Linux /proc filesystem for MySQL DBAs

Sampling /proc content for troubleshooting

Valerii Kravchuk, Principal Support Engineer, MariaDB

vkravchuk@gmail.com

Who am I and What Do I Do?

Valerii (aka **Valeriy**) **Kravchuk**:

- MySQL Support Engineer in MySQL AB, Sun and Oracle, 2005-2012
- Principal Support Engineer in Percona, 2012-2016
- Principal Support Engineer in MariaDB Corporation since March 2016
- **http://mysqlentomologist.blogspot.com** - my blog about MariaDB and MySQL (including some **HowTo**s, not just bugs marketing)
- **https://www.facebook.com/valerii.kravchuk** - my Facebook page
- **http://bugs.mysql.com** - used to be my personal playground
- **@mysqlbugs** **#bugoftheday** - links to interesting MySQL bugs, few per week
- **MySQL Community Contributor of the Year 2019**
- I speak about MySQL and MariaDB in public. Some slides from previous talks are here and there…
- "I solve problems", "I drink and I know things"

Disclaimers

- Since September, 2012 I act as an Independent Consultant providing services to different companies
- All views, ideas, conclusions, statements and approaches in my presentations and blog posts are mine and may not be shared by any of my previous, current and future employees, customers and partners
- All examples are either based on public information or are truly fictional and has nothing to do with any real persons or companies. Any similarities are pure coincidence :)
- The information presented is true to the best of my knowledge

Sources of information on **mysqld** in production

- Extended slow query log
- **show [global] status**; **show engine innodb status**\G
- InnoDB-related tables and different plugins in the INFORMATION_SCHEMA
- **userstat** - per user, client, table or index
- **show profiles;**
- PERFORMANCE_SCHEMA
- OS-level tracing and profiling tools:
  - **/proc** filesystem and related utilities
  - **ftrace**
  - Profilers, simple like **pt-pmp** or real like **perf**
  - **eBPF** and related tools, **bpftrace**
- tcpdump analysis

What is this session about?

- It’s about troubleshooting tools and approaches based on **/proc** sampling (like **0x.tools** by **Tanel Poder** or ad hoc scripts), that allow to monitor individual thread level activity in MySQL server on Linux, like thread states, currently executing system calls and kernel wait locations…
- … and few other useful **/proc** files and features
  - Why not about Performance Schema?
  - Why not about **perf**?
  - Why not about eBPF, bcc tools and **bpftrace**?
- Performance impact of the off-CPU profiling, availability on production servers (still not running kernels 5.x.y)

Linux /proc filesystem basics

- The **proc** filesystem is a pseudo-filesystem which provides an interface to kernel data structures. It is commonly mounted at **/proc**:

```
openxs@ao756:~$ mount | grep '/proc'
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
openxs@ao756:~$ sudo ls -F /proc/30580
attr/           cpuset   limits      net/          projid_map stat
autogroup       cwd@      loginuid   ns/           root@       statm
auxv            environ map_files/ numa_maps   sched       status
cgroup          exe@      maps       oom_adj     schedstat  syscall
clear_refs      fd/       mem        oom_score   sessionid  task/
cmdline         fdinfo/   mountinfo oom_score_adj setgroups timers
comm            gid_map   mounts     pagemap     smaps       uid_map
coredump_filter io      mountstats personality stack       wchan
```

- See also:
  - **man 5 proc**
  - My blog post about **/proc** basics

How to identify threads of the mysqld process?

- MySQL server is a multi-threaded process:
  *"The MySQL Server (**mysqld**) executes as a single OS process, with multiple threads executing concurrent activities. MySQL does not have its own thread implementation, but relies on the thread implementation of the underlying OS."*
- In MySQL 5.7+ use **performance_schema.threads**:

```
mysql> select thread_id, thread_os_id, name from
performance_schema.threads where type = 'BACKGROUND';
+-----------+--------------+----------------------------------------+
| thread_id | thread_os_id | name                                   |
+-----------+--------------+----------------------------------------+
|       1   |        30580 | thread/sql/main                        |
|       2   |        30581 | thread/sql/thread_timer_notifier       |
|       3   |        30582 | thread/innodb/io_ibuf_thread
...
|       13 |        30592 | thread/innodb/page_cleaner_thread      |
|       14 |        30593 | thread/innodb/buf_lru_manager_thread   |
...
```

- See my blog post for more details

Poor man’s threads monitoring with shell scripts

- Consider this simple loop to check something for every thread of MySQL:

```
openxs@ao756:~$ for dir in `ls /proc/$(pidof mysqld)/task`
> do
>   echo -n $dir': '
>   2>/dev/null sudo strings /proc/$dir/wchan
> done
...
2393: 2394: futex_wait_queue_me
2488: futex_wait_queue_me
30580: poll_schedule_timeout
30581: do_sigtimedwait
...
30591: read_events
30592: futex_wait_queue_me
30593: hrtimer_nanosleep
...
31000: jbd2_log_wait_commit
...
```

- See my blog post for more details

Something more advanced? **0x.tools**!

- **0x.tools**. - a useful set of programs to access, summarize and record **/proc** details created and recently shared by famous **Tanel Poder**
- Get them from GitHub:

```
git clone https://github.com/tanelpoder/0xtools
cd 0xtools/
make
sudo make install
```

- You need **python** (**2**!), **gcc** and **make**
- **xcapture** - low-overhead thread state sampler based on reading **/proc** files. Can run continuously and save captured data into **.csv** files
- **psn** - shows current top thread activity by sampling **/proc** files
- **schedlat** - shows CPU scheduling latency for the given PID as a % of runtime
- **run_xcapture.sh** - a simple “daemon” script for keeping **xcapture** running
- **run_xcpu.sh** - low-frequency continuous stack sampling for threads on CPU (using **perf**)
- Check my blog post for more details.

0x.tools: process snapper (**psn**) in action

- Let’s check what we can get from **psn** while **sysbench** load is running:

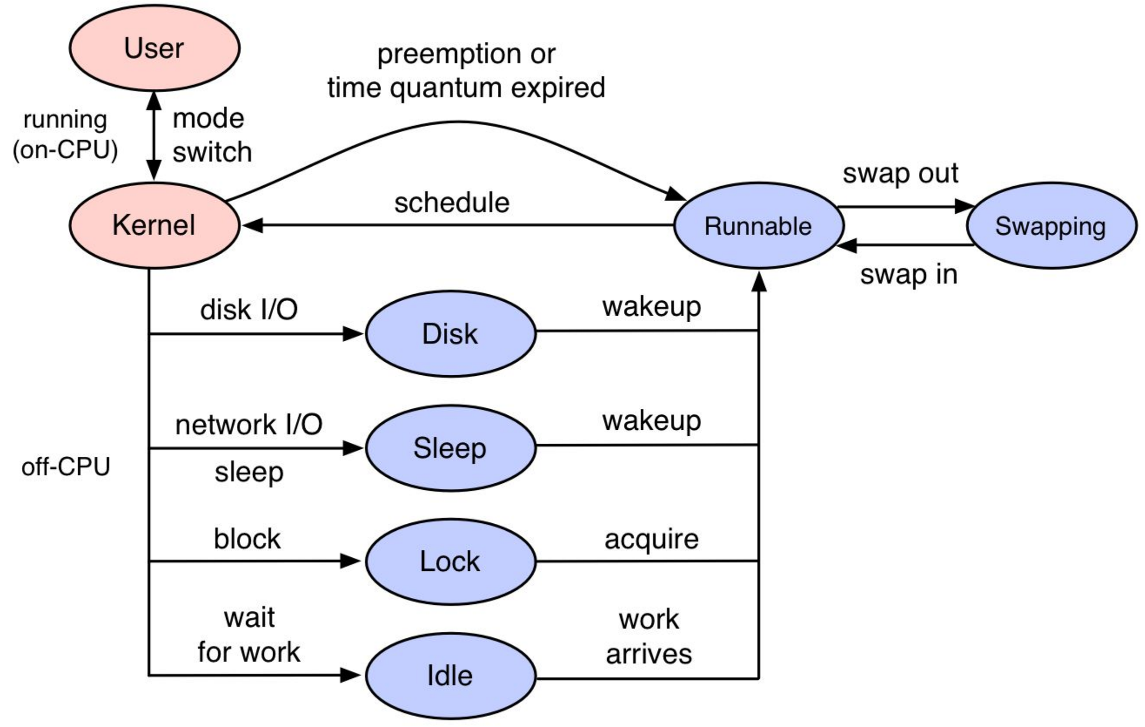
```
openxs@ao756:~/git/0xtools$  sudo psn -p `pidof mysqld` -G kstack

Linux Process Snapper v0.18 by Tanel Poder [https://0x.tools]
Sampling /proc/stat, stack for 5 seconds... finished.
...
 samples | avg_threads | comm     | state                  | kstack
...
    101 |        1.01 | (mysqld) | Disk (Uninterruptible) |
entry_SYSCALL_64_fastpath()->SyS_fsync()->do_fsync()->vfs_fsync_range()->ext4_
sync_file()->jbd2_complete_transaction()->jbd2_log_wait_commit()
...
     15 |        0.15 | (mysqld) | Running (ON CPU)       |
int_ret_from_sys_call()->syscall_return_slowpath()->exit_to_usermode_loop()
     11 |        0.11 | (mysqld) | Disk (Uninterruptible) |
entry_SYSCALL_64_fastpath()->SyS_fsync()->do_fsync()->vfs_fsync_range()->ext4_
sync_file()->blkdev_issue_flush()->submit_bio_wait()
...
```

- Check my blog post for more details.

Generic thread states on Linux

- You may be wondering why we care about kernel stacks, wait channels etc, how this may help?
- I'll use a chart of generic thread states presented by **Brendan Gregg** here:



Classification of performance issues

- As **Brendan Gregg** pointed out, performance issues can be categorized into one of two types:
  - **On-CPU**: where threads are spending time running on CPU
  - **Off-CPU**: where time is spent waiting while blocked on I/O, locks, timers, paging/swapping, etc.
- Different approaches to the analysis are used, depending on the type:
  a. **CPU Sampling** - checking stack traces on all CPUs at a certain rate and summarizing to understand where CPU cycles are mostly spent. See **perf -F99** ...
  b. **Application Tracing** - application functions are instrumented to collect timestamps when they begin and end, so that the time spent in functions can be calculated.. This is what Performance Schema provides for MySQL server.
  c. **Off-CPU Tracing** - only the kernel functions that switch the thread off-CPU are traced, along with timestamps and user-land stack traces. This is possible (but hard) with **perf** tracing and is usually done with eBPF-based **bcc** tools and **bpftrace** that can summarize data in kernel space on the fly.
  d. **Off-CPU Sampling** - sampling to capture blocked stack traces from threads that are not running on-CPU. This is what we can do by checking **/proc** content once in a while, with ad hoc scripts or **0x.tools**: **xcapture** or **psn**.

Brendan Gregg on Off-CPU sampling

- As **Brendan Gregg** pointed out, before eBPF made the overhead of Off-CPU tracing acceptable for some cases:

    “At one point I wrote a simple wall-time kernel-stack profiler called proc-profiler.pl, which sampled **/proc/PID/stack** for a given PID. It worked well enough. I'm not the first to hack up such a wall-time profiler either, see poormansprofiler and Tanel Poder's quick'n'dirty troubleshooting.”

- This is exactly what I am suggesting to do here, and what **xcapture** and **psn** tools allow to do easily, per thread.

Performance impact of /proc sampling vs bcc tools vs perf for off-CPU analysis

- Consider MySQL 8.0.22 started with **--no-defaults** running **sysbench** (I/O bound) test on Q8300 @ 2.50GHz Fedora 31 box:

```
[openxs@fc31 ~]$ sysbench oltp_read_write --db-driver=mysql --tables=5
--table-size=100000 --mysql-user=root ... --threads=32 --time=80
--report-interval=10 run
```

- I’ve executed it without tracing or sampling, and with the following (compatible?) off-CPU data collections working for 60 seconds:

```
[openxs@fc31 ~]$ sudo psn -d 60 --sample-hz 99 -p `pidof mysqld` -G kstack
-o /tmp/psnstacks
[openxs@fc31 ~]$ sudo xcapture -c kstack -o /tmp
[openxs@fc31 ~]$ sudo /usr/share/bcc/tools/offcputime -K -f 60 -p `pidof
mysqld` > /tmp/stacks.txt
WARNING: 5 stack traces lost and could not be displayed.
[openxs@fc31 tmp]$ sudo perf record -g -F 1 -a -- sleep 60
[ perf record: Captured and wrote 1.254 MB perf.data (83 samples) ]
```

- QPS: 1720 | 1676 (**97%**) | 1727 (**100%**) | 1614 (**93%**) | 1490 (**86%**)
- More realistic benchmarks and detailed blog post are yet to come

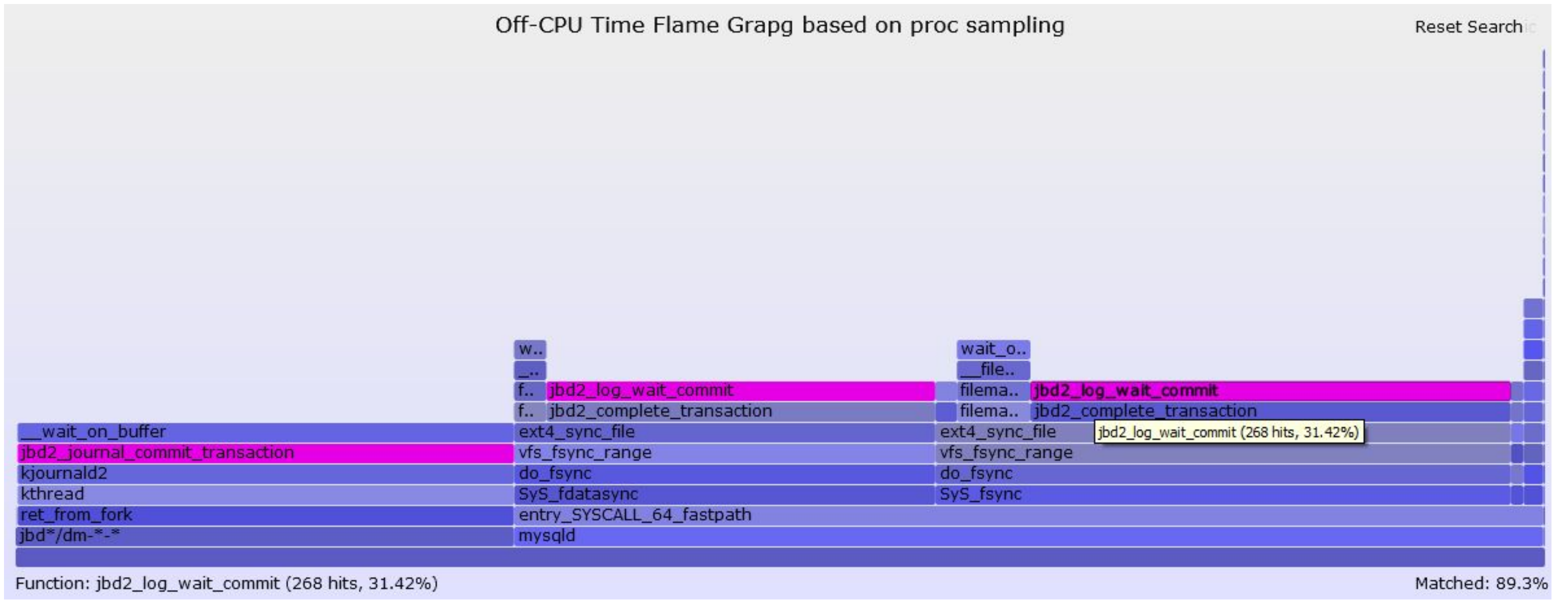
What we can get from stacks: Flame Graphs

- **http://www.brendangregg.com/flamegraphs.html**
- *Flame graphs* are a visualization (as **.svg** file to be checked in browser) of profiled software, allowing the most frequent code-paths to be identified quickly and accurately.
- The x-axis shows the stack profile population, sorted *alphabetically* (it is not the passage of time), and the y-axis shows stack depth. Each rectangle represents a stack frame. The wider a frame is, the more often it was present in the stacks.
- **Off-CPU Flame Graphs** ← tracing file I/O, block I/O or scheduler
- **https://github.com/brendangregg/FlameGraph** + **perf** + ... or **bcc** tools like **offcputime.py**
- Can we we try to build off-CPU flame graph from samples of kernel stacks?

Flame Graph based on off-CPU sampling

- Created based on these steps (while **oltp_read_write.lua** was running):

```
openxs@ao756:~$ sudo psn -d 60 -G kstack | grep -v Running | awk -F\| '{
print $3, $5, $1 }' |  sed 's/->/;/g' | grep '^.(' | sed 's/(//g' | sed
's/)//g' | awk '{ print $1";"$2, $3 }' > /tmp/psnstacks.txt
openxs@ao756:~$ git/FlameGraph/flamegraph.pl --color=io --title="Off-CPU
Time Flame Grapg based on proc sampling" --countname=hits <
/tmp/psnstacks.txt > ~/Documents/psn1.svg
```



Problems of /proc sampling:

- **root**/**sudo** access is required for many interesting files
- Storing and processing of the information collected:
  - Data can be presented as **.csv** files and loaded into the database
  - Kernel stacks can be processed for presentation as off-CPU flame graphs
  - Other options?
- Writing ad hoc shell scripts (**0x.tools** is one of answers)
- How to get both on-CPU and off-CPU sampling in a unified low impact manner? Any options other than **bcc** tools or **bpftrace**?

Q & A

- Thank you!
- Please, report bugs to **https://bugs.mysql.com**!