International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 04 Issue 05
April 2017

Y2K38 BUG

1. Velaga.Sriman Sandeep,2. G. Venkata Prasad,3. J. Deepthi

1.Pg Scholar, Department of CSE,Sphoorthy Engineering College, Nadergul(vill),Sagar Road, Saroornagar(Mdl), RR Dist TS.

2. Assistant Professor, Department of CSE,Sphoorthy Engineering College, Nadergul(vill),Sagar Road, Saroornagar(Mdl), R R Dist TS.

3. Associate Professor & HOD, Department of CSE,Sphoorthy Engineering College, Nadergul(vill),Sagar Road, Saroornagar(Mdl), R R Dist TS

ABSTRACT

The Y2k38 bug was detected a few months ago, but there was no actual testing done to prove that this will affect our current computers. Now this bug can actually start to cause some damage. To test this we can try using any chatting client either AOL Messenger , MSN Messenger, Yahoo Messenger, Trillian, or Gaim will do. As for my testing this bug affected all of these applications (windows). To test these for yourself do the following: First double click on the time in the bottom right-hand corner. Change the date to something after January 19, 2038 (2039 is fine!). Now start up any of the chatting clients mentioned above. Try sending a message to someone else (or to yourself if it supports it). As soon as you do this, the entire program should crash within a few seconds and will display some sort of error message. It will then bring up a message asking to send an error report or debug. From the actual technical problem, apparently as soon as the year 2038 strikes, all certain computers will immediately get confused and switch the date to December 13, 1901 from January 2038.

INTRODUCTION

The Year 2000 problem is understood by most people these days because of the large amount of media attention it received. Most programs written in the C programming language are relatively immune to the Y2K problem, but suffer instead from the Year 2038 problem. This problem arises because most C programs use a library of routines called the standard time library (time.h). This library establishes a standard 4-byte format for the storage of time values, and also provides a number of functions for converting, displaying and calculating time values.

The Y2K38 problem has been described as a non-problem, given that we are expected to be running 64-bit operating systems well before 2038.

Just as Y2K problems arise from programs not allocating enough digits to the year, Y2K38 problems arise from programs not allocating enough bits to internal time. Unix internal time is commonly stored in a data structure using a long int containing the number of seconds since 1970. This time is

used in all time-related processes such as scheduling, file timestamps, etc. In a 32-bit machine, this value is sufficient to store time up to 18-jan-2038. After this date, 32-bit clocks will overflow and return erroneous values such as 32-dec-1969 or 13-dec-1901.

WHAT IS THE YEAR 2038 BUG?

In the first month of the year 2038 C.E. many computers will encounter a date-related bug in their operating systems and/or in the applications they run. This can result in incorrect and wildly inaccurate dates being reported by the operating system and/or applications. The effect of this bug is hard to predict, because many applications are not prepared for the resulting "skip" in reported time - anywhere from 1901 to a "broken record" repeat of the reported time at the second the bug occurs. Also, leap seconds may make some small adjustment to the actual time the bug expresses itself. This bug expects to cause serious problems on many platforms, especially Unix and Unix-like platforms, because these systems will "run out of time". Starting at GMT 03:14:07, Tuesday, January 19, 2038, we fully expect to see lots of systems around the world breaking magnificently: satellites falling out of orbit, massive power outages (like the 2003 North American blackout), hospital life support system failures, phone system interruptions (including 911 emergency services), banking errors, etc. One second after this critical second, many of these systems will have wildly inaccurate date settings, producing all kinds of unpredictable consequences. In short, many of the dire predictions for the year 2000 are much more likely to actually occur in the year 2038! Consider the year 2000 just a dry run. In case you think we can sit on this issue for another 30 years before addressing it, consider that reports of temporal echoes of the 2038 problem are already starting to appear in future date calculations for mortgages and vital statistics!

WHAT HAPPENS IN YEAR 2038?

Most programs written in the C programming language are relatively immune to the Y2K problem, but suffer instead from the Year 2038 problem. This problem arises because most C programs use a library of routines called the standard time library (time.h). This library establishes a standard 4-byte format for the storage of time values, and also provides a number of functions for converting, displaying and calculating time values.

The standard 4-byte format assumes that the beginning of time is January 1, 1970, at 12:00:00 a.m. This value is 0. Any time/date value is expressed as the number of seconds following that zero value. So the value 919642718 is 919,642,718 seconds past 12:00:00 a.m. on January 1, 1970, which is Sunday, February 21, 1999, at 16:18:38 Pacific time (U.S.). This is a convenient format because if you subtract any two values, what you get is a number of seconds that is the time difference between them. Then you can use

other functions in the library to determine how many minutes/hours/days/months/years have passed between the two times.

The maximum value of time before it rolls over to a negative (and invalid) value is 2,147,483,647, which translates into January 19, 2038. On this date, any C programs that use the standard time library will start to have problems with date calculations.

This problem is somewhat easier to fix than the Y2K problem on mainframes, fortunately. Well-written programs can simply be recompiled with a new version of the library that uses, for example, 8-byte values for the storage format. This is possible because the library encapsulates the whole time activity with its own time types and functions (unlike most mainframe programs, which did not standardize their date formats or calculations). So the Year 2038 problem should not be nearly as hard to fix as the Y2K problem was.

The cause of the Y2K problem is pretty simple. Until recently, computer programmers have been in the habit of using two digit placeholders for the year portion of the date in their software.

For example, the expiration date for a typical insurance policy or credit card is stored in a computer file in MM/DD/YY format (e.g. - 08/31/99). Programmers have done this for a variety of reasons, including:

- That's how everyone does it in their normal lives. When you write a check by hand and you use the "slash" format for the date, you write it like that.
- It takes less space to store 2 digits instead of 4 (not a big deal now because hard disks are so cheap, but it was once a big deal on older machines).
- Standards agencies did not recommend a 4-digit date format until recently.
- No one expected a lot of this software to have such a long lifetime. People writing software in 1970 had no reason to believe the software would still be in use 30 years later.

The 2-digit year format creates a problem for most programs when "00" is entered for the year. The software does not know whether to interpret "00" as "1900" or "2000". Most programs therefore default to 1900. That is, the code that most programmer's wrote either prepends "19" to the front of the two-digit date, or it makes no assumption about the century and therefore, by default, it is "19". This wouldn't be a problem except that programs perform lots of calculations on dates. For example, to calculate how old you are a program will take today's date and subtract your birthdate from it. That subtraction works fine on two-digit year dates until today's date and your birthdate are in different centuries. Then the calculation no longer works. For example, if the program thinks that today's date is 1/1/00 and your birthday is 1/1/65, then it may calculate that you are -65 years old

rather than 35 years old. As a result, date calculations give erroneous output and software crashes or produces the wrong results.

The important thing to recognize is that that's it. That is the whole Year 2000 problem. Many programmers used a 2-digit format for the year in their programs, and as a result their date calculations won't produce the right answers on 1/1/2000. There is nothing more to it than that.

The solution, obviously, is to fix the programs so that they work properly. There are a couple of standard solutions:

- Recode the software so that it understands that years like 00, 01, 02, etc. really mean 2000, 2001, 2002, etc.
- "Truly fix the problem" by using 4-digit placeholders for years and recoding all the software to deal with 4-digit dates. [Interesting thought question - why use 4 digits for the year? Why not use 5, or even 6? Because most people assume that no one will be using this software 8,000 years from now, and that seems like a reasonable assumption. Now you can see how we got ourselves into the Y2K problem...]

Either of these fixes is easy to do at the conceptual level - you go into the code, find every date calculation and change them to handle things properly. It's just that there are millions of places in software that have to be fixed, and each fix has to be done by hand and then tested. For example, an insurance company might have 20 or 30 million lines of code that performs its insurance calculations. Inside the code there might be 100,000 or 200,000 date calculations. Depending on how the code was written, it may be that programmers have to go in by hand and modify each point in the program that uses a date. Then they have to test each change. The testing is the hard part in most cases - it can take a lot of time.

If you figure it takes one day to make and test each change, and there's 100,000 changes to make, and a person works 200 days a year, then that means it will take 500 people a year to make all the changes. If you also figure that most companies don't have 500 idle programmers sitting around for a year to do it and they have to go hire those people, you can see why this can become a pretty expensive problem. If you figure that a programmer costs something like $150,000 per year (once you include everything like the programmer's salary, benefits, office space, equipment, management, training, etc.), you can see that it can cost a company tens of millions of dollars to fix all of the date calculations in a large program.

The year-2038 bug is similar to the Y2K bug in that it involves a time wrap not handled by programmers. In the case of Y2K, many older machines did not store the century digits of dates, hence the year 2000 and the year 1900 would appear the same.

Of course we now know that the prevalence of computers that would fail because of this error was greatly exaggerated

by the media. Computer scientists were generally aware that most machines would continue operating as usual through the century turnover, with the worst result being an incorrect date. This prediction withstood through to the new millennium. Effected systems were tested and corrected in time, although the correction and verification of those systems was monumentally expensive.

There are however several other problems with date handling on machines in the world today. Some are less prevalent than others, but it is true that almost all computers suffer from one critical limitation. Most programs use Coordinated Universal Time (UTC) to work out their dates. Simply, UTC is the number of seconds elapsed since Jan 1 1970. A recent milestone was Sep 9 2001, where this value wrapped from 999'999'999 seconds to 1'000'000'000 seconds. Very few programs anywhere store time as a 9 digit number, and therefore this was not a problem.

Modern computers use a standard 4 byte integer for this second count. This is 31 bits, storing a value of 231. The remaining bit is the sign. This means that when the second count reaches 2147483647, it will wrap to -2147483648.

The precise date of this occurrence is Tue Jan 19 03:14:07 2038. At this time, a machine prone to this bug will show the time Fri Dec 13 20:45:52 1901, hence it is possible that the media will call this The Friday 13th Bug

C LANGUAGE AND Y2K 38

For the uninitiated, time_t is a data type used by C and C++ programs to represent dates and times internally. (You Windows programmers out there might also recognize it as the basis for the CTime and CTimeSpan classes in MFC.) time_t is actually just an integer, a whole number, that counts the number of seconds since January 1, 1970 at 12:00 AM Greenwich Mean Time. A time_t value of 0 would be 12:00:00 AM (exactly midnight) 1-Jan-1970, a time_t value of 1 would be 12:00:01 AM (one second after midnight) 1-Jan-1970, etc.. Since one year lasts for a little over 31 000 000 seconds, the time_t representation of January 1, 1971 is about 31 000 000, the time_t representation for January 1, 1972 is about 62 000 000, et cetera. If you're confused, here are some example times and their exact time_t representations:

Date & time	time_t representation
1-Jan-1970, 12:00:00 AM GMT	0
1-Jan-1970, 12:00:01 AM GMT	1
1-Jan-1970, 12:01:00 AM GMT	60

Date & time	time_t representation
1-Jan-1970, 01:00:00 AM GMT	3 600
2-Jan-1970, 12:00:00 AM GMT	86 400
3-Jan-1970, 12:00:00 AM GMT	172 800
1-Feb-1970, 12:00:00 AM GMT	2 678 400
1-Mar-1970, 12:00:00 AM GMT	5 097 600
1-Jan-1971, 12:00:00 AM GMT	31 536 000
1-Jan-1972, 12:00:00 AM GMT	63 072 000
1-Jan-2003, 12:00:00 AM GMT	1 041 379 200
1-Jan-2038, 12:00:00 AM GMT	2 145 916 800
19-Jan-2038, 03:14:07 AM GMT	2 147 483 647

By the year 2038, the time_t representation for the current time will be over 2 140 000 000. And that's the problem. A modern 32-bit computer stores a "signed integer" data type, such as time_t, in 32 bits. The first of these bits is used for the positive/negative sign of the integer, while the remaining 31 bits are used to store the number itself. The highest number these 31 data bits can store works out to exactly 2 147 483 647. A time_t value of this exact number, 2 147 483 647, represents January 19, 2038, at 7 seconds past 3:14 AM Greenwich Mean Time. So, at 3:14:07 AM GMT on that fateful day, every time_t used in a 32-bit C or C++ program will reach its upper limit.

One second later, on 19-January-2038 at 3:14:08 AM GMT, disaster strikes.

International Journal of Research
Available at https://edupediapublications.org/journals
p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 04 Issue 05
April 2017

What will the time_t's do when this happens?

Signed integers stored in a computer don't behave exactly like an automobile's odometer. When a 5-digit odometer reaches 99 999 miles, and then the driver goes one extra mile, the digits all "turn over" to 00000. But when a signed integer reaches its maximum value and then gets incremented, it wraps around to its lowest possible negative value. (The reasons for this have to do with a binary notation called "two's complement") This means a 32-bit signed integer, such as a time_t, set to its maximum value of 2 147 483 647 and then incremented by 1, will become -2 147 483 648. Note that "-" sign at the beginning of this large number. A time_t value of -2 147 483 648 would represent December 13, 1901 at 8:45:52 PM GMT.

So, if all goes normally, 19-January-2038 will suddenly become 13-December-1901 in every time_t across the globe, and every date calculation based on this figure will go haywire. And it gets worse. Most of the support functions that use the time_t data type cannot handle negative time_t values at all. They simply fail and return an error code. Now, most "good" C and C++ programmers know that they are supposed to write their programs in such a way that each function call is checked for an error return, so that the program will still behave nicely even when things don't go as planned. But all too often, the simple, basic, everyday functions they call will "almost never" return an error code, so an error condition simply isn't checked for. It would be too tedious to check everywhere; and besides, the extremely rare conditions that result in the function's failure would "hardly ever" happen in the real world. (Programmers: when was the last time you checked the return value from printf() or malloc()?) When one of the time_t support functions fails, the failure might not even be detected by the program calling it, and more often than not this means the calling program will crash. Spectacularly

An example C program

The follow C program demonstrates this effect. It is strict ANSI C so it should compile on all systems that support an ANSI C compiler.

```
#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <time.h>

int main (int argc, char **argv)

{

   time_t t;

   t = (time_t) 1000000000;

   printf ("%d, %s", (int) t, asctime (gmtime
(&t)));

   t = (time_t) (0x7FFFFFFF);

   printf ("%d, %s", (int) t, asctime (gmtime
```

```
(&t)));

   t++;

   printf ("%d, %s", (int) t, asctime (gmtime
(&t)));

   return 0;

}
```

The program produces the output:

```
10000000000,  Sun Sep  9 01:46:40  2001

2147483647,  Tue Jan 19 03:14:07  2038

-2147483648,  Fri Dec 13 20:45:52  1901
```

FIXING Y2K38 BUG

Time_t is never, ever at fault in any Year 2000 bug. Year 2000 bugs usually involve one of three things: The user interface, i.e., what year do you assume if the user types in "00"; a database where only the last two digits are stored, i.e., what year do you assume if the database entry contains a 00 for its year; and, in rare instances, the use of data items (such as the struct tm data structure's tm_year member in a C or C++ program) which store the number of years since 1900 and can result in displays like "19100" for the year 2000.

Year 2038 bugs, on the other hand, occur when a program reads in a date and carries it around from one part of itself to another.

You see, time_t is a convenient way to handle dates and times inside a C or C++ program. For example, suppose a program reads in two dates, date A and date B, and wants to know which date comes later. A program storing these dates as days, months, and years would first have to compare the years, then compare the months if the years were the same, then compare the days if the months were the same, for a total of 3 comparison operations. A program using time_t's would only have to compare the two time_t values against each other, for a total of 1 comparison operation. Additionally, adding one day to a date is much easier with a time_t than having to add 1 to the day, then see if that puts you past the end of the month, then increase the month and set the day back to 01 if so, then see if that puts you past the end of the year, et cetera. If dates are manipulated often, the advantage of using time_t's quickly becomes obvious. Only after the program is done manipulating its time_t dates, and wants to display them to the user or store them in a database, will they have to be converted back into days, months, and years.

So, even if you were to fix every Year 2000 Bug in a program in such a way that users and databases could use years as large as 9999, it wouldn't even brush on any of the Year 2038 Bugs lurking within the same program.

The Problem with Pooh-Poohing

Admittedly, some of our colleagues don't feel that this impending disaster will strike too many people. They reason that, by the time 2038 rolls around, most programs will be running on 64-bit or even 128-bit computers. In a 64-bit program, a

time_t could represent any date and time in the future out to 292 000 000 000 A.D., which is about 20 times the currently estimated age of the universe.

The problem with this kind of optimism is the same root problem behind most of the Year 2000 concerns that plagued the software industry in previous years: Legacy Code. Developing a new piece of software is an expensive and time-consuming process. It's much easier to take an existing program that we know works, and code one or two new features into it, than it is to throw the earlier program out and write a new one from scratch. This process of enhancing and maintaining "legacy" source code can go on for years, or even decades. The MS-DOS layer still at the heart of Microsoft's Windows 98 and Windows ME was first written in 1981, and even it was a quick "port" (without many changes) of an earlier operating system called CP/M, which was written in the 1970s. Much of the financial software hit by the various Year 2000 bugs had also been used and maintained since the 1970s, when the year 2000 was still thought of as more of a science fiction movie title than an actual impending future. Surely, if this software had been written in the 1990s its Year 2000 Compliance would have been crucial to its authors, and it would have been designed with the year 2000 in mind. But it wasn't.

We should also mention that computer designers can no longer afford to make a "clean break" with the computer architectures of the past. No one wants to buy a new kind of PC if it doesn't run all their old PC's programs. So, just as the new generation of Microsoft Windows operating systems has to be able to run the old 16-bit programs written for Windows 3 or MS-DOS, so any new PC architecture will have to be able to run existing 32-bit programs in some kind of "backward compatibility" mode.

Even if every PC in the year 2038 has a 64-bit CPU, there will be a lot of older 32-bit programs running on them. And the larger, more complex, and more important any program is, the better are its chances that that it'll be one of these old 32-bit programs.

What about making time_t unsigned in 32-bit software?

One of the quick-fixes that has been suggested for existing 32-bit software is to re-define time_t as an unsigned integer instead of a signed integer. An unsigned integer doesn't have to waste one of its bits to store the plus/minus sign for the number it represents. This doubles the range of numbers it can store. Whereas a signed 32-bit integer can only go up to 2 147 483 647, an unsigned 32-bit integer can go all the way up to 4 294 967 295. A time_t of this magnitude could represent any date and time from 12:00:00 AM 1-Jan-1970 all the way out to 6:28:15 AM 7-Feb-2106, surely giving us more than enough years for 64-bit software to dominate the planet.

It sounds like a good idea at first. We already know that most of the standard time_t handling functions don't accept negative time_t values anyway, so why not just make time_t into a data type that only represents positive numbers?

Well, there's a problem. time_t isn't just used to store absolute dates and times. It's also used, in many applications, to store differences between two date/time values, i.e. to answer the question of "how much time is there between date A and date B?". (MFC's CTimeSpan class is one notorious example.) In these cases, we do need time_t to allow negative values. It is entirely possible that date B comes before date A. Blindly changing time_t to an unsigned integer will, in these parts of a program, make the code unusable.

Conclusion

Though the Y2k38 problem does not affect the IT industry as that done by Y2k, it has significance in the area of embedded systems. Many real time system that purely depends on system clock may hit by the problem . The solution to the problem is done by some patch programs that will skip the effects . Majorities of the bug programs will be done in C language. Converting the present applications to 64 bit one is the final word on the solution. This can be implemented professionally by adding library functions that do the conversion.

References

1. http://www.merlyn.demon.co.uk/critdate.htm#Dates
2. fi.wikipedia.org/wiki/Y2K38
3. www.seminarsonly.com
4. blogger.sahaskatta.com/2005/07/y2k

AUTHOR'S PROFILE:

Velaga.Sriman Sandeep
B.Tech Student,
Department of CSE,
Sphoorthy Engineering College,
Nadergul(vill),Sagar Road,
Saroornagar(Mdl), R R Dist TS.

G.Venkata Prasad
Assistant Professor,
Department of CSE,
Sphoorthy Engineering College,
Nadergul(vill),Sagar Road,
Saroornagar(Mdl), R R Dist TS.