timefd_create`

```
#include <sys/timerfd.h>
#include <time.h>
#include <unistd.h>
#include <inttypes.h>      /* Definition of PRIu64 */
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>        /* Definition of uint64_t *

#define handle_error(msg) \
        do { perror(msg); exit(EXIT_FAILURE); } while

static void
print_elapsed_time(void)
{
    static struct timespec start;
    struct timespec curr;
    static int first_call = 1;
    int secs, nsecs;

    if (first_call) {
        first_call = 0;
        if (clock_gettime(CLOCK_MONOTONIC, &start) ==
            handle_error("clock_gettime");
    }

    if (clock_gettime(CLOCK_MONOTONIC, &curr) == -1)
        handle_error("clock_gettime");

    secs = curr.tv_sec - start.tv_sec;
    nsecs = curr.tv_nsec - start.tv_nsec;
```

```
    if (nsecs < 0) {
        secs--;
        nsecs += 1000000000;
    }
    printf("%d.%03d: ", secs, (nsecs + 500000) / 10000
}

int
main(int argc, char *argv[])
{
    struct itimerspec new_value;
    int max_exp, fd;
    struct timespec now;
    uint64_t exp, tot_exp;
    ssize_t s;

    if ((argc != 2) && (argc != 4)) {
        fprintf(stderr, "%s init-secs [interval-secs m
                argv[0]);
        exit(EXIT_FAILURE);
    }

    if (clock_gettime(CLOCK_REALTIME, &now) == -1)
        handle_error("clock_gettime");

    /* Create a CLOCK_REALTIME absolute timer with ini
        expiration and interval as specified in comman

    new_value.it_value.tv_sec = now.tv_sec + atoi(argv
    new_value.it_value.tv_nsec = now.tv_nsec;

    if (argc == 2) {
        new_value.it_interval.tv_sec = 0;
        max_exp = 1;
    } else {
        new_value.it_interval.tv_sec = atoi(argv[2]);
        max_exp = atoi(argv[3]);
    }
    new_value.it_interval.tv_nsec = 0;

    fd = timerfd_create(CLOCK_REALTIME, 0);
    if (fd == -1)
        handle_error("timerfd_create");
```

```
    if (timerfd_settime(fd, TFD_TIMER_ABSTIME, &new_va
        handle_error("timerfd_settime");

    print_elapsed_time();
    printf("timer started\n");

    for (tot_exp = 0; tot_exp < max_exp;) {
        s = read(fd, &exp, sizeof(uint64_t));
        if (s != sizeof(uint64_t))
            handle_error("read");

        tot_exp += exp;
        print_elapsed_time();
        printf("read: %" PRIu64 "; total=%" PRIu64 "\n
    }

    exit(EXIT_SUCCESS);
}
```

timefd_create_monotonic_clock.c

modify a bit, change `CLOCK_REALTIME` to `CLOCK_MONOTONIC`

```
#include <sys/timerfd.h>
#include <time.h>
#include <unistd.h>
#include <inttypes.h>      /* Definition of PRIu64 */
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>        /* Definition of uint64_t *

#define handle_error(msg) \
        do { perror(msg); exit(EXIT_FAILURE); } while

static void
print_elapsed_time(void)
{
    static struct timespec start;
    struct timespec curr;
    static int first_call = 1;
    int secs, nsecs;
```

```
    if (first_call) {
        first_call = 0;
        if (clock_gettime(CLOCK_MONOTONIC, &start) ==
            handle_error("clock_gettime");
    }

    if (clock_gettime(CLOCK_MONOTONIC, &curr) == -1)
        handle_error("clock_gettime");

    secs = curr.tv_sec - start.tv_sec;
    nsecs = curr.tv_nsec - start.tv_nsec;
    if (nsecs < 0) {
        secs--;
        nsecs += 1000000000;
    }
    printf("%d.%03d: ", secs, (nsecs + 500000) / 10000
}

int
main(int argc, char *argv[])
{
    struct itimerspec new_value;
    int max_exp, fd;
    struct timespec now;
    uint64_t exp, tot_exp;
    ssize_t s;

    if ((argc != 2) && (argc != 4)) {
        fprintf(stderr, "%s init-secs [interval-secs m
                argv[0]);
        exit(EXIT_FAILURE);
    }

    // T_NOTE: comment
    // if (clock_gettime(CLOCK_REALTIME, &now) == -1)
        // handle_error("clock_gettime");

    /* Create a CLOCK_REALTIME absolute timer with ini
        expiration and interval as specified in comman

    // new_value.it_value.tv_sec = now.tv_sec + atoi(a
    // new_value.it_value.tv_nsec = now.tv_nsec;
```

```
    new_value.it_value.tv_sec = atoi(argv[1]);
    new_value.it_value.tv_nsec = 0;
    if (argc == 2) {
        new_value.it_interval.tv_sec = 0;
        max_exp = 1;
    } else {
        new_value.it_interval.tv_sec = atoi(argv[2]);
        max_exp = atoi(argv[3]);
    }
    new_value.it_interval.tv_nsec = 0;

    // fd = timerfd_create(CLOCK_REALTIME, 0);
    fd = timerfd_create(CLOCK_MONOTONIC, 0);
    if (fd == -1)
        handle_error("timerfd_create");

    // if (timerfd_settime(fd, TFD_TIMER_ABSTIME, &new
    if (timerfd_settime(fd, 0, &new_value, NULL) == -1
        handle_error("timerfd_settime");

    print_elapsed_time();
    printf("timer started\n");

    for (tot_exp = 0; tot_exp < max_exp;) {
        s = read(fd, &exp, sizeof(uint64_t));
        if (s != sizeof(uint64_t))
            handle_error("read");

        tot_exp += exp;
        print_elapsed_time();
        printf("read: %" PRIu64 "; total=%" PRIu64 "\n
    }

    exit(EXIT_SUCCESS);
}
```

1. compile both and run in 2 tabs in same terminal

   `./timefd_create_monotonic_clock 3 1 100`

   `./timefd_create_realtime_clock 3 1 100`

2. **put my Ubuntu Desktop into sleep**

3. Wait a few miniutes
4. Wake up my Ubuntu by pressing power button once
5. Check the terminal output

Output:

The realtime clock stopped immedicately. Because it've counted the time elapsed when the computer is suspended/asleep.

```
tian@tian-B250M-Wind:~/playground/libuv-vs-
libevent$ ./timefd_create_realtime_clock 3 1 100
0.000: timer started
3.000: read: 1; total=1
4.000: read: 1; total=2
5.000: read: 1; total=3
6.000: read: 1; total=4
7.000: read: 1; total=5
8.000: read: 1; total=6
9.000: read: 1; total=7
10.000: read: 1; total=8
11.000: read: 1; total=9
12.000: read: 1; total=10
13.000: read: 1; total=11
14.000: read: 1; total=12
15.000: read: 1; total=13
16.000: read: 1; total=14
17.000: read: 1; total=15
18.000: read: 1; total=16
19.000: read: 1; total=17
20.000: read: 1; total=18
21.000: read: 1; total=19
22.000: read: 1; total=20
23.000: read: 1; total=21
24.000: read: 1; total=22
25.000: read: 1; total=23
26.000: read: 1; total=24
27.000: read: 1; total=25
28.000: read: 1; total=26
```

```
29.000: read: 1; total=27
30.000: read: 1; total=28
31.000: read: 1; total=29
33.330: read: 489; total=518 # wake up here
tian@tian-B250M-Wind:~/playground/libuv-vs-
libevent$
```

```
tian@tian-B250M-Wind:~/Desktop/playground/libuv-
vs-libevent$ ./timefd_create_monotonic_clock 3 1
100
0.000: timer started
3.000: read: 1; total=1
3.1000: read: 1; total=2
4.1000: read: 1; total=3
6.000: read: 1; total=4
7.000: read: 1; total=5
7.1000: read: 1; total=6
9.000: read: 1; total=7
10.000: read: 1; total=8
11.000: read: 1; total=9
12.000: read: 1; total=10
13.000: read: 1; total=11
14.000: read: 1; total=12
15.000: read: 1; total=13
16.000: read: 1; total=14
16.1000: read: 1; total=15
18.000: read: 1; total=16
19.000: read: 1; total=17
19.1000: read: 1; total=18
21.000: read: 1; total=19
22.001: read: 1; total=20
23.000: read: 1; total=21
25.482: read: 2; total=23
26.000: read: 1; total=24
26.1000: read: 1; total=25
28.000: read: 1; total=26
28.1000: read: 1; total=27
29.1000: read: 1; total=28
30.1000: read: 1; total=29
31.1000: read: 1; total=30
32.1000: read: 1; total=31
33.1000: read: 1; total=32
```

```
35.000: read: 1; total=33
36.000: read: 1; total=34
```

Share Improve this answer Follow

answered Nov 14, 2022 at 1:25

Rick

7,466 2 54 91