Tom Once Again, Does Stupid Things with a Computer: activitymonitor.sh

BY TOM JONES

At EuroBSDCon in Vienna this year, I spoke about my work examining the performance of QUIC on FreeBSD and Linux

One core part of my measurement approach was CPU saturation during a network transfer that is using all the available CPU cycles for a sender. I used CPU saturation two ways, the first was to detect if the CPU was the bottleneck in the system. If it wasn't the bottleneck, then I needed to be sure that the network was the bottleneck, and not some other component I wasn't trying to measure. And second, once I had controlled for CPU saturation and made sure another bottleneck wasn't at play, I could use the actual CPU usage for a test to estimate how fast the system could send if it could saturate the network card with UDP. This was really helpful, as with all my measurements, my network interfaces can only manage ~6Gbit/s with UDP traffic in any form on any of the tested operating systems. But this doesn't saturate the CPU, instead it runs at about 70% utilization. With good CPU measurements, I could invent a metric to optimistically predict what the processor could do if the network interface wasn't getting in the way.

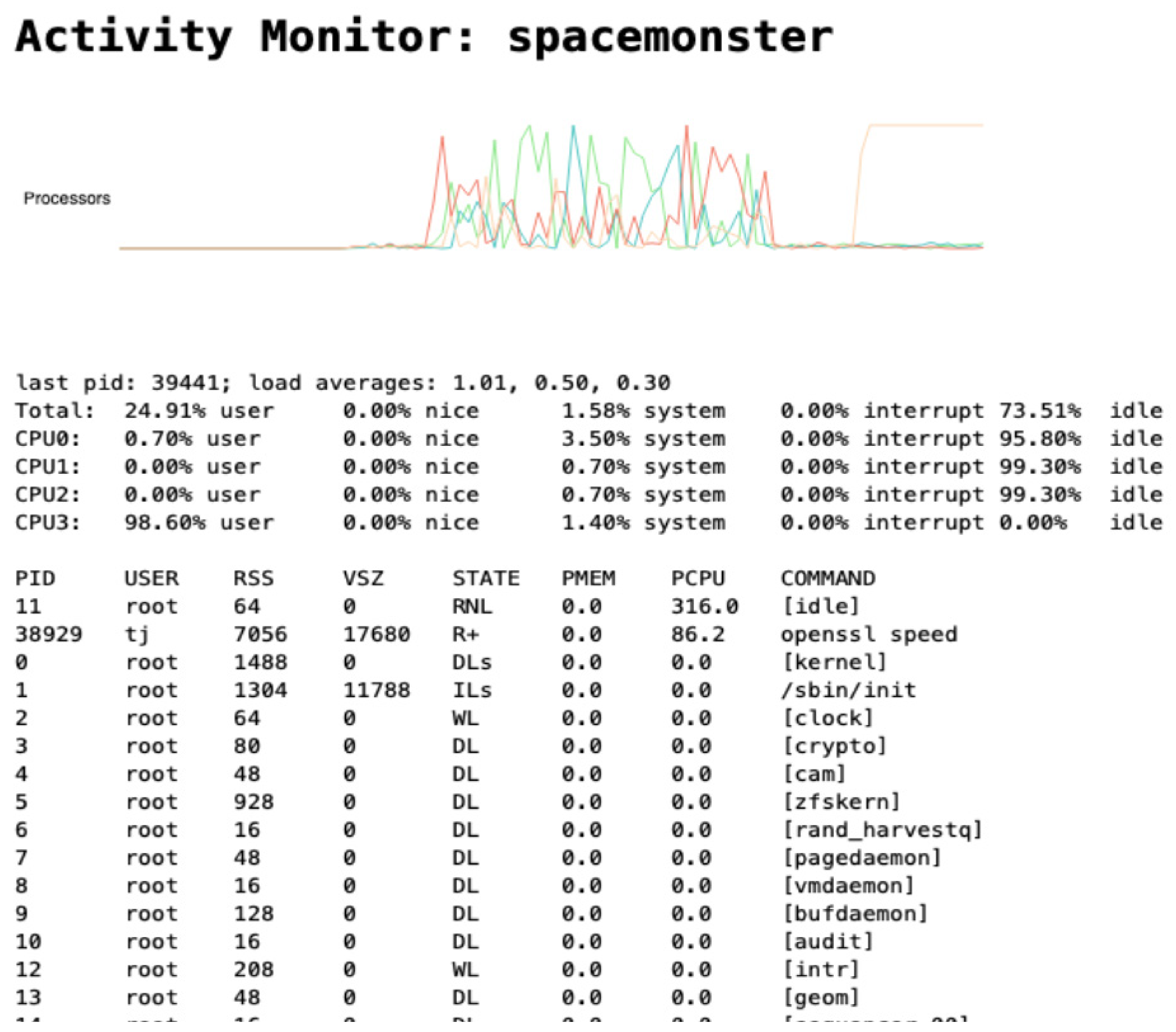
It seemed a good idea to find a well-reasoned approach to looking at CPU performance during a network test.

My EuroBSDCon presentation was based on yet unpublished academic work, and it seemed a good idea to find a well-reasoned approach to looking at CPU performance during a network test. Some of my tests were inspired by work that Fastly did to compare the computational efficiency of QUIC compared to TCP. While Fastly seems to have established a great testbed, the only mechanism I could figure out from their writing was to "eyeball" top.

This wasn't really good enough for publishing results or for accurate evaluation of thousands of test results. If looking at top wasn't good enough, maybe I could, instead, figure out how top does its own looking and reimplement that?

How Does top Estimate CPU Utilization?

top is probably the first tool we go to when we wonder what is happening on a system. Even with its universal usefulness, top is a very simple program that actually draws things nicely on the screen and it has some abstractions on the machine to make the code a little more portable. The CPU utilization interface is provided by FreeBSD in the form of two sysctl nodes:



```
Activity Monitor: spacemonster
```

```
last pid: 39441; load averages: 1.01, 0.50, 0.30
Total:  24.91% user     0.00% nice      1.58% system     0.00% interrupt 73.51%  idle
CPU0:   0.70% user      0.00% nice      3.50% system     0.00% interrupt 95.80%  idle
CPU1:   0.00% user      0.00% nice      0.70% system     0.00% interrupt 99.30%  idle
CPU2:   0.00% user      0.00% nice      0.70% system     0.00% interrupt 99.30%  idle
CPU3:   98.60% user     0.00% nice      1.40% system     0.00% interrupt 0.00%   idle

PID     USER    RSS     VSZ     STATE   PMEM    PCPU    COMMAND
11      root    64      0       RNL     0.0     316.0   [idle]
38929   tj      7056    17680   R+      0.0     86.2    openssl speed
0       root    1488    0       DLs     0.0     0.0     [kernel]
1       root    1304    11788   ILs     0.0     0.0     /sbin/init
2       root    64      0       WL      0.0     0.0     [clock]
3       root    80      0       DL      0.0     0.0     [crypto]
4       root    48      0       DL      0.0     0.0     [cam]
5       root    928     0       DL      0.0     0.0     [zfskern]
6       root    16      0       DL      0.0     0.0     [rand_harvestq]
7       root    48      0       DL      0.0     0.0     [pagedaemon]
8       root    16      0       DL      0.0     0.0     [vmdaemon]
9       root    128     0       DL      0.0     0.0     [bufdaemon]
10      root    16      0       DL      0.0     0.0     [audit]
12      root    208     0       WL      0.0     0.0     [intr]
13      root    48      0       DL      0.0     0.0     [geom]
```

```
$ sysctl hw.ncpu
4
$ sysctl kern.cp_time
kern.cp_time: 3832370 11408 3650627 44926 2061043745
$ sysctl kern.cp_times
kern.cp_times: 1080959 3035 954019 4354 515101995 1062132 2823 815176 1143
515263088 960320 3419 980090 37760 515162773 728956 2131 901334 1668 515510273
```

The first node `kern.cp_times`, returns 5 values for the processor which report the total time since boot for time spent in user, nice, system, interrupt and idle. `kern.cp_times` reports the same 5 values of the each of the processors in the system. By using `kern.cp_times` with `hw.ncpu`, we can break down this list. By working with both sysctls we get the total system time usage since boot and the per-processor time usage since boot.

Usage since boot can be helpful in understanding what the machine has been up to and it is useful to see how the system usage is changing over short periods of time. top on startup displays the total system time breakdown, but once it refreshes (by default after 1 second), it then shows how these fields have changed over that second.

Plotting

With the ability to very easily reproduce what top does, I wondered if I could grab the system CPU utilization periodically and plot it out. I figured I could include these plots next to throughput plots to show that between tests, the host was almost entirely idle, and it saturated the expected single core when the test was running.

Over the years, I have had different go-to tools when it comes to plotting, but with this work, I got tired of custom things and wanted simplicity. I could have plotted out the utilizations with the easiest method—a spreadsheet, but I thought something that gave me a little more control would be nice.

In other recent work, I have used web-based tooling for plots. The c3js library gives nice interactive charts but struggles when there are a lot of data points (more than the low 10s of thousands). Given that I was also going to look at network usage, the amount of data spread out over a minute of recordings was going to be a lot.

When thinking about tooling, I recalled a recent article I had written for Klara Systems on inetd. When writing that article, I created my own simple inetd service that implemented the datetime service with a shell script.

Could I deliver my CPU usage information live from the host using inetd?

activitymonitor.sh

This leads us to activitymonitor.sh, a hacky creation that abuses all good norms to give simple plots in a web browser.

activitymonitor.sh is a single shell script that consists of three parts:

- A script run by inetd.
- A basic html page.
- Some javascript to update a live page.

inetd is an Internet service runner. The full history and features of inetd fall a little outside the scope of this short article, but inetd was used for on-demand launch applications when hosts were too small to have waiting services hanging around in memory. inetd handles listening for traffic. When connections are received or datagrams arrive (for UDP based protocols), inetd launches either a built-in handler or a specified program. The program is given reads from the Internet connection on standard in. Any writes to standard out are sent out over the network. This is a really simple interface, but powerful enough to implement plain text server protocols.

A Small Script

The small script is quite simple. It has two main components and then a blob of data appended to it which is the web page and javascript.

First, the shell script deals with being a guest of inetd and parsing the http headers. The input to the script will be the HTTP headers the client sends when making its request.

```
# Read client headers, we only really care if one is data.json.
h=""
while read -t 1 h
do
    log $h

    if echo $h | grep -q "data.json";
    then
        page="data.json"
        contenttype="application/json"
    else
    fi
done
```

```
echo "HTTP/1.0 200 OK"
echo "Content-Type: $contenttype"
echo
```

The script uses the **`read`** built-in command with a timeout, meaning the script will consume all the input on the socket until there is a 1-second gap between incoming lines before proceeding. This **`read`** timeout is the rate-limiting mechanism. The headers are checked to see if the data url is being requested, if not then it delivers the base html page.

The data url path of the script is where we gather up interesting data about the host. activitymonitor.sh implements most of the default interface of top. To do so, it gathers up the required information using FreeBSD base commands and then encodes them into a JSON blob delivered to the requester.

```
if["$contenttype" == "text/html" ]
then
    indexstart=$((cat -n $0 | grep -e 'INDEX START'\
       | awk '{print $1}' | tail -n 1+1))
    sed -n"$indexstart"',$p' $0
elif["$contenttype" == "application/json" ]
then
     psout=$(ps -ax -o \
        "user,pid,%cpu,cpu %mem,vsz,rss,state,command"\
        --libxo json)
    vmstatout=$(vmstat -libxo json)
    netstatout=$(netstat -bi -libxo json)

    # kern.cp_time(s) gives us 5 numbers for the system:
    # user nice system interrupt idle
    # kern.cp_times gives us hw.ncpu entries for those 5 values
    totalcputime=$(sysctl -n kern.cp_time)
    percputime=$(sysctl -n kern.cp_times)
    ncpu=$(sysctl -n hw.ncpu)
    loadavg=$(sysctl -n vm.loadavg)
    lastpid=$(sysctl -n kern.lastpid)
    hostname=$(sysctl -n kern.hostname)

    system=$(printf ‘{"hostname":"%s",
        "cp_time":"%s", "cp_times":"%s", "ncpu":"%s",
        "loadavg":"%s", "lastpid":"%s"}’ "$hostname"
        "$totalcputime" "$percputime" "$ncpu"
        "$loadavg" "$lastpid")
    log $system

    physmem=$(sysctl -n hw.physmem)
    pagesize=$(sysctl -n hw.pagesize)
    pagecount=$(sysctl -n vm.stats.vm.v_page_count)
    wirecount=$(sysctl -n vm.stats.vm.v_wire_count)
    activecout=$(sysctl -n vm.stats.vm.v_active_count)
    inactivecount=$(sysctl -n vm.stats.vm.v_inactive_count)
    cachecount=$(sysctl -n vm.stats.vm.v_cache_count)
    freecount=$(sysctl -n vm.stats.vm.v_free_count)

    memory=$(printf '{"physmem":"%s", "pagesize":"%s",
        "pagecount":"%s", "wirecount":"%s",
        "activecout":"%s", "inactivecount":"%s",
        "cachecount":"%s", "freecount":"%s" }'
        "$physmem" "$pagesize" "$pagecount"
        "$wirecount" "$activecout" "$inactivecount"
        "$cachecount" "$freecount")

    log $totalcputime
    # deliver the data json
    printf '{"system":%s, "memory":%s, "ps":%s,
        "vmstat":%s, "netstat":%s}'"$system"
        "$memory" "$psout" "$vmstatout" "$netstatout"

fi
exit    # don't continue into the web page
```

The first set of information the script collects comes from FreeBSD tools that have libxo support. Libxo is a very powerful FreeBSD feature—base tools with support can give output in JSON natively. We grab the output of `ps`, `vmstat` and `netstat`. This lets us display processes, vm system information, and network statistics such as interface rates.

The second set in the script is concerned with getting information directly from the sysctl interface. Right now, we get the kern.cp_time(s), number of cpus, hostname, load average and base statistics about memory. Each of these has to be bundled into a JSON object by hand by using `printf`.

All of this information is then built into a JSON object which the script prints out after a simple response header. inetd then feeds back into the connecting socket and returns to the client as the body of the http response.

A Basic Web Page

The activitymonitor.sh script embeds a tiny webpage within itself which it delivers if the data url isn't requested. The page has a header, some canvases to give the javascript somewhere to draw plots, and a pre block for the top-style process list. It also embeds the javascript that causes all the magic to happen.

The html page (and javascript) is appended to the end of the shell script and marked with a `'INDEX START'` tag. activitymonitor.sh searches itself for this tag and takes everything after the tag as content to deliver, using sed to cut it up.

Some Javascript

The javascript does all the heavy lifting to parse out information from the data and to give us a user interface. It pulls out and processes the `kern.cp_times` values and calculates deltas we need to draw the plots.

The main functional thing it does other than drawing is to request data from the data side of activitymonitor.sh. Once the basic web page has loaded, the script will kick off a task to fetch the `'/data.json'` url.

When data successfully arrives it pulls in the fields it needs to from the JSON result and merges in new data to calculate what should be displayed.

Finally, at the end it calls `getdata` again to start off this task. Because of the `read` time-out in activitymonitor.sh, this will happen with a 1-second gap between results.

This is a Bad Idea

activitymonitor.sh was a fun little project that got away from me and almost became a usable tool. The CPU plots helped me understand that the FreeBSD scheduler moves processes between CPUs very eagerly and helped me devise a measurement strategy that would account for this.

While a fun project, it is not something that should be used by anyone in the real world. Instead, it is an example of the power of composability of the tools in the FreeBSD base system. Other than sysctl, all the tools we use natively output JSON and can be fed into powerful user interface languages.

This native support for JSON makes it easy to consume output from standard tools and lets automation occur with the data a human would consume trivially. This gives us power to build systems that are machine readable with the same data we read on the screen. It is a small enhancement beyond the traditional UNIX interfaces, but an incredibly powerful one.

activitymonitor.sh is available here

inetd configuration such as the following is required:

```
http-alt stream tcp     nowait  tj      /home/tj/code/activitymonitor.sh
activitymonitor.sh
```

TOM JONES wants FreeBSD-based projects to get the attention they deserve. He lives in the North East of Scotland and offers FreeBSD consulting.