LINUTRONIX

Surviving 19 Jan 2038 on 32 Bit Platforms: Lessons Learned and Common Problems

Alexander Kanavin
alex@linutronix.de
alex.kanavin@gmail.com

What is time? What time is it?

- Obtaining time on Unix: issue a POSIX API call time(), and get a single signed integer that contains amount of seconds since 1 January 1970
- That is super convenient: uses an elementary fixed-size datatype, one can perform direct arithmetic operations with it (add, subtract). Can be used for both absolute time and intervals.
- There is no standard specifying how many bits in the integer
- The obvious implementation choice (with less obvious problems): let's just use one of C integer types

What's the problem? Why the problem?

Binary : 01111111 11111111 11111111 11111111

Decimal : 2147483647

Date : 2038-01-19 03:14:07 (UTC)

Date : 2038-01-19 03:14:07 (UTC)



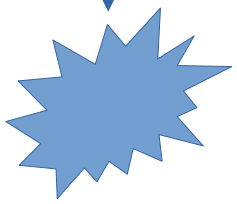
Binary : 10000000 00000000 00000000 00000000

Decimal : -2147483648

Date : 1901-12-13 20:45:52 (UTC)

Date : 2038-01-19 03:14:08 (UTC)





Source: https://en.wikipedia.org/wiki/Year_2038_problem

- Until C99, the biggest integer type in C was ‚long'
- ‚long' is defined as „at least 32 bits"
- On 32 bit hardware that many bits can be handled directly with 32 bit registers and cpu instructions
- Never mind future-proof correctness of time

Demo one: Yocto 4.0 ‚kirkstone‘ (released in spring 2022)

```
Poky (Yocto Project Reference Distro) 4.0.19 qemux86 /dev/ttyS0

qemux86 login: root
root@qemux86:~# date -s "2038-01-19 03:14"
Tue Jan 19 03:14:00 UTC 2038
root@qemux86:~# date
Tue Jan 19 03:14:02 UTC 2038
root@qemux86:~# date
Thu Jan  1 00:00:00 UTC 1970
root@qemux86:~# date
Thu Jan  1 00:00:00 UTC 1970
root@qemux86:~# ls /
bin         boot        dev         etc         home        lib         lost+f
root@qemux86:~# ls /
ls: /: Value too large for defined data type
root@qemux86:~# reboot
INIT: Switching to runlevel: 6
INIT: Sending processes configured via /etc/inittab the TERM signalroot@qemux8
INIT: /run/initctl is not a fifo
hwclock: RTC_SET_TIME: Invalid argument
INIT: /run/initctl is not a fifo
stopped syslogd (pid 286)
INIT: /run/initctl is not a fifo
stopped klogd (pid 289)
done
/etc/init.d/rc: /etc/rc6.d/K31umountnfs.sh: line 19: halt: Value too large for
Unmounting remote filesystems...
Deconfiguring network interfaces... ifdown: interface lo not configured
done.
Sending all processes the TERM signal...
killall5: /proc unavailable.

root@qemux86:~#
```

Demo two: Yocto 5.0 ‚scarthgap‘ (released in spring 2024)

```
Poky (Yocto Project Reference Distro) 5.0.2 qemux86 /dev/ttyS0

qemux86 login: root

WARNING: Poky is a reference Yocto Project distribution that should be used for
testing and development purposes only. It is recommended that you create your
own distribution for production use.

root@qemux86:~# date -s "2038-01-19 03:14"
Tue Jan 19 03:14:00 UTC 2038
root@qemux86:~# date
Tue Jan 19 03:14:02 UTC 2038
root@qemux86:~# date
Tue Jan 19 03:14:09 UTC 2038
root@qemux86:~# date
Tue Jan 19 03:14:16 UTC 2038
root@qemux86:~# ls /
bin         boot        dev         etc         home        lib         lost+found  m
root@qemux86:~# ls /
bin         boot        dev         etc         home        lib         lost+found  m
root@qemux86:~# ls /
bin         boot        dev         etc         home        lib         lost+found  m
root@qemux86:~#
```

Y2038 safety on 32 bit systems: how to get there?

- Replace 32 bit time with 64 bit time throughout the software stack
  - should be good for the next 292 billion years; check https://en.wikipedia.org/wiki/Future_of_an_expanding_universe for a detailed future timeline according to the current consensus in cosmology
  - Kernel
    - Userspace API
    - Internal time handling
    - Filesystems
  - C library
- Full rebuild of the stack from source, including evil vendor blobs
- Simulated testing of Y2038 actually happening (or having already happened)
- Through that testing, ensure userspace doesn't write 64 bit time into 32 bits internally
  - By programmer mistake (C makes it easy)
  - Deliberately

The kernel API towards userspace

- 64 bit systems always used 64 bit time API
- Recent 32 bit systems (x86-x32, 32 bit risc-v) had 64 bit time API from the start
- ‚Classic‘ 32 bit systems got an API extension in 5.1 linux kernel release, adding 64 bit versions of time related API calls; 32 bit API stayed as it was (but can be optionally disabled via COMPAT_32BIT_TIME option)
- Those systems are: x86, arm, ppc, mips, and a few others

```
commit 48166e6ea47d23984f0b481ca199250e1ce0730a
Author: Arnd Bergmann <arnd@arndb.de>
Date:   Thu Jan 10 12:45:11 2019 +0100

    y2038: add 64-bit time_t syscalls to all 32-bit architectures

    This adds 21 new system calls on each ABI that has 32-bit time_t
    today. All of these have the exact same semantics as their existing
    counterparts, and the new ones all have macro names that end in 'time64'
    for clarification.

    This gets us to the point of being able to safely use a C library
    that has 64-bit time_t in user space. There are still a couple of
    loose ends to tie up in various areas of the code, but this is the
    big one, and should be entirely uncontroversial at this point.
```

LINUTRONIX

The kernel internals

```
7eeca8ccd106 ARM: OMAP: timer32K: fix all kernel-doc warnings
46dae32fe625 time: Correct the prototype of ns_to_kernel_old_timeval and ns_to_timespec64
7df621a3eea6 Merge git://git.kernel.org/pub/scm/linux/kernel/git/netdev/net
c1ff12dac465 perf bench futex: Add support for 32-bit systems with 64-bit time_t
4c1e34c0dbff vsock: Enable y2038 safe timeval for timeout
90c90cda05ae Merge tag 'xfs-5.15-merge-6' of git://git.kernel.org/pub/scm/fs/xfs/xfs-linux
d0e3fc69d00d powerpc/vdso: Provide __kernel_clock_gettime64() on vdso32
b2f1c3db2887 kselftest: Extend vdso correctness test to clock_gettime64
cb47755725da time: Prevent undefined behaviour in timespec64_to_ns()
2fc61f25fb29 Merge tag 'xfs-5.10-merge-2' of git://git.kernel.org/pub/scm/fs/xfs/xfs-linux
4ea1ff3b4968 xfs: widen ondisk quota expiration timestamps to handle y2038+
f93e5436f0ee xfs: widen ondisk inode timestamps to deal with y2038+
5631c5e0eb90 Merge tag 'xfs-5.9-merge-7' of git://git.kernel.org/pub/scm/fs/xfs/xfs-linux
a990f7a84edc xfs: improve ondisk dquot flags checking
fc94cf2092c7 sh: include linux/time_types.h for sockios
b0dd1eb220c0 Merge branch 'akpm' (patches from Andrew)
c766d1472c70 y2038: hide timeval/timespec/itimerval/itimerspec types
412c53a680a9 y2038: remove unused time32 interfaces
595abbaff5db y2038: remove ktime to/from timespec/timeval conversion
08dffcc7d94b Merge tag 'nfsd-5.6' of git://linux-nfs.org/~bfields/linux
11648b8339f8 sparc64: fix adjtimex regression
846de71bedef Merge tag 'media/v5.6-1' of git://git.kernel.org/pub/scm/linux/kernel/git/mchehab/linux-media
9ca4c6429f92 Merge git://git.kernel.org/pub/scm/linux/kernel/git/davem/sparc
d68712ee3506 y2038: sparc: remove use of struct timex
22b17db4ea05 Merge tag 'y2038-drivers-for-v5.6-signed' of git://git.kernel.org:/pub/scm/linux/kernel/git/arnd/playground
```

- How to review: git log -i --grep=y2038 –oneline
- The bulk of the changes merged in 5.6
  - https://lkml.org/lkml/2020/1/29/355?anz=web
- XFS fixes continued to happen until 5.15
- I can’t say what to do with „evil vendor kernels“ except don’t get your kernels from evil vendors
- Make sure the filesystem in your products is not using 32 bit timestamps (e.g. ext2 or ufs or unfixed xfs)
- A moment of appreciation for Arnd Bergmann

Extending time in standard C library and its API – how to redefine time_t?

- In libc API time is represented with **time_t** type
- „Used for time in seconds. According to POSIX, it is an integer type.“ (says ‚man time_t‘)

- Musl C library switched over to 64 bits in time_t with 1.2 release: https://musl.libc.org/time64.html
- Musl users: simply rebuild your complete product with that (including evil vendor blobs)
- Glibc story is more complicated as it’s used on binary Linux distros (that don’t offer a full-rebuild option) and can’t break backwards compatibility this way

Extending time in glibc

- On 32 bit systems glibc has pairs of 32 bit and 64 bit versions of every time-related API call in its binary (since 2.34), for example (can be seen with nm on libc.so.6):

  000cf370 T __time64@@GLIBC_2.34

  000cf3c0 W time@@GLIBC_2.0

- Glibc headers will define time_t as 32 bit, and link with 32 bit API *by default*
- To opt into 64 bit versions, pass in -D_TIME_BITS=64 to compiler and rebuild the full system. Glibc headers will do magic to make it happen.
  - Don’t forget to rebuld the ‚evil vendor blobs‘ from source!
- _TIME_BITS=64 is set globally in Yocto since version 4.3 ‚nanbield‘
  - This is why Yocto 4.0 (kirkstone) system collapses in the demo, and 5.0 (scarthgap) survives :-)
- ‚new‘ 32 bit systems (risc-v, x86-x32) have had only 64 bit time_t from the start in glibc; the problems are on arm, x86, ppc, and mips (and more esoteric retro computing targets) (this is same as in the kernel api)

Extending time in glibc again

- Glibc's header tricks won't help when someone does not rely on C headers and is
  - directly linking to 32 bit symbols in glibc (there's a yocto qa test for it)
  - using dlsym() to look up symbols at runtime
- An example (and fix) in glib-networking:

tls/tests/connection.c: look up the 64bit version of clock_gettime as well

Merged **Alexander Kanavin** requested to merge `alex.kanavin/glib-networ…` into `master` 10 months ago

Overview 4 | Commits 1 | Pipelines 5 | **Changes 1**

Compare **master** and **latest version**

tls/tests/connection.c

```
@@ -588,7 +588,11 @@ clock_gettime (clockid_t clk_id,
588 588    int ret = -1;
589 589    if (!original_clock_gettime)
590 590      {
    591 + #if (_TIME_BITS == 64)
    592 +       original_clock_gettime = dlsym (RTLD_NEXT, "__clock_gettime64");
    593 + #else
591 594        original_clock_gettime = dlsym (RTLD_NEXT, "clock_gettime");
    595 + #endif
592 596        if (!original_clock_gettime)
593 597          {
594 598            errno = EINVAL;
```

Extending time in glibc with rust (and how they messed it up)

- Rust's libc crate, too, bypasses glibc headers, and ends up linking to 32 bit API
  - This was caught by the no-32-bit-symbol yocto QA test (mentioned on previous slide)
- There's a pull request https://github.com/rust-lang/libc/pull/3175/ but it's stuck in protracted ‚review hell' since spring 2023
- Assuming it merges, these are still needed:
  - A libc crate release with it
    - Every crate updating to that release in their Cargo.toml (rust ecosystem vendors everything into the build tree, ‚system libraries' are not a thing in that world)
      - Every crate making a release with that Cargo.toml update
        - Every crate updating to those releases in their own Cargo.toml
          - Every crate making a release with that Cargo.toml update
            - It isn't clear when to stop the recursion, but limited slide space forces me to do it here
  - Distributions updating to those fixed releases

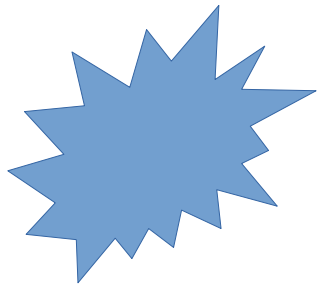
Everything mentioned so far has been checked and done; how to test a system's Y2038 safety?

- Have a clear definition of a ‚working system'
- Have **tests** implementing that definition
- Have **automated** tests implementing that definition
- Have automated **target-installable** tests implementing that definition
- Run them with the system clock set to beyond Y2038 rollover
  - Better option: boot the system with the clock set to just before, and wait until it rolls over (this tests the transition too, and not just the ‚static' post-2038 situation)

How is Yocto's Y2038 readiness tested (work is ongoing)

- Yocto's ‚working system' definition is here: https://git.yoctoproject.org/yocto-autobuilder-helper/tree/config.json
- A set of build and test configurations, executed on the ‚autobuilder cluster' against every proposed set of changes
- Target testing only in qemu; the project has no resources for real silicon testing
- For Y2038 I privately modified the definition to:
  - Start qemu with the system clock set to 2040:

    QB_OPT_APPEND:append = " -rtc base=2040-02-02" (put this in local.conf or similar)
  - Y2038 rollover hasn't yet been tested
- Covered in the next slides:



Before we talk about failures and issues, let's talk about importance of installable tests

- Tests that can be installed and executed on the target are especially important for Y2038, as Y2038 issues will not be exposed on build host (that has 64 bit time and runs in 2024)
- GNOME gets it right: https://wiki.gnome.org/Initiatives/GnomeGoals/InstalledTests (required reading as a homework assignment!)
- Python gets it mostly right too (via pytest)
- GNU (and by extension everything using autotools) does not (‚make check' problem):
  - Yocto obtains installable tests from the most important of those projects through awful recipe hacks (manual copy out of build tree into ${D}, then butchering Makefiles with write-only sed expressions to run ‚make check' on target)
- Yocto users: more installable tests (known as ‚ptests') are welcome! Add them to recipes that matter to you and send us patches.

What are the issues exposed by target testing in Yocto?

Broadly four types:

- Bypassing the redirection tricks in glibc headers, and directly linking to functions with 32 bit time (already covered earlier)
- Test data that assumes the test is run in ‚the present‘ and fails when taken with a time machine to ‚the future‘
- Deliberately using 32 bit integers for generic integer handling, including timestamps
- The worst: accidentally writing timestamps into 32 bit integers

Let’s look at the last three one by one.

Test data that can’t handle the ‚future‘

- These are typically things that have ‚expiration dates‘ inside them, and those dates are set to before Y2038 (sometimes intentionally so, to allow testing on unfixed 32 bit systems until they altogether collapse)
  - Certificates
  - HTTP cookies
- Fixes are often easy, but it can be surprisingly hard to land them upstream:
  - Upstream doesn’t know how to regenerate the data because the developer that had produced it has disappeared (this is not good, and gives me ‚Jia Tan‘ vibes). Example: https://github.com/openssl/openssl/issues/21671#issuecomment-1878080242
  - Upstream repeatedly doesn’t react: https://github.com/openssh/openssh-portable/pull/425
  - Upstream has vague fears that something somewhere might break (even when their own CI says otherwise), and the pull request then languishes in ‚review hell‘: https://github.com/python/cpython/pull/107594

Intentionally using 32 bits for integers (and then storing timestamps in them)

Typically some kind of ‚runtime' that can execute ‚scripts', and provide an integer type to those scripts. Then the scripts use that integer type to store timestamps.

Fortunately both known examples can be reconfigured at build time:

- Perl: ./configure -Duse64bitint
- Busybox: CONFIG_FEATURE_SH_MATH_64=y in defconfig

Yocto 5.0 ‚scarthgap' already does this. You're welcome.

Why C is not my favorite programming language (and today it's not because of memory safety)

- ‚Classic' pre-C99 integer types (and long long in C99) are defined as „at least this many bits, and should be a natural fit to the hardware". Check your copy of K&R (section 2.2) !
- Compiler doesn't care if a something with more bits is assigned to something with less bits, truncating data at runtime (and the language specifically allows it: K&R 2.7)
- It is assumed that C programmers always know what they're doing with the types, and mistakes like the below are the programmer's fault.
- Reality is something else: mistakes like these are easy to make even for experienced programmers, and this is primarily due to the compiler and the language not helping.

```
  GNU nano 7.2
#include <stdio.h>
#include <time.h>

int main()
{
    printf("size of long %d\n",sizeof(long));
    printf("size of time_t %d\n",sizeof(time_t));

    time_t sometime;
    long somel;

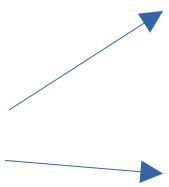
    sometime = time(NULL);
    somel = sometime;

    printf("current time_t %lld\n", sometime);
    printf("assigned to long %ld\n", somel);

    sometime = 0x10000000002;
    somel = sometime+1;

    printf("post-y2038 time_t %lld\n", sometime);
    printf("assigned to long %ld\n", somel);

}
```



```
alex@alex-lx-laptop:~/development/y2038-demos/c-types$ make
i686-linux-gnu-gcc -Wall -Wextra -D_TIME_BITS=64 -D_FILE_OFFSET_BITS=64 ./test_types_time_t.c -o test_type_time_t
alex@alex-lx-laptop:~/development/y2038-demos/c-types$ ./test_type_time_t
size of long 4
size of time_t 8
current time_t 1719842893
assigned to long 1719842893
post-y2038 time_t 1099511627778
assigned to long 3
alex@alex-lx-laptop:~/development/y2038-demos/c-types$
```

Accidentally writing 64 bit time_t into 32 bit long: the discouraging list of examples

- Perl: https://github.com/Perl/perl5/pull/21379
- Tcl: https://core.tcl-lang.org/tcl/tktview/86dd172271
- Dbus: https://gitlab.freedesktop.org/dbus/dbus/-/merge_requests/444
- Util-linux: https://github.com/util-linux/util-linux/pull/2430
- Glib: https://gitlab.gnome.org/GNOME/glib/-/merge_requests/3547
- Openssl: https://github.com/openssl/openssl/pull/21206

All of these are key ingredients in a Linux system, and the issues were found only because Yocto has **installable tests** for them that were executed on a **target 32 bit system with the date set to the future.**

All 32 bit systems are affected by such issues, including **32 bit Risc V** (that was designed for 64 bit kernel API and time_t from the ground up).

Why I'm worried: how many more such issues are currently **unknown**? We have tests for some of the key pieces; many more core components aren't tested in this way.

Ask the audience: can time_t → long be checked with static analysis? (without drowning in irrelevant noise)

Should we just migrate to 64 bit hardware? Is 64 bit Y2038 safe?

- It can be more viable to replace deployed hardware products with 64 bit versions (if the total cost is less than the cost of fixing and verifying 32 bit stack). Consider this option seriously.
- 64 bit systems are much *safer* but they are not *safe*. Why?
  - 32 bit compatibility libraries used to run legacy 32 bit apps (sometimes delivered decades ago as evil vendor blobs). Exact same issues as just covered! Don't forget to check your system :)
  - Accidentally writing time_t into int – not as likely as writing time_t to long, but not impossible
  - Specific issues like squeezing timestamps into 32 bit in networking protocols, databases or in particular application-specific protocols/software.
- 64 bit systems should be verified in the same way as 32 bit systems for y2038 safety

Conclusion-ish

- There's no way to conclusively ‚prove' that any given system is Y2038-safe. There are only degrees of likelihood.
- The scariest sub-problem is the last one: accidentally truncating bits in C integers.
- Have tests for a complete target system, run them with the clock set to post-2038.
- We need **legally required SBOMs**, so the public can know what is **inside critical infrastructure** that our lives depend on.
  - Ideally complete source code as well. I hope this will be in the public discussion before a serious accident with multiple (or mass!) casualties forces the issue.
- Are you confident to travel or stay in a hospital on 19 Jan 2038? Do you have a wood-heated cabin to stay in?

Resources for Yocto users:

- https://git.yoctoproject.org/poky/tree/meta/conf/distro/include/time64.inc - settings for qemu, settings for affected 32 bit hardware targets, list of known issues, exceptions for few components with good reasons to be excepted etc.
- https://git.yoctoproject.org/poky-contrib/log/?h=akanavin/y2038 - my private branch used for testing, with component patches, links to upstream tickets and pull requests etc. - you are free to cherry-pick from it as needed. Periodically rebased, as fixes land upstream, and releases with them are integrated into Yocto proper – hopefully this branch will be reduced to nothing over time.

LINUTRONIX GmbH
Bahnhofstr. 3 | D – 88690 Uhldingen
Phone +49 7556 25 999 0
sales@linutronix.de

www.linutronix.de



Thanks for your attention

Any questions ?