by Peter Collinson, Hillside Systems

PAUL SCHULENBURG

The Time

Happy New Year. As I write this, I have more than the usual feeling of posting an article into the future. I am on one side of a great divide, you on the other. I am pre, you are post January 1, 2000. I'm finding it really hard to get excited about the new millennium, I'm afraid. I am sure that it will just be like a birthday. When you wake up on the morning of a birthday, you always expect to feel different somehow. Sadly you don't, and you always feel cheated because you don't. For the millennium, there is a faint worry that the computing systems that I run will break in some way. You are able to read this knowing one way (or the other) what happened to you. I actually expect to feel cheated that things worked out fine.

What now? You should probably make sure your systems know that 2000 is a leap year, if you haven't already done so. My fax machine thinks "00" is 1900, so it will supply the correct day and date until March. Then there's a bunch of other fun questions to consider. We have a year to worry about what to call 2001–are we allowed to use 1 or 01, or will we be forced to write four digits? There seems to be some date standard now that forces the use of four-digit dates. I think that suddenly Roman numerals seem attractive again. After all, MM is a short string. We also have a year to complain that any mention of "being in the 21st century" is bogus. There was no year zero and 2000 is the last year of the 20th century.

Once the dust has settled, let's all start worrying seriously about 2038; specifically Tuesday, January 19, 2038 at 03:14:07 GMT. At this point, there will have been exactly 2,147,483,647 seconds since midnight on January 1, 1970, which is the time that Ken Thompson decided to set as the epoch for UNIX internal time measurement. The next clock tick after that makes the 32-bit integer that holds the time a negative number.

Number Theory

Incrementing a value and obtaining a negative number may seem odd, but it's a fact of life on a computer. The effect is due to the way that integer numbers are represented. Numbers are stored in computer "words" that contain a fixed number of binary digits, or "bits." We want to perform arithmetic on the numbers, so there has to be a way of expressing positive and negative numbers. For most users, computers use signed integers where the "top" bit, or leftmost bit, is a sign bit. When the top bit is "0" we have a positive number and when it's "1" we have a negative number. If we look at a computer with a three-bit word, then:

```
0 is binary 000
1 is binary 001
2 is binary 010
3 is binary 011
```

If, for example, we add 1 to 2, we are adding binary 001 to binary 010, which is 011 and that equals 3. Negative numbers have the top bit set:

-1 is binary 111
-2 is binary 110
-3 is binary 101
-4 is binary 100

If we add 1 to -2, we are adding binary 001 to binary 110, deriving 111, which is -1. The coding means that we don't have to make the computer subtract, we just make it add negative values.

I've picked some easy examples above, ones that don't create any "carry." When you add any numbers together, you start with the right-most pair of digits and add them together to derive the sum for that digit position. Of course, there's always the possibility of a carry value that needs to be added into the pair of digits to the left of the first pair. When adding binary digits, a carry is generated when we add 1 to 1. The result is 0 and a carry of 1 is added into the computation of the next digit position. Bearing this in mind, and returning to our three-bit word computer, try adding 1 to -1, or binary 001 to binary 111. The carry that's generated by the addition of the right-most bits ripples up the three-bit word and drops off the top, and the final result is 000, which is decimal 0.

When the result of addition is "in range," the mechanism works well. However, what happens when we add a 1 to 3? We are creating a number that is larger than the three-bit word computer can handle; we have no coding for 4. But we get a valid result, the binary value 100, which in our coding system is -4. What's happened is that we've incremented a positive number and ended up with a negative one. This effect is exactly what will happen on January 19, 2038 at 03:14:07 GMT.

Looking After the Time

The way that UNIX maintains the time is simple. The hardware has a clock that's usually ticking at 1,000 times per second. On each of these ticks, the processor is interrupted and runs code in the kernel that increments a central location. Programs interrogate the kernel to obtain this value. When the system is bootstrapped, the central value is set to a starting value that we interpret as the time by using the `date` program.

We normally run UNIX systems with this central value set to GMT, and layers in the system libraries perform the computation to adjust for the local time zone, or daylight savings time. It's not "part of UNIX" to run the clock on GMT, you can run the system with the local time loaded into the kernel's clock. However, when you put the clocks back at the start of winter, you'll need to change the kernel's value. You'll subtract 3,600 seconds, and time stamps that are derived from the clock will be incorrect because that hour will happen again. It's more convenient to let the system do the work of displaying the correct time based on a continuously advancing kernel value.

However, as we've seen, this continuously advancing value will eventually "go negative," creating an out-of-range time.

Why has UNIX picked a 32-bit signed integer to represent time and not a 32-bit unsigned integer? UNIX practice goes back to the days of PDP-11, which operated with 16-bit integers, but `time` was always returned as a 32-bit value. It's just simple expediency. A 16-bit clock, even unsigned, doesn't last very long (about 18 hours) and the next step is a 32-bit value.

When UNIX migrated to 32-bit machines in the 1980s, this 32-bit value was preserved, but became the same size as the standard integer on the machine. There are loads of programs out there in userland that expect to be able to perform computations on time, and for these purposes, an integer is an ideal representation. For example, when evaluating elapsed time, we can just subtract one time value from another. When comparing two times, we can use the normal comparison instructions that the hardware uses to compare integers.

The ANSI C standard carefully avoided placing any limits on the number of bits that are used to represent time. It defined that the `time` system call should return a value with a special `time_t` type that implied nothing about how many bits should be used to represent time. The POSIX standard extended this definition to state that the system call should return time in seconds from 00:00 on January 1, 1970. Neither standard said anything about how many bits should be used to represent time.

As we approach 2038, and get closer to the point where our time representation goes out of range, we have two options. First, we could make the `time` routine return an unsigned 32-bit integer. Doing this gives us approximately until 2106 before we run out of bits again. The second alternative is to go to a 64-bit signed internal time representation. The number of years that this value supplies is mind-boggling geologically: we run out of bits around the year 292,471,210,647 (I am not worrying about leap years in that computation).

Actually, Sun Microsystems has perhaps preempted the use of a 32-bit unsigned number. If you are running a current Solaris 2.6 system in January 2038 when the clock's time value turns negative, your `date` command will show `Fri, Dec 13 21:45:52 1901`. The effect on other systems is undefined. When I ran my little test program to obtain the December date on Windows NT 4.0, the `time` routine returned (`null`).

Times and Files

The choice between going to a 32-bit unsigned value or moving to a 64-bit value is important because kernel times are central to the system's operation. They are not simply maintained by the kernel to be displayed by the `date` command. A time stamp is placed with every file on your file system. In fact, on UNIX each file has three stored times. You've got time values expressed as signed 32-bit integers sprinkled liberally all over your system.

The times that UNIX stores for each file are returned to a calling program by the `stat` system call and so the usual names are prefaced by `st`. The first time stamp, `st_atime`, contains the time of last access to the file, basically the time the file was last read. The second, `st_mtime`, is the time the file was last modified. Usually, this means the time the file was last written.

The final time, `st_ctime`, tells you the time the `inode` information for the file was last changed. Remember that an `inode` is the small block of information that the system stores on a disk to refer to each file. It's a common mistake to think that `ctime` means *Creation* time. The `st_ctime` is set when the file is created, but it's altered whenever the `inode` is updated. The `inode` is changed when we write to a file, because usually the size of the file changes. The `inode` is also updated when we change permission bits or ownership.

To investigate these times on a file or set of files, you can use the `ls` command. Here's a shell script that can be given a list of files and will print three lines for each file:

```
#!/bin/sh
for f in "$@"
do
    echo "atime: \c"; ls -dlu $f
    echo "mtime: \c"; ls -dl $f
    echo "ctime: \c"; ls -dlc $f
done
```

The script should work in all Bourne shell variants. I've put the guts of the solution in a loop so you can use it to look at a bunch of files easily. The `\c` at the end of each `echo` command inhibits the command from printing a newline at the end of its output. Some shells may prefer to use

```
echo -n "atime: "
```

to achieve the same effect.

Each `ls` command in the script is given the `d` option so that when it's applied with a directory argument, it deals with the directory itself and not its contents. I am sure you are familiar with the `l` (*ell*) option that makes `ls` print a long listing. The final letters in the options list tell `ls` to print the appropriate time for the nominated file. Of course, you could pass the output of the `ls` command into `sed` or `awk` to reduce what it prints to just the times in which you are interested, but I didn't want to get into that here.

Seeing the times on files may be illuminating, but the time stamps can also be used in the `find` command to tell you things about your file system. The `find` command searches a file tree applying a set of tests to each of the files it locates. Here's an example I've adapted from the `find` manual page:

```
find $HOME \( -name a.out -o -name '*.o' \) \
          -atime +7 -print
```

The command searches the tree starting at your home directory (`$HOME`) for files named `a.out`, or anything ending in `.o` that was accessed more than seven days ago. When it finds a matching file, it prints its name. Notice that the `find` command uses `+` before the number to mean more than the specified number of days. We have two other options: We can just use the number by itself to mean exactly seven days ago, or we can prefix the number with a `-` (*minus sign*), which means less than seven days ago. Of course, this example is aimed at programmers who will be creating `.o` and `a.out` files. You can replace the name selection by any set of temporary files that you use.

You can get `find` to execute a command whenever it finds a matching file. In the `find` manual page, the `-print` in the example above is replaced by

```
-exec rm {} \;
```

The `-exec` option is followed by a command that possibly has a set of options. Here we have one `{}` that is replaced by the name of the file that's been matched. The revised `find` command will delete any matched file. The backslash before the semicolon at the end of the line ensures that the semicolon is passed into the `find` command and is used to announce the end of the command you have typed.

This type of `find` statement is often used to clear out entire file systems, deleting common temporary files. I realize that I may be using the system when the clean-up command runs and will always insert something like

```
-atime +1
```

into the command so that it leaves today's files alone. This means that I am not suddenly surprised when a file disappears from under my feet.

As you might expect, the `find` command also has `-mtime` and `-ctime` predicates, both of which have their uses. For example, my mail system places my mail in a `Sent` folder after I have composed it and pressed the appropriate transmit button. Because I use `exmh`, each piece of mail is stored as a separate file and I run a script every night that eliminates any file older than three days. This keeps my `Sent` folder down to manageable proportions.

Size Choice

I started with some worry about what is going to happen to dates in about 38 years, and now it's time to come clean. I think this issue will be resolved, just like the millennium bug has been (I hope). It will be interesting to see what will happen. If we go to a 64-bit time stamp in `inodes`, then we will reduce the total amount of usable disk space that exists on each disk. We increase the amount of information that we need to pull off the disk to process a single file. Perhaps three years ago, loss of space might have been a worry, but disk technology has advanced in leaps and bounds. We may have disk space to spare for this. Worrying about pulling more data from a disk may also not be relevant for the same reason.

All operating systems now have the ability to deal with different file system formats, and many of these formats use different time stamp formulae. At the moment, the presence of different formats means systems have to translate the dates on file systems into something the local system understands. It would be pretty easy to create a system that has an internal 64-bit date format, but could deal with 32-bit dates in files. Of course, the old file systems would expire when we reach 2038 or 2106, because there just aren't enough bits.

What? Me Worry?

Worrying about these issues may seem somewhat academic. I am unconvinced that I will get to 2038, I certainly won't be around in 2106, and the human race as we know it may not be around at all in 292,471,210,647.

However, I think the writing is on the wall, and that writing is digital. I believe that in the next few years we're all going to start owning huge digital file systems. You may already, but at the moment most people don't. My personal set of electronic baggage is growing because I've acquired a digital camera. My son's disks are groaning with MP3s and he's managed to fill some significant portion of his system with music. We haven't started to see much digital video retained on rewritable media yet, but we will. I've spotted digital video cameras in electronics stores already.

We are all going to start collecting digital file systems that hold things that we really want to keep. Quite a lot of what I have on disk is of no interest to anyone else at all. I have a lot of source code of things that I am running, an immense amount of email, odd letters and general personally relevant information. I can live without this and so can everyone else.

However, official documents, sound recordings of family members, family photos and video are all going to be on digital media in a very short time, and these will be things that people will want to pass down the generations. To support this need, the computing world will need to think seriously, and soon, about generating and supporting an extensible file system that's vendor-independent and guaranteed to be around and usable over a considerable time period.

For example, I have an audio cassette somewhere that's a recording of my grandmother talking about her grandmother who was born in 1815. My grandmother's life spanned from having horses on the streets to seeing men on the moon. I'm not sure that changes in my lifetime will be so visibly dramatic. I recorded her originally on a reel-to-reel recorder and had the presence of mind to put it onto a C90 at some point. Of course, the copying process lost quality and the C90's magnetic tape has probably print through by now.

The story is relevant to this discussion. We need file systems that can be replicated from media to media without worrying about individual file contents so the data can be preserved for long periods.

The file system may be an archival system, not designed for speed of access, but designed to be portable. We've been learning about portability for a long time. A key change in the `tar` file format in the early days of UNIX altered the meta-information held about each file on the archive into text format. Originally, the information was binary, but consequently it wasn't portable across systems. The dates and sizes moved from binary integers to text to allow receiving systems to read and understand them.

However, I am not sure I want an archive-only file system for my persistent files. I suspect that we will want to write and read randomly accessible data that is organized hierarchically from our live systems in this portable format. The problem with archiving and dumping is that you have to take specific action to archive the file or files, and you have to work out how to retrieve those files. I like to keep all my "stuff" online so I can get to it at any time.

Networking has proved that having a ubiquitous transport format from your desktop to mine is a win. I suspect the same will be true of file systems. The worry is that such a system will be developed to support one vendor's products. We will end up with the mess that occurred with CD-ROM file system formats, where the people sitting on the standards body were more concerned with promoting their own needs and didn't seem to even consider other people's requirements.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.