Use strace to Understand Linux

Harald König
Harald.Koenig2@Bosch-Sensortec.com
koenig@linux.de

strace is one of my favourite tools in Linux. Simply use it – and by using it you will learn a lot about the internals of Linux. strace allows you to observe one or multiple running processes at the level of system (kernel) calls. strace may provide you with valuable information for many problem cases, for example: which config files really were read, which was the last file or shared library read before your programm crashed, and so on. In contrast to interactive debuggers like gdb the program of interest runs more or less in real time under strace, and you may already obtain a lot of information on kernel calls while your program is still executing, allowing you to follow the flow of the process "live", and/or save the strace output, in order to comfortably analyze it afterwards "offline".

Even for tracking performance problems you may obtain interesting information from strace: how often is a system call executed, how much time does a kernel call consume, how much compute time does the program itself use between kernel calls. With strace, one may also elegantly log the complete I/O of a program (disk or network) to later analyze it offline, or, if so desired, "replay" the whole I/O (even in "real time" thanks to the preceise time stamps strace produces).

This talk is intended to encourage you to investigate weird UNIX-effects, -problems or program crashes with strace and hopefully solve your problems more quickly, while gaining new insights into the inner working of the the program and Linux as you go.

1 Let's use strace

strace has many command line options and variants. Here, I will address and introduce you to only the most important use cases – you may consult the man page (RTFM: man strace) for much more information and help.

1.1 First Steps

```
$ strace emacs
```

or for an already running emacs process

```
$ strace -p $( pgrep emacs )
```

and if you must trace multiple processes simultaneously (e.g. all instances of Apache httpd):

```
$ strace $( pgrep httpd | sed 's/^/-p/' )
```

The "strace -p ..." which is attached to a process may be stopped at any time by typing CTRL-C; the tested program then continues to proceed normally and at full speed. If the program was *started* under strace, typing CTRL-C will not only abort strace, but also the launched program.

strace emits its entire output to stderr, but you may write the output to a file or redirect it, which then includes the stderr output of the process (here: emacs) itself:

```
$ strace -o OUTFILE emacs
$ strace emacs 2> OUTFILE
```

Usually strace prints one line of output per system call. This line contains the name of the kernel routine called, its parameters, and the return value of the call.

strace-output looks very C-like, which should not surprise, because the kernel API of Linux/UNIX is indeed defined as an ANSI-C interface (mostly didactic useful abridged edition):

```
$ strace cat /etc/HOSTNAME
execve("/bin/cat", ["cat", "/etc/HOSTNAME"], [/* 132 vars */]) = 0
open("/etc/HOSTNAME", O_RDONLY)         = 3
read(3, "harald.mydomain.de\n", 32768) = 19
write(1, "harald.mydomain.de\n", 19) = 19
read(3, "", 32768)                      = 0
close(3)                                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
```

Even with no knowledge of C-programming, you should be able to understand this output and interpret it reasonably well. You may read up on all the details of the individual calls in the corresponding the man pages, because all system calls are documented in section 2 of the Linux man pages (man execve ; man open 2 ; man 2 read write etc.). Only from the definition of each system call you will be able to determine whether the values of the arguments were *passed from the process* to the kernel, or *returned from the kernel* to the executing process (for example, the string values, passed as the second arguments of read() and write() above).

For some system calls, (e.g. stat() and execve()) strace with the -v option supplied, will produce more detailed "verbose" output.

If you are looking for more information (stat() details of files, or complete environment for execve()), the verbose-option -v often helps.

```
$ strace -e execve cat /etc/HOSTNAME
execve("/bin/cat", ["cat", "/etc/HOSTNAME"], [/* 132 vars */]) = 0
harald.mydomain.de
$
$ strace -v -e execve cat /etc/HOSTNAME
execve("/bin/cat", ["cat", "/etc/HOSTNAME"], ["LESSKEY=/etc/lesskey.bin", "MANPATH=/usr/
local/man:/usr/loca"..., "XDG_SESSION_ID=195", "TIME=\\t%E real,\\t%U user,\\t%S sy"...,
 "HOSTNAME=harald", "GNOME2_PATH=/usr/local:/opt/gnom"..., "XKEYSYMDB=/usr/X11R6/lib/X11
/XKe"..., "NX_CLIENT=/usr/local/nx/3.5.0/bi"..., "TERM=xterm", "HOST=harald", "SHELL=/bi
n/bash", "PROFILEREAD=true", "HISTSIZE=5000", "SSH_CLIENT=10.10.8.66 47849 22", "VSCMBOO
T=/usr/local/scheme/.sche"..., "CVSROOT=/soft/.CVS", "GS_LIB=/usr/local/lib/ghostscrip".
.., "TK_LIBRARY=/new/usr/lib/tk4.0", "MORE=-sl", "WINDOWID=14680077", "QTDIR=/usr/lib/qt
3", "JRE_HOME=/usr/lib64/jvm/jre", "USER=harald", "XTERM_SHELL=/bin/bash", "TIMEFORMAT=\
t%R %3lR real,\t%U use"..., "LD_LIBRARY_PATH=/usr/local/nx/3."..., "LS_COLORS=no=00:fi=0
0:di=01;34:l"..., "OPENWINHOME=/usr/openwin", "PILOTPORT=usb:", "XNLSPATH=/usr/share/X11
[ ... 22 lines deleted ...]
rap"..., "_=/usr/bin/strace", "OLDPWD=/usr/local/nx/3.5.0/lib/X"...]) = 0
```

A special mode of strace is turned on with the option -c. In this mode, not every single kernel call is shown separately, but statistics of the number of individual calls, as well as the CPU time consumed by each call will be reported. This mode may be useful to obtain an initial overview of CPU-time consumption in order to track down performance problems, however, strace reports only very few details about the program schedule and the output of the program flow is no longer "live". To combine both live trace and final statistics, use the option -C.

1.2 Restricting the Output of strace

The output of strace can quickly become very large and synchronous output to strerr or a log file may greatly impair program performance. If you know up front which system calls are of importance, you may limit the output to one or a few kernel calls (or e.g. with -e file system calls with file names). This slows down program execution considerably less and makes subsequent evaluation of the strace output much easier. However, the -e option offers many more possibilities. Here are just a few simple examples, which very often are sufficient, everything else is documented in the man page:

```
$ strace -e open            cat /etc/HOSTNAME
open("/etc/HOSTNAME", O_RDONLY)         = 3

$ strace -e open,read,write  cat /etc/HOSTNAME
open("/etc/HOSTNAME", O_RDONLY)         = 3
read(3, "harald.mydomain.de\n", 32768) = 19
write(1, "harald.mydomain.de\n", 19harald.mydomain.de
) = 19
read(3, "", 32768)                      = 0

$ strace -e open,read,write  cat /etc/HOSTNAME > /dev/null
open("/etc/HOSTNAME", O_RDONLY)         = 3
read(3, "harald.mydomain.de\n", 32768) = 19
write(1, "harald.mydomain.de\n", 19) = 19
read(3, "", 32768)                      = 0

$ strace -e file            cat /etc/HOSTNAME
execve("/bin/cat", ["cat", "/etc/HOSTNAME"], [/* 132 vars */]) = 0
open("/etc/HOSTNAME", O_RDONLY)         = 3
```

1.3 Tracing Several Processes and Child Processes

strace is able to trace multiple processes simultaneously (multiple -p PID options or -p PID1,PID2,...). strace is also able to track *all* child processes with option -f. In these cases, the PID of the process will be printed at the beginning of each line of output. Alternativly strace may write the output for each process to a separate file (option -ff).

If you do not exactly know what you are actually tracing or looking for, then it's a good idea to use either option -f or -ff, initially, as you do not know whether any subprocesses or scripts are involved. Just running strace without -f and -ff is not advisable as you are risking to miss vital information from other processes. In my simple examples below (with emacs) the program invoking emacs is a wrapper shell script. Here are a few variants as a little quiz to test and speculate on the inner workings of bash:

```
$ strace -e execve bash -c true
execve("/bin/bash", ["bash", "-c", "true"], [/* 132 vars */]) = 0

$ strace -e execve bash -c /bin/true
execve("/bin/bash", ["bash", "-c", "/bin/true"], [/* 132 vars */]) = 0
execve("/bin/true", ["/bin/true"], [/* 130 vars */]) = 0
$ strace -e execve bash -c "/bin/true ; /bin/false"
execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"], [/* 132 vars */]) = 0

$ strace -f -e execve bash -c "/bin/true ; /bin/false"
execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"], [/* 132 vars */]) = 0
execve("/bin/true", ["/bin/true"], [/* 130 vars */]) = 0
execve("/bin/false", ["/bin/false"], [/* 130 vars */) = 0

$ strace -o OUTFILE -f -e execve bash -c "/bin/true ; /bin/false" ; grep execve OUTFILE
21694 execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"], [/* 132 vars */]) = 0
21695 execve("/bin/true", ["/bin/true"], [/* 130 vars */]) = 0
21696 execve("/bin/false", ["/bin/false"], [/* 130 vars */]) = 0

$ strace -o OUTFILE -ff -e execve bash -c "/bin/true ; /bin/false" ; grep execve OUTFILE*
OUTFILE.22155:execve("/bin/bash", ["bash", "-c", "/bin/true ; /bin/false"], [/*...*/]) = 0
OUTFILE.22156:execve("/bin/true", ["/bin/true"], [/* 130 vars */]) = 0
OUTFILE.22157:execve("/bin/false", ["/bin/false"], [/* 130 vars */]) = 0
```

1.4 Time Stamps and Performance Considerations

If you must address performance issues, it is often helpful to complement each line of `strace` output for a kernel call with a time stamp. By adding the option `-t` once, twice or three times, one may select different time stamp formats: HH:MM:SS, HH:MM:SS.microseconds, or the "UNIX time" in seconds and microseconds since 1970 GMT/UTC respectively. The last time stamp format is suited best for performing calculations with the time-stamps later, while the other two formats usually are better readable for humans and easier compared with other time stamps from e.g. `syslog` and other log files, where very specific events must be analyzed.

The `-r` option of `strace` will output the time difference between the last kernel call and the current one in microseconds (which may also be calculated and complemented later with `-ttt`). All time stamps always indicate *the entry* of the kernel call, it is not possible to obtain the time at which the kernel call returns control to the program directly.

The option `-T` adds the time being spent in the kernel call at the end of the output line in angle brackets <>. The time spent in the kernel is the only datum you can use if you require the time at which the system call returned control to the running program, as up to the next kernel call the program may spend a long time in user mode, so that the time of subsequent kernel call entry is not always equivalent to when the last kernel call ended..

```
$ strace -t    -e open,read,write,close  cat /etc/HOSTNAME > /dev/null
22:22:25 open("/etc/HOSTNAME", O_RDONLY) = 3
22:22:25 read(3, "harald.mydomain.de\n", 32768) = 19
22:22:25 write(1, "harald.mydomain.de\n", 19) = 19
22:22:25 read(3, "", 32768)             = 0
22:22:25 close(3)                       = 0
```

```
$ strace -tt   -e open,read,write,close  cat /etc/HOSTNAME > /dev/null
22:22:46.314380 open("/etc/HOSTNAME", O_RDONLY) = 3
22:22:46.314566 read(3, "harald.mydomain.de\n", 32768) = 19
22:22:46.314698 write(1, "harald.mydomain.de\n", 19) = 19
22:22:46.314828 read(3, "", 32768)      = 0
22:22:46.314945 close(3)                = 0

$ strace -ttt  -e open,read,write,close  cat /etc/HOSTNAME > /dev/null
1325971387.860049 open("/etc/HOSTNAME", O_RDONLY) = 3
1325971387.860143 read(3, "harald.mydomain.de\n", 32768) = 19
1325971387.860186 write(1, "harald.mydomain.de\n", 19) = 19
1325971387.860249 read(3, "", 32768)    = 0
1325971387.860283 close(3)              = 0

$ strace -r    -e open,read,write,close  cat /etc/HOSTNAME > /dev/null
     0.000067 open("/etc/HOSTNAME", O_RDONLY) = 3
     0.000076 read(3, "harald.mydomain.de\n", 32768) = 19
     0.000042 write(1, "harald.mydomain.de\n", 19) = 19
     0.000039 read(3, "", 32768)        = 0
     0.000033 close(3)                  = 0

$ strace -ttT  -e open,read,write,close  cat /etc/HOSTNAME > /dev/null
22:24:09.339915 open("/etc/HOSTNAME", O_RDONLY) = 3 <0.000010>
22:24:09.340001 read(3, "harald.mydomain.de\n", 32768) = 19 <0.000012>
22:24:09.340052 write(1, "harald.mydomain.de\n"
```

Usually you'll first write the `strace` output to a file, followed by various steps of "offline" analysis. Below I present a little "live" example to answer the question "What is `acroread` doing in the background all the time?"

```
$ strace -p $( pgrep -f intellinux/bin/acroread ) -e gettimeofday -tt
22:48:19.122849 gettimeofday({1325972899, 122962}, {0, 0}) = 0
22:48:19.123038 gettimeofday({1325972899, 123055}, {0, 0}) = 0
22:48:19.140062 gettimeofday({1325972899, 140154}, {0, 0}) = 0
22:48:19.140347 gettimeofday({1325972899, 140482}, {0, 0}) = 0
22:48:19.142097 gettimeofday({1325972899, 142261}, {0, 0}) = 0
22:48:19.146188 gettimeofday({1325972899, 146335}, {0, 0}) = 0
22:48:19.159878 gettimeofday({1325972899, 160023}, {0, 0}) = 0
CTRL-C
$ strace -p $( pgrep -f intellinux/bin/acroread ) -e gettimeofday -t 2>&1 \
  | cut -c-8 | uniq -c
    136 22:48:26
    138 22:48:27
    140 22:48:28
CTRL-C
$ strace -p $( pgrep -f intellinux/bin/acroread ) -e gettimeofday -t 2>&1 \
  | cut -c-5 | uniq -c
   8309 22:50
   8311 22:51
CTRL-C
```

or as another example, some small statistics about where `acroread` spends its time during a 10 second interval:

```
$ strace -p $( pgrep -f intellinux/bin/acroread )  -c & sleep 10 ; kill %1
[1] 24495
Process 18559 attached - interrupt to quit
[ Process PID=18559 runs in 32 bit mode. ]
Process 18559 detached
System call usage summary for 32 bit mode:
$ % time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 79.13    0.010992           5      2114           poll
  7.88    0.001094           0      2939           clock_gettime
  7.00    0.000972           0      2124      1716 read
  3.07    0.000426           0      1388           gettimeofday
  2.93    0.000407           1       406           writev
  0.00    0.000000           0         1           restart_syscall
  0.00    0.000000           0         2           time
------ ----------- ----------- --------- --------- ----------------
100.00    0.013891                8974      1716 total
```

1.5 Why Breaks XY, and which Files are Relevant ?

When a program terminates with either a "segmentation violation" or a reasonable error message, you have already won half the battle: now you *"only"* must run the program with `strace`, and hope that the error is reproducible so that perhaps you may get an *"obvious"* hint at the end of the `strace` output on the last steps executed which caused the problem or crash. So perhaps the problem may be a required shared library which can't be loaded anymore (because was erased by accident). Here is an example of an `emacs` instance, which is no longer able to find its `libX11.so.6`:

```
25243 open("/usr/lib64/x86_64/libX11.so.6", O_RDONLY) = -1 ENOENT (No such file or directory)
25243 stat("/usr/lib64/x86_64", 0x7fff4f3568c0) = -1 ENOENT (No such file or directory)
25243 open("/usr/lib64/libX11.so.6", O_RDONLY) = -1 ENOENT (No such file or directory)
25243 stat("/usr/lib64", {st_mode=S_IFDIR|0755, st_size=212992, ...}) = 0
25243 writev(2, [{"/usr/bin/emacs", 14}, {": ", 2}, {"error while loading shared libra"...
25243 exit_group(127)                   = ?
```

As new issues are reported, you often hear the statement: "We haven't changed anything since the last successful run". In such cases, using `strace`, you may easily create a complete list of all opened files, sought files, or files not found and subsequently check their time stamps, in order to determine whether they are unchanged or were overwritten by the program in the meantime:

```
$ strace -o OUT -e open,execve -f emacs
$ grep \" OUT | cut -d\" -f2 | sort | uniq -c | sort -nr | less
     35 /opt/kde3/share/icons/hicolor/icon-theme.cache
      9 /home/harald/.icons/DMZ/index.theme
      6 /usr/share/emacs/site-lisp/anthy/leim-list.el
      5 /usr/share/emacs/23.3/etc/images/splash.xpm
      4 /usr/share/icons/hicolor/icon-theme.cache
$ ls -ltcd $( grep \" OUT | cut -d\" -f2 | sort -u )  | less
-r--r--r--  1 root   root        0 Jan  7 23:43 /proc/filesystems
-r--r--r--  1 root   root        0 Jan  7 23:43 /proc/meminfo
drwx------  2 harald users    28672 Jan  7 23:40 /home/harald/.emacs.d/auto-save-list/
drwxrwxrwt 48 root   root     12288 Jan  7 23:33 /tmp//
-rw-r--r--  1 root   root   3967384 Jan  7 14:29 /usr/local/share/icons/hicolor/icon-theme.cache
-rw-r--r--  1 root   root  94182076 Jan  7 11:34 /usr/share/icons/hicolor/icon-theme.cache
```

```
$ strace -o OUT -e open,execve -e signal=\!all  -f emacs /etc/HOSTNAME
$ cut -d\" -f2 OUT | sort -u > FILELIST

$ strace -o OUT -e open -f emacs /etc/HOSTNAME
$ grep open OUT | grep -v ENOENT | cut -d\" -f2 | sort -u > FILELIST
```

or a list of all failed file accesses (cf. difference between `-e open` and `-e file`!):

```
$ strace -o OUT -e file,execve -f emacs
$ grep ENOENT OUT | cut -d\" -f2 | sort -u
$ grep ' = -1 E'  OUT | cut -d\" -f2 | sort -u
```

By creating a complete `FILELIST`, you may easily check which config files were, for example, read and subsequently locate the file, in which you find some magic string from the program's output (macro, config setting, strange text in the output, etc.), in order to track down the problem. However, to simplify things, you should previously remove the entries `/dev/` or `/proc/` etc.:

```
$ strace -o OUT -e open,execve  -f emacs /etc/HOSTNAME
$ grep -v ENOENT OUT | grep \" | cut -d\" -f2 | sort -u > FILELIST
$ egrep -v '^/dev/|^/proc/' FILELIST | xargs grep "Welcome to GNU Emacs"
Binary file /usr/bin/emacs-gtk matches
```

1.6 Which Login Scripts Were Executed ?

…is also easy to check and verify with `strace`.

One may trust that the corresponding shell with e.g. option `-login` really behaves identically to a "real" login. With this assumption it is sufficient to create a file list and analyze it.

In the following example the file `FILELIST` is unsorted, because the *order* of the login scripts is of course interesting:

```
$ strace -o OUT -e open -f bash --login -i
exit
$ grep -v ENOENT OUT | grep \" | cut -d\" -f2 > FILELIST1
$ strace -o OUT -e file -f bash --login -i
exit
$ grep \" OUT | cut -d\" -f2 > FILELIST2

$ egrep "^/etc|$HOME" FILELIST1
/etc/profile
/etc/bash.bashrc
/etc/venus/env.sh
/etc/bash.local
/home/koenig/.bashrc
/etc/bash.bashrc
/etc/venus/env.sh
/etc/bash.local
/home/koenig/.bash_profile
/home/koenig/.profile.local
/home/koenig/.bash_history
/etc/inputrc
/home/koenig/.bash_history
```

If you are not so familiar with that shell, you should the trace the "source" of a login itself! You may even try this as a non-root user, e.g. with xterm -ls. Alternatively, you may trace the SSH daemon directly on port 22, as root or trace a getty process and all subsequent processes (see below on safety aspects of strace logs of sshd, etc.):

```
$ lsof -i :22
COMMAND  PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
sshd    2732 root    3u  IPv4  15551      0t0  TCP *:ssh (LISTEN)
$ strace -p 2732 -ffv -e file -o sshd.strace ...
```

1.7 See even more Data

For local files it's sufficient to obtain a file list and use grep to look for the source of a certain string. With network connections or similar it is unfortunately not that easy. Long strings in read(), write(), etc. are by default chopped off after 32 characters, so that you may not reasonably search for strings in strace output. In such cases it is helpful to increase the maximum length of the strings strace outputs. Often enough, the first 999 characters are sufficient, but sometimes you really need the complete records ("more is better"):

```
$ strace -s999    -o OUTFILE -f wget http://www.google.de/
```

(or even -s 999999) to search for a known string (e.g. HTTP). Thus we find the system calls recvfrom() and read() and may focus on the read() call, if you only want to examine the incoming data (corresponding to the transmitted data stream sendto()).

You may obtain a good overview of the network connection by executing

```
$ strace -o OUT -e connect,bind,recvfrom,sendto,read wget http://www.google.de/
```

In order to output all data received, you may e.g. extract the strings from the output values of the recvfrom() calls, output the nicely quoted strings with a shell command and convert all the quoted characters with echo -e:

```
$ strace -o OUT -s 999999 -e read wget http://www.google.de/
$ grep 'read(4, "' OUT | cut -d, -f2- | sed -e '1,/google.com/d' \
    -e s'/. [0-9]*) * = [0-9]*$//'  -e 's@^@/bin/echo -en @' | sh
```

Now only the length records of the HTTP protocol are a bit distracting, but the content of interest is fully visible and searchable.

2 Problems

But even a wonderful tool such as strace is not without shadows. Here are a few hints and warnings, the man page or google will be able to tell you the "rest of the story".

2.1 strace Interferes with the Process Flow

The use of strace is mostly transparent to the process being traced, so a "measure" with strace should usually not distort its result. However, one should always consider that the communication between strace and the process(es) being traced causes additional computational overhead and many context switches between the program and strace. Statements on program performance on systems with high load you should therefore be worded very carefully.

Now and then I have experienced some lockups of the processes, usually with IPC or real time signals. "In the past" (older Linux kernel, perhaps to 2.6.<smallnumber> and even older?!) these problems have occurred more frequently in a reproducible fashion, than with newer kernels such problem are only rarely observed (if at all?).

However, should an attached process hang, although strace was killed (strace itself sometimes only can be killed with kill -9), then it may be worth sending SIGCONT (kill -CONT ...) to the hanging processes, so that they may (perhaps) continue execution.

2.2 ptrace() is NOT Cascadable

The system call tracer used by strace, gdb and other debuggers, uses the (ptrace()) system call to control the debugged process and obtain information from and on it. Because only a single process is able to control another process using ptrace(), at any given time, this implies the restriction that processes being debugged with gdb/strace or another debugger. may not again be observed with another strace instance:

```
$ strace sleep 999 &
$ strace -p $( pgrep sleep )
attach: ptrace(PTRACE_ATTACH, ...): Operation not permitted
```

2.3 Tracing SUID Programs

SUID programs – for safety reasons – may not be debugged or (s)traced by ordinary users in Linux/UNIX in SUID mode, because this would require and hence allow the tracing user to execute and control code under a different UID and may be able e.g. to change data going into or coming from kernel calls. SUID programs are therefore executed without SUID, so that they run under the user's own UID (which in some cases may also be helpful). If strace is run as root, this is not an issue anymore: root may do everything anyway, so the execution as root under other UIDs is not an security problem.

But if you try to strace a single SUID command (eg. sudo) in a more complex script that may/must *not* run as root, you can achieve this with strace by changing the strace binary itself to SUID root.

But be careful: every program executed and traced with this SUID-enabled strace binary will *automatically* run with root privileges – a major security issue if other users have access to this SUID-strace! Therefore better put this "dangerous" version of strace as a copy into a directory inaccessible to other users.

2.4 strace Output Readable to All Users

The output of strace may contain and disclose quite "interesting" contents and therefore better should not be readable by other users! Thus, parts of files may be displayed in the output of read(), which are

not readable (e.g. your private SSH key while tracing the execution of `ssh ...`) or even the characters of your own password or similarly sensitive data when they are being typed..

The output of `strace` sessions from `root` of course has a much higher potential to disclose sensitive information unintentionally!!

2.5 Dead Locks

By straceing "yourself", you may create wonderful deadlocks, e.g. by tracing a part of the output chain for the output of `strace` itself: your own `ssh` session or the `xterm` in which the shell with `strace` is running, or even the X Server on which the `strace` output is gushing. Blessed is ye, who has another way to access the system to kill the `strace` process!

3 SEE ALSO

There are other trace tools on Linux besides `strace`, which may be helpful, as well:

`ltrace`: instead of "s"ystem calls, this tool traces the "l"ibrary calls of most common shared libraries You'll even get *much* more output than from `strace` (which conversely also slows down the execution considerably), but explicitly tracing individual libc calls such as e.g. `getenv()` or string functions such as `strlen()` or `strcpy()`, etc. may yield valuable information about the program execution. Many command line options of `ltrace` are same as for `strace`, but `ltrace` is *only* able to trace ELF binaries and hence is not able to trace shell scripts etc., which can make its use difficult in some cases.

`LTTng` may be the future for both kernel and user-mode tracing. Unfortunately, I do not have any practice with `LTTng` (thanks to `strace`?!).

4 Conclusion

Well have a lot of fun, success and new UNIX insights `strac`ing your problems!

References

[1] RTFM: `man strace ltrace ptrace`

[2] http://linux.die.net/man/1/strace man page of `strace`

[3] http://linux.die.net/man/2/ptrace man page of `ptrace()`

[4] http://linux.die.net/man/1/ltrace man page of `ltrace`

[5] http://lttng.org/ The LTTng Project