Who's Afraid Of 2038?

Michael G Schwern
schwern@pobox.com
or
Saving The Future From The Past
Hi

I'm Michael Schwern
#!/usr/bin/perl -w
use Test::More tests => 2;
is( $testing => "easy" );
perl Makefile.PL
make
make test
make install

Maintain MakeMaker
package Films;

use base qw(Class::DBI);

Films->table('movies');
Please ask questions.
It's a long talk.
schwern@pobox.com
Y2K?

Remember y2k?
When the whole world was going to explode?
And we'd be living in our y2k shelters
Dogs & cats, living together.
Mass hysteria!
y2k came and it didn't happen.
So here it is in 2008 and we're home free, right?
If I can't have my flying car, I should at least be able to have a computer that knows what day it is.
J.R. Stockton's
"Critical and Significant Dates"
http://www.merlyn.demon.co.uk/critdate.htm

Interesting read.
Which is not about...
...your dating history,
but time and date values in the past and future,
from...
13,700,000,000 BC

The beginning of the Universe, generally considered to be a bad idea
13,700,000,000 BC
(around teatime)

The beginning of the Universe,
generally considered to be a bad idea
1970-01-01
Thu 00:00:00:00 GMT

To the beginning of the Unix epoch
Unix counts the number of seconds since 1970.
To the beginning of the Unix epoch

Unix counts the number of seconds since 1970.
600,000,000,000 AD

To the end of the world
And all the crazy exceptions in between.
600,000,000,000,000 AD
(Tower of Hanoi completed)

To the end of the world
And all the crazy exceptions in between.
600,000,000,000 AD
(Tower of Hanoi completed)
(just after Perl 6.0.0)

To the end of the world
And all the crazy exceptions in between.
So what's the problem?
It turns out that in dating...
Size matters
$time = 2**31-1;
print scalar gmtime($time);
$time = 2**31-1;
print scalar gmtime($time);

Tue Jan 19 03:14:07 2038
Uh oh

We've gone from...
$time += 1;
print scalar gmtime($time);

Fri Dec 13 20:45:52 1901

Uh oh

We've gone from...
No car

Normally Perl lets us forget that we're running software on actual silicon
But hardware has limits.
And this is because of...
...or whatever the C folks say.
Perl is supposed to shield us from all that.
But sometimes it doesn't.
32 bit signed double floating point static registers

...or whatever the C folks say.
Perl is supposed to shield us from all that.
But sometimes it doesn't.
32 bit signed integer
31 bits for the integer
1 bit for the sign
$2^{31} == 2,147,483,648$

$-2^{31} == -2,147,483,648$
\[(2^{31}) - 1 \equiv 2,147,483,647\]
\[-2^{31} \equiv -2,147,483,648\]
2038-01-19
Tue 03:14:07
2**31 seconds

in 2038, 32 bit signed Unix time fails
292,277,026,596-12-04
Sun 15:30:08
2**63 seconds

Y 292 billion
64 bit signed Unix time fails
5,391,559,471,918,239,497,011,222,876,596-04-18
Mon 16:02:08
2**127 seconds

128 bit fails at the year
5 non-illions
256 bit at the year
1 million vig-int-illion
There are only $10^{80}$ particles in the universe. So in year $2^{10^{80}}$, it is now impossible to write the year.
Oddly enough, there was an argument about what to do with 2038 on p5p. (Imagine that) The C programmers on p5p said that...
gmtime() uses time_t

(internally)
time_t is a signed 32 bit integer!

(on a 32 bit machine)
signed 32 bit numbers overflow at $2^{31}$!
2147483647 + 1

Of course.
Therefore...
2147483647 + 1

==
Of course.
Therefore...
$ perl -wle 'print scalar gmtime(2**31)'

Fri Dec 13 20:45:52 1901

...you got exactly what you asked for.

And BTW, you should be using a 64 bit clean machine.
But this is Perl.
And Perl is supposed to...
And I don't care about...
More importantly, my clients don't care. When they get ...
32 bit signed double floating point static registers

More importantly, my clients don't care. When they get ...
...1901
I can't say "Oh, that's because of..."
...and expect them to nod knowingly.

No, they'll say...
32 bit signed double floating point static registers

...and expect them to nod knowingly.

No, they'll say...
...you're FIRED!
A number is a number.

(Perl is a closet Ayn Rand fan)
I don't care how it's represented internally
Perl gives no indication how it's represented internally.
There's no reason a Perl programmer can suspect there's any limits.
So we have to fix this.
Dude...

Wait, what?

Why do you care?
29 years

There's less than 30 years to go!
In 30 years, I'll be eligible for the...
...I can only hope I look that good in 30 years.

But...
Stuff happens in the future

This might seem obvious, but people seem to pretend it doesn't.
How many people have a 30 year mortgage?

Perhaps you want to construct..
...some sort of death clock.

I plan on living past 2038.
Stuff happens in the past.

Like... photographs!
Pittsburgh, 1941
1941, arsenal of democracy

Apparently Picassa doesn't do anything before 1970!
April 20th, 1975
167,220,000 seconds

The most important date in my life.
I was born.
Fortunately after the Unix epoch.
Some people were born before 1970.
Some systems can't handle negative times.
Windows can't.
How many people here are still using (or know someone using) 5.6?

Seven years from now, people will still be using 5.10. They will still be using 32 bit machines. Or 64 bit machines with 32 bit time_t.
How many people here are still using (or know someone using) 5.6?

Seven years from now, people will still be using 5.10. They will still be using 32 bit machines. Or 64 bit machines with 32 bit time_t.
Let's do the math...

In 2015, with 22 years to go, people will likely still be using 5.10.0

How many people have a 20 year mortgage?

So we'd better fix this NOW
Let's do the math...

In 2015, with 22 years to go, people will likely still be using 5.10.0

How many people have a 20 year mortgage?

So we'd better fix this NOW
Let's do the math...

In 2015, with 22 years to go, people will likely still be using 5.10.0

How many people have a 20 year mortgage?

So we'd better fix this NOW
So that's the problem.
Solution?

Before we look at the solution
Let's look at what's not a solution.
Blame the user!

No, only C programmers think this is a feature.
We'll all upgrade to 64 bit!
And if you use Linux, and use a 64 bit distribution, that's fine.

How many Macs out there?
New Macs use 64 bit processors.
They do not use a 64 bit time_t.
Why?
Computers store numbers in a fixed size.
Storing 3 32 bit numbers is 96 bits
(the embarassing part is I had to calculate that in Ruby) But 3 64 bit integers is 192 bits This means the same binary program can't run on a 32 and a 64 bit machine. Linux users are used to recompiling and multiple distributions Apple and Windows users aren't. Apple has clever ways to deal with this Did it for Motorola -> PPC, PPC -> Intel but they didn't think it's worth the effort Why?
Windows tried it.
Most PCs these days use a 64 bit processor.
Nobody uses 64 bit Windows

People are really bad about keeping their types straight
Assume that an integer is 32 bit and such.
And, of course, not everyone is going to upgrade their hardware.

Really what "let's all upgrade to 64 bit" is saying is...
works on
my
machine
Don't use the built-ins, use DateTime!

Yes, DateTime is fantastic, but a lot of things depend on the built-ins. And a lot of other languages and projects depend on localtime and gmtime. C, Python, Perl and Ruby all make use of it. And leaving the built-ins broken leaves Perl with no built in safe date handling. We could suck in DateTime, but there's a problem with that I'll get to in a moment.

This is special case of the...
Just use <insert library here>

libtai is one
It's fast
It's reliable
It avoids the 2038 problem...
It doesn't do time zones.
Time zones are important
Time zones are insane
Time zones change
Oh, and there's daylight savings time
All this information is in your operating system's time zone database.
It gets updated along with the operating system.
There is no portable API to the OS' time zone database.

Things like DateTime ship their own time zone database.
But that's a lot of work.
And now you have two time zone databases to keep up to date.
(That you probably don't know exist)
Daylight Saving Time was first suggested as a joke by Benjamin Franklin in his whimsical essay "An Economical Project for Diminishing the Cost of Light" published in the Journal de Paris (1784-04-26).
ENERGY POLICY ACT OF 2005
(b) Effective Date.--Subsection (a) shall take effect 1 year after the date of enactment of this Act or March 1, 2007, whichever is later.

So we had about a year to adjust all the computers to match. Probably cost far more money and energy than it saved.
SEC. 110. DAYLIGHT SAVINGS.

(a) Amendment.--Section 3(a) of the Uniform Time Act of 1966 (15 U.S.C. 260a(a)) is amended--
(1) by striking ``first Sunday of April''
and inserting ``second Sunday of March''; and
(2) by striking ``last Sunday of October'' and
inserting ``first Sunday of November''.
(c) Report to Congress.--Not later than 9 months after the effective date stated in subsection (b), the Secretary shall report to Congress on the impact of this section on energy consumption in the United States.
(d) Right to Revert.--Congress retains the right to revert the Daylight Saving Time back to the 2005 time schedules once the Department study is complete.

So we might have to put it right back!
A) Handle time zones
b) Use the system tz database
3) Be portable (ANSI C 89)
iv) Compatible license w/Perl
É) localtime/gmtime compatible

Nothing exists.
A) Handle time zones
b) Use the system tz database

GIVEN

No tz database API

This seems impossible.

Well, I lied. There is, sort of.
date = localtime(time);
time = mktime(date);

ANSI C 89 has two functions which talk to the time zone database. This is the wedge we'll use

mktime() is the inverse of localtime()
How to do it?
Step 1: 
Write 64 bit gmtime() 

And that's relatively easy. It's just a bunch of math. Perl already detects a native 64 bit integer type (Quad_t).

"Easy" because writing ANSI C 89 is hard.

But if strings aren't involved it's fine.
Before we get into step 2
Let's talk about calendars.
The calendar we all think we use is the Julian Calendar, introduced in the west in 46 BC by Julius Caesar.

It estimates that a year is 365 1/4 days. Thus a leap year every 4 years. This is slightly wrong, but we'll get to that.

Looking at on a calendar, there are two important attributes.
What week day is Jan 1st?
\[365 \mod 7 == 1\]

2002    Tuesday
     +1
2003    Wednesday
     +1
2004    Thursday
366 \% 7 == 2

2004 Thursday
   +2
2005 Saturday
   +1
2006 Sunday
2. Is it a leap year?
7 possible starting days
Leap year every 4 years

7 \times 4 = 28

28 year Julian cycle
Given any year on the Julian calendar
We can add or subtract 28 years to get the same calendar year

Ah ha!
2038  ->  2010

We can map future dates which localtime() can't handle back to earlier ones
And we can map past dates to ones inside localtime's safe range.

1971 – 2037 is safe
But, of course, we don't use the Julian Calendar
Else we'd be 2 weeks ahead of the sun.
We use the Gregorian calendar
This has more complex leap year rules.
It changes the rules for centuries.
400 year cycle.
2038 - 1970 < 400 years

localtime() doesn't have 400 years to do the mapping.

But we can fake it!
Taking another look at the Julian cycle
<table>
<thead>
<tr>
<th>2099</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Here's 2099
2100 (not a leap year)
Where does 2101 go?

Non-leap years only add 1.

5 + 1 = 6
There it is.
It turns out you just add 16 to the year to adjust for exceptional centuries.

There's a little more numerology, but that's the hack.
# Get date in GMT
$date = gmtime64($time);
my $real_year = $date->{year};

# Map GMT year to 1970-2038
$date->{year} = safe_year($real_year);

# Get the epoch time for that
my $safe_time = timegm($date);

# Then the localized date
$safe_date = localtime($safe_time);

# Map the date forward again
$safe_date->{year} = $real_year;

That's the basic algorithm.
There's some issues around the new year.
gmtime Jan 1st might be Dec 31 in the local zone
How accurate is this?

Calculating future local times is always dicey, the rules might change.

Past local times will become less accurate, but at a certain point the time zone simply no longer exists. What's does Eastern Standard Time mean in 1589?

The real danger is mapping near future dates to a point just before the rules changed. As we get closer to 2038 the problem will become more acute.

So at worst you'll be off by an hour or two.
$ perl -wle 'print scalar gmtime 2**31-1'
Tue Jan 19 03:14:07 2038

$ perl -wle 'print scalar gmtime 2**31'
Fri Dec 13 20:45:52 1901

This is better than being off by 138 years.
Rather than just patch it in Perl
I've decided to write a portable 64 bit clean POSIX time.h
http://y2038.googlecode.com/

You can get it here.

By making it generic, now we can fix...
Perl
localtime()
gmtime()

(Time::Local)
timelocal()
timegm()
Python

time module
date.timestamp
Ruby

Time.at
Time.UTC
Time.local
bleadperl?
But I want it NOW!
Older module

Did a similar hack with localtime, but without the careful year mapping (just subtracted 60 years).

Hopefully the patched version will be released soon. There's a patch on y2038. You can use that to make your Perl code y2038 safe.
use Time::Local::Extended;

print scalar gmtime(2**31);
print scalar localtime(2**31);

# Tue Jan 19 03:14:08 2038
# Mon Jan 18 22:14:08 2038

Does timelocal() and timegm() too.

How far does it go?

For practical purposes, it can do about 142 million years accurately.
use Time::Local::Extended;
print scalar localtime 2**52;
print scalar localtime -2**52;

# Fri Dec 5 22:48:16 142715360
# Mon Jan 25 15:11:44 -142711421

This is because Perl has no portable 64 bit integer scalar type. (I hope to fix that)
So it has to use double floats which are accurate to 2**52.
After that it loses accuracy.

How far can it go?
That -513 is because of floating point error.
292,277,026,596

292 billion
13.7 billion years
(age of the universe)

So it gives us some room to work.
Once you break the 2038 barrier you start to run into new problems.
The struct in which C stores the year is defined to be an integer.

This leads to the...
y2,147,483,648 bug

Have to be careful to store the year as a 64 bit int. Many 64 operating systems miss this. y2038 defines its own Year type to help with that.

So time64.h has the option of using a y2 billion safe time struct. bleadperl uses that.
Windows' C functions have a y3001 bug, I have no idea why.
y2038 gets around that
y10k

HP/UX has a y10k bug

y2038 gets around that
In the opposite direction we have the
Year Zero Problem

Year Zero problem

What does 0 mean?
If you ask a historian
They'll say "there is no year zero"
1 BC + 1 year = 1 AD

And historians are fine with this
Because they still do their work on scrolls
And it's not an exact science
And being a year off isn't too important

This is also why 2001 is the "real millennium".

For math and computers, the discontinuity is difficult.
This sucks.

So what does the Gregorian calendar say?
Nope, no year 0
ISO 8601?

How about the International date standard?

Nope.
Values in the range [0000] through [1582] shall only be used by **mutual agreement of the partners in information interchange.**

Which is ISO-speak for "we couldn't decide"

But they do have a year 0 and they have the best cop out.
What is year \( -2 \) in ISO?
So what's year zero?
It's the year before year one.

So simple!
Turns out, this is how astronomers do it.

They're the only people who need to worry about really long dates
And do a lot of math where being one year off matters.
1 = 1 AD
0 = 1 BC
-1 = 2 BC
Step 3: Profit!
Fortunately TPF funded my project to do all this.
(and thus you all, through your donations)
Merijn Brand is helped hooking it into the Perl guts
http://y2038.googlecode.com/

The project needs help.

I'm the only developer.
I'm not a C programmer.

Only a handful of time.h functions have been implemented.
Python needs a lot more.
If you're a C programmer, or want to learn, please sign up.
There's a user's list. I'm liberal with handing out commit bits.
Be bold.
Thanks...
...for your money.
...for your money.

;)