iosnoop For Linux

16 Jul 2014

I'm probably dreaming. I just ported my popular iosnoop tool to Linux:

```
# ./iosnoop
Tracing block I/O. Ctrl-C to end.
COMM             PID    TYPE DEV      BLOCK        BYTES     LATms
supervise        1689   W    202,1    17040376     4096       0.49
supervise        1684   W    202,1    17039744     4096       0.59
supervise        1687   W    202,1    17039752     4096       0.62
supervise        1684   W    202,1    17039768     4096       0.59
java             17705  R    202,1    5849560      4096      16.48
supervise        3055   W    202,1    12862528     4096       5.56
java             17705  R    202,1    5882792      4096       6.31
java             17705  R    202,1    5882984      40960     17.79
[...]
```

This shows a one line summary per block device I/O (disk I/O). It's like tcpdump for your disks.

Here's a random I/O workload, on a system with rotational disks:

```
COMM             PID    TYPE DEV      BLOCK        BYTES     LATms
randread         6199   R    202,16   71136208     8192       5.32
randread         6199   R    202,16   83134400     8192       9.26
randread         6199   R    202,16   88477928     8192       3.46
randread         6199   R    202,16   66953696     8192      10.69
randread         6199   R    202,16   87832704     8192       3.68
randread         6199   R    202,16   74963120     8192       4.62
[...]
```

Notice the random disk location (BLOCK), which results in latencies (I/O time) between 3 and 10 milliseconds.

Running the same workload on SSDs:

```
COMM             PID    TYPE DEV      BLOCK        BYTES     LATms
randread         20125  R    202,16   24803976     8192       0.15
randread         20125  R    202,32   17527272     8192       0.15
randread         20125  R    202,16   13382360     8192       0.15
randread         20125  R    202,32   29727160     8192       0.19
randread         20125  R    202,32   26965272     8192       0.18
randread         20125  R    202,32   27222376     8192       0.17
[...]
```

Excellent, latencies are much better!

In this post I'll summarize iosnoop(8) options and usage, provide an example of debugging a latency outlier, and discuss caveats, tool internals, overhead, and why I wrote this.

Options

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

```
# ./iosnoop -h
USAGE: iosnoop [-hQst] [-d device] [-i iotype] [-p PID] [-n name]
               [duration]
                 -d device       # device string (eg, "202,1)
                 -i iotype       # match type (eg, '*R*' for all reads)
                 -n name         # process name to match on I/O issue
                 -p PID          # PID to match on I/O issue
                 -Q              # use queue insert as start time
                 -s              # include start time of I/O (s)
                 -t              # include completion time of I/O (s)
                 -h              # this usage message
                 duration        # duration seconds, and use buffers
  eg,
       iosnoop                   # watch block I/O live (unbuffered)
       iosnoop 1                 # trace 1 sec (buffered)
       iosnoop -Q                # include queueing time in LATms
       iosnoop -ts               # include start and end timestamps
       iosnoop -i '*R*'          # trace reads
       iosnoop -p 91             # show I/O issued when PID 91 is on-CPU
       iosnoop -Qp 91            # show I/O queued by PID 91, queue time

See the man page and example file for more info.
```

I use the -s and -t options when post-processing the output in R or gnuplot, or for digging into harder issues, including latency outliers.

Latency Outlier

My randread program on an SSD server (which is an AWS EC2 instance) usually experiences about 0.15 ms I/O latency, but there are some outliers as high as 20 milliseconds. Here's an excerpt:

```
# ./iosnoop -ts > out
# more out
Tracing block I/O. Ctrl-C to end.
STARTs          ENDs            COMM         PID    TYPE DEV      BLOCK      BYTES   LATms
6037559.121523 6037559.121685 randread     22341  R    202,32   29295416   8192     0.16
6037559.121719 6037559.121874 randread     22341  R    202,16   27515304   8192     0.16
[...]
6037595.999508 6037596.000051 supervise    1692   W    202,1    12862968   4096     0.54
6037595.999513 6037596.000144 supervise    1687   W    202,1    17040160   4096     0.63
6037595.999634 6037596.000309 supervise    1693   W    202,1    17040168   4096     0.68
6037595.999937 6037596.000440 supervise    1693   W    202,1    17040176   4096     0.50
6037596.000579 6037596.001192 supervise    1689   W    202,1    17040184   4096     0.61
6037596.000826 6037596.001360 supervise    1689   W    202,1    17040192   4096     0.53
6037595.998302 6037596.018133 randread     22341  R    202,32   954168     8192    20.03
6037595.998303 6037596.018150 randread     22341  R    202,32   954200     8192    20.05
6037596.018182 6037596.018347 randread     22341  R    202,32   18836600   8192     0.16
[...]
```

It's important to sort on the I/O completion time (ENDs). In this case it's already in the correct order.

So my 20 ms reads happened after a large group of supervise writes were completed (I truncated dozens of supervise write lines to keep this example short). Other latency outliers in this output file showed the same sequence: slow reads after a batch of writes.

Note the I/O request timestamp (STARTs), which shows that these 20 ms reads were issued *before* the supervise writes – so they had been sitting on a queue. I've debugged this type of issue many times before, but this one is different: those writes were to a different device (202,1), so I would have assumed they would be on different queues, and wouldn't interfere with each other. Somewhere in this system (Xen guest) it looks like there is a shared queue. (Having *just* discovered this using iosnoop, I can't yet tell you which queue, but I'd hope that after identifying it there would be a way to tune its queueing behavior, so that we can eliminate or reduce the severity of these outliers. UPDATE: See comments.)

While the output contains COMM and PID columns, as explained in the man page, you should treat these (and the -p and -n options) as best-effort.

COMMs and PIDs

Like my original iosnoop tool, the COMM and PID columns show who is on-CPU when the I/O was issued, which is usually the task who issued it, but it could be unrelated. I had driven a system under high write load,

and started seeing entries like these:

```
COMM             PID    TYPE DEV      BLOCK        BYTES     LATms
awk              6179   W    202,16   597439928    4096      73.86
sshd             2357   W    202,16   597448128    4096      73.94
awk              6179   W    202,16   597456328    4096      73.94
awk              6179   W    202,16   597464528    4096      74.08
awk              6179   W    202,16   597472728    4096      74.30
awk              6179   W    202,16   597480928    4096      74.91
sshd             2357   W    202,16   597489128    4096      74.19
awk              6179   W    202,16   597497328    4096      74.94
[...]
```

Well, while awk and sshd were on-CPU, but aren't likely to be the origin of the I/O. This can be misleading.

I captured stack traces when block I/O was issued (using perf_events) to find you an example. Most of the stacks showed the correct process, but here's one showing awk instead:

```
awk 11774 [000] 3495819.879203: block:block_rq_issue: 202,16 W 0 () 19337224 + 256 [awk]
        ffffffff8133c980 blk_peek_request ([kernel.kallsyms])
        ffffffff8149d114 do_blkif_request ([kernel.kallsyms])
        ffffffff81336377 __blk_run_queue ([kernel.kallsyms])
        ffffffff813364ad blk_start_queue ([kernel.kallsyms])
        ffffffff8149d9f5 kick_pending_request_queues ([kernel.kallsyms])
        ffffffff8149dd86 blkif_interrupt ([kernel.kallsyms])
        ffffffff810c258d handle_irq_event_percpu ([kernel.kallsyms])
        ffffffff810c2788 handle_irq_event ([kernel.kallsyms])
        ffffffff810c50a7 handle_edge_irq ([kernel.kallsyms])
        ffffffff81421764 evtchn_2l_handle_events ([kernel.kallsyms])
        ffffffff8141f210 __xen_evtchn_do_upcall ([kernel.kallsyms])
        ffffffff81420e1f xen_evtchn_do_upcall ([kernel.kallsyms])
        ffffffff81709f3e xen_do_hypervisor_callback ([kernel.kallsyms])
            7f8e91ff2b1a __brk (/lib/x86_64-linux-gnu/libc-2.15.so)
```

awk is doing a brk() syscall, likely normal behavior as it expands its heap. Since this is an AWS EC2 Xen guest, it performs a hypercall (xen_do_hypervisor_callback) to optimize memory operations. This happens to call kick_pending_request_queues(), to issue more work that is already queued, which calls _blk_run_queue(), leading to issuing block I/O that was likely queued by another process.

I can fetch the originating PID by tracing higher up the stack, although it may be best to keep iosnoop as a simple block device-level tracer. I could also dig out the file system pathname (if any) that this I/O is related to. My earlier iosnoop versions did this, but, (like PID and COMM) it wasn't reliable, and depended on the file system type. iosnoop with ZFS can't really get filenames at the moment, but with HFS+ on OS X it does get partial pathnames, which can be very useful.

UPDATE: I've added a -Q option to iosnoop, which uses block queue insertion as the starting tracepoint instead of block I/O issue. This provides times that include block I/O queueing, to analyze problems of queueing and load. But it also improves the acccuracy of the PID and COMM columns, since the originating process is much more likely to be still on-CPU at the time of queue insert. I'm tempted to trace both insert and issue tracepoints at the same time so I can do it all at once – better PID and COMM, and show both I/O queued time and device time separately – but this would cost a lot more overhead. (I'll do that when the Linux kernel has an enhanced tracing framework, and I can do this all more cheaply.)

Why iosnoop

Why not use blktrace for disk I/O? I first wrote iosnoop before blktrace existed. Why port it now? Well...

I showed the source to a colleague. He was stunned.

> How does this work? This is just a shell script?!

Not only that, but it works on the large number of servers I use that are running the 3.2 kernel, without any software additions. I'm not using SystemTap, ktap, dtrace4linux, or even perf_events.

I'm using the bash shell, awk, and some files under /sys. Those files control the kernel's **ftrace** framework.

This was an experiment that began with perf_events, but I hit some problems with its scripting mode, since my use case is a little different to what's it's been designed for. perf_events provides the "perf" command, and in some ways is the nicer and safer front-end to the same frameworks (tracepoints and kprobes) that ftrace uses. iosnoop should at least be a perf_events script (after it can handle my use case), if not a script for a more advanced framework (if/when they are integrated).

So that leaves the present. I realized I could whip up an ftrace-based iosnoop, and use it to solve real issues until the nicer future arrived. I was also curious to see what problems I'd run into with the ftrace interface. It worked really well. (I had some issues with snapshots losing events, so I didn't use them, and also write errors with marker events, so I didn't use them either. I'm still debugging, and I don't know if I was just using them wrong, of if there are some issues with their implementation.)

My previous post used a lower-overhead mode of ftrace for function counting, which does the counts in-kernel. iosnoop(8) reads individual event data from ftrace and processes it in user space, so I was particularly interested in the overhead of this approach.

Overhead

This iosnoop version will print events as they happen. This produces snappy output, but for high IOPS systems it can consume noticeable CPU resources. The duration mode uses buffering instead, which populates the buffer once, then reads it. For example, using a duration of 5 seconds:

```
# time ./iosnoop -ts 5 > out

real    0m5.566s
user    0m0.336s
sys     0m0.140s
# wc out
  27010  243072 2619744 out
```

iosnoop ate half a second of CPU (all at once) to process the buffer of 27k events. That's really not bad. (I did analyze and tune awk performance for this script, and reduced CPU overhead by 3.2x.) The downside of using a buffer like this is that it has a limited size, and if you overflow it you'll lose events. With the default settings (bufsize_kb), that will happen at around 50k I/O.

There's a lot of different ways to handle this if you need to. The easiest way is drop the duration, so that events are printed without buffering. It does cost much more CPU; here's 30 seconds:

```
# time ./iosnoop -ts > out
^C
real    0m33.992s
user    0m4.704s
sys     0m7.524s
# wc out
  169687  1527164 16459401 out
# ls -lh out
-rwxrwsr-x 1 root other 16M Jul 16 05:58 out
```

It captured 170k events, and wrote a 16 Mbyte output file, but did cost over 12 seconds of CPU time to do so. About 36% of one CPU, during the run. This also means we can calculate the upper bound for the unbuffered approach (on this hardware and software versions): which would be about 15k IOPS.

But if you really want to trace more than 10k IOPS, for long durations, you're beyond the scope of this temporary tool. In the future, I imagine ditching this iosnoop implementation and rewriting it, at which point it will handle higher loads.

Conclusion

iosnoop(8) is a useful tool, and can be implemented using nothing more than Linux ftrace, which has been part of the kernel since the 2.6.x days. It should be implemented in something more robust, like perf_events or an advanced tracer (SystemTap, ktap), which can handle buffering and concurrent users more sensibly. This ftrace-

based iosnoop was really an experiment to see how far I could go. I surprised myself – just dropping a shell script on different kernels and having it work has been amazing. I hope I'm not dreaming.

Warnings apply: this ftrace tool is a hack, so know what you are doing, test first, and use at your own risk. With future kernel developments it should, one day, be rewritten, and move from a hack to a stable tool.

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).