The hardest part of a performance investigation can be knowing where to start: which analysis tools to run first, which metrics to read, and how to interpret the output. The choices on FreeBSD are numerous, with standard tools including top(1), vmstat(8), iostat(8), netstat(1), and more advanced options such as pmcstat(8) and DTrace. These tools collectively provide hundreds of metrics, and can be customized to provide thousands more. However, most of us aren’t full-time performance engineers and may only have the time to check familiar tools and metrics, overlooking many possible areas of trouble.

The Utilization, Saturation, and Errors (USE) method addresses this problem, and is intended for those—especially systems administrators—who perform occasional performance analysis. It is a process for quickly checking system performance early in an investigation, identifying common bottlenecks and errors. Instead of beginning with the tools and their statistics, the USE method begins with the questions we’d like answered. That way, we ensure that we don’t overlook things due to a lack of familiarity with tools or a lack of the tools themselves.

WHAT IS IT?

The USE Method can be summarized in a single sentence: For every resource, check Utilization, Saturation, and Errors. Resources are any hardware in the datapath, including CPUs, memory, storage devices, network interfaces, controllers, and busses. Software resources, such as mutex locks, thread pools, and resource control limits, can also be studied. Utilization is usually the portion
of time a resource was busy servicing work, with
the exception of storage devices where utilization
can also mean used capacity. Saturation is the
degree to which a resource is overloaded with
work, such as the length of a backlog queue.

Usually individual resource statistics also need to
be investigated. Averaging CPU utilization across
32 CPUs can hide individual CPUs that are running
at 100%, and the same can also be true for disk
utilization. Identifying these is the target of the
USE method, as they can become systemic bottlenecks.

I employed the USE method recently during a
performance evaluation, where our system was
benchmarked and compared to a similar one run-
ing Linux. The potential customer was dissatisfied
with how ZFS performed in comparison, despite it
being backed by SSDs in our system. Having
worked both ZFS and SSD pathologies before, I
knew that this could take days to unravel, and
would involve studying complex behaviors of ZFS
internals and SSD firmware. Instead of heading in
those directions, I began by performing a quick
check of system health using the USE method,
while the prospect’s benchmark was running. This
immediately identified that the CPUs were at
100% utilization, while the disks (and ZFS) were
completely idle. This helped reset the evaluation
and shifted blame to where it was due (not ZFS),
saving both my time and theirs.

Employing the USE method involves developing
a checklist of combinations of resource
and USE statistics, along with the tools in
your current environment for observing
them. I’ve included examples here, devel-
oped on FreeBSD 10 alpha 2, for both
hardware and some software resources.
This includes some new DTrace one-liners
for exposing various metrics. You may
copy these to your own company documentation
(wiki), and enhance them with any additional tools
you use in your environment.

These checklists include a terse summary of the
targets, tools, and metrics to study. In some cases,
the metric is straightforward reading from com-
mmand line tools. Many others require some math,
inference, or quite a bit of digging. This will hope-
fully get easier in the future, as tools are developed
to provide USE method metrics more easily.

[1] e.g., using per-CPU run queue length as the
### PHYSICAL RESOURCES

<table>
<thead>
<tr>
<th>COMPONENT</th>
<th>TYPE</th>
<th>METRIC</th>
</tr>
</thead>
</table>
| CPU                  | utilization  | system-wide: vmstat 1, "us" + "sy"; per-cpu: vmstat -P; per-process: top, "WCPUs"
| CPU                  | saturation   | system-wide: uptime, "load averages" > CPU count; vmstat 1, "procs:r" > CPU count
| CPU                  | errors       | dmesg; /var/log/messages; pmstat for PMC and whatever error counters are supported |
| Memory capacity      | utilization  | system-wide: vmstat 1, "fre" is main memory free; top, "Mem:"; per-process: top -d "RSS" is resident set size (Kbytes), "VSZ" is virtual memory size (Kbytes)
| Memory capacity      | saturation   | system-wide: vmstat 1, "sr" for scan rate, "w" for swapped threads (was saturated, physical: dmesg?; /var/log/messages?; virtual: DTrace failed malloc()s
| Network Interfaces   | utilization  | system-wide: netstat -i 1, assume one very busy interface and use input/output "byts"
| Network Interfaces   | saturation   | system-wide: netstat -s, for saturation related metrics, eg netstat -s l egrep 'retrans'|
| Storage device I/O   | utilization  | system-wide: iostat -x 1, "%b"; per-process: DTrace io provider, eg, iosnoop or iotop
| Storage device I/O   | saturation   | system-wide: iostat -x 1, "qlen"; DTrace for queue duration or length [4]
| Storage capacity     | utilization  | DTrace io:::done probe when [args[0]-b_error != 0] /
| Storage capacity     | errors       | file systems: df -h, "Capacity"; swap: swapinfo, "Capacity"; psstat -T, also shows swap info not sure this one makes sense - once its full, ENOSPC
| Storage controller   | utilization  | iostat -x 1, sum IOPS & tput metrics for devices on the same controller, and compare with `check_utilization and look for kernel queueing`
| Storage controller   | saturation   | DTrace the driver
| Storage controller   | errors       | see network interface saturation |
| Network controller   | utilization  | system-wide: netstat -i 1, assume one busy controller and examine input/output "byts"
| Network controller   | saturation   | see network interface saturation |
| CPU interconnect     | utilization  | pmcstat (PMC) for CPU interconnect ports, tput / max
| CPU interconnect     | saturation   | pmcstat and relevant PMCs for CPU interconnect stall cycles
| CPU interconnect     | errors       | pmcstat and relevant PMCs for whatever is available |
| Memory interconnect  | utilization  | pmcstat and relevant PMCs for memory bus throughput / max, or, measure CPI and
| Memory interconnect  | saturation   | pmcstat and relevant PMCs for memory stall cycles
| Memory interconnect  | errors       | pmcstat and relevant PMCs for whatever is available |
| I/O interconnect     | utilization  | pmcstat and relevant PMCs for tput / max if available; inference via known tput from
| I/O interconnect     | saturation   | pmcstat and relevant PMCs for I/O bus stall cycles |
| I/O interconnect     | errors       | pmcstat and relevant PMCs for whatever is available |

saturation metric: dtrace -n ‘profile-99 { @[cpu] = lquantize(‘tdq_cpu[cpu].tdq_load, 0, 128, 1); } tick-1s { printa(@); trunc(@); }’ where > 1 is saturation. If you’re using the older BSD scheduler, profile runq_length[]. There are also the sched::load-change and other sched probes.

[2] For this metric, we can use time spent in TDS_RUNQ as a per-thread saturation (latency) metric. Here is an (unstable) fbt-based one-liner: dtrace -n ‘fbt::tdq_runq_add:entry { ts[arg1] = timestamp; } fbt::choosethread:returm /ts[arg1] { @[stringof(args[1]->td_name), "runq (ns)"] = quantize(timestamp - ts[arg1]); ts[arg1] = 0; }’. This would be better (stability) if it can be rewritten to use the DTrace sched probes. It would also be great if there were simply high resolution thread state times in struct rusage or rusage_ext, eg, cumulative times for each state in td_state and more, which would make reading this metric easier and have lower overhead.

[3] e.g., for swapping: dtrace -n ‘fbt::cpu_thread_swapin:entry, fbt::cpu_thread_swapout:entry { [probefunc, stringof(args[0]->td_name)] = count(); }’ (NOTE, I would trace vm_thread_swapin() and vm_thread_swapout(), but their probes don’t exist). Tracing paging is tricker until the vminfo provider is added; you could try tracing from `swap_page_putpages()` and `swap_page_getpages()`, but I didn’t see an easy way to walk back to a thread struct; another approach may be via `vm_fault_hold()`. Good luck. See thread states [2], which could make this much easier.

[4] e.g., sampling GEOM queue length at 99 Hertz: dtrace -n ‘profile-99 { @["geom qlen"] = lquantize(g_bio_run_down.bio_queue_length, 0, 256, 1); }’

[5] This approach is different from storage
For weighted and recent usage; per-kernel-process: `top -S, “WCPU”`

per-cpu: DTrace to profile CPU run queue lengths \([1]\); per-process: DTrace of scheduler events \([2]\)

So res, “RES” is resident main memory size, “SIZE” is virtual memory size; `ps -auxw,`

may not be now); `swaponf, “Capacity” also for evidence of swapping/paging; per-process: DTrace \([3]\)

n bytes” / known max (note: includes localhost traffic); per-interface: `netstat -l interface 1, input/output “bytes” / known max`

`dropout-of-order/memory problems/overflow’; per-interface: DTrace

`etstat -i, “Ierrs”, “Oerrs” (eg, late collisions), “Colls” \([5]\)

drop|out-of-order|memory problems|overflow’; per-interface: DTrace

detstat -i, “Ierrs”, “Oerrs” (eg, late collisions), “Colls” \([5]\)

To known limits \([5]\)

bytes” / known max (note: includes localhost traffic)

treat, say, 5+ as high utilization

from iostat/netstat/…

device (disk) utilization. For controllers, percent busy has much less meaning, so we’re calculating utilization based on throughput (bytes/sec) and IOPS instead. Controllers typically have limits for these based on their busses and processing capacity. If you don’t know them, you can determine them experimentally.

PMC == Performance Monitoring Counters, aka CPU Performance Counters (CPC), Performance Instrumentation Counters (PICs), and more. These are processor hardware counters that are read via programmable registers on each CPU. The availability of these counters is dependent on the processor type. See `pmc(3)` and `pmcstat(8)`.

`pmcstat(8): the FreeBSD tool for instrumenting PMCs. You might need to run a kidload hwpmc first before use. To figure out which PMCs you need to use and how, it usually takes some serious time (days) with the processor vendor manuals; eg, the Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B, Appendix A-E: Performance-Monitoring Events.

DTT == DTraceToolkit (http://www.brendan-gregg.com/dtrace.html#DTraceToolkit) scripts. These are in the FreeBSD source (http://svn-web.freebsd.org/base/head/cddl/contrib/dtrace-toolkit/) under cddl/contrib/dtrace-toolkit, and `dtruss` is under `/usr/sbin`. As features are added to DTrace see the freebsd-dtrace mailing list (https://lists.freebsd.org/mailman/listinfo/freebsd-dtrace), more scripts can be ported.

CPI == Cycles Per Instruction (others use IPC == Instructions Per Cycle).

I/O interconnect: this includes the CPU to I/O controller busses, the I/O controller(s), and device busses (eg, PCIe).
lockstat: you may need to run kldload ksyms before lockstat will work (otherwise: “lockstat: can’t load kernel symbols: No such file or directory”).

[6] e.g., showing adaptive lock block time totals (in nanoseconds) by calling function name: dtrace -n 'lockstat:::adaptive-block { @[caller] = sum(arg1); } END { printa("%40a%@16d ns\n", @); }'

IN PRACTICE
You may notice that many metrics are difficult to observe, especially those involving pmcstat(8), which could take days to figure out. In practice, you may only have time to perform a subset of the USE method, studying those metrics that can be determined quickly, and acknowledging that some remain unknown due to pressures of time.

Knowing what you don’t know can still be enormously valuable: there may come a time when a performance issue is of vital importance to your company to resolve, and you are already armed with a to-do list of extra work than can be performed to more thoroughly check resource performance.

OTHER TOOLS
I didn’t include procstat, sockstat, gstat or others, as the USE method is designed to begin with questions, and only uses the tools that answer them. This is very different from making a list of all the tools available and then trying to find a way to use them all. Those other tools are useful for other methodologies, which can be used after this one.

It’s hoped that more tools are developed to make the USE method easier, expanding the subset of metrics that you have time to regularly check. If you are a FreeBSD developer, then you may be the first to develop a bustop, for example, which could be a PMC-based tool for showing busses and interconnects, and their utilization, saturation, and errors.

CONTINUOUS ANALYSIS
The USE method is intended as an early methodology to identify common bottlenecks and errors. These may then be studied using other methodologies, such as drill-down analysis and latency analysis. There are also performance issues which the USE method will miss entirely, and it should be treated as only one methodology in your toolbox.

REFERENCES
Thinking Methodically about Performance, Brendan Gregg, ACM Queue, vol. 10, no. 12, 2012
• The USE Method: FreeBSD Performance Checklist summary on my blog
• FreeBSD source code and man pages
• FreeBSD Wiki PmcTools
• FreeBSD Handbook

Brendan Gregg is a senior performance architect at Netflix. He is the author of the book Systems Performance (Prentice Hall, 2014), primary author of DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD (Prentice Hall, 2011), and recipient of the USENIX 2013 LiSA Award for Outstanding Achievement in System Administration. He was previously a performance lead and kernel engineer at Sun Microsystems where he developed the ZFS L2ARC and various DTrace providers.
Up with dtrace -n 'profile-997 { @[stack()] = count(); }'
for status

...to pthread_mutex_unlock() entry
...to return times
...(3C) etc.

"..." also shows current

counts; per-process: can figure out using fstat -p PID and ulimit -n
array, it errors; see fdalloc()
returning fds (eg, open(), accept(), ...)

The Dynamic Edition format offers subscribers the same features as the App, but permits viewing the Journal through your favorite browser.

Read It Today!

The DE, like the App, is an individual product. You will get an email notification each time an issue is released. A one-year subscription is $19.99, and a single copy is $6.99—the same pricing as the App version.

www.freebsdfoundation.com

The DE, like the App, is an individual product. You will get an email notification each time an issue is released. A one-year subscription is $19.99, and a single copy is $6.99—the same pricing as the App version.

www.freebsdfoundation.com