USE Method: Linux Performance Checklist

The USE Method provides a strategy for performing a complete check of system health, identifying common bottlenecks and errors. For each system resource, metrics for utilization, saturation and errors are identified and checked. Any issues discovered are then investigated using further strategies.

This is an example USE-based metric list for Linux operating systems (eg, Ubuntu, CentOS, Fedora). This is primarily intended for system administrators of the physical systems, who are using command line tools. Some of these metrics can be found in remote monitoring tools.

Physical Resources

component	type	metric
CPU	utilization	system-wide: `vmstat 1`, "us" + "sy" + "st"; `sar -u`, sum fields except "%idle" and "%iowait"; `dstat -c`, sum fields except "idl" and "wai"; per-cpu: `mpstat -P ALL 1`, sum fields except "%idle" and "%iowait"; `sar -P ALL`, same as `mpstat`; per-process: `top`, "%CPU"; `htop`, "CPU%"; `ps -o pcpu`; `pidstat 1`, "%CPU"; per-kernel-thread: `top/htop` ("K" to toggle), where VIRT == 0 (heuristic). [1]
CPU	saturation	system-wide: `vmstat 1`, "r" > CPU count [2]; `sar -q`, "runq-sz" > CPU count; `dstat -p`, "run" > CPU count; per-process: /proc/PID/schedstat 2nd field (sched_info.run_delay); `perf sched latency` (shows "Average" and "Maximum" delay per-schedule); dynamic tracing, eg, SystemTap schedtimes.stp "queued(us)" [3]
CPU	errors	`perf` (LPE) if processor specific error events (CPC) are available; eg, AMD64's "04Ah Single-bit ECC Errors Recorded by Scrubber" [4]
Memory capacity	utilization	system-wide: `free -m`, "Mem:" (main memory), "Swap:" (virtual memory); `vmstat 1`, "free" (main memory), "swap" (virtual memory); `sar -r`, "%memused"; `dstat -m`, "free"; `slabtop -s c` for kmem slab usage; per-process: `top/htop`, "RES" (resident main memory), "VIRT" (virtual memory), "Mem" for system-wide summary
Memory capacity	saturation	system-wide: `vmstat 1`, "si"/"so" (swapping); `sar -B`, "pgscank" + "pgscand" (scanning); `sar -W`; per-process: 10th field (min_flt) from /proc/PID/stat for minor-fault rate, or dynamic tracing [5]; OOM killer: `dmesg | grep killed`
Memory capacity	errors	`dmesg` for physical failures; dynamic tracing, eg, SystemTap uprobes for failed malloc()s
Network Interfaces	utilization	`sar -n DEV 1`, "rxKB/s"/max "txKB/s"/max; `ip -s link`, RX/TX tput / max bandwidth; /proc/net/dev, "bytes" RX/TX tput/max; nicstat "%Util" [6]
Network Interfaces	saturation	`ifconfig`, "overruns", "dropped"; `netstat -s`, "segments retransmited"; `sar -n EDEV`, *drop and *fifo metrics; /proc/net/dev, RX/TX "drop"; nicstat "Sat" [6]; dynamic tracing for other TCP/IP stack queueing [7]

Network Interfaces	errors	`ifconfig`, "errors", "dropped"; `netstat -i`, "RX-ERR"/"TX-ERR"; `ip -s link`, "errors"; `sar -n EDEV`, "rxerr/s" "txerr/s"; /proc/net/dev, "errs", "drop"; extra counters may be under /sys/class/net/...; dynamic tracing of driver function returns 76]
Storage device I/O	utilization	system-wide: `iostat -xz 1`, "%util"; `sar -d`, "%util"; per-process: iotop; `pidstat -d`; /proc/PID/sched "se.statistics.iowait_sum"
Storage device I/O	saturation	`iostat -xnz 1`, "avgqu-sz" > 1, or high "await"; `sar -d` same; LPE block probes for queue length/latency; dynamic/static tracing of I/O subsystem (incl. LPE block probes)
Storage device I/O	errors	/sys/devices/.../ioerr_cnt; `smartctl`; dynamic/static tracing of I/O subsystem response codes [8]
Storage capacity	utilization	swap: `swapon -s`; `free`; /proc/meminfo "SwapFree"/"SwapTotal"; file systems: "df -h"
Storage capacity	saturation	not sure this one makes sense - once it's full, ENOSPC
Storage capacity	errors	`strace` for ENOSPC; dynamic tracing for ENOSPC; /var/log/messages errs, depending on FS
Storage controller	utilization	`iostat -xz 1`, sum devices and compare to known IOPS/tput limits per-card
Storage controller	saturation	see storage device saturation, ...
Storage controller	errors	see storage device errors, ...
Network controller	utilization	infer from `ip -s link` (or /proc/net/dev) and known controller max tput for its interfaces
Network controller	saturation	see network interface saturation, ...
Network controller	errors	see network interface errors, ...
CPU interconnect	utilization	LPE (CPC) for CPU interconnect ports, tput / max
CPU interconnect	saturation	LPE (CPC) for stall cycles
CPU interconnect	errors	LPE (CPC) for whatever is available
Memory interconnect	utilization	LPE (CPC) for memory busses, tput / max; or CPI greater than, say, 5; CPC may also have local vs remote counters
Memory interconnect	saturation	LPE (CPC) for stall cycles
Memory interconnect	errors	LPE (CPC) for whatever is available

I/O interconnect	utilization	LPE (CPC) for tput / max if available; inference via known tput from iostat/ip/...
I/O interconnect	saturation	LPE (CPC) for stall cycles
I/O interconnect	errors	LPE (CPC) for whatever is available

- [1] There can be some oddities with the %CPU from top/htop in virtualized environments; I'll update with details later when I can.
- CPU utilization: a single hot CPU can be caused by a single hot thread, or mapped hardware interrupt. Relief of the bottleneck usually involves tuning to use more CPUs in parallel.
- `uptime` "load average" (or /proc/loadavg) wasn't included for CPU metrics since Linux load averages include tasks in the uninterruptable state (usually I/O).
- [2] The man page for vmstat describes "r" as "The number of processes waiting for run time", which is either incorrect or misleading (on recent Linux distributions it's reporting those threads that are waiting, *and* threads that are running on-CPU; it's just the wait threads in other OSes).
- [3] There may be a way to measure per-process scheduling latency with perf's sched:sched_process_wait event, otherwise `perf probe` to dynamically trace the scheduler functions, although, the overhead under high load to gather and post-process many (100s of) thousands of events per second may make this prohibitive. SystemTap can aggregate per-thread latency in-kernel to reduce overhead, although, last I tried schedtimes.stp (on FC16) it produced thousands of "unknown transition:" warnings.
- LPE == Linux Performance Events, aka perf_events. This is a powerful observability toolkit that reads CPC and can also use static and dynamic tracing. Its interface is the `perf` command.
- CPC == CPU Performance Counters (aka "Performance Instrumentation Counters" (PICs) or "Performance Monitoring Events" (PMUs) or "Hardware Events"), read via programmable registers on each CPU by `perf` (which it was originally designed to do). These have traditionally been hard to work with due to differences between CPUs. LPE `perf` makes life easier by providing aliases for commonly used counters. Be aware that there are usually many more made available by the processor, accessible by providing their hex values to `perf stat -e`. Expect to spend some quality time (days) with the processor vendor manuals when trying to use these. (My short video about CPC may be useful, despite not being on Linux).
- [4] There aren't many error-related events in the recent Intel and AMD processor manuals; be aware that the public manuals may not show a complete list of events.
- [5] The goal is a measure of memory capacity saturation - the degree to which a process is driving the system beyond its ability (and causing paging/swapping). High fault latency works well, but there isn't a standard LPE probe or existing SystemTap example of this (roll your own using dynamic tracing). Another metric that may serve a similar goal is minor-fault rate by process, which could be watched from /proc/PID/stat. This should be available in `htop` as MINFLT.
- [6] Tim Cook ported nicstat to Linux; it can be found on sourceforge or his blog.
- [7] Dropped packets are included as both saturation and error indicators, since they can occur due to both types of events.
- [8] This includes tracing functions from different layers of the I/O subsystem: block device, SCSI, SATA, IDE, ... Some static probes are available (LPE "scsi" and "block" tracepoint events), else use dynamic tracing.
- 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).

- Dynamic Tracing: Allows custom metrics to be developed, live in production. Options on Linux include: LPE's "perf probe", which has some basic functionality (function entry and variable tracing), although in a trace-n-dump style that can cost performance; SystemTap (in my experience, almost unusable on CentOS/Ubuntu, but much more stable on Fedora); DTrace-for-Linux, either the Paul Fox port (which I've tried) or the OEL port (which Adam has tried), both projects very much in beta.

Software Resources

component	type	metric
Kernel mutex	utilization	With CONFIG_LOCK_STATS=y, /proc/lock_stat "holdtime-totat" / "acquisitions" (also see "holdtime-min", "holdtime-max") [8]; dynamic tracing of lock functions or instructions (maybe)
Kernel mutex	saturation	With CONFIG_LOCK_STATS=y, /proc/lock_stat "waittime-total" / "contentions" (also see "waittime-min", "waittime-max"); dynamic tracing of lock functions or instructions (maybe); spinning shows up with profiling (`perf record -a -g -F 997 ...`, `oprofile`, dynamic tracing)
Kernel mutex	errors	dynamic tracing (eg, recusive mutex enter); other errors can cause kernel lockup/panic, debug with kdump/`crash`
User mutex	utilization	`valgrind --tool=drd --exclusive-threshold=...` (held time); dynamic tracing of lock to unlock function time
User mutex	saturation	`valgrind --tool=drd` to infer contention from held time; dynamic tracing of synchronization functions for wait time; profiling (oprofile, PEL, ...) user stacks for spins
User mutex	errors	`valgrind --tool=drd` various errors; dynamic tracing of pthread_mutex_lock() for EAGAIN, EINVAL, EPERM, EDEADLK, ENOMEM, EOWNERDEAD, ...
Task capacity	utilization	`top/htop`, "Tasks" (current); `sysctl kernel.threads-max`, /proc/sys/kernel/threads-max (max)
Task capacity	saturation	threads blocking on memory allocation; at this point the page scanner should be running (sar -B "pgscan*"), else examine using dynamic tracing
Task capacity	errors	"can't fork()" errors; user-level threads: pthread_create() failures with EAGAIN, EINVAL, ...; kernel: dynamic tracing of kernel_thread() ENOMEM
File descriptors	utilization	system-wide: `sar -v`, "file-nr" vs /proc/sys/fs/file-max; `dstat --fs`, "files"; or just /proc/sys/fs/file-nr; per-process: `ls /proc/PID/fd | wc -l` vs `ulimit -n`
File descriptors	saturation	does this make sense? I don't think there is any queueing or blocking, other than on memory allocation.
File descriptors	errors	`strace` errno == EMFILE on syscalls returning fds (eg, open(), accept(), ...).

- [8] Kernel lock analysis used to be via lockmeter, which had an interface called "lockstat".

What's Next

See the USE Method for the follow-up strategies after identifying a possible bottleneck. If you complete this checklist but still have a performance issue, move onto other strategies: drill-down analysis and latency analysis.

Acknowledgements

Resources used:

- dstat documentation
- 20 Linux monitoring tools every sysadmin should know
- Rosetta Stone for Unix
- SystemTap Example Scripts for schedtimes.stp
- PerfUserGuide Linux profiling with Perf (LPE)
- perf announcements on LKML by Ingo Molnar, Peter Zijlstra, ...
- Perf homepage
- lock_stat by Peter Zijlstra
- lockmeter, for historical interest (now done via lock_stat)
- oprofile by John Levon
- Valgrind DRD which has a good list (8.2.4) of detected lock errors
- Linux kernel source
- man pages

Filling this this checklist has required a lot of research, testing and experimentation. Please reference back to this post if it helps you develop related material.

It's quite possible I've missed something or included the wrong metric somewhere (sorry); I'll update the post to fix these up as they are understood.

Last updated: 29-Sep-2013