execsnoop For Linux: See Short-Lived Processes

28 Jul 2014

Every time I can't immediately explain CPU usage, I wonder if short-lived processes are to blame. On Linux systems I can debug this using atop, which uses process accounting to catch these fleeting processes. But I wish I had my execsnoop tool, which creates a live log of each process for later study.

I just ported it to Linux. Here's an example, where "man ls" is run in another window:

```
# ./execsnoop
Tracing exec()s. Ctrl-C to end.
   PID   PPID ARGS
 20139  20135 mawk -W interactive -v o=1 -v opt_name=0 -v name= [...]
 20140  20138 cat -v trace_pipe
 20171  16743 man ls
 20178  20171 preconv -e UTF-8
 20181  20171 pager -s
 20180  20171 nroff -mandoc -rLL=173n -rLT=173n -Tutf8
 20179  20171 tbl
 20184  20183 locale charmap
 20185  20180 groff -mtty-char -Tutf8 -mandoc -rLL=173n -rLT=173n
 20186  20185 troff -mtty-char -mandoc -rLL=173n -rLT=173n -Tutf8
 20187  20185 grotty
```

Great! The first two lines, showing mawk and cat, are from execsnoop initializing. The remaining show the arcane workings of the man command.

Here's catching part of a Linux build:

```
# ./execsnoop
Tracing exec()s. Ctrl-C to end.
   PID   PPID ARGS
 25753  25588 /bin/sh -c echo 'python' | grep ^/ -q && echo y
 25755  25753 grep ^/ -q
 25756  25588 sh -c command -v 'python' | awk 'NR==1 {t=$0} NR>1 {t=t "m822df3020w6a44id34bt574ct
 25758  25756 awk NR==1 {t=$0} NR>1 {t=t "m822df3020w6a44id34bt574ctac44eb9f4n" $0} END {printf t
 25759  25588 /bin/sh -c echo 'python' | grep ^/ -q && echo y
 25761  25759 grep ^/ -q
 25762  25588 sh -c command -v 'python' | awk 'NR==1 {t=$0} NR>1 {t=t "m822df3020w6a44id34bt574ct
 25764  25762 awk NR==1 {t=$0} NR>1 {t=t "m822df3020w6a44id34bt574ctac44eb9f4n" $0} END {printf t
 25765  25588 /bin/sh -c echo '/usr/bin/python-config' | grep ^/ -q && echo y
 25767  25765 grep ^/ -q
 25768  25588 sh -c test -f '/usr/bin/python-config' -a -x '/usr/bin/python-config' && echo y
 25769  25588 /bin/sh -c echo '/usr/bin/python-config' | grep ^/ -q && echo y
 25771  25769 grep ^/ -q
 25772  25588 sh -c test -f '/usr/bin/python-config' -a -x '/usr/bin/python-config' && echo y
 25773  25588 /bin/sh -c '/usr/bin/python-config' --ldflags 2>/dev/null
 25774  25773 /usr/bin/python-config --ldflags
 25775  25588 /bin/sh -c '/usr/bin/python-config' --cflags 2>/dev/null
 25776  25775 /usr/bin/python-config --cflags
 25777  25588 /bin/sh -c touch PERF-FEATURES; cat PERF-FEATURES
 25778  25777 touch PERF-FEATURES
 25779  25777 cat PERF-FEATURES
 25780  25588 printf ...%30s: [ \033[32mon\033[m  ] dwarf
 25781  25588 printf ...%30s: [ \033[31mOFF\033[m ] dwarf
 25782  25588 printf ...%30s: [ \033[32mon\033[m  ] glibc
 25783  25588 printf ...%30s: [ \033[31mOFF\033[m ] glibc
 25784  25588 printf ...%30s: [ \033[32mon\033[m  ] gtk2
[...]
```

Wow. Application startups can also run a surprising number of processes. execsnoop can help you identify areas for performance improvement: excessive sh/grep/sed/awk invocations, that can often be rewritten to use more advanced shell or awk features.

You can also use execsnoop to catch unexpected behavior. Running it with -t for timestamps:

```
# ./execsnoop -t
Tracing exec()s. Ctrl-C to end.
TIMEs                PID   PPID ARGS
[...]
3799932.757407    29403  29390 hostname
3799932.763324    29405  29404 cat /sys/class/net/eth0/address
3799932.768432    29407  29406 grep -l ^[[:space:]]*EC2SYNC=no\([[:space:]#]\|$\) /etc/sysconfig/
3799932.778556    29412  29411 curl -s -f http://169.254.169.254/latest/meta-data/network/interfa
3799932.796555    29415  29413 grep inet .* secondary eth0
3799932.798570    29416  29413 awk {print $2}
3799932.801021    29414  29413 /sbin/ip addr list dev eth0 secondary
3799932.802559    29417  29413 cut -d/ -f1
3800159.574170    29419  29418 /bin/sh -c /usr/lib64/sa/sa1 1 1
```

Why is my system running curl on that address? Hm.

Options

execsnoop options are summarized by the USAGE message (there's also a man page and examples file):

```
# ./execsnoop -h
USAGE: execsnoop [-hrt] [-a argc] [-d secs] [name]
                 -d seconds      # trace duration, and use buffers
                 -a argc         # max args to show (default 8)
                 -r              # include re-execs
                 -t              # include time (seconds)
                 -h              # this usage message
                 name            # process name to match (REs allowed)
  eg,
       execsnoop                 # watch exec()s live (unbuffered)
       execsnoop -d 1            # trace 1 sec (buffered)
       execsnoop grep            # trace process names containing grep
       execsnoop 'log$'          # filenames ending in "log"
```

execsnoop traces events as they happen, unless the -d option is used, which uses in-kernel buffering.

What, Why, and How

This is another ftrace-based hack for my perf-tools collection. These are designed to work with fewest dependencies (including no kernel debuginfo, if possible), and on older Linux kernel versions, particularly my Linux 3.2 systems. I expect to rewrite them when new tracing features are added to Linux in the future.

This turned out to be difficult, and a number of times it I thought it might be impossible in this environment.

My first attempt, execsnoop-proc, traced sched:sched_process_exec with process arguments from /proc/PID/cmdline. This worked on many systems, but not all, as I don't think /proc could always be read quickly enough for some processes. The sched:sched_process_exec tracepoint was missing on some systems as well.

The version I'm now using dynamically traces either stub_execve() or do_execve(), and walks the %si register as an array of strings using an unrolled loop. This is an enormous hack that I can hardly believe works, but it does work on all the systems I need it to.

Here's the essence of what I'm doing, using the perf(1) command (perf_events):

```
# perf probe --add 'do_execve +0(+0(%si)):string +0(+8(%si)):string +0(+16(%si)):string +0(+24(%s
# perf record --no-buffering -e probe:do_execve -a -o - | PAGER="cat -v" stdbuf -oL perf script -
  :10007 10007 [000] 557516.214765: probe:do_execve: (ffffffff811cccb0) arg1="ls" arg2="--color=a
  :10008 10008 [000] 557516.219168: probe:do_execve: (ffffffff811cccb0) arg1="sleep" arg2="1" arg
# perf probe --del do_execve
```

In that case, I only included three arguments, but you can see how the unrolled loop works. The first one-liner is for a specific kernel version and platform, and may not work for you for many reasons, without first adjusting it to match what you have.

If you want to analyze at this level, you might find my kprobe tool easier to work with. Compare the above to:

```
# ./kprobe 'p:do_execve +0(+0(%si)):string +0(+8(%si)):string +0(+16(%si)):string +0(+24(%si)):st
Tracing kprobe do_execve. Ctrl-C to end.
  kprobe-12484 [000] d... 7149163.889695: do_execve: (do_execve+0x0/0x20) arg1="cat" arg2="trace_
    bash-12499 [001] d... 7149164.593106: do_execve: (do_execve+0x0/0x20) arg1="ls" arg2="--color
    bash-12500 [001] d... 7149164.597399: do_execve: (do_execve+0x0/0x20) arg1="sleep" arg2="1" a
^C
Ending tracing...
```

kprobe(8) is nice. It automatically adds and removes kprobes, and has options for showing stacks (-s) and column headers (-H).

Were I to use SystemTap, I'd do something like this:

```
# stap -ve 'probe process.begin { printf("%6d %6d %s\n", pid(), ppid(), cmdline_str()); }'
```

(When I run this, for some reason it begins by listing all the current processes, followed by the new ones. I don't know if it's a feature or a bug.)

The future of Linux should support a tracer that does a one-liner like this. In the meantime, I can use this hacked version of execsnoop to solve performance issues on older systems.

Conclusion

Short-lived processes can cause performance problems, and aren't visible from interval-sampling tools like top(1). In this post I described a Linux port of my popular execsnoop tool, which I've used for many years to identify and study short-lived processes, and solve the performance problems they can cause.

This is also another proof of concept for older Linux kernels, like opensnoop and iosnoop, using the existing ftrace and kprobes tracing frameworks. There are other ways to do this: you might have kernel debuginfo and SystemTap available, or auditing enabled. This implementation is handy for my basic Linux 3.2 cloud instances, that don't have kernel debuginfo installed.

Warnings apply: execsnoop and these one-liners use dynamic tracing on Linux, which has had kernel panic bugs in the past, so know what you are doing, test first, and use at your own risk.

You can comment here, but I can't guarantee your comment will remain here forever: I might switch comment systems at some point (eg, if disqus add advertisements).

About this blog

USE Method
Off-CPU Analysis
Active Bench.
Flame Graphs
Heat Maps
Frequency Trails
Colony Graphs
perf Examples
eBPF Tools
DTrace Tools
DTraceToolkit
DtkshDemos
Guessing Game
Specials
Books
Other Sites