|  |  |
| --- | --- |
|  | |
| The Linux Performance Handbook | |
|  | |
|  | |
|  | |
| Classification: <classification>  Version: <version> |  |
| ID: <CRM-ID> |
| YYYY-MM-DD |

Table of Contents

1 Reason For This Handbook 4

2 Introduction to Linux 6

2.1 Rapid Growth 6

2.2 UNIX Clone 7

2.3 Advantages as Compared With Proprietary Unix-like Systems 8

2.4 Advantages as Compared With Microsoft Windows 8

2.5 What’s your Problem? 9

3 Methodologies 10

3.1 Anti-Method 10

3.2 Street Light Anti-Method 10

3.3 Drunk Man Anti-Method 10

3.4 Blame Someone Else Anti-method 11

3.5 Problem Statement Method 11

3.6 Workload Characterization Method 12

3.7 USE Method 12

3.8 CPU Profile Method 12

3.9 RTFM Method (https://en.wikipedia.org/wiki/RTFM) 13

3.10 Ad Hoc Methods 13

4 Linux – The Basics 14

4.1 CPU 14

4.2 Memory 14

5 Tool Types 17

6 Simple Tools 18

7 Intermediate Tools 21

8 Advanced Tools 25

9 Benchmarking Tools 29

10 Linux Tracing Tools 30

11 Conclusions 34

12 Appendix A: A Question Based Methodology 35

13 Appendix B: Notable Events 42

14 Appendix C: Collectl 43

15 Appendix D: ftrace 46

16 Appendix E: eBPF 47

17 Appendix F: LinuxKI 50

18 Appendix G: Perf\_events 53

19 Appendix H: perf\_tools 62

20 Appendix I: sar 64

20.1 sadf and gnuplot 64

20.2 kSar 66

21 Change log 69

# Reason For This Handbook

Is performance self-evident? All end users and others in your organization, who use IT systems and supported business applications, take it for granted that these systems and applications will provide instant results, a good user experience, and a quick response time.

The reality is that all stakeholders and key businesses have to work hard every day to maintain and improve the performance figures within the system and application landscape. Continuous proactive monitoring of performance figures and trends are essential to ensure long-term usability of IT assets, and in-depth problem investigation as reactive response to performance incidents requires high attention and specialized knowledge.

Increasing response time and decreasing performance leads to user dissatisfaction, lower productivity, and customer complaints.

Basics of Performance Management

Firstly, let’s distinguish between Proactive Performance Management and Reactive Performance Troubleshooting, also known as Root Cause Analysis (RCA). Proactive activities are long term—such activities enable smooth planning while protecting your business interests and IT investments. On the other hand, reactive activities typically come with high pressure, management attention, escalations and time constraints. It is essential to stay calm and handle reactive activities in a structured and organized manner and be well prepared for reactive analysis.

Proactive Performance Management

* What should be measured and what are the metrics required to give the satisfaction to the questions asked?
* Where should we measure? The application, database, virtualization, OS or storage subsystem?
* What tools should be used?
* How to evaluate the findings?

An experienced performance consultant can help with these answers**.**

Reactive Performance Management

In this position you will most likely you engaging in a RCA and answering the questions:

* When the application is running, what is it doing?
* When the application is waiting what it is waiting for?
* What resources is it using?
* What tools and collectors are needed to answer these questions?

This handbook does not address the subject of Linux Tuning. This may actually be the required result of a Linux performance analysis but is not included in this handbook.

# Introduction to Linux

*Linux* is a high performance, yet completely free, [Unix-like](http://www.linfo.org/unix-like.html) [operating system](http://www.linfo.org/operating_systems_list.html) that is suitable for use on a wide range of computers and other products. Most [*distributions*](http://www.linfo.org/distributions_list.html) (i.e., versions) consist of a [*kernel*](http://www.linfo.org/kernel.html) (i.e., the core of the operating system) together with hundreds of free utilities and [application programs](http://www.linfo.org/appslist.html) in a coordinated package.

A narrower, and somewhat less common, meaning of the term *Linux* is just the kernel itself. However, when referring to just the kernel, usually the expression *the Linux kernel* is used.

Linux was started as a hobby in 1991 by [Linus Torvalds](http://www.linfo.org/linus.html) while a student at the University of Helsinki (in [Finland](http://www.linfo.org/finland.html)) because he was unhappy with the [MS-DOS](http://www.linfo.org/ms-dos.html) operating system that came with his new personal computer. He greatly preferred the much more powerful and stable [UNIX](http://www.linfo.org/unix_upper.html) that he had been using on the university's computers, but he was not able to afford the high licensing fees for any of the commercial versions then available. Today, Torvalds remains the spiritual leader of the Linux movement, and he still coordinates the development of the Linux kernel.

## Rapid Growth

The use of Linux by individuals, corporations, government agencies and academic institutions around the world has been growing swiftly, and many computer experts think that it will eventually become the most widely used operating system for many or most types of applications.

This rapid growth is a result of several factors including

* the major advantages that Linux has over other operating systems (including over the other Unix-like operating systems and the Microsoft Windows family of operating systems),
* the rapid progress that is being made on further improving the performance and increasing the functions of Linux,
* the expanding array of high-quality application programs,
* the growing awareness by individuals, businesses and other organizations throughout the world of the advantages of Linux and
* an increase in the number of people who are familiar with installing, administering and using Linux.

Well in excess of a hundred (and possibly more than two hundred) Linux distributions have been developed by a diverse range of companies, non-commercial organizations and individuals. Some of the most popular are Red Hat, SuSE, Mandrake, Debian, Slackware, Linspire and Ubuntu. In addition to these mainstream distributions, numerous specialized distributions are also available, including those optimized for specific types of computers or applications (e.g., for use on notebook computers or [routers](http://www.linfo.org/router.html)), those for specific languages or countries (e.g., Polish or Chinese) and ultra-miniature distributions (some of which can even fit on just a single floppy disk, such as [muLinux](http://www.linfo.org/mulinux)).

## UNIX Clone

UNIX was originally developed by [Ken Thompson](http://www.linfo.org/thompson.html) in 1969 at [Bell Labs](http://www.linfo.org/bell_labs), the highly innovative research arm of AT&T. Much subsequent work was carried out at the University of California at Berkeley (UCB).

Linux is a [*clone*](http://www.linfo.org/clone.html) of UNIX; that is, it was developed to mimic the form and function of UNIX but its [*source code*](http://www.linfo.org/source_code.html) was written completely independently (i.e., none of it was copied from UNIX source code). Source code is the version of an operating system or other software as it is originally *written* (i.e., typed into a computer) by a human in a programming language (e.g., the [C language](http://www.linfo.org/c.html) in the case of the Linux kernel).

Linux incorporates all of the features that have made Unix-like systems the longest-lived and what many consider to be the best operating systems still in widespread use. That is, it is a *multiuser* (i.e., allows multiple simultaneous users), [*multitasking*](http://www.linfo.org/multitasking.html), highly flexible (with regard to configuration), inherently secure (including high resistance to viruses, spyware and other [*malware*](http://www.linfo.org/malware.html)) and extraordinarily [*robust*](http://www.linfo.org/robust.html) (i.e., resistance to crashing and needing rebooting) operating system. A multitasking operating system is one in which multiple programs or [*processes*](http://www.linfo.org/process.html) (also referred to as *tasks*) can execute (i.e., run) on a single computer seemingly simultaneously and without interfering with each other.

As is the case with most of the Unix-like operating systems, Linux is a highly mature (and very sophisticated) work of engineering that has been skillfully crafted by the collective efforts of thousands of the best minds in computer science. There is no planned (and little unplanned) obsolescence.

Yet Linux is much more than just a clone of another highly successful operating system. It also represents a philosophy, one which not only incorporates the simple but elegant Unix philosophy but which also has also taken it a big step further and made it a truly free operating system.

Moreover, Linux is a product of the [Internet](http://www.linfo.org/internet.html) era. In contrast to [*proprietary*](http://www.linfo.org/proprietary.html) (i.e., commercial) operating systems, which have been developed mostly by paid programmers employed at corporations, Linux has been developed virtually since its inception by an informal, world-wide network of unpaid (but highly skilled and motivated) volunteers who communicate via the Internet.

## Advantages as Compared With Proprietary Unix-like Systems

Linux has several important advantages over the proprietary Unix-like operating systems (e.g., AIX, HP-UX and Solaris). One is that it is free software. This means that it is free both in a monetary sense and with regard to use. That is, everyone is permitted to download Linux from the Internet (or obtain it from other sources, including from friends) at no cost And everyone is also permitted to use it for any desired purpose, including studying, modifying, extending, installing on as many computers as desired, making copies as many copies as desired and redistributing. This is possible because Torvalds wisely released it under a free software license, the [*GNU*](http://www.linfo.org/gnu.html)*General Public License* (GPL).

Yet another advantage of Linux as compared with proprietary Unix-like operating systems is that it can generally run on a much wider range of hardware, including both system types and processor types. For example, it can run on cell phones, game machines, notebook computers, desktop computers, workstations, mainframes, supercomputers -- and even some wristwatches.

Linux is no longer the only Unix-like operating system that available under a free software license. There are several others, most notably the BSD (Berkeley Software Distribution) systems, which descended from work done on UNIX at UCB. Each of these systems, which include FreeBSD, NetBSD, NetBSD and Darwin (which is used by Mac OS X), has its own advantages and disadvantages. However, the number of users of Linux is much greater than that of the BSDs, mainly because it is easier to use, particularly for the less technically proficient.

## Advantages as Compared With Microsoft Windows

Linux also has some very big advantages as compared with the Microsoft Windows family of operating systems. The most obvious is that businesses and other organizations can save vast sums of money because there are no licensing fees nor is there any pressure for costly (and often disruptive) upgrades (so-called *forced upgrades*).

Linux can also cut administration and maintenance costs as compared with the Microsoft Windows operating systems because it is considerably more stable (it rarely crashes or needs rebooting) and is highly resistant to viruses and other malicious attacks.

In addition, Linux has the advantage that it can operate on older hardware that is unsuitable for newer versions of Microsoft Windows. This is because it is much more compactly written. Whereas upgrading to newer versions of Microsoft Windows generally requires costly outlays for new hardware, it is often possible to upgrade to newer versions of Linux without buying any new equipment.

The availability of the source code for Linux can also offer substantial benefits to users as compared with the closed (i.e., secret) source code for the Microsoft Windows operating systems. For example, corporations, government agencies and other organizations can monitor the code for security holes, including secret [*backdoors*](http://www.linfo.org/backdoor.html) that allow others (e.g., government agencies) to access or change data. Having the source code also allows users to customize Linux to a far greater extent than can be done with closed source operating systems.

Thousands of application programs are available for Linux. Many of them offer performance and functions at least equal to those available for Microsoft Windows and other operating systems. Moreover, most of them are also free software, and many are included on the same CDROMs that contain Linux and can be installed automatically during Linux installation.

This wealth of diversity gives plenty of scope for differences in performance and hence plenty of diversity in the tools and methods available for investigating perceived performance problems.

This handbook tries to give the reader a helping hand in the investigation and solving of perceived performance problems.

## What’s your Problem?

When somebody rings you in a grim panic and says “it’s not working properly” or “it’s going too slow” a thousand and one questions could go through your mind. If the problem really is performance based then what the person is really asking is for you to solve the riddle

**If its running what is it doing ?**

**If its waiting - what is it waiting for ?**

It is now up to you as the performance specialist to diagnose the lack of information you have been presented with and answer the two questions above in a way which the customer can understand.

Be kind to your customer - he is under pressure and has probably looked at the problem already for several hours staring into a screen with the simple tools at his fingers.

It is up to you as the expert to come with “added value” and to understand and then solve his problem

# Methodologies

There are a number of approaches to any problem. Here are a number of common approaches to fixing performance problems some of them which are unfortunately all too common.

Methods can be combined and or ignored!

## Anti-Method

This is the use of any sort of methodology. In this approach you basically go into a panic and like a headless chicken flounder around looking for someone or something who can get you off the hook!

## Street Light Anti-Method

Here you look at the things which are most familiar to you or easily found on the internet and try them. If you are lucky you can make a square peg fit into a round hole.



You can run your chosen tool and see if it helps.

## Drunk Man Anti-Method

There are plenty of items which can be tuned in Linux. If fact Linux is pretty good at coming pre-tuned and over the years it comes with a reasonably good set of defaults. However there are occasions in which tuning can and should be carried out. In this method you simply take the tuning possibilities at random until one of them does or doesn’t help.

You will be looking to update your CV pretty quickly!

## Blame Someone Else Anti-method

It’s always good to pass the buck.

You can easily claim that your change management process is 100% and nothing has changed and hence it’s not your problem. In our modern complex world the end-to-end traffic is pretty complex and system problems can be provoked by systems which are very much remote to yours. For example a denial of use hacker attack which brings your network cards and your systems to their knees!

The engineer in this case would do something like:

* *Find a system or environment for which he is not responsible*
* *State that the issue is with that environment*
* *Re-direct the issue/ticket to the responsible team*
* *Case closed - Grab a cup of coffee !!!*

Once again have your CV ready!

## Problem Statement Method

Do you have a problem at all? Just because you get one phone call saying “your system is slow” then how does one start looking at the complaint?

Firstly one should ask if there is a problem at all. Think of the following questions and their answers:

* What makes you think there is a performance problem?
* Has the system ever performed well?
* What has changed recently? (software, hardware, load)
* Does the problem affect other people or applications or is it just you?
* Has an employee just left and maybe compromised the system?
* Is the problem constant or changing with time?
* Find the record of performance tests before the system went live and compare the observations. Are there any similarities?

## Workload Characterization Method

In this method one looks basically at the incoming load.

*Who or what is causing the load ? pid, uid, ip address …*

*How is the load changing over time ?*

*What is the load?*

This method may work if your performance problem is actually caused by excessive load and load alone which is rather short sighted!

## USE Method

In this method you could check the characteristics of the resources. That is for resources such as

* Busses
* Interconnects
* Controllers
* Fans
* Power supplies
* Disk
* Network
* Memory
* Swap
* Cache
* Cpu

You should for each resource check:

* Utilization
* Saturation
* Errors

So here you start with the questions and then find the tools to document the ability of the resource to perform the requested load.

## CPU Profile Method

Here you look at what the CPU is doing and understand what it is doing and why it is doing it. For example does the process get a reasonable time to execute on the CPU or does it get pre-empted too quickly and too frequently.

Understanding what your CPU’s are doing helps you discover a wide range or problems which are CPU related.

## RTFM Method (<https://en.wikipedia.org/wiki/RTFM>)

Knowledge is an asset which cannot be underestimated. To understand and solve performance issues usually requires significant experience with the Operating system and its tools. There are no tool sets which are good enough to identify the problem and suggest the remedy. Hence for now we must use our own brains to assemble the knowledge, examine the data resulting from the output of the tools and interpret our way to a solution.

We need to:

* Read man pages
* Read books
* Search the web for similar experiences
* Ask our peers and fellow engineers
* Ask our support services tree
* Examine source code
* Experiment
* Be inquisitive
* Be suspicious

## Ad Hoc Methods

There is no reason why an ad hoc method cannot be the correct one for you. If it works for you then document it, maintain it and use it to guide you.

See Appendix A for an example of a complete Ad Hoc method

# Linux – The Basics

## CPU

CPU utilization is a straight forward event. At any given time the CPU can be doing any one of seven things

1. It can be idle which means the processor is not actually doing any work and is waiting for something to do
2. It can be running user code which is specified as “user” time.
3. It can be executing code in the Linux kernel on behalf of the application code. This is the system time.
4. It can be executing user code that has been ”nice”ed or set to run at a lower priority than normal processes.
5. It can be in iowait which means the system is spending its time waiting for I/O to complete.
6. It can be in irq state which means it is high-priority kernel code handling a hardware interrupt.
7. It is in softirq mode which means it is executing kernel code that was also triggered by an interrupt but it is running at a lower priority.

## Memory

The memory subsystem is more complex to describe. In modern processors saving information to and retrieving from the memory subsystem usually takes longer than the CPU executing the code and manipulating the information. The CPU usually spends a significant amount of time idle, waiting for the instructions and data to be retrieved from memory before it can execute them or operate based on them. Processors have various levels of cache that compensate form the slow memory performance. When performance is slow there is talk of a “cache miss”.

Any given Linux system has a certain amount of RAM or physical memory. Linux breaks this up into chunks or “pages” of memory. When allocating or moving around memory Linux operates on page-sized pieces rather than individual bytes. When reporting some memory statistics the Linux kernel reports the number of pages per second.

On the IA32 architecture the page size is 4KB. In rare cases these page-sized chunks of memory can cause too much overhead to track so the kernel manipulates memory in much bigger chunks known as HugePages. These are on the order of 2048KB rather than 4KB and greatly reduce the overhead for managing very large amounts of memory. Certain applications such as Oracle use these huge pages to load an enormous amount of data in memory while minimizing the overhead that the Linux kernel needs to manage it. If HugePages are not completely filled with data these can waste a significant amount of memory.

All systems have a fixed amount of fixed amount of physical memory in the form of RAM chips. The Linux kernel allows applications to run even if they require more memory than available with the physical memory. The Linux kernel uses the hard drive as a temporary. This hard drive is called the swap space.

Although swap is an excellent way to allow processes to run it is terribly slow as it is access the disk with its moving parts. If a system is performing poorly it usually proves helpful to determine how much swap the system is using.

If your system has much more physical memory than required by your applications Linux will cache recently used files in physical memory so that subsequent accesses to that file do not require an access to the hard drive. This can greatly speed up applications that access the hard disk frequently.

In addition to cache Linux also uses extra memory as buffers. To further optimize applications Linux sets aside memory to use for data that needs to be to be written to disk. These set-asides are called buffers. If an application has to write something to the disk which would usually take a long time Linux lets the application continue immediately but saves he file data into a memory buffer. At some point in the future the buffer is flushed to disk but the application can continue immediately.

It can be discouraging to see very little free memory in a system because of the cache and buffer usage but this is not necessarily a bad thing. By default Linux tries to use as much memory as possible. This is good. If Linux detects any free memory it caches applications and data in the free memory to speed up future accesses. Because it is usually a few orders of magnitude faster to access memory rather than disk this can dramatically affect system performance. When the system needs the cache memory for more important things the cache memory is erased and given to the system.

Active memory is currently being used by a process. Inactive memory is memory that is allocated but has not been used for a while. When required the Linux kernel takes a process’s least recently used memory pages and moves them form the active to the inactive list. When choosing which memory will be swapped to disk the kernel chooses from the inactive memory list.

In addition to the memory that applications allocate the Linux kernel consumes a certain amount for book keeping purposes. This book keeping includes for example keeping track of data arriving from network or disk I/O devices as well as keeping track of which processes are running and which are sleeping. To manage this book keeping the kernel the kernel has a series of caches that contains one or more slabs of memory. Each slab consists of one or more objects. The amount of slab memory consumed by the kernel depends on which parts of the Linux kernel are being used and can change as the type of load on the machine changes.

The basic problem we need to attack and answer is how we measure the elements of the kernel.



There are a number of notable events and affects which are notable. These are compiled in Appendix B.

# Tool Types

Tools can be fitted into several classes.

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  | **Observability** | With these tools you watch what the system is doing. They are usually safe.  Some tools add kernel drivers and may have a “slow down” effect and should only be used in short periods. |
|  | **Benchmarking** | This sort if activity should be carried before your system goes live so it probably has little use in your current performance analysis. It is included here to remind you that next time it must be included as part of your project process ! |
|  | **Tuning** | You can change many parameters of which some may require a reboot. Of course changes should be tested on a dedicated test system first and the change should be expected to produce the solution.  This is a danger area and many engineers tend to change parameters based only on a hunch! |
|  | **Static** | Here you only check current configurations without any change. A further analysis may indicate a change after having being tested properly. |

|  |  |  |
| --- | --- | --- |
|  |  |  |

# Simple Tools

Here are some of the simple tools. Use the man pages to see all the options.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Tool** | **Explanation** | **Example** |
|  | uptime | uptime gives a one line display of the following information.  The current time, how long the system has been running, how many users are currently logged on, and the system load averages for the past 1, 5, and 15 minutes.  System load averages is the average number of processes that are either in a runnable or uninterruptable state. A process in a runnable state is either using the CPU or waiting to use the CPU. A process in uninterruptable state is waiting for some I/O access, eg. waiting for disk. The averages are taken over the three time intervals. Load averages are not normalized for the number of CPUs in a system, so a load average of 1 means a single CPU system is loaded all the time while on a 4 CPU system it means it was idle 75% of the time. |  |
|  | top | The top program provides a dynamic real-time view of a running system. It can display system summary information as well as a list of processes or threads currently being managed by the Linux kernel. The types of system summary information shown and the types, order and size of information displayed for processes are all user configurable and that configuration can be made persistent across restarts.  See also “htop” and “glances” |  |
|  | ps | ps displays information about a selection of the active processes.  There are many options which control what and how it is displayed. For example  ps -eo user,sz,rss,minflt,majflt,pcpu,args |  |
|  | vmstat | vmstat reports information about processes, memory, paging, block IO, traps, disks and cpu activity. | See the result of vmstat 5 times with a delay of 5 seconds. |
|  | mpstat | The mpstat command writes to standard output activities for each available processor, processor 0 being the first one.  Use “mpstat –A” to see all statistics.  This is a good tool to tell you what each processor is doing. |  |
|  | free | free displays the total amount of free and used physical and swap memory in the system, as well as the buffers and caches used by the kernel.  “free –h” gives the output in a human readable form. |  |
|  | iostat | The iostat command is used for monitoring system input/output device loading by observing the time the devices are active in relation to their average transfer rates.  The options and output are important. Read the man pages for further information.  Importantly here note the “%steal” column which shows the wait while hypervisor is servicing another virtual machine. In our world or cloud systems and shared hardware this can often be another customer’s virtual machine which we cannot see. The options here are to move the virtual machine to another piece of hardware or ensure you own all the hardware and can control the utilization yourself. |  |

This can be represented by the following visualization.



# Intermediate Tools

Some of these tools may not be part the standard installation and may have to be installed

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Tool** | **Explanation** | **Example** |
|  | strace | Trace system calls and signals.  It can trace existing process using a pid or spawn its own process with a given command.  It slows the execution of the program massively so use with caution on existing running programs.  Can be good for detecting “Permission Denied” characteristics from already running daemons. |  |
|  | tcpdump | Dump network traffic.  Dump’s can be saved and passed to wireshark for easier analysis.  Good to detect network “resets” initiate by a remote network entity. |  |
|  | netstat | Print network connections, routing tables, interface statistics, masquerade connections, and multicast memberships.  A real catch-all for network diagnosis. Read the man page!  For example to show listening ports  netstat –anlp | grep LISTEN |  |
|  | pidstat | The pidstat command is used for monitoring individual tasks currently being managed by the Linux kernel. |  |
|  | swapon | swapon and swapoff is usually used to specify devices on which paging and swapping are to take place.  However using it with the option “-s” gives you the current usage. This is equivalent to using “cat /proc/swaps” |  |
|  | lsof | List open files.  Too many options to mention here. Read the man page. | List TCP based open files. |
|  | sar | Collect, report or save system activity information.  A really good multipurpose tool to look at cpu, disk, system calls and interrupts.  Provided by the sysstat package.  To record the standard data to a file called “datafile” 500 times with an interval of 5 seconds:  # sar -o datafile 5 500 count >/dev/null 2>&1 &  Collect all data once  # sar –A  See Appendix I. |  |
|  | collectl | Collects data that describes the current system status. Is a good candidate for gathering data in a longer term basis to export to an analysis system.  Not generally a part of the normal installed based  # yum install –y epel-release  # yum install –y collect  See Appendix C for a more detailed use of collectl. |  |
|  | dstat | A versatile tool for generating system resource statistics.  Gathers the same as vmstat and iostat.  Not generally a part of the normal installed base. Use “yum install dstat” |  |
|  | oprofile | Profile is a profiling system for systems running Linux 2.6 and greater. Profiling runs transparently in the background and profile data can be collected at any time. OProfile makes use of the hardware performance counters provided on Intel, AMD, and other processors, and uses a timer-interrupt based mechanism on CPUs without counters. OProfile can profile the whole system in high detail.  # yum install oprofile |  |
|  | cachegrind | See <https://kcachegrind.github.io/html/Home.html> |  |
|  | nm | List symbols from object files. |  |
|  | valgrind | Valgrind is a flexible program for debugging and profiling Linux executables. It consists of a core, which provides a synthetic CPU in software, and a series of debugging and profiling tools. The architecture is modular, so that new tools can be created easily and without disturbing the existing structure.  # yum install valgrind |  |
|  | memprof | See  <https://github.com/GNOME/memprof> |  |

This can be represented by the following visualization



# Advanced Tools

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Tool** | **Explanation** | **Example** |
|  | ltrace | A library call tracer. It runs the specified command until it exists. It intercepts and records the dynamic library calls which are called by the executed process and the signals which are received by that process.  Very similar to strace.  Install using:  # yum install ltrace |  |
|  | ss | A utility to investigate sockets similar to netstat.  For example see the listening connections:  # ss –lt |  |
|  | iptraf | Interactive colorful IP lan monitor.  Iptraf-ng is an ncurses-based IP LAN monitor that generates various network statistics including TCP info, UDP counts, ICMP and OSPF information, Ethernet load info, node stats, IP checksum errors, and others  Install using “yum install iptraf”.  Run using “iptraf-ng” |  |
|  | ethtool | Query or control network driver and hardware tools |  |
|  | tiptop | The tiptop program provides a dynamic real-time view of the tasks running in the system. tiptop is very similar to top, but the information displayed comes from hardware counters.  Cannot be used on clouds since it needs PMCs enabled. |  |
|  | iotop | iotop watches I/O usage information output by the Linux kernel and displays a table of current I/O usage by processes or threads on the system. |  |
|  | blktrace | Generates traces of the i/o traffic on block devices  blktrace is a block layer IO tracing mechanism which provides detailed information about request queue operations up to user space. There are three major components: a kernel component, a utility to record the i/o trace information for the kernel to user space, and utilities to analyse and view the trace information. |  |
|  | slabtop | slabtop displays detailed kernel slab cache information in real time  Slab allocation is a memory management mechanism intended for the efficient memory allocation of kernel objects. It eliminates fragmentation caused by allocations and deallocations. The technique is used to retain allocated memory that contains a data object of a certain type for reuse upon subsequent allocations of objects of the same type. It is analogous to an object pool but only applies to memory, not other resources. |  |
|  | /proc | This is the basic method of extracting data.  Some examples are:  cat /proc/cpuinfo  cat /proc/devices  cat /proc/version  cat /proc/sys/kernel/ostype  cat /proc/meminfo  cat /proc/filesystems  cat /proc/sys/dev/cdrom/info  cat /proc/mounts  cat /proc/locks  cat /proc/uptime |  |
|  | pcstat | Get page cache statistics from files.  This helps to answer the question “is linux caching my data or not?” Helpful when dealing with databases or IO-intensive applications.  Install with:  go get golang.org/x/sys/unix  go get github.com/tobert/pcstat/pcstat | A silly example to see of /etc/passwd is cached |
|  | rdmsr | A utility to read Model specific Registers (MSRs).  Fetch using:  git clone <https://github.com/01org/msr-tools.git>  gcc rdmsr.c  ./a.out |  |

# Benchmarking Tools

Although benchmarking is not the target of this handbook we can name some the popular benchmarking tools. Go and investigate them for yourself!

Some of them will require the EPEL repository.

# yum install –y epel-release

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Area** | **Tool** | **Source** |
|  | Multi | Perf bench | # yum install perf |
|  |  | UnixBench | https://aur.archlinux.org/packages/unixbench/ |
|  |  | imbench | <https://github.com/daneroo/im-bench> |
|  |  | sysbench | <https://github.com/akopytov/sysbench> |
|  | Disk | dd | # yum install coreutils |
|  |  | hdparm | # yum install hdparm |
|  |  | fio | # yum install fio |
|  | Applications | ab | # yum install httpd-tools |
|  |  | wrk | Fetch from <https://github.com/wg/wrk> |
|  |  | jmeter | Fetch from <http://www.gtlib.gatech.edu/pub/apache//jmeter/binaries/> |
|  |  | openssl | # yum install openssl |
|  | Networking | ping | # yum install iputils |
|  |  | hping3 | # yum install hping3 |
|  |  | iperf | # yum install iperf |
|  |  | ttcp | From <https://centos.pkgs.org/7/repoforge-x86_64/ttcp-3.7-1.2.el7.rf.x86_64.rpm.html> |
|  |  | traceroute | # yum install traceroute |
|  |  | mtr | # yum install mtr |
|  |  | pchar | Fetch from <ftp://ftp.ee.lbl.gov/pathchar/> |
|  |  |  |  |

# Linux Tracing Tools

Tracing Tools are defined to be those tools which in general provide a lot more information in much greater detail. They often go into the kernel space.

This is where the majority of effort will go with complex performance problems.

Once of the problems here is to find out which one is best to use or should you decide to concentrate on one or two and be expert in those. In reality customers do not take kindly to the installation of new tools which are possible not-classified or verified by the security department. Hence it is probably best to try and use the tools which are already on the system and move to others only if absolutely necessary. Hence ftrace, eBPF and perf\_events should be first on your list!

|  |  |  |
| --- | --- | --- |
|  | **Tool** | **Explanation** |
|  | ftrace | Added since 2.6.27. Already enabled on 3.2+  One of the diagnostic facilities is ftrace, which is can be used to analyse and debug latency and performance issues that occur outside of user-space.  Ftrace is built into all modern kernels and it usually used by other utilities who want to fetch complex data from kernel events.  See <https://lwn.net/Articles/365835/>  See appendix D for a more detailed treatment.  See also Appendix H for an easy to use set of tools called perf-tools to utilize ftrace. |
|  | eBPF | The Berkeley Packet Filter (BPF) provides a raw interface to [data link layers](https://en.wikipedia.org/wiki/Data_link_layer), permitting raw link-layer packets to be sent and received. It is available on most [Unix-like](https://en.wikipedia.org/wiki/Unix-like) operating systems. In addition, if the driver for the network interface supports [promiscuous mode](https://en.wikipedia.org/wiki/Promiscuous_mode), it allows the interface to be put into that mode so that all packets on the [network](https://en.wikipedia.org/wiki/Computer_network) can be received, even those destined to other hosts.  BPF supports filtering packets, allowing a [userspace](https://en.wikipedia.org/wiki/Userspace) [process](https://en.wikipedia.org/wiki/Process_(computer_science)) to supply a filter program that specifies which packets it wants to receive. For example, a [tcpdump](https://en.wikipedia.org/wiki/Tcpdump) process may only want to receive packets that initiate a TCP connection. BPF only returns packets that pass the filter that the process supplies. This avoids copying unwanted packets from the [operating system](https://en.wikipedia.org/wiki/Operating_system) [kernel](https://en.wikipedia.org/wiki/Kernel_(computer_science)) to the process, greatly improving performance.  BPF is sometimes used to refer just to the filtering mechanism, rather than to the entire interface. Some systems, such as [Linux](https://en.wikipedia.org/wiki/Linux) and [Tru64 UNIX](https://en.wikipedia.org/wiki/Tru64_UNIX), provide a raw interface to the data link layer other than the BPF raw interface but use the BPF filtering mechanisms for that raw interface  Kernel-mode interpreters for that same virtual machine language are used in raw data link layer mechanisms in other operating systems, such as [Tru64 Unix](https://en.wikipedia.org/wiki/Tru64_Unix), and for socket filters in the [Linux kernel](https://en.wikipedia.org/wiki/Linux_kernel) and in the WinPcap packet capture mechanism. Since version 3.18, the Linux kernel includes an extended BPF virtual machine, termed extended BPF (eBPF). It can be used for non-networking purposes, such as for attaching eBPF programs to various [tracepoints](https://en.wikipedia.org/wiki/Tracepoint). Since kernel version 3.19, eBPF filters can be attached to [sockets](https://en.wikipedia.org/wiki/Network_socket) and, since kernel version 4.1, to [traffic control](https://en.wikipedia.org/wiki/Tc_(Linux)) classifiers for the ingress and egress networking data path.  This is treated in more detail later in this handbook. |
|  | Perf\_events | perf is powerful: it can instrument CPU performance counters, tracepoints, kprobes, and uprobes (dynamic tracing).  It is capable of lightweight profiling. It is also included in the Linux kernel, under tools/perf, and is frequently updated and enhanced.  perf began as a tool for using the performance counters subsystem in Linux, and has had various enhancements to add tracing capabilities.  Performance counters are CPU hardware registers that count hardware events such as instructions executed, cache-misses suffered, or branches mispredicted. They form a basis for profiling applications to trace dynamic control flow and identify hotspots. perf provides rich generalized abstractions over hardware specific capabilities. Among others, it provides per task, per CPU and per-workload counters, sampling on top of these and source code event annotation.  Tracepoints are instrumentation points placed at logical locations in code, such as for system calls, TCP/IP events, file system operations, etc. These have negligible overhead when not in use, and can be enabled by the perf command to collect information including timestamps and stack traces. perf can also dynamically create tracepoints using the kprobes and uprobes frameworks, for kernel and userspace dynamic tracing. The possibilities with these are endless.    See <https://perf.wiki.kernel.org/index.php/Main_Page>  And  <https://perf.wiki.kernel.org/index.php/Tutorial>  See appendix G for more information.  See also Appendix H for an easy to use set of tools called perf-tools to utilize perf-events |
|  | SystemTap | [SystemTap](https://sourceware.org/systemtap/wiki/SystemTap) provides free software (GPL) infrastructure to simplify the gathering of information about the running Linux system. This assists diagnosis of a performance or functional problem. [SystemTap](https://sourceware.org/systemtap/wiki/SystemTap) eliminates the need for the developer to go through the tedious and disruptive instrument, recompile, install, and reboot sequence that may be otherwise required to collect data.  [SystemTap](https://sourceware.org/systemtap/wiki/SystemTap) provides a simple command line interface and scripting language for writing instrumentation for a live running system. We are publishing samples, as well as enlarging the internal "tapset" script library to aid reuse and abstraction.  See <https://sourceware.org/systemtap/wiki> |
|  | linuxKI | A tool developed by Hewlett Packard and now available on github at  <https://github.com/HewlettPackard/LinuxKI>  It is generally only run for 20 seconds at a time and produces a wealth of data from investigation and interpretation.  It uses Linux kernel Instrumentation [ see <https://elinux.org/Kernel_Instrumentation> ] for pulling information from the kernel.  LinuxKI goes a little bit further by providing a very nice http map of the results.  This makes LinuxKI a very strong candidate for Linux performance Analysis tasks.  See also Appendix F for a detailed usage. |
|  | ktap | ktap is a scripting dynamic tracing tool for Linux, it uses a scripting language and lets users trace the Linux kernel dynamically. ktap is designed to give operational insights with interoperability that allows users to tune, troubleshoot and extend the kernel and applications. It's similar to Linux Systemtap and Solaris Dtrace  <https://github.com/ktap/ktap>  Development has been suspending whilst eBPF is evolved. |
|  | Dtrace4linux | A port of the Sun DTrace user and kernel code to Linux. No linux kernel code is touched in this build, but what is produced is a dynamically loadable kernel module  https://github.com/dtrace4linux/linux |
|  | LTTng | An open source tracing framework for Linux.  <https://lttng.org/>  Looks like a good tracer but needs a lot of knowledge ramp up and significant effort.  It claims to have little effect on the system! |
|  | sysdig | A tracer based on simple expressions.  https://github.com/draios/sysdig/wiki/Tracers |

# Conclusions

You can see from this handbook that performance is a very large topic and you can approach it at a very basic high level with a few simple commands or you can delve deeply into the operating system with some very complex tracers.

Hence performance anaylsis is a flexible service and you need to be flexible in your approach and use any and all tools which become apparently necessary.

This handbook will never be complete.

# Appendix A: A Question Based Methodology

|  |  |  |
| --- | --- | --- |
| **Area** | **Question?** | **Action** |
| **Application** |  |  |
| 1 | Is memory usage a problem? | Use top or “ps –eal” to determine how much memory the application is using  If the application is using more memory than it should go to 27. |
| 2 | Is startup time a problem? | If not then go to 4. |
| 3 | Is the loader introducing a delay? | Set the environment variables to investigate the loader.  env LD\_DEBUG=statistics LD\_DEBUG\_OUTPUT=filename <command>  If the loader appears to be the problem go to 41. |
| 4 | Is CPU usage a problem? | Use top or ps to determine the amount of CPU that the application uses.  If CPU appears to be a problem then go to 17. |
| 5 | Is the application’s disk usage a problem? | If the application is known to cause a large amount of disk I/O then go to 34. |
| 6 | Is the applications network usage a problem? | If the application is known to cause a large amount of network traffic then go to 40.  Otherwise you have reached a condition that this method cannot diagnose. Seek help another way!!!!! |
| **System** |  |  |
| 7 | Is the system CPU bound? | Use top, procinfo or mpstat and determine where the system is spending its time. If the entire system is spending less than 5% of the total time in idle and wait modes then the system is CPU-bound and proceed to 9. |
| 8 | Is a single processor CPU-bound? | Use top or mpstat to determine whether an individual CPU had less than 5% in idle and wait modes. In this case go to 10. |
| 9 | Are more than one processes using most of the system CPU? | The next step is to figure out whether any particular application is using the CPU.  Run top which by default sorts the processes in descending order. If the time used is significantly less than the system-wide system plus user time the kernel is doing significant work that is not on the behalf of applications. Go to section 11.  Otherwise go to section 17. |
| 10 | Are one or more processes using most of an individual CPU? | The next step is to figure out whether any particular application is using the individual CPU’s. Use top to investiage.  If the time used on applications is less than 25% go to section 11.  Otherwise the problem is likely to be the application and go to section 17 |
| 11 | Is the kernel servicing many interrupts? | The kernel appears to be doing work not part of an application. One explanation for this is an I/O card that is raising many interrupts such as a busy network card.  Run procinfo or “cat /proc/interrupts” to determine how many interrupts are being fired, how often they are fired and what devices are causing them. Proceed to the next section. |
| 12 | Where is the time spent in the kernel? | Now we need to know exactly what the kernel is doing. Run a kernel proofing tool such as “oprofile” and record those which are using more than 10% of the time.  Go to section 41. |
| 13 | Is the amount of swap being used increasing? | Use top, vmstat, procinfo to see if the swap space is increasing. You need to find out what part of the system is using more memory. If it is increasing go to section 25. |
| 14 | Is the system I/O bound? | Use top to see of the system is spending a high percentage of time in the wait state. If this is more than 50% then go to section 15.  Otherwise there is a problem this procedure cannot detect. |
| 15 | Is the system using Disk I/O? | Run vmstat or iostat and see how many blocks are being written to and from the disk. If this is large then the disk may a bottleneck and go to section 32. |
| 16 | Is the system using network I/O? | Use iptraf, ifconfig, sar and see how much data is being transferred on each network device. If the network traffic is near capacity then go to section 35. |
| **CPU** |  |  |
| 17 | Is the process spending time in user or kernel space? | Use the time command to determine whether an application is spending its time in kernel or user mode. Oprofile can also be used to determine where time is spent. If the application is not spending significant time in kernel space ( < 25% ) then go to section 19 |
| 18 | Which system calls is the process making and how long do they take to complete? | Run strace and see which system calls are made and how long they take to complete.  Some of the system calls may be unexpected and result in calls to various libraries. Run ltrace to determine why they are being made.  Now you have the information you need so go to section 41. |
| 19 | In which functions does the process spend time? | Run oprofile on the application using the cycle event to determine which functions are using all the CPU cycles. |
| 20 | What is the call tree to the hot functions? | Run the application with gprof to show the call tree for each function. If time consuming functions are in a library use ltrace to see which functions. Is it possible to eliminate calls to the hot functions with a re-compilation? If so go to section 41. |
| 21 | Do cache missies correspond to the hot functions or source lines? | Run oprofile, cachegrind and kcache to see whether the time-consuming functions or source lines are those with a high number of cache misses. If they are then try to re-arrange or compress data structures and accesses to make them more data friendly. If the hot lines do not correspond to high cache misses try to rearrange the algorithm to reduce the number of times that the particular line or function is executed.  Go to section 41. |
| **Memory** |  |  |
| 22 | Is the kernel memory usage increasing? | Run slabtop and see whether the total size of the kernel memory is increasing. If not then go to section 24. |
| 23 | What type of memory is the kernel using? | Run slabtop and see what type of memory the kernel is allocating. The name of the slab can give some indication about why that memory is being allocated.  You can try and tune the amount of maximum memory that the particular subsystem can consume.  Go to section 41. |
| 24 | Is a particular processes resident set size increasing? | Use top or ps to see whether a particular process resident size is increasing. It is easiest to add the “rss” field to top and sort by memory usage. If it is increasing you need to find out what type of memory it is using. Do this in section 27. |
| 25 | Is shared memory usage increasing? | Use ipcs to determine whether the amount of shared memory being used is increasing.  If not then you have a memory leak and you need to start using a tool such as valgrind to find your memory leaks.  Go to section 41. |
| 26 | Which processes are using the shared memory? | Use ipcs to determine which process are using shared memory. Determine why each process is using shared memory. That is is each invocation of shmget and shmat really required?  Try to reduce shared the use of shared memory.  Go to section 41. |
| 27 | What type of memory is the process using? | Use “cat /proc/<pid>/status to get a breakdown of the process memory usage. If the process has a large and increasing VmStk this means the process stack size is increasing.  If the process has a large VmExe that means that the executable size is big. If so go to section 29.  If the process has a large VmLib that means the process is using either a large number of shared libraries or a few large-sized shared libraries. If so then go to section 30.  If the process has a large VmData this means the process data area or heap is increasing. If so go to section 31. |
| 28 | What functions are using all of the stack? | You can use “gdb” to attach to the process and use it to find out which functions are taking up all of the stack space.  This is outside the scope of this handbook. Find a good developer!  Go to section 41. |
| 29 | What functions have the biggest text size? | If the executable has a sizeable amount of memory being used it may be useful to determine which functions are taking up the greatest amount of space and try and reduce it.  Use  # nm –S –size-sort “executable”  If you have a knowledge of each function you may be able to reduce the text size. |
| 30 | How big are the libraries that the process uses? | Show the size of the process code and data with  “cat /proc/<pid>/map”  See if you can eliminate the largest of the libraries if they are really not required!  If any other applications are using the library (determine this with lsof) the libraries will already be loaded into memory. So switching to a new library may actually increase memory!!!!  The best solution is to shrink the size of the library themselves. Section 29 may help in determine the size of the functions in the library.  Go to section 41. |
| 31 | What functions are allocating heap memory? | If you application is written in C or C++ you can find out which functions are allocating heap memory by using the memory profiler memprof. This can dynamically show how memory usage grows as the application is used. |
| **DISK I/O** |  |  |
| 32 | Is the system stressing a particular disk? | Run iostat and look for partitions that have an average wait (avwait) greater than zero. This is the average number of milliseconds that requests are waiting to be filled. The higher the number then the more you wait!  If many file are accessed on a single drive it may be possible to increase performance by spreading these files out to multiple disks. |
| 33 | What application is accessing disk? | Use top and look for the processes which are not idle. For each of these use section 34. |
| 34 | Which files are accessed by the applications? | Use strace to trace all system calls that an application is making to do with file I/O using “strace –e trace=file”  You can map the file descriptors back to files on the system by using “ls –la /proc/<pid>/fd  Use this information to help spread disk access more evenly over multiple disks.  Go to section 41. |
| **Network I/O** |  |  |
| 35 | Is any network device sending/receiving near the theoretical limit? | Use ethtool to determine the theoretical limit of each network card. Then use iptraf to determine if a card is nearing its limit.  If the limit is being reached then go to section 37. |
| 36 | Is any network device generating a large number of errors? | Use ipconfig to see if the any of the interfaces are generating a large number of errors. If so then get in details of the card and its proper settings. Maybe it is simply defect!  Go to section 41. |
| 37 | What type of traffic is running on that device? | Use iptraf to track down exactly what type of traffic is being sent and received. When you can characterize what you have you will better know what to do with it. |
| 38 | Is a particular process responsible for that traffic? | We would like to know if one or more particular processes is responsible for the network traffic. Use “netstat –p” to see whether any process is handling the type of traffic flowing over the port.  If an application is responsible go to section 40. |
| 39 | What remote system is sending the traffic? | Use iptraf or etherape to determine wich system is sending all this traffic.  If the traffic is unwanted then you can set up your own host based firewall using ipfilter or firewalld. (but not both).  You may well be the subject of a denial of service attack so speak with you network support services. |
|  |  |  |
| 40 | Which application socket is responsible for the traffic? | Determine which socket is being used is a two step process.  First use # strace –e trace=file  This shows which file descriptors the process is reading and writing from.  Second we map the file descriptors back to a socket by looking at  # ls –la /proc/<pid>/fd  Use this information to map which application is creating all the traffic. |
| 41 | When you have reached here the problem may or may not be solved but you will have a lot of information to characterize it. |  |

|  |
| --- |
|  |
|  |

# Appendix B: Notable Events

|  |  |  |
| --- | --- | --- |
|  | **Event** |  |
|  | There is a scenario known as the "thundering herd".  This occurs when many tasks are sleeping and waiting for the same resource. Depending on how the resource is handled in the kernel, either one task will be awakened when the resource is available or all waiting tasks will be awakened. If there are many sleeping tasks that are all woken up at the same time, you have a thundering herd of tasks contending for the resource. One task will get the resource and continue execution. The other tasks will be put back to sleep. This flurry of scheduling activity can impact system performance if it happens frequently. |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Appendix C: Collectl

There are a number of times in which you find yourself needing performance data. These can include benchmarking, monitoring a system's general heath or trying to determine what your system was doing at some time in the past. Sometimes you just want to know what the system is doing right now. Depending on what you're doing, you often end up using different tools, each designed to for that specific situation.

Unlike most monitoring tools that either focus on a small set of statistics, format their output in only one way, run either interatively or as a daemon but not both, collectl tries to do it all. You can choose to monitor any of a broad set of subsystems which currently include buddyinfo, cpu, disk, inodes, infiniband, lustre, memory, network, nfs, processes, quadrics, slabs, sockets and tcp.

The major site for collect is here <http://collectl.sourceforge.net/index.html>

but it is built directly into many vendor repositories and can hence be considered stable and verified software in the mainstream of use.

On rpm based system is can be installed as follows:

# yum install collectl

# service collectl start

# systemctl enable collectl

Change /etc/collectl.conf

DaemonCommands = -f /var/log/collectl -r00:00,7 –i10:60:300 -m -F60 -s+cCdDmYZ -P –ocz

# service collectl restart

Check its running

# ps –ef | grep collectl

The files in /var/log/collect should be tar’d up and set for analysis

cd /var/log/collect

tar –czvf /tmp/`hostname.tar.gz`\*

**COLPLOT**

Colplot is a small program which can process and view of the plot able files directly.

If you are collecting for colplot processing only use:

It needs a web server installed.

# yum install httpd

# service httpd start

# systemctl enable httpd

Get colplot from <https://sourceforge.net/projects/collectl-utils/files/latest/download?source=directory>

Unpackit and run INSTALL

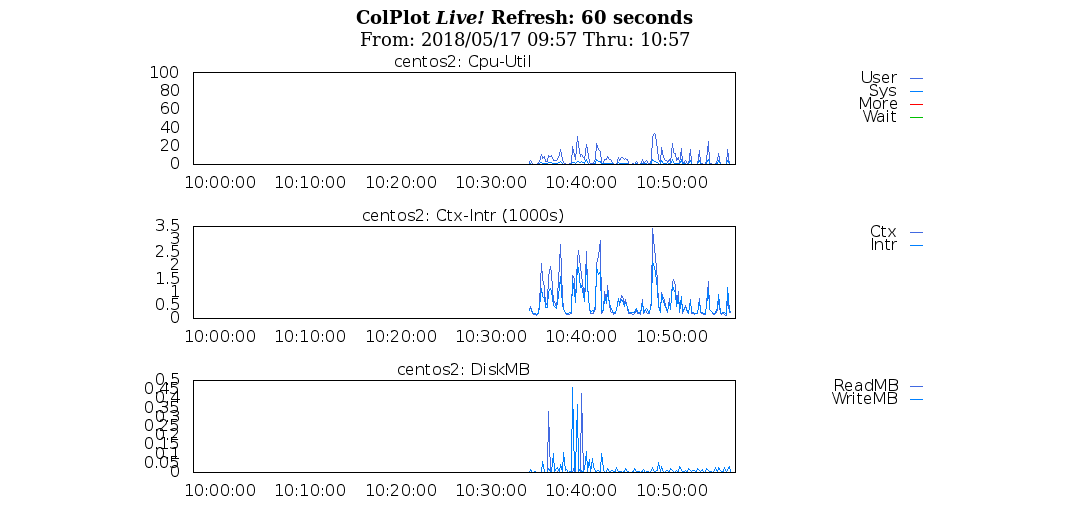
Check /etc/httpd/conf.d/colplot-apache.conf and ensure:

Alias /colplot "/var/www/html/colplot"

Colplot should now be available on <http://127.0.0.1/colplot>

By default it points at the default place /var/log/collect

It uses gnuplot to create graphs.



# Appendix D: ftrace

Ftrace is the Linux kernel internal tracer that was included in the Linux kernel in 2.6.27. Although Ftrace is named after the function tracer it also includes many more functionalities. But the function tracer is the part of Ftrace that makes it unique as you can trace almost any function in the kernel and with dynamic Ftrace, it has no overhead when not enabled.

See <https://elinux.org/Ftrace>

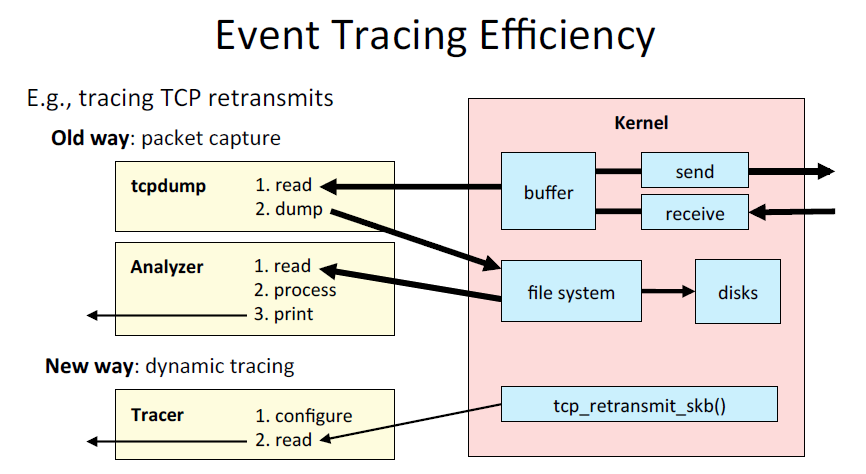
Using the Ftrace debugfs interface can be awkward and time consuming.

It is recommended that you use the perf-tools documented in Appendix H.

# Appendix E: eBPF

eBPF is the latest and possible best of all the tracers. It strongest point is that it runs in a kernel container and does not insert a kernel device driver. Hence we should consider this a safe method of tracing which is also very lightweight in comparison to other tracers such as “strace”.

Consider the old and new ways to doing “tcpdump” as follows:



In fact this use case is why BPF was originally conceived. Now its use has been extended to many more utilities which give us a lightweight and programmable access to the kernel.

Unfortunately a Linux kernel of version 4.1 is required. This puts an unfortunate restriction on us that it cannot be used for any RedHat or Centos based kernel at the present time.

In addition, the kernel should have been compiled with the following flags set:

CONFIG\_BPF=y

CONFIG\_BPF\_SYSCALL=y

# [optional, for tc filters]

CONFIG\_NET\_CLS\_BPF=m

# [optional, for tc actions]

CONFIG\_NET\_ACT\_BPF=m

CONFIG\_BPF\_JIT=y

CONFIG\_HAVE\_BPF\_JIT=y

# [optional, for kprobes]

CONFIG\_BPF\_EVENTS=y

It can be fetched from

<https://github.com/iovisor/bcc/blob/master/INSTALL.md>

Additionally if can be installed on debian based systems using

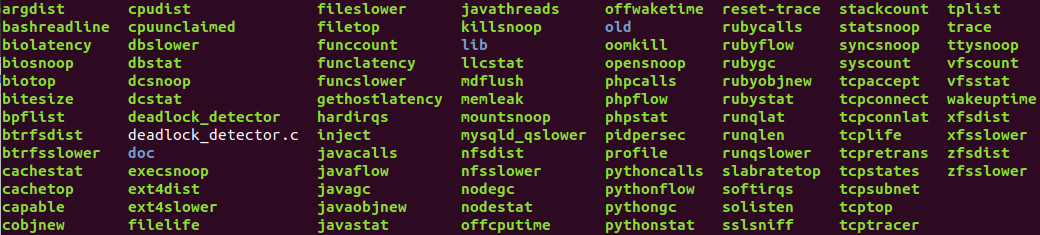
# echo "deb [trusted=yes] https://repo.iovisor.org/apt/xenial xenial-nightly main" |\

sudo tee /etc/apt/sources.list.d/iovisor.list

# sudo apt-get update

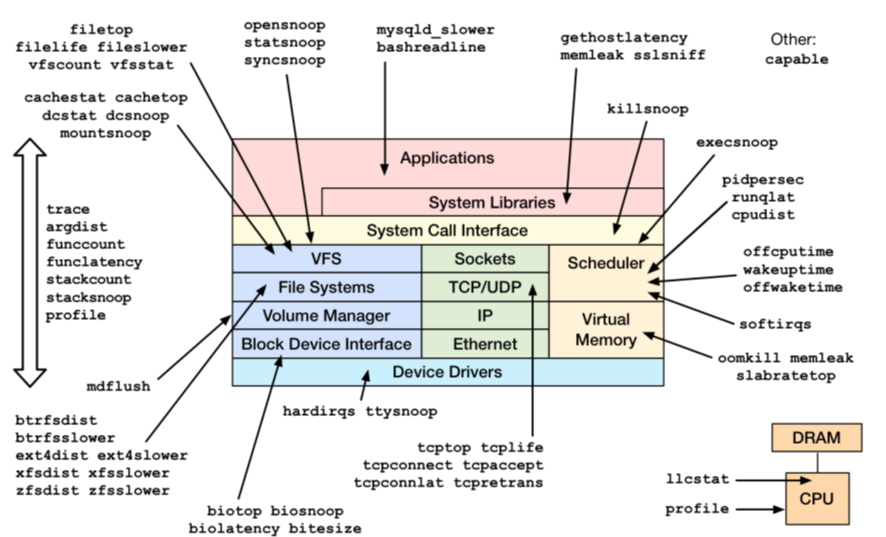
# sudo apt-get install bcc-tools

Tools are installed in /usr/share/bcc/tools



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | |  | execsnoop | Trace new process and debug short-lived process issues | |  | opensnoop | Find config, log and data files and inefficient file usage | |  | ext4slower | Trace slow FS I/O to better identify I/O issues and outliers | |  | biolatency | Identify multimodal latency and outliers with a histogram | |  | biosnoop | Dump disk I/O events for detailed analysis. Tcpdump for disks. | |  | cachestat | Measure file system cache hit ratio statistics | |  | tcpconnect | Trace active - outbound TCP connections | |  | tcpaccept | Trace passive – inbound TCP connections | |  | tcpretrans | Trace TCP retransmits with kernel state | |  | gethostlatency | Trace DNS latency system-wide | |  | runqlat | Examine CPU scheduler run queue latency | |  | profile | An efficient profiler. Count unique stacks in kernel | |  |  |  | |  |  |

The tools can be characterized as profiling as follows:



# Appendix F: LinuxKI

Overview

This is a tool developed by Hewlett Packard. It is used by the Americas Performance center. For many years it was kept confidential but they have now released it on github. I have talked with Mary Ray one of the developers and he stated that they would have a better understanding from customers by doing this.

Building

We can find it at

<https://github.com/HewlettPackard/LinuxKI>

Download using

# git clone <https://github.com/HewlettPackard/LinuxKI.git>

There are no mandatory pre-requisites. LinuxKI should install and run on most Linux systems from 2.6.32 through 4.12.8.

However, if you would like to use the LiKI tracing mechanism (preferred method), you will need the following packages installed to compile the LiKI module from source code:

* kernel-devel
* kernel-headers
* gcc
* make

Download and Installation

Prepackaged RPM and DEB files are available from

<https://github.com/HewlettPackard/LinuxKI/raw/master/rpms/linuxki-5.4-1.noarch.rpm>

<https://github.com/HewlettPackard/LinuxKI/raw/master/rpms/linuxki_5.4-1_all.deb>

The LinuxKI Toolset is provided in an RPM Package and can be installed as follows:

# rpm --install --nodeps linuxki.<version>.noarch.rpm

Or for Debian-related kernels, the toolset can be installed using the dpkg command:

# dpkg --install linuxki.<version>\_all.deb

Data Collection

When the system is experiencing performance problems, the runki script can be executed to collect data. By default 20 seconds of trace data will be collected, and then runki will spend some time gathering other configuration and supplemental data, and then bundle this into a single gzip archive. It might take several minutes in all to complete. Root/superuser privilege is required to collect the trace data. The data is stored in the current working directory, and may require several hundred megabytes or gigabytes of space per collection run, depending on the size of the system and amount of trace data generated. The filesystem on which data is stored should be enabled to use the filesystem cache; directIO is not recommended. If sufficient memory is available, the current working directory can be changed to /dev/shm and the runki script can collect the data in-memory and then copied to persistent storage later.

After installing the LinuxKI Toolset, a 20-second trace dump can easily be obtained as follows:

$ export PATH=$PATH:/opt/linuxki

$ /dev/shm # optional, to collect data in memory

$ runki # use LiKI tracing mechanism

or

$ runki -f # use ftrace tracing mechanism

When the data collection is complete, you will see a message similar to the following...

=== Trace completed and archived as ki\_all.localhost.0613\_1310.tgz

Trace Analysis

After data collection, a default set of reports can be generated as follows:

$ kiall -r # -r option creates a nodename/timestamp directory structure

This will produce a directory tree starting with the hostname. Point a browser at the single and only html file for an easy start point to an analysis.



This is extremely powerful and takes much of the hard work out of understanding what the system is doing.

# Appendix G: Perf\_events

Perf is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple command line interface. Perf is based on the perf\_events interface exported by recent versions of the Linux kernel.

Install with

# yum install linux-common-tools

For great information see:

<https://perf.wiki.kernel.org/index.php/Tutorial>

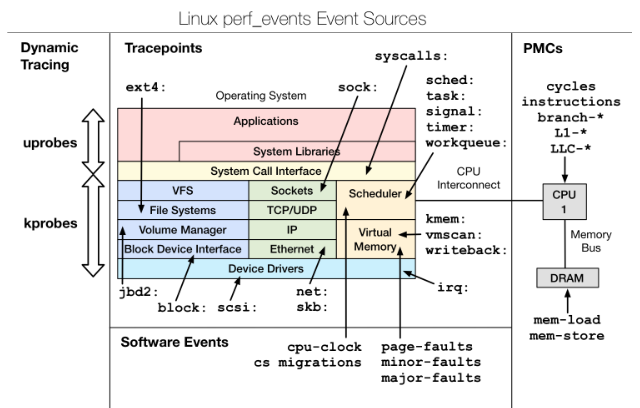
<http://www.brendangregg.com/perf.html>

perf\_events is an event-oriented observability tool, which can help you solve advanced performance and troubleshooting functions. Questions that can be answered include:

* Why is the kernel on-CPU so much? What code-paths?
* Which code-paths are causing CPU level 2 cache misses?
* Are the CPUs stalled on memory I/O?
* Which code-paths are allocating memory, and how much?
* What is triggering TCP retransmits?
* Is a certain kernel function being called, and how often?
* What reasons are threads leaving the CPU?

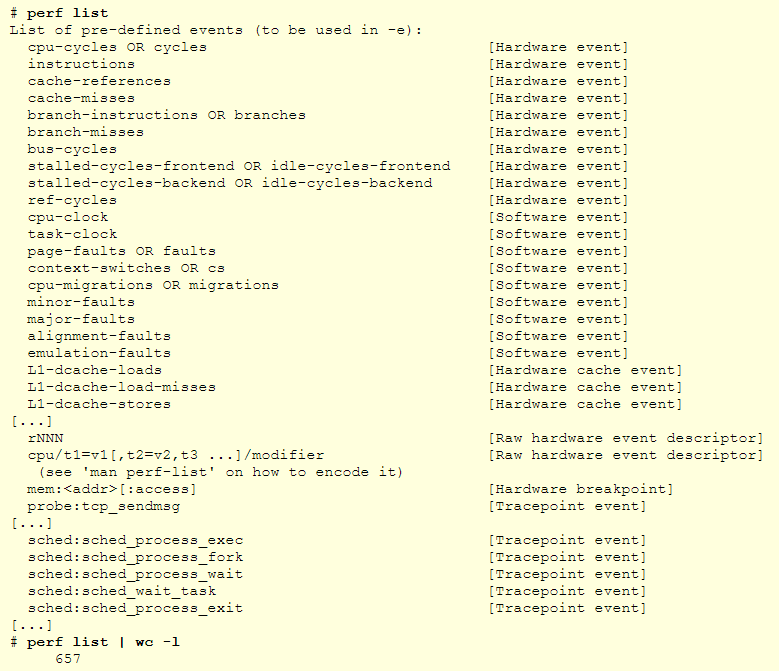
The types of events are:

* **Hardware Events**: CPU performance monitoring counters.
* **Software Events**: These are low level events based on kernel counters. For example, CPU migrations, minor faults, major faults, etc.
* **Kernel Tracepoint Events**: These are static kernel-level instrumentation points that are hardcoded in interesting and logical places in the kernel.
* **User Statically-Defined Tracing (USDT)**: These are static tracepoints for user-level programs and applications.
* **Dynamic Tracing**: Software can be dynamically instrumented, creating events in any location. For kernel software, this uses the kprobes framework. For user-level software, uprobes.
* **Timed Profiling**: Snapshots can be collected at an arbitrary frequency, using perf record -F*Hz*. This is commonly used for CPU usage profiling, and works by creating custom timed interrupt events.



Details about the events can be collected, including timestamps, the code path that led to it, and other specific details. The capabilities of perf\_events are enormous, and you're likely to only ever use a fraction.

Currently available events can be listed using the list subcommand:



|  |  |  |  |
| --- | --- | --- | --- |
|  | **Events** | **Command(s)** |  |
|  | **EXAMPLE TRACING DISK** |  |  |
|  | Tracing Disk I/O | perf record -e block:block\_rq\_issue –ag  CTRL C  perf report |  |
|  | **LISTING EVENTS** |  |  |
|  | List all known events | “perf list”  Current this give 1360 lines of output! |  |
|  | **COUNTING EVENTS** |  |  |
|  | CPU counter statistics | perf stat *command* |  |
|  | Detailed CPU counter statistics | perf stat –d *command* |  |
|  | CPU counter statistics for the specified PID, until Ctrl-C | perf stat –p *PID* |  |
|  | CPU counter statistics for the entire system, for 5 seconds | perf stat -a sleep 5 |  |
|  | Various basic CPU statistics, system wide, for 10 seconds | perf stat -e cycles,instructions,cache-references,cache-misses,bus-cycles -a sleep 10 |  |
|  | Various CPU level 1 data cache statistics for the specified command | perf stat -e L1-dcache-loads,L1-dcache-load-misses,L1-dcache-stores *command* |  |
|  | Various CPU data TLB statistics for the specified command | perf stat -e dTLB-loads,dTLB-load-misses,dTLB-prefetch-misses *command* |  |
|  | Various CPU last level cache statistics for the specified command | perf stat -e LLC-loads,LLC-load-misses,LLC-stores,LLC-prefetches *command* |  |
|  | Using raw PMC counters, eg, unhalted core cycles | perf stat -e r003c -a sleep 5 |  |
|  | PMCs: cycles and frontend stalls via raw specification: | perf stat -e cycles -e cpu/event=0x0e,umask=0x01,inv,cmask=0x01/ -a sleep 5 |  |
|  | Count syscalls per-second system-wide (this should be in /proc): | perf stat -e raw\_syscalls:sys\_enter -I 1000 -a |  |
|  | Count system calls by type for the specified PID, until Ctrl-C: | perf stat -e 'syscalls:sys\_enter\_\*' -p *PID* |  |
|  | Count system calls by type for the entire system, for 5 seconds: | perf stat -e 'syscalls:sys\_enter\_\*' -a sleep 5 |  |
|  | Count scheduler events for the specified PID, until Ctrl-C | perf stat -e 'sched:\*' -p *PID* |  |
|  | Count scheduler events for the specified PID, for 10 seconds | perf stat -e 'sched:\*' -p *PID* sleep 10 |  |
|  | Count ext4 events for the entire system, for 10 seconds: | perf stat -e 'ext4:\*' -a sleep 10 |  |
|  | Count block device I/O events for the entire system, for 10 seconds: | perf stat -e 'block:\*' -a sleep 10 |  |
|  | ount all vmscan events, printing a report every second | perf stat -e 'vmscan:\*' -a -I 1000 |  |
|  | Show system calls by process, refreshing every 2 seconds | perf top -e raw\_syscalls:sys\_enter -ns comm |  |
|  | Show sent network packets by on-CPU process, rolling output (no clear): | stdbuf -oL perf top -e net:net\_dev\_xmit -ns comm | strings |  |
|  | **PROFILING** |  |  |
|  | Sample on-CPU functions for the specified command, at 99 Hertz: | perf record -F 99 *command* |  |
|  | Sample on-CPU functions for the specified PID, at 99 Hertz, until Ctrl-C: | perf record -F 99 -p *PID* |  |
|  | Sample on-CPU functions for the specified PID, at 99 Hertz, for 10 seconds: | perf record -F 99 -p *PID* sleep 10 |  |
|  | Sample CPU stack traces (via frame pointers) for the specified PID, at 99 Hertz, for 10 seconds: | perf record -F 99 -p *PID* -g -- sleep 10 |  |
|  | Sample CPU stack traces for the PID, using dwarf (dbg info) to unwind stacks, at 99 Hertz, for 10 seconds: | perf record -F 99 -p *PID* --call-graph dwarf sleep 10 |  |
|  | Sample CPU stack traces for the entire system, at 99 Hertz, for 10 seconds (< Linux 4.11): | perf record -F 99 -ag -- sleep 10 |  |
|  | Sample CPU stack traces for the entire system, at 99 Hertz, for 10 seconds (>= Linux 4.11): | perf record -F 99 -g -- sleep 10 |  |
|  | If the previous command didn't work, try forcing perf to use the cpu-clock event: | perf record -F 99 -e cpu-clock -ag -- sleep 10 |  |
|  | Sample CPU stack traces for a container identified by its /sys/fs/cgroup/perf\_event cgroup: | perf record -F 99 -e cpu-clock --cgroup=docker/1d567f4393190204*...etc...* -a -- sleep 10 |  |
|  | Sample CPU stack traces for the entire system, with dwarf stacks, at 99 Hertz, for 10 seconds: | perf record -F 99 -a --call-graph dwarf sleep 10 |  |
|  | Sample CPU stack traces for the entire system, using last branch record for stacks, ... (>= Linux 4.?): | perf record -F 99 -a --call-graph lbr sleep 10 |  |
|  | Sample CPU stack traces, once every 10,000 Level 1 data cache misses, for 5 seconds: | perf record -e L1-dcache-load-misses -c 10000 -ag -- sleep 5 |  |
|  | Sample CPU stack traces, once every 100 last level cache misses, for 5 seconds: | perf record -e LLC-load-misses -c 100 -ag -- sleep 5 |  |
|  | Sample on-CPU kernel instructions, for 5 seconds | perf record -e cycles:k -a -- sleep 5 |  |
|  | Sample on-CPU user instructions, for 5 seconds: | perf record -e cycles:u -a -- sleep 5 |  |
|  | Sample on-CPU user instructions precisely (using PEBS), for 5 seconds: | perf record -e cycles:up -a -- sleep 5 |  |
|  | Perform branch tracing (needs HW support), for 1 second: | perf record -b -a sleep 1 |  |
|  | Sample CPUs at 49 Hertz, and show top addresses and symbols, live (no perf.data file): | perf top -F 49 |  |
|  | Sample CPUs at 49 Hertz, and show top process names and segments, live: | perf top -F 49 -ns comm,dso |  |
|  | **STATIC TRACING** |  |  |
|  | Trace new processes, until Ctrl-C: | perf record -e sched:sched\_process\_exec -a |  |
|  | Trace all context-switches, until Ctrl-C: | perf record -e context-switches -a |  |
|  | Trace context-switches via sched tracepoint, until Ctrl-C: | perf record -e sched:sched\_switch -a |  |
|  | Trace all context-switches with stack traces, until Ctrl-C: | perf record -e context-switches -ag |  |
|  | Trace all context-switches with stack traces, for 10 seconds: | perf record -e context-switches -ag -- sleep 10 |  |
|  | Trace all CS, stack traces, and with timestamps (< Linux 3.17, -T now default): | perf record -e context-switches -ag -T |  |
|  | Trace CPU migrations, for 10 seconds: | perf record -e migrations -a -- sleep 10 |  |
|  | Trace all connect()s with stack traces (outbound connections), until Ctrl-C: | perf record -e syscalls:sys\_enter\_connect -ag |  |
|  | Trace all accepts()s with stack traces (inbound connections), until Ctrl-C: | perf record -e syscalls:sys\_enter\_accept\* -ag |  |
|  | Trace all block device (disk I/O) requests with stack traces, until Ctrl-C: | perf record -e block:block\_rq\_insert -ag |  |
|  | Trace all block device issues and completions (has timestamps), until Ctrl-C: | perf record -e block:block\_rq\_issue -e block:block\_rq\_complete -a |  |
|  | Trace all block completions, of size at least 100 Kbytes, until Ctrl-C: | perf record -e block:block\_rq\_complete --filter 'nr\_sector > 200' |  |
|  | Trace all block completions, synchronous writes only, until Ctrl-C: | perf record -e block:block\_rq\_complete --filter 'rwbs == "WS"' |  |
|  | Trace all block completions, all types of writes, until Ctrl-C: | perf record -e block:block\_rq\_complete --filter 'rwbs ~ "\*W\*"' |  |
|  | Trace all minor faults (RSS growth) with stack traces, until Ctrl-C: | perf record -e minor-faults -ag |  |
|  | Trace all page faults with stack traces, until Ctrl-C: | perf record -e page-faults -ag |  |
|  | Trace all ext4 calls, and write to a non-ext4 location, until Ctrl-C: | perf record -e 'ext4:\*' -o /tmp/perf.data -a |  |
|  | Trace kswapd wakeup events, until Ctrl-C: | perf record -e vmscan:mm\_vmscan\_wakeup\_kswapd -ag |  |
|  | Add Node.js USDT probes (Linux 4.10+): | perf buildid-cache --add `which node` |  |
|  | Trace the node http\_\_server\_\_request USDT event (Linux 4.10+): | perf record -e sdt\_node:http\_\_server\_\_request -a |  |
|  | **DYNAMIC TRACING** |  |  |
|  | Add a tracepoint for the kernel tcp\_sendmsg() function entry ("--add" is optional): | perf probe --add tcp\_sendmsg |  |
|  | Remove the tcp\_sendmsg() tracepoint (or use "--del"): | perf probe -d tcp\_sendmsg |  |
|  | Add a tracepoint for the kernel tcp\_sendmsg() function return: | perf probe 'tcp\_sendmsg%return' |  |
|  | Show available variables for the kernel tcp\_sendmsg() function (needs debuginfo): | perf probe -V tcp\_sendmsg |  |
|  | Show available variables for the kernel tcp\_sendmsg() function, plus external vars (needs debuginfo): | perf probe -V tcp\_sendmsg --externs |  |
|  | Show available line probes for tcp\_sendmsg() (needs debuginfo): | perf probe -L tcp\_sendmsg |  |
|  | Show available variables for tcp\_sendmsg() at line number 81 (needs debuginfo): | perf probe -V tcp\_sendmsg:81 |  |
|  | Add a tracepoint for tcp\_sendmsg(), with three entry argument registers (platform specific): | perf probe 'tcp\_sendmsg %ax %dx %cx' |  |
|  | Add a tracepoint for tcp\_sendmsg(), with an alias ("bytes") for the %cx register (platform specific): | perf probe 'tcp\_sendmsg bytes=%cx' |  |
|  | Trace previously created probe when the bytes (alias) variable is greater than 100: | perf record -e probe:tcp\_sendmsg --filter 'bytes > 100' |  |
|  | Add a tracepoint for tcp\_sendmsg() return, and capture the return value: | perf probe 'tcp\_sendmsg%return $retval' |  |
|  | Add a tracepoint for tcp\_sendmsg(), and "size" entry argument (reliable, but needs debuginfo): | perf probe 'tcp\_sendmsg size' |  |
|  | Add a tracepoint for tcp\_sendmsg(), with size and socket state (needs debuginfo): | perf probe 'tcp\_sendmsg size sk->\_\_sk\_common.skc\_state' |  |
|  | Tell me how on Earth you would do this, but don't actually do it (needs debuginfo): | perf probe -nv 'tcp\_sendmsg size sk->\_\_sk\_common.skc\_state' |  |
|  | Trace previous probe when size is non-zero, and state is not TCP\_ESTABLISHED(1) (needs debuginfo): | perf record -e probe:tcp\_sendmsg --filter 'size > 0 && skc\_state != 1' -a |  |
|  | Add a tracepoint for tcp\_sendmsg() line 81 with local variable seglen (needs debuginfo): | perf probe 'tcp\_sendmsg:81 seglen' |  |
|  | Add a tracepoint for do\_sys\_open() with the filename as a string (needs debuginfo): | perf probe 'do\_sys\_open filename:string' |  |
|  | Add a tracepoint for myfunc() return, and include the retval as a string: | perf probe 'myfunc%return +0($retval):string' |  |
|  | Add a tracepoint for the user-level malloc() function from libc: | perf probe -x /lib64/libc.so.6 malloc |  |
|  | Add a tracepoint for this user-level static probe (USDT, aka SDT event): | perf probe -x /usr/lib64/libpthread-2.24.so %sdt\_libpthread:mutex\_entry |  |
|  | List currently available dynamic probes: | perf probe -l |  |
|  | **MIXED** |  |  |
|  | Sample stacks at 99 Hertz, and, context switches: | perf record -F99 -e cpu-clock -e cs -a -g |  |
|  | Sample stacks to 2 levels deep, and, context switch stacks to 5 levels (needs 4.8): | perf record -F99 -e cpu-clock/max-stack=2/ -e cs/max-stack=5/ -a -g |  |
|  | **SPECIAL** |  |  |
|  | Record cacheline events (Linux 4.10+): | perf c2c record -a -- sleep 10 |  |
|  | Report cacheline events from previous recording (Linux 4.10+): | perf c2c report |  |
|  | **REPORTING** |  |  |
|  | Show perf.data in an ncurses browser (TUI) if possible: | perf report |  |
|  | Show perf.data with a column for sample count: | perf report -n |  |
|  | Show perf.data as a text report, with data coalesced and percentages: | perf report --stdio |  |
|  | Report, with stacks in folded format: one line per stack (needs 4.4): | perf report --stdio -n -g folded |  |
|  | List all events from perf.data: | perf script |  |
|  | List all perf.data events, with data header (newer kernels; was previously default): | perf script --header |  |
|  | List all perf.data events, with customized fields (< Linux 4.1): | perf script -f time,event,trace |  |
|  | List all perf.data events, with customized fields (>= Linux 4.1): | perf script -F time,event,trace |  |
|  | List all perf.data events, with my recommended fields (needs record -a; newer kernels): | perf script --header -F comm,pid,tid,cpu,time,event,ip,sym,dso |  |
|  | List all perf.data events, with my recommended fields (needs record -a; older kernels): | perf script -f comm,pid,tid,cpu,time,event,ip,sym,dso |  |
|  | Dump raw contents from perf.data as hex (for debugging): | perf script -D |  |
|  | Disassemble and annotate instructions with percentages (needs some debuginfo): | perf annotate --stdio |  |
|  |  |  |  |

# Appendix H: perf\_tools

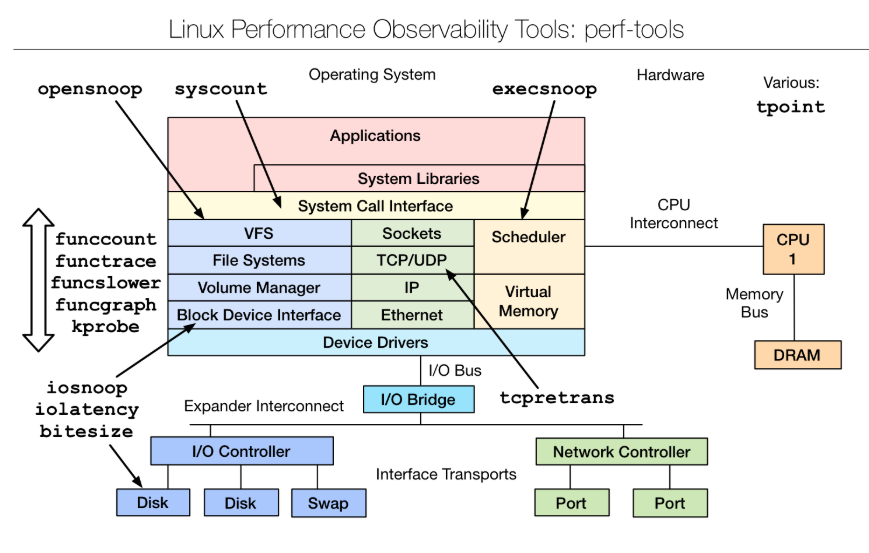
Perf-tools is a set of scripts written by Brendan Gregg and takes the hard work out of using ftrace and perf\_events.

See <https://github.com/brendangregg/perf-tools>

To download use:

# git clone <https://github.com/brendangregg/perf-tools.git>

It provides a number of tools as follows:



Using ftrace:

* [iosnoop](https://github.com/brendangregg/perf-tools/blob/master/iosnoop): trace disk I/O with details including latency.
* [iolatency](https://github.com/brendangregg/perf-tools/blob/master/iolatency): summarize disk I/O latency as a histogram.
* [execsnoop](https://github.com/brendangregg/perf-tools/blob/master/execsnoop): trace process exec() with command line argument details.
* [opensnoop](https://github.com/brendangregg/perf-tools/blob/master/opensnoop): trace open() syscalls showing filenames.
* [killsnoop](https://github.com/brendangregg/perf-tools/blob/master/killsnoop): trace kill() signals showing process and signal details.
* fs/[cachestat](https://github.com/brendangregg/perf-tools/blob/master/fs/cachestat): basic cache hit/miss statistics for the Linux page cache.
* net/[tcpretrans](https://github.com/brendangregg/perf-tools/blob/master/net/tcpretrans): show TCP retransmits, with address and other details.
* system/[tpoint](https://github.com/brendangregg/perf-tools/blob/master/system/tpoint): trace a given tracepoint.
* kernel/[funccount](https://github.com/brendangregg/perf-tools/blob/master/kernel/funccount): count kernel function calls, matching a string with wildcards.
* kernel/[functrace](https://github.com/brendangregg/perf-tools/blob/master/kernel/functrace): trace kernel function calls, matching a string with wildcards.
* kernel/[funcslower](https://github.com/brendangregg/perf-tools/blob/master/kernel/funcslower): trace kernel functions slower than a threshold.
* kernel/[funcgraph](https://github.com/brendangregg/perf-tools/blob/master/kernel/funcgraph): trace a graph of kernel function calls, showing children and times.
* kernel/[kprobe](https://github.com/brendangregg/perf-tools/blob/master/kernel/kprobe): dynamically trace a kernel function call or its return, with variables.
* user/[uprobe](https://github.com/brendangregg/perf-tools/blob/master/user/uprobe): dynamically trace a user-level function call or its return, with variables.
* tools/[reset-ftrace](https://github.com/brendangregg/perf-tools/blob/master/tools/reset-ftrace): reset ftrace state if needed.

Using perf\_events:

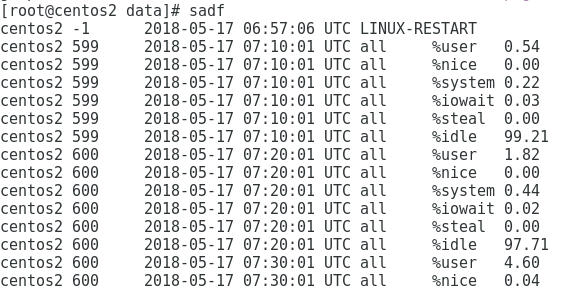
* misc/[perf-stat-hist](https://github.com/brendangregg/perf-tools/blob/master/misc/perf-stat-hist): power-of aggregations for tracepoint variables.
* [syscount](https://github.com/brendangregg/perf-tools/blob/master/syscount): count syscalls by syscall or process.
* disk/[bitesize](https://github.com/brendangregg/perf-tools/blob/master/disk/bitesize): histogram summary of disk I/O size.

# Appendix I: sar

The package “systat” which provides numerious utilities such as “sar” is generally installed on Linux servers. Data is gather and stored in /var/log/sa.

## sadf and gnuplot

In addition to sar, the [sysstat](http://sebastien.godard.pagesperso-orange.fr/) package also has a utility called sadf to display collected data.  Simply running sadf without any arguments will show you the recently aggregated processor utilization data resulting in:



However you can format the data ready for usage in other applications using “sadf”

sadf -d > sar.dat

Then create a file called gnuplot.txt with the following contents:

datafile = "sar.dat"

set datafile commentschar ""

set title "Processor Utilization"

set xdata time

set timefmt "%Y-%m-%d %H:%M:%S"

set datafile separator ";"

set terminal pngcairo size 800,500

set output 'sar.png'

set xlabel "Time"

set ylabel "% Utilization"

set xtics rotate

set key below

set grid

plot for [i=5:8:1] \

datafile using 3:(sum [col=i:8] column(col)) \

title columnheader(i) \

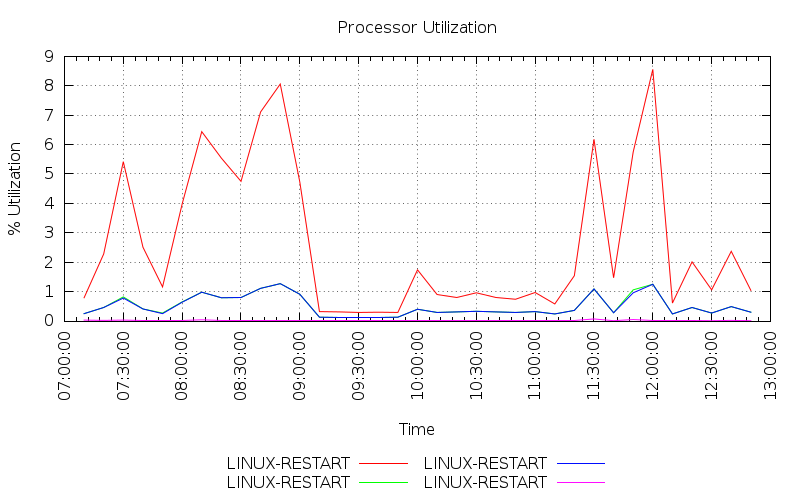
with lines

Now pass it to gnuplot as follows:

# cat gnuplot.txt | gnuplot

Now view the resulting file sar.png

# xdg-open sar.png



## kSar

Whilst this looks quite nice there is a better way and this is to use kSar. Get it from

<https://sourceforge.net/projects/ksar/?source=typ_redirect>

Since kSar is a Java application, you'll need to have Java installed -- preferably Sun Java. Unzip the download package and run it with:

# bash run.sh

After kSar starts up, you can start graphing sar results. If you have sar running on your local machine, you can do a quick test with kSar by going to its File menu and selecting Run local command. The default is sar -A, but any valid sar command should work just fine. For instance, if you'd like to see your system performance in almost real time, you could choose sar -A 1 120 to display all available system data with an update every second for 120 seconds; the first number after sar's other arguments is the interval -- in this case one second -- and the second number is the number of times sar should update before exiting.

Another way to go is to view past data. On Ubuntu and Debian systems, sar stores its data in /var/log/sysstat with the data for individual days stored in the format saXX, where XX is the day of the month. For example, if you want to see system data from the 15th, you could tell kSar to parse the file for the 15th:

sar -A -f /var/log/sysstat/sa15

The -f option tells sar to read from a specified file. You can run that as a local command, or if you like you can save sar data as a text file that can be read by kSar. To do this, you'll need to export one variable first

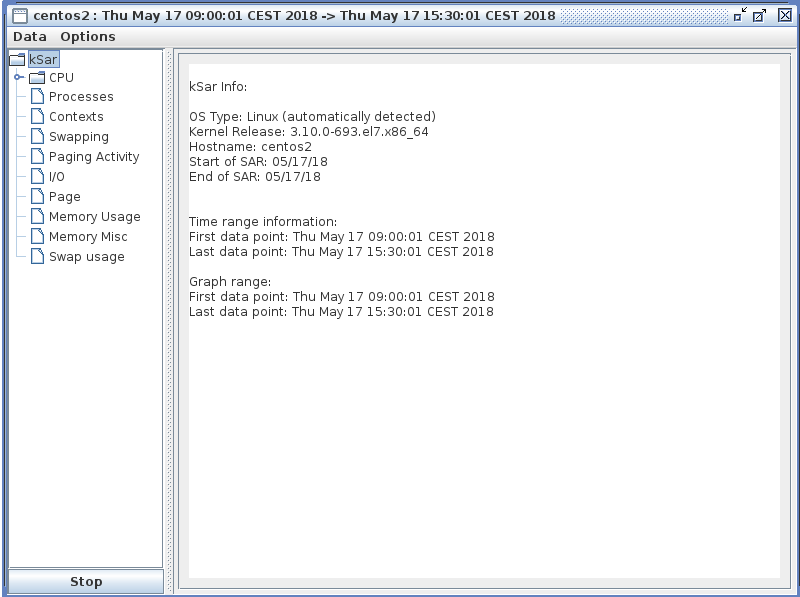
export LC\_ALL=C

Then, run sar -A -f /var/log/sysstat/sa15 > sardata.txt

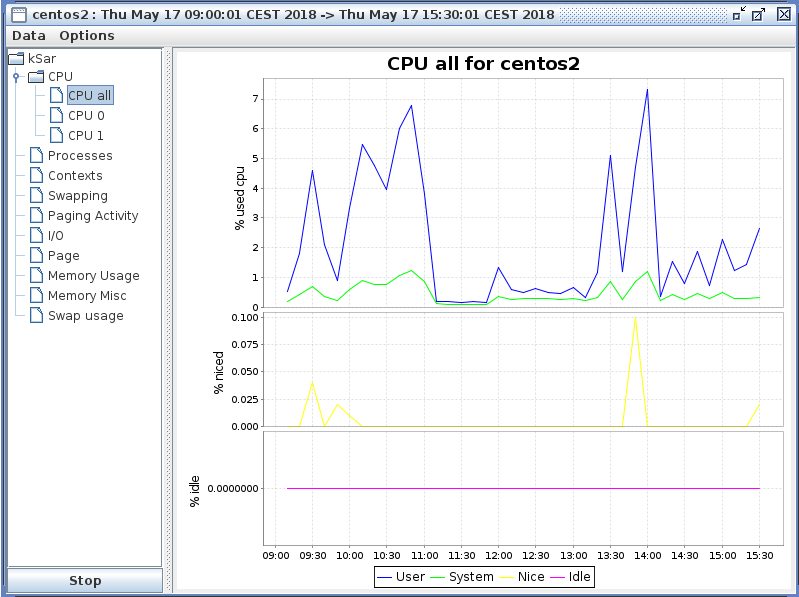
To load the data in kSar, go to File -> Load from text file... and select the sardata.txt file.

Outputs from kSar look as follows:

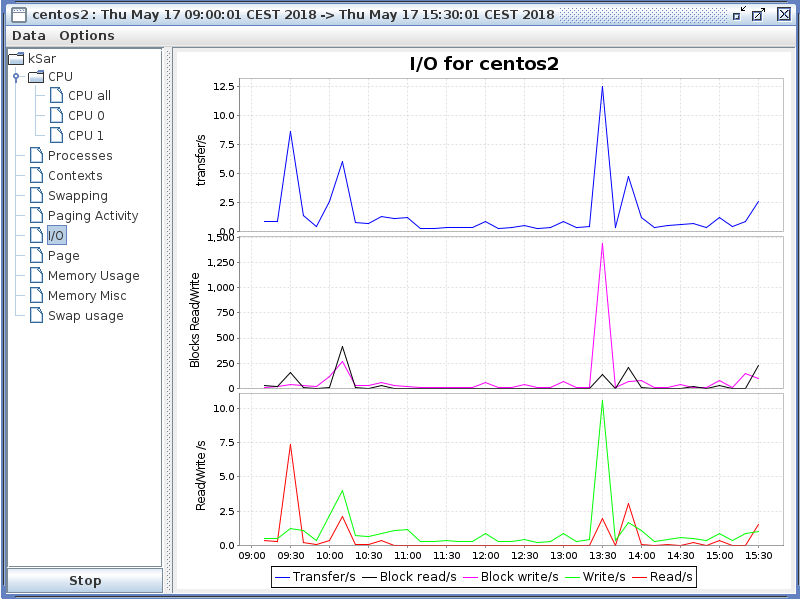
Available screen metrics:



CPU metrics:



I/O Metrics:



# Change log

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description of Changes** | **Initials** |
| 2018-05-23 | 1.0 | Initial Release | MDHT |