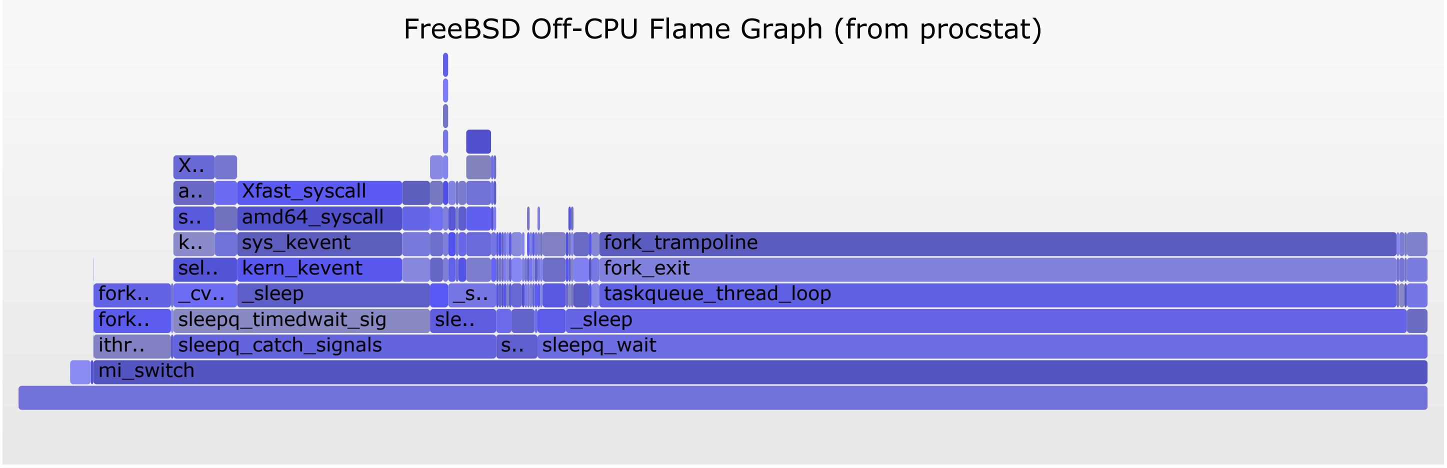
FreeBSD Off-CPU Flame Graphs

12 Mar 2015

While I was giving a talk on FreeBSD flame graphs, a new technique was suggested by the audience for gathering off-CPU stacks, which is something I'd like to see improved and included in all operating systems. It involved using `procstat -ka` to gather sleeping thread stacks, from which I created an off-CPU flame graph:



This visualization lets us browse the sleeping thread call stacks to look for anything unusual that may be a problem. Most of these will be normal, as threads usually spend most of their time sleeping while waiting for work. Some may not be normal, and can be the source of latency and performance issues.

Background

CPU profiling can help solve all performance issues when threads are on-CPU, but what about issues when threads are blocked and sleeping off-CPU? Time may be spent waiting for storage I/O, network I/O, lock contention, run queue latency, etc. Collectively, I call this *off-CPU time*, which can be profiled and examined to help solve these issues.

I recently showed how Linux perf_events can measure off-CPU time and create flame graphs. This approach captures all kernel scheduler events involved in sleeping and switching threads, to a trace file, and then uses user-level programs to stitch together events, calculating off-CPU time by call stack. While it works, tracing all scheduler events to a file can incur crazy-high overhead, and I'd be very nervous about using this on a busy production system.

An improvement is to calculate off-CPU time in kernel context, and to pass only aggregated call stacks with total times to user-level. This needs a programmable tracer, like SystemTap on Linux, or DTrace on FreeBSD. Slide 48 of my talk showed the DTrace commands, but this still comes with a high overhead warning, even though it's much better than the perf_events approach.

Off-CPU tracing costs overhead that is relative to the event rate. A system doing over a million IOPS can have over a million scheduler events per second. Compare this to CPU profiling, where we wake up at a fixed rate (eg, 99 Hertz) on all CPUs and gather a stack sample. On an 8 CPU system, CPU profiling can mean about 800 stack samples per second, no matter how busy the server is. Not the millions per second that off-CPU tracing can reach.

Can we profile off-CPU time in the same way as CPU profiling? Instead of tracing every scheduler event, can we wake up at a timed interval and gather all the *sleeping* stacks, not the on-CPU ones? This would mean the overhead was relative to the thread count instead of the scheduler event rate. Eg, to sample these at, say, 9 Hertz, on a system with 500 threads, would mean taking about 4,500 stack samples per second.

FreeBSD procstat -ka

During my talk, Adrian Chadd suggested I use procstat -ka to dump sleeping stacks (thanks!):

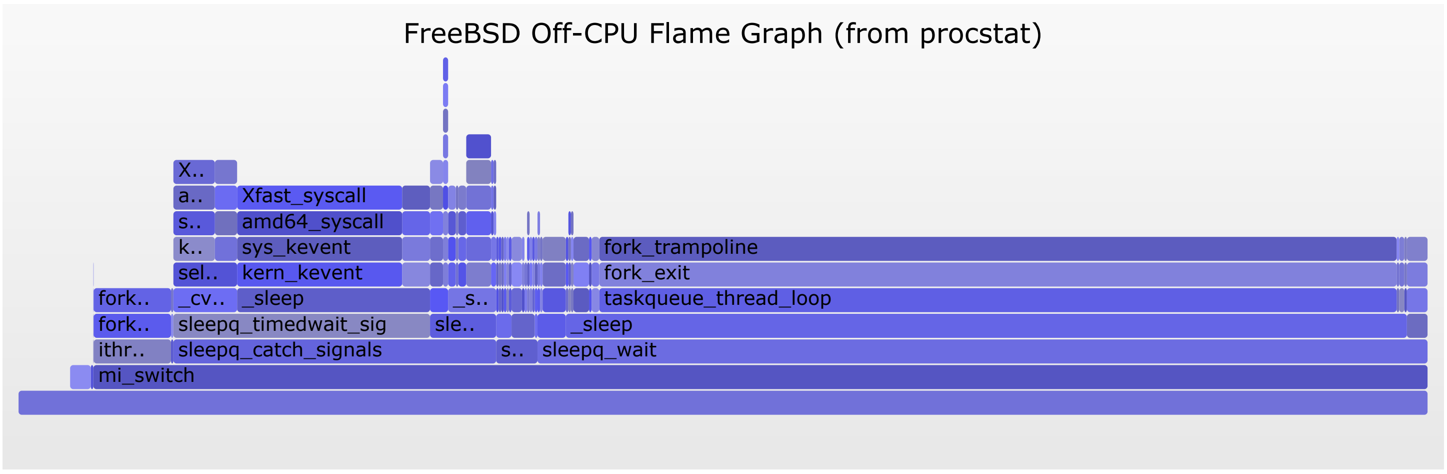
```
# procstat -ka
  PID    TID COMM             TDNAME           KSTACK
    0 100000 kernel           swapper          mi_switch sleepq_timedwait _sleep swapper btext
    0 100023 kernel           firmware taskq   mi_switch sleepq_wait _sleep taskqueue_thread_loop
    0 100025 kernel           kqueue taskq     mi_switch sleepq_wait _sleep taskqueue_thread_loop
    0 100026 kernel           ffs_trim taskq   mi_switch sleepq_wait _sleep taskqueue_thread_loop
    0 100027 kernel           acpi_task_0      mi_switch sleepq_wait msleep_spin_sbt taskqueue_th
    0 100028 kernel           acpi_task_1      mi_switch sleepq_wait msleep_spin_sbt taskqueue_th
    0 100029 kernel           acpi_task_2      mi_switch sleepq_wait msleep_spin_sbt taskqueue_th
    0 100033 kernel           aiod_bio taskq   mi_switch sleepq_wait _sleep taskqueue_thread_loop
    0 100035 kernel           thread taskq     mi_switch sleepq_wait _sleep taskqueue_thread_loop
    0 100037 kernel           mps0 taskq       mi_switch sleepq_wait _sleep taskqueue_thread_loop
    0 100049 kernel           mps1 taskq       mi_switch sleepq_wait _sleep taskqueue_thread_loop
[...]
```

Wow, so FreeBSD *already has this*. I'd really like to do this on Linux, too (gdb macro, at least?).

The KSTACK column shows a leaf to origin call stack. It just needs a little text processing for flamegraph.pl. For example, collecting ten of these and generating an off-CPU flame graph (in Bourne shell):

```
# git clone --depth 1 https://github.com/brendangregg/FlameGraph
# cd FlameGraph
# for i in `seq 1 10`; do procstat -ka >> out.offstacks01; sleep 1; done
# sed 's/^.\{47\}//;s/ /;/g;s/;$/ 1/' out.offstacks01 | \
    ./flamegraph.pl --color=io --title="Off-CPU" > offstacks01.svg
```

That generates the off-CPU flame graph I demonstrated in the talk, and is also at the top of this page (SVG):



Click to zoom.

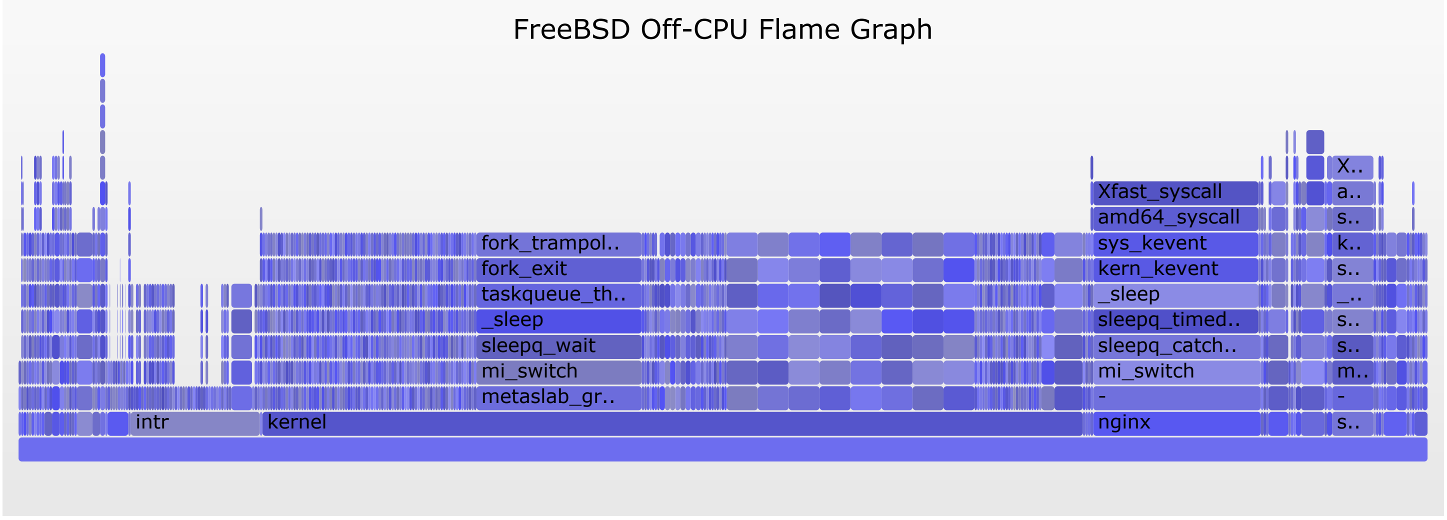
By Thread

Since procstat has the process thread names as the third and fourth fields, we can include them too. This gets a bit tricker as the thread name can contain spaces. Switching to awk:

```
# awk '{ comm = $3; thread = substr($0, 30, 16); gsub(/ *$/, "", thread);
    stack = substr($0, 48); gsub(/ /, ";", stack); gsub(/;$/, "", stack);
    printf "%s;%s;%s 1\n", comm, thread, stack }' out.offstacks01 | \
    ./flamegraph.pl --color=io --title="Off-CPU" > offstacks01byt.svg
```

(This awk is getting a bit long, and I should put it in a stackcollapse-procstat.awk file, like I have for other profiler output.)

Example output, now including process and thread names (SVG):



This has more detail, as we're now splitting time for each thread name, creating many thin towers. The wider towers mean pools of threads. There are 64 threads in ZFS metaslab_group_t (truncated), and 65 in nginx.

The per-thread name detail might be overdoing it in many cases. You can adjust the awk to only include the process name if desired.

Improving procstat

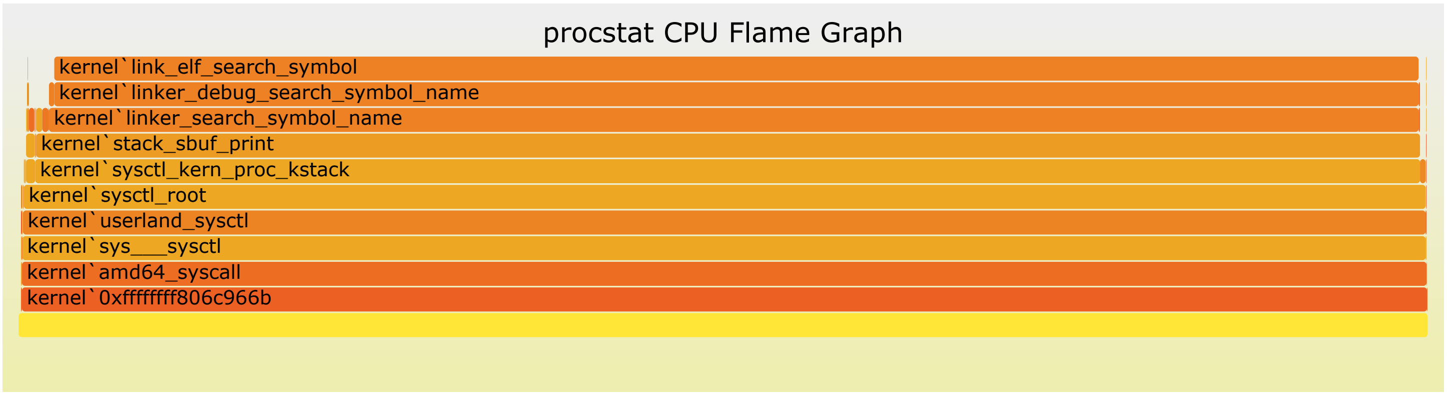
There is room for improvement with procstat. First, its stacks are kernel only. It would be helpful for it to include user-level frames, to fully explain blocked events.

Second, the overhead is higher that I would have hoped:

```
# procstat -ka | wc -l
     546
# time procstat -ka > /dev/null
        1.27 real         0.00 user         1.27 sys
```

That's 1.27 seconds of system CPU time to walk 546 threads. We won't be able to take even one sample of all threads per second, let alone taking it to 9 Hertz (like I suggested earlier) or higher.

Here's a CPU flame graph of procstat's system CPU time:



Ok, so we're in link_elf_search_symbol() (that's so obvious I didn't need to visualize this as a flame graph).

From sys/kern/link_elf.c:

```
static int
link_elf_search_symbol(linker_file_t lf, caddr_t value,
    c_linker_sym_t *sym, long *diffp)
{
[...]
        for (i = 0, es = ef->ddbsymtab; i < ef->ddbsymcnt; i++, es++) {
                if (es->st_name == 0)
                        continue;
                st_value = es->st_value + (uintptr_t) (void *) ef->address;
                if (off >= st_value) {
                        if (off - st_value < diff) {
                                diff = off - st_value;
                                best = es;
                                if (diff == 0)
                                        break;
                        } else if (off - st_value == diff) {
                                best = es;
                        }
                }
        }
```

This is not only a sequential search of a list for a symbol, but it also checks the entire list unless a direct match is found (offset 0), which usually won't be the case. Usually, it'll return the closest match. This is also called by another loop in linker_debug_search_symbol() (not visible in the flame graph, probably due to compiler optimizations), which iterates over all linker symbol files. In this case, it's searching all the kernel modules, the largest of which is zfs, which has 4,532 symbols.

So for every frame in every stack trace, we usually compare the program counter to every known kernel symbol, sequentially. This can be improved: eg, using a btree lookup, or at least bailing out early when a decent match is found. Another approach could be to build an off-CPU profiler in the kernel, which sampled and then aggregated stacks before doing symbol lookup. Many approaches are possible. I'd suspect there's never been a reason to optimize this further until now.

Conclusion

FreeBSD has an interesting way to list blocked thread stacks using `procstat -ka`, which can be used to generate off-CPU flame graphs. In this post I described how it worked, and some room for improvement with this approach. It would be great to see similar facilities in other operating systems, so that off-CPU flame graphs can be generated in a cheaper fashion, for solving issues of blocked time.

For more about flame graphs, see my previous post about my FreeBSD flame graphs talk.

Comments for this thread are now closed ×

0 Comments **Brendan Gregg's Blog** 1 **Login**

♡ **Recommend** 2 Tweet Share **Sort by Best**

This discussion has been closed.

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Disqus' Privacy PolicyPrivacy PolicyPrivacy**

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).

Copyright 2017 Brendan Gregg.
About this blog

Homepage
Blog
Full Site Map
Sys Perf book
Linux Perf
Perf Methods
USE Method
TSA 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