UNIX Time

UNIX time data: seconds elapsed since

01.01.1970, 00:00:00 UTC.

Stored in a signed 32–bit integer.

UTC = universal time coordinated
GMT = Greenwich Mean Time
CET = Central Europe Time
CEST = Central Europe Summer Time

UNIX Time Overflow

One year has 31536000 seconds.

One leap year has 31622400 seconds.

Four years have 126230400 seconds.

2^{31} seconds are 2147483648 seconds (signed 32-bit).

$2147483648 = 17 \cdot 126230400 + 1566848$

$1566848 = 18 \cdot 24 \cdot 3600 + 11648$

$11648 = 3 \cdot 3600 + 14 \cdot 60 + 8$

So the overflow occurs after

$17 \cdot 4 = 68$ years, 18 days, 3 hours, 14 min and 8 sec

which is the

19.01.2038, 03:14:08 GMT.

UNIX Time Overflow (2)

Solutions to the overflow problem:

- use *unsigned* 32-bit integer,
  overflow occurs after $2 \cdot 68 = 136$ years in the February of 2106

  problem: programmers rely on signed integer,
  including positive and negative differences of `time_t` values
- use signed *64-bit* integer, overflow in the year 292.277.026.596
  (default on 64-bit operating systems)

open question: will there be still 32-bit systems in the year 2038?

- embedded CPUs?
- file systems?

Hardware Clock and Time Zones

booting $\rightsquigarrow$ read RTC (real time clock, 32Hz) $\rightsquigarrow$ system time

Problems when RTC has local time:

- computer travels in different time zones
- daylight savings time
- virtualization, two operating systems

$\rightsquigarrow$ RTC *should not* have local time (Windows 7/8 has)

Windows solution: create registry key

`SYSTEM\CurrentControlSet\Control\TimeZoneInformation\RealTimeIsUniversal`

UTC, universal time coordinated

Set your hardware system clock to

UTC.

basis of the worldwide system of civil time since 1972.

UTC replaced *Greenwich Mean Time* (GMT), because of ambiguity.

Before 1925, the GMT day started at noon.

UTC is the basis of the worldwide system of civil time. UTC is kept in time laboratories using atomic clocks. UTC is distributed via satellite/radio signal.

Set your local system time by using time zones.

UT1

Problem: a second is 1/86400 of a day (one rotation of earth)

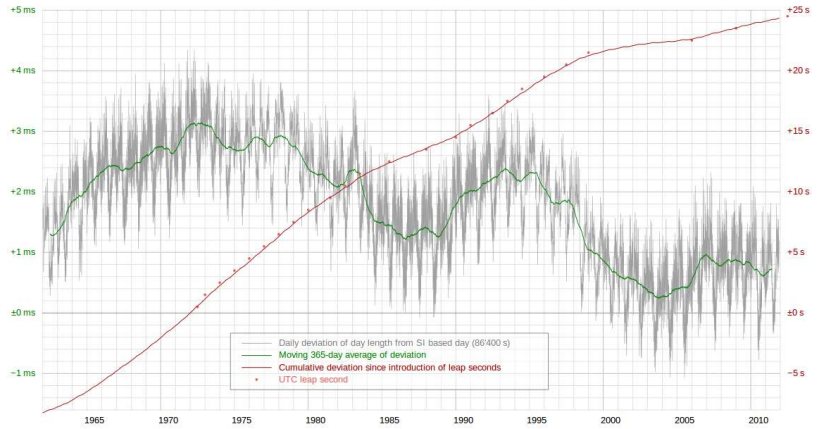
. . . but earth rotations don't have constant duration

- earth rotates slower and slower over time
- day length gets longer and longer over time
- by definition, the length of a second gets longer and longer over time

The time determined by the rotation of the Earth is called **UT1**.

Rotation time of earth needed for locations of satellites.

UTC/UT1 Deviation



UTC and UT1

UTC and UT1 should not deviate too much

$\rightsquigarrow$ correct UTC by 1 leap second every 12-18 months (27s since 1972)
6 month advance notice

latest corrections:

```
Jun 30, 1997, 23:59:60
Dec 31, 1998, 23:59:60
Dec 31, 2005, 23:59:60
Dec 31, 2008, 23:59:60
Jun 30, 2012, 23:59:60
Jun 30, 2015, 23:59:60
Dec 31, 2016, 23:59:60
```

NB: GPS time = UTC as of Jan 6, 1980 – no leap seconds

UTC – Decision to Retain Leap Seconds

Civil Global Positioning System Service Interface Committee 2007

mailed vote on stopping leap seconds
reported computer problems after June 30, 2012

decision, World Radio Conference in 2015
pro elimination: France, Italy, Japan, Mexico, USA
contra elimination: Canada, China, Germany, UK

see `https://en.wikipedia.org/wiki/Leap_second`

UTC

Cesium Clock CS2 of PTB Braunschweig, origin of DCF77 signal

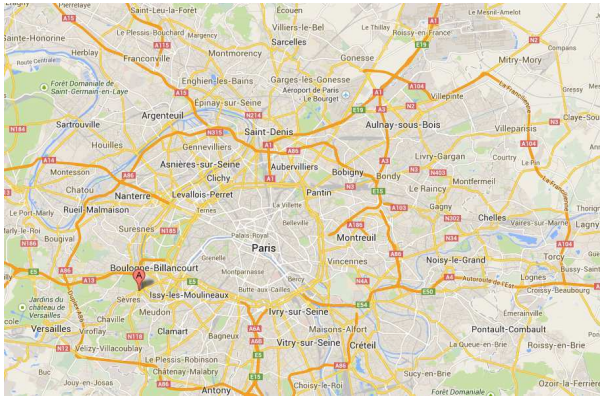
TAI (1)

Temps atomique international

weighted average of the time kept by over 200 atomic clocks

TAI (2)

International Bureau of Weights and Measures (BIPM, France)



Time Zones

Which time does the user see?

```
$ date
Sun May 11 20:53:35 CEST 2014
```

Where does the `CEST` come from?

Time zone information contained in `zoneinfo` files, for example

`/usr/share/zoneinfo/Europe/Berlin`

Each user may define his own time zone (New York):

```
$ TZ=EST date
Sun May 11 13:53:36 EST 2014
```


Time Zone Data

See

`/usr/share/zoneinfo/`

for example

```
$ zdump  /usr/share/zoneinfo/Europe/*
/usr/share/zoneinfo/Europe/Amsterdam    Tue May  7 08:51:40 2014 CEST
/usr/share/zoneinfo/Europe/Andorra      Tue May  7 08:51:40 2014 CEST
/usr/share/zoneinfo/Europe/Athens       Tue May  7 09:51:40 2014 EEST
/usr/share/zoneinfo/Europe/Belgrade     Tue May  7 08:51:40 2014 CEST
/usr/share/zoneinfo/Europe/Berlin       Tue May  7 08:51:40 2014 CEST
/usr/share/zoneinfo/Europe/Bratislava   Tue May  7 08:51:40 2014 CEST
/usr/share/zoneinfo/Europe/Brussels     Tue May  7 08:51:40 2014 CEST
/usr/share/zoneinfo/Europe/Bucharest    Tue May  7 09:51:40 2014 EEST
/usr/share/zoneinfo/Europe/Budapest     Tue May  7 08:51:40 2014 CEST
```


Time Zones

at system installation time, such a file is copied to

`/etc/localtime`

dump the contents

```
$ zdump -v /etc/localtime |grep 2014
Sun Mar 30 00:59:59 2014 UTC = Sun Mar 30 01:59:59 2014 CET isdst=0
Sun Mar 30 01:00:00 2014 UTC = Sun Mar 30 03:00:00 2014 CEST isdst=1
Sun Oct 26 00:59:59 2014 UTC = Sun Oct 26 02:59:59 2014 CEST isdst=1
Sun Oct 26 01:00:00 2014 UTC = Sun Oct 26 02:00:00 2014 CET isdst=0
```


Time Zones: Which one is installed?

Which one is it? A soft link would be better:

`/etc/localtime -> /usr/share/zoneinfo/Europe/Berlin`

If not, can find it by checksums:

```
sha1 /usr/share/zoneinfo/Europe/* | head -5
SHA1 (/usr/share/zoneinfo/Europe/Amsterdam)
= aee37bc42d7fb5061913609ce1155bc4a53d9000
SHA1 (/usr/share/zoneinfo/Europe/Andorra)
= 1ce238588cd3cbca3f9b620fe93fbff8a2f9d2bc
...
SHA1 (/usr/share/zoneinfo/Europe/Berlin)
= b065fae6bda0f0642ca6a52b665768e34a99d213
...

SHA1 (/etc/localtime)
= b065fae6bda0f0642ca6a52b665768e34a99d213
```

References (Time)

Why the RTC clock should keep UTC time

`http://www.cl.cam.ac.uk/~mgk25/mswish/ut-rtc.html`

On time zones, an astronomical view

`http://aa.usno.navy.mil/faq/docs/UT.html`

On the chrystal oscillator, used by the BIOS

`http://en.wikipedia.org/wiki/Crystal_oscillator`

NTPD – Network Time Protocol Daemon

Server which receives or distributes its system time

Protocol specification in RFC 958 (1985)

Port 123/udp

stratum 0 atomic clock

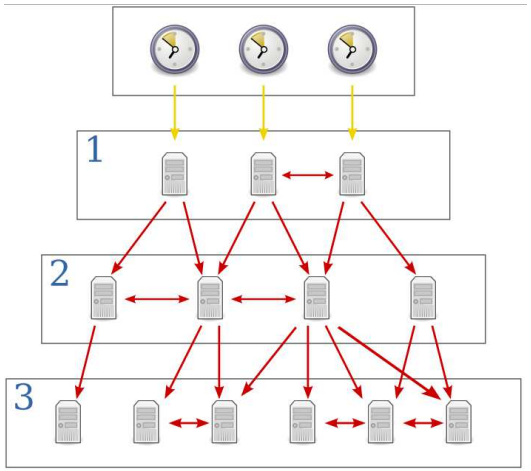
stratum 1 NTPD with signal from atomic clock
(for example `ptbtime1.ptb.de`)

stratum 2 NTPD with signal from stratum 1

stratum 3 NTPD with signal from stratum 2

⋮

NTP–Stratum Concept



NTP–Clients

ntpd

rdate

ntpdate

sntp

NTP–Clients Query Server

```
$ ntpdate -q ntp1.rz.uni-saarland.de ntp2.rz.uni-saarland.de
             ntp3.rz.uni-saarland.de

server 134.96.7.2, stratum 3, offset 0.074765, delay 0.02667

server 134.96.7.14, stratum 2, offset 0.056386, delay 0.02605

server 134.96.7.18, stratum 2, offset 0.059031, delay 0.02626

11 May 17:22:55 ntpdate[39524]: adjust time server 134.96.7.14
offset 0.056386 sec
```


The Shell

The Shell

The standard shell is the Bourne Shell `/bin/sh`.

The Bourne Shell is available on *every* UNIX system.

There are related shells of `sh`: `bash`, `ksh`, `ash`, `zsh`...

There are C–syntax based shells: `csh`, `tcsh`,...

The Shell creates processes and waits for them to terminate if the command is not followed by &.

FreeBSD's `sh` History

```
HISTORY
A sh command, the Thompson shell, appeared in Version 1 AT&T UNIX.  It
was superseded in Version 7 AT&T UNIX by the Bourne shell, which inher-
ited the name sh.

This version of sh was rewritten in 1989 under the BSD license after the
Bourne shell from AT&T System V.4 UNIX.
```

(from sh's manual page)

Overview of sh

- started after login (change shell with `chsh`)
- reads lines (from terminal/file)
- interactive/non-interactive
- programming language with control constructs

The Shell: Executing Commands

- tries to execute first command line argument
- built-in command (cd, echo, fg, bg,...) ?
- contains „/" ? Attempt to call it directly.
- otherwise do PATH search

The Shell: Parser

- reads whole *lines* (until `[newline]` = ASCII 10)
- *words* are separated by *meta* or *control characters*:
  `[Space] [Tab] | & ( ) ; < >`
- *control operators* perform control functions:
  `|| & && ; ;; ( ) | [newline]`
- meta or control characters lose their special meaning by quoting them
  - by \ (backslash affects next char)
  - by ' (single quote affects all chars till next ')
  - by " (single quote affects most chars till next ")

The Shell: Expansion

Some expressions are substituted by other strings.

The order of the substitutions is important.

1. brace expansion (`{}`)
2. tilde expansion (`~`)
3. variable/parameter expansion (`$`)
4. command substitution (`` `cmd` `` or `$(cmd)`)
5. arithmetic expansion (`$((expression))`)
6. word splitting (meta+control characters)
7. pathname expansion (`* ? []`)
8. quote removal (`"..."`, `'...'`,\)

The Shell: Expansion Examples


```
$ ls -l file{1,2}
-rw-r--r--    1 dw       users           0 Apr 12 13:11 file1
-rw-r--r--    1 dw       users           0 Apr 12 13:11 file2
$ echo ~root
/root
$ echo ~{sysi01,sysi07}
/home/sysi01 /home/sysi07

$ echo $USER
dweber
$ echo ~$USER
~dweber

$ echo $TERM
xterm
$ echo today is ‘date‘
today is Tue May  7 09:56:48 CEST 2013
```



```
$ echo $((3+9*5))
48
$ ls -l file?
-rw-r--r--    1 dw       users           0 Apr 12 13:11 file1
-rw-r--r--    1 dw       users           0 Apr 12 13:11 file2
-rw-r--r--    1 dw       users           0 Apr 12 13:11 file3
$ ls -l *2
-rw-r--r--    1 dw       users           0 Apr 12 13:11 file2
-rw-r--r--    1 dw       users           0 Apr 12 13:11 otherfile2
```


Shell Programming

- invoking arbitrary commands, programs, shell scripts
- using control structures
- setting/reading environment variables

all variables are *strings*

may occasionally be interpreted as an *integer*

Shell Programming: Variables

predefined are

- command line arguments in `$0,$1,...$9`
- number of command line arguments in `$#`
- all parameters in `$*` and `$@`
- return value of last command in `$?`
- own PID in `$$`

a user may set his own variables such as

`var=value`

(no spaces here!)

Shell Programming: Comments

use a *hash sign* = ,,#"

some German notations (from Wikipedia)

`Doppelkreuz Gartenzaun Gatter *Hash* Kanalgitter Knastfenster Lattenkreuz Lattenzaun Mengenkreuz Nummer Nummernzeichen Oktothorp Quadrat Raute Rhombus Schweinegatter Teppich Tic-Tac-Toe`

special case: `#!` in first line identifies *shell interpreter*

```
#!/bin/sh
```


Shell Programming: Control Structures

1. `for ... do ... done`
2. `for ... in ... do ... done`
3. `while ... do ... done`
4. `until ... do ... done`
5. `if ... then ... else ... fi`, see also `elif`
6. `case ... esac`

there is a `break` statement to leave loops

Shell Programming: Control Operators (AND)

`command1 && command2`

command2 is executed only if command1 returns *true*

example:

`mkdir /my/new/dir && cd /my/new/dir`

Shell Programming: Control Operators (OR)

```
command1 || command2
```

command2 is executed only if command1 returns *false*

example:

```
mkdir /my/new/dir || echo "could not create new directory"
```


Shell Programming: Conditions (if/while)

the test *command* is used for all conditions

see the manual page, mostly needed conditions are

`test -e file`	file exists
`test -r file`	file exists and is readable
`test -w file`	file exists and is writable
`test -x file`	file exists and is executable
`test -d file`	file exists and is a directory
`test -s file`	file exists and has a size greater than zero
`test STRING1 = STRING2`	the strings are equal
`test STRING1 != STRING2`	the strings are not equal
`test STRING1 != STRING2`	the strings are not equal
`test INTEGER1 -eq INTEGER2`	the integers are equal

for integers we analogously have `-ne -ge -gt -le -lt`

Shell Programming: Example `for` (1)

```
#!/bin/sh

# our for loop starts here
for x in 1 2 3 ; do
   cp $x.doc $x.txt
done
# our for loop ends here

exit 0 # return sucessfully
```


Shell Programming: Example `for` (2)

executing this gives nasty error messages:

```
$ chmod +x job
$ ./job
cp: cannot stat ‘1.doc’: No such file or directory
cp: cannot stat ‘2.doc’: No such file or directory
cp: cannot stat ‘3.doc’: No such file or directory
```

Shell Programming: Example `for` (3)

```
#!/bin/sh
# our for loop starts here
for x in 1 2 3 ; do
   if test -r $x.doc ; then # check the file
      cp $x.doc $x.txt
   fi
done
# our for loop ends here
exit 0 # return sucessfully
```

note: the `;` *terminates the condition*