Linux CPU Usage Statistics

Conceptions and Misconceptions

Author: Sawan Kumar

Abstract

It is customary to refer to “top” (and related utilities) during debugging or more generally to monitor the behavior of Linux applications. “Top” provides a bunch of per-process and global information including CPU usage and CPU load averages. It is important first to identify the ideal nature of these software metrics and then whether the implementation does justice to the idea. This paper discusses both the ideal nature and the implementation of the metrics followed by the sources of error. The goal is to develop an intuitive understanding of these numbers while not missing out on the factors which affect the calculation of the metrics. Such an understanding requires a knowledge of how process scheduling is carried out by the Linux kernel. A discussion on the same is also added in this paper and serves as a prerequisite for the other sections.

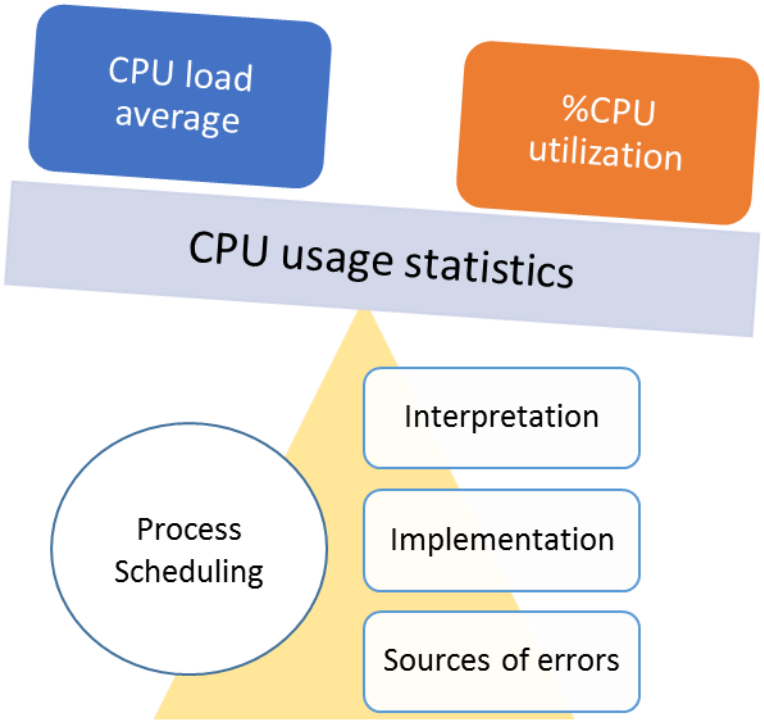

Figure 1 - Overview

Contents

Figures

Introduction

The goal of this paper is to develop an intuitive understanding of CPU usage information and CPU load averages provided by Linux considering not only the ideal nature of these metrics but also the implementation specific details and the errors introduced therein.

"CPU usage [%CPU utilization]" and "CPU load average metrics" are distinguished below.

- **%CPU utilization** provides the percentage of the CPU resource used by different processes including the idle process.
- **CPU load average** gives an estimate of the average demand for the CPU in terms of the number of processes wanting to run (the ones which are ready to run and not blocked on any resource).

Linux kernel version 2.6.37 is the reference for the following sections. And a system with 1 CPU is assumed.

With a single-processor system, the sum total of the %CPU utilization numbers for all processes (including the idle process) would be 100. There is no bound on the CPU load average. Consider N CPU-bound processes of equal priority. We can expect the %CPU utilization number for each such process to be 100/N and the load average to be N.

For a multi-processor system with P processors, the sum total of the %CPU utilization numbers for all processes is either 100 or P*100 depending on the choice of the reporting application (the latter being the default behavior of 'top'). So, by default, if a process is using one of the processors 40% of the time, 'top' would display the CPU utilization as 40%. There is no difference with respect to the interpretation of CPU load average numbers.

The next section covers some aspects of process scheduling in Linux. Keeping this information in mind will help in better understanding of the subsequent sections.

Process scheduling

This section discusses specifically the context and call-flow associated with process scheduling and not the details of process selection in specific scheduling policies.

The kernel does a lot of work at regular intervals (the interval being build-time configurable; typically referred to as a jiffy (and set to 10ms/1ms for x86/ARM platforms) including:

1. Timekeeping
2. **Updating resource usage information (including %CPU information)**
3. Running dynamic timers (jiffy-based timers)
4. Updating scheduler statistics and deciding whether a process switch is required
5. **Load average calculation**

The periodic events are provided by the "**system timer**".

Invocation of the scheduler happens under the following conditions:

1. A process yields voluntarily
2. During the switch from IRQ context/kernel mode to user mode
3. During the switch from IRQ context to kernel mode (for pre-emptive kernels)

For #2 and #3, the context switch to another process is conditional, the condition being a flag – "***need_resched***" – which is set on the following conditions:

1. The time slice allocated for the process has expired and there is another process which needs to run based on the scheduling policy.
2. A higher priority process (than the current process) moved from blocked state to runnable state.

Since the calculation of %CPU and load averages are dependent on the processing of the periodic system timer event which in turn depends on the certain specific configurations of the kernel, it is worth examining how these configurations qualify the preceding discussion which applies generally.

Periodic ticks mode without hrtimer support

Without hrtimer support, any process which needs to sleep for a certain duration of time (as opposed to a busy wait loop) or do something at a certain time in the future would use dynamic timers which have the granularity of a jiffy. Essentially, whether a timer has expired or not is checked when a jiffy expires. Therefore, context switch to a new process can potentially be caused by the following.

- The current process moves a higher priority process to be runnable (can happen any time)
- In an ISR, a higher priority process is marked as runnable.
- The time slice of the current process expires (happens in system timer interrupt handler).
- A dynamic timer expires marking a higher priority process to be runnable (happens in system timer interrupt handler).

Periodic ticks mode with hrtimer support

Hrtimer support is not dependent on the value of a jiffy and events can be programmed independently of a jiffy.

When hrtimer support is enabled, processes can sleep more accurately as compared to the older dynamic timers. A hrtimer can also be used for accurately programming an event in the future, e.g. running a work function.

Therefore, in addition to the previous section, there is another condition in which a new process can be scheduled:

- A hrtimer expires subsequently marking a higher priority process as runnable (happens in the hrtimer interrupt handler and possibly subsequent SOFTIRQ context).

Dyntick-idle mode with hrtimer support

The periodic ticks described in the previous sections are not required if there are no processes which need to run. In fact, optimization in power consumption can be achieved by disabling the ticks when not required. Linux provides a build time configurability for the same.

When the dyntick-idle mode is enabled, the periodic ticks are disabled when there are no processes to be run on the CPU.

This happens in the idle process loop.

The idle process executes the following steps in a loop:

1. Disable periodic ticks
2. Loop until *need_resched* is set. *need_resched* will be set whenever any process is marked as runnable.
3. Restart periodic ticks
4. Invoke scheduler

#1 disables periodic ticks only if the next event is more than a tick into the future.

The system timer events are handled by a hrtimer object. This timer is always reprogrammed with successive intervals which are multiples of a jiffy.

With this background on the handling of system timer and process scheduling, the following sections discuss how Linux collects relevant information and calculates the CPU usage and load average metrics.

CPU Utilization

The %CPU numbers reflect how the CPU is shared among different processes.

The kernel collects samples regularly to identify how the CPU is being shared.

The sample is simply the current process which is using the CPU. Ideally (i.e., if we had enough and accurate samples), these numbers should show how the CPU is shared by different processes (including the idle process) over time.

So, if in a span of say, 1s, process A runs for 300ms, process B runs for 100ms and the CPU is idle for the rest of the time, we would obtain the following %CPU numbers:

Process A – 30%, Process B – 10%, idle – 60%.

Note that this calculation only cares about a process when it actually runs on the CPU and not in any other state (including processes which are ready to run or are blocked).

The %CPU numbers can be obtained using the “top” utility. Figure 2 is a snapshot of the output of “top”. The %CPU numbers for each process is displayed in the corresponding row. The second row displays how the CPU is shared between user mode (usr), kernel mode (sys) and so on.

```
Mem: 83592K used, 120228K free, 0K shrd, 832K buff, 7488K cached
CPU:  16% usr  64% sys   0% nic  18% idle   0% io   0% irq   0% sirq
Load average: 0.03 0.04 0.02 1/47 1348
  PID  PPID USER     STAT   VSZ %MEM %CPU COMMAND
 1343  1337 root     S     1556   1%  81% ./a.out 8 2 0
 1348  1346 root     R     3128   2%   0% top
 1294     1 messageb S     3324   2%   0% /usr/bin/dbus-daemon --system
 1337  1336 root     S     3128   2%   0% -sh
 1334  1332 root     S     3128   2%   0% -sh
 1346  1345 root     S     3128   2%   0% -sh
 1304     1 root     S     3000   1%   0% /sbin/syslogd -n -C64 -m 20
 1299     1 root     S     2940   1%   0% /usr/sbin/telnetd
 1256     1 root     S     2940   1%   0% udhcpc -R -b -p /var/run/udhcpc.eth0.p
 1306     1 root     S     2936   1%   0% /sbin/klogd -n
 1336  1299 root     S     2520   1%   0% login -- root
 1345  1299 root     S     2520   1%   0% login -- root
 1332     1 root     S     2516   1%   0% login -- root
 1315     1 root     S     2172   1%   0% /usr/sbin/thttpd -d /srv/www -u root -
 1333     1 root     S     1968   1%   0% /sbin/getty 38400 tty1
   70     1 root     S <   1956   1%   0% /sbin/udevd -d
    1     0 root     S     1708   1%   0% init [5]
   26     2 root     SW       0   0%   0% [scsi_eh_0]
   39     2 root     SW       0   0%   0% [flush-1:0]
    3     2 root     SW       0   0%   0% [ksoftirqd/0]
   17     2 root     SW       0   0%   0% [kworker/0:1]
    2     0 root     SW       0   0%   0% [kthreadd]
    9     2 root     SW       0   0%   0% [irq/74-serial i]
    4     2 root     SW       0   0%   0% [kworker/0:0]
```

Figure 2 - %CPU utilization and CPU load averages displayed by "top"

The kernel populates the information in a proc entry - /proc/[PID]/stat[3]. For each process, time spent (in terms of jiffies) in user mode and kernel mode are maintained. "Top" adds these two to display the per-process %CPU numbers.

Accounting CPU time

As mentioned earlier, the kernel accounts for CPU resource in the system timer interrupt handler which happens every jiffy (or some time-varying multiple of jiffy in case of dyntick-idle mode).

The default mechanism accounts an entire system timer tick (10ms if HZ=100) to the process running when the system timer event is processed. The processes which ran during the last jiffy but yielded the CPU voluntarily before the system timer tick are not considered.

With this information, we'll now to try to analyze what can go wrong in this method of accounting and reporting CPU usage.

Sources of error

If a process needs to be running/sleeping for durations close to a jiffy, the %CPU numbers are expected to be inaccurate. Additionally, if the process happens to have a periodicity in the multiple of a jiffy, the numbers can be wildly inaccurate. For example, if a process runs for 5ms and sleeps for 15ms in a loop, and 1 jiffy is 10ms, the %CPU value reported would depend on when the scheduler timer tick events occur. It could be 50% (if timer tick event happens during the run time – the process would be found running every alternate jiffy) or 0% otherwise. A situation where the %CPU utilization incorrectly shows up as 0% is described in Figure 3.

Figure 3 - A situation where %CPU numbers can be very inaccurate

Getting more accurate numbers

If we change the tick interval (jiffy) to a different value (not aligned with the process execution periodicity), then the %CPU would be expected to be more accurate.

Alternatively, we can reduce the tick interval to a lower number. This is again expected to provide more accurate %CPU numbers. However, we would also expect an increase in the overall system load as the scheduler processing would get invoked quite often.

It would be advisable, however, not to change tick interval just for the sake of correct reporting of %CPU unless explicitly desired. There are other ways of getting better accuracy (at a performance cost) which, however, are not available for all platforms [4].

The next section discusses CPU load averages which should be used in conjunction with %CPU numbers to monitor the health of an application.

CPU Load average

The kernel maintains three load averages to track the CPU demand trend.

These numbers are provided by "top" and "uptime". In Figure 2, the third line displays the three load averages.

The averages are exponentially damped and maintained for 1 min, 5 min and 15 mins. For the 1 min average, for example, the samples collected in the last 1 minute will contribute the most to the load.

The samples which Linux collects are discussed later in this section.

The kernel populates these numbers at /proc/loadavg.

Samples collected

The **CPU demand** is reflected by the number of processes waiting to run or running on the CPU. This number is what needs to be sampled.

The Linux kernel however collects samples of a number which is obtained as a sum of the following:

1. Number of **runnable processes (TASK_RUNNING)** – includes the processes ready to run and running, if any.
2. Number of process in **uninterruptible sleep (TASK_UNINTERRUPTIBLE)**.

Frequency of sample collection and processing context

The calculation is expected to be done every **LOAD_FREQ** jiffies (~5HZ jiffies or ~5s).

The calculation happens in the system timer interrupt handling context.

The sample collection and load average calculation, therefore, happens every jiffy if dyntick-idle mode is disabled. If dyntick-idle mode is enabled, the accuracy of load average calculation reduces.

When dyntick-idle mode is enabled, the system timer ticks are not programmed every jiffy. The frequency of scheduler ticks depends on the demand of the CPU. This, in turn, can mean that a system timer tick was not available and processed when LOAD_FREQ jiffies expired and load average calculation was supposed to happen.

With this background, the following section discusses the possible sources of error in the calculation of load averages.

Sources of error

So, in dyntick-idle mode, the CPU can be idle for long durations of time or periodically around the time when the system timer tick is supposed to happen. This would mean that the system timers are not programmed and processed under such conditions. Whenever a tick is finally processed, the kernel needs to account for the time that elapsed since the last tick was processed. The sample value for the intermediate intervals for the idle CPU is taken as zero. This has the potential of causing huge errors in the load average calculation. The description of one such situation follows.

For processes which use accurate timer interrupts – hrtimers - to sleep/schedule themselves would not need too many scheduler interrupts in dyntick-idle mode. This is because such processes would be scheduled when the timer duration specified by them expires and would yield the CPU themselves when the work is done.

Under circumstances not too rare, the kernel would see that there is no demand for CPU during a system timer tick. The CPU load average in such circumstances can often show up as zero.

Consider the following example:

Kernel is configured for HZ=100 and dynamic ticks are enabled

Process A has the following sequence of operation

- *Do some CPU operations for 2ms*
- *Program a sleep for 8 ms (using nanosleep which uses hrtimers)*

When this process executes, the average load reported (1 min average) is often found to be 0.00.

There are two reasons why the CPU load average can show up as zero here.

1. The samples are always collected at times when process A is sleeping.
2. Although a few representative samples are collected, the system timer ticks (and therefore the load average calculations) are not programmed for long durations of time. Such periods of time are accounted as if there was no load on the CPU.

   The first is likely to happen only if the process periodicity is aligned with LOAD_FREQ. The situation is similar to the one described in Figure 3. This in the long run doesn't happen even with process A (which does have the same periodicity as a jiffy). This is demonstrated by the observation that the load averages are representative of the actual demand for CPU (~0.2) when the same process is run with a Kernel which is configured for HZ=100 and dyntick-idle mode disabled.

   The second reason above, therefore, is likely to cause more pronounced errors in load average calculation, often determining the value to be zero.

Note that although some of the errors mentioned above can be reduced by using a kernel configured with dynamic ticks disabled, that wouldn't be desirable. Having dynamic ticks disabled can often be sub-optimal for certain systems and cause a performance hit.

Beyond the sources of error discussed above, there is another source of error which comes inherent in the samples collected.

3. The processes which are in uninterruptible sleep contribute to the load average calculation.

This will cause an overshoot in the average load calculated in certain cases where the uninterruptible sleeps are long.

For example, a load average of around 0.7 is observed with no processes launched after the default processes are launched by the filesystem for a specific ARM-based embedded system. The cause for this happens to be the "khubd" thread which goes into non interruptible sleeps often. The state of current processes can be obtained by looking at the "top" or "ps" output.

Interpreting Load average numbers

As discussed in the previous section, one needs to be careful before interpreting or acting on load averages.

An important consideration is the nature of the processes running.

A CPU-bound process which is non-time critical has different requirements than a **Time critical one** (specific jobs must be performed at specific times with minimal jitter). The requirements of **CPU-bound process** would be met as long as it gets a reasonable share of the CPU and a high CPU load average doesn't necessarily imply an alarm. The **Time critical one** would require that the average number of processes running (and therefore the average CPU load) be small which would imply a smaller latency in running that process.

However, one can think of complex systems with processes of both kinds running. In this particular case, a high load average alone doesn't necessarily imply a problem with the system. In fact, as long as the time-critical processes get the share of the CPU they want and when they want it, there is no problem. This would require appropriate prioritization of the processes and in some cases a design based on a good understanding of the scheduling policy. For example, consider two processes:

Process A: CPU-bound, not time-critical

Process B: IO-bound, time critical, needs to use the CPU resource every 10ms for a duration of 5ms.

In this case, the expected load average is 1.5.

What we need to know additionally is whether process B is getting sufficient time to run and whether it is getting the CPU at regular intervals. The %CPU utilization numbers can help in identifying whether process B is getting sufficient share of the CPU.

Generally, the following ideas should help in interpreting and using the load average numbers:

1. Analyze the %CPU and load average numbers together.
   If the CPU is not too heavily loaded as would be the case for processes with real-time requirements, the %CPU numbers and load average number should in fact be consistent with each other - the total utilization of the CPU against the load average.
   The desired load average in that case would be less than 1. However, this is not a strict rule, and there is dependency on the types of processes running.
   Also, %CPU numbers can be wildly inaccurate when the process periodicity matches the system timer tick frequency. To get consistent %CPU numbers, one can choose to change the periodicity of the process or the kernel (perhaps only for debugging). An extremely high or low load average with expectedly consistent and low %CPU numbers would indicate an error in the average load calculation.
2. Understand the behavior of the processes running – for example, the method of sleeping and execution periodicity along with the kernel configuration with respect to scheduling and use it to build an expectation from the load average numbers.
3. Identify processes which can go into uninterruptible sleep for long durations or periodically.

References

1. Robert Love: Linux Kernel Development (3rd Edition)
2. Linux kernel source code V2.6.37.
3. http://man7.org/linux/man-pages/man5/proc.5.html
4. http://www.linux.org/threads/the-linux-kernel-configuring-the-kernel-part-2.4318/
5. http://www.linuxjournal.com/article/9001?page=0,0
6. http://www.linuxjournal.com/article/8144

Disclaimer

This white paper is for informational purposes only. Ittiam makes no warranties, express, implied or statutory, as to the information in this document. The information contained in this document represents the current view of Ittiam Systems on the issues discussed as of the date of publication. It should not be interpreted to be a commitment on the part of Ittiam, and Ittiam cannot guarantee the accuracy of any information presented after the date of publication.