UNIX Load Average
Part 1: How It Works

Neil J. Gunther

Performance Dynamics
Castro Valley, California, USA

Have you ever wondered how those three little numbers that appear in the UNIX load average (LA) report are calculated?

This online article explains how and how the load average (LA) can be reorganized to do better capacity planning.

In this multi-part series I want to explore the use of averages in performance analysis and capacity planning. There are many manifestations of averages e.g., arithmetic average (the usual one), moving average (often used in financial planning), geometric average (used in the SPEC CPU benchmarks), harmonic average (not used enough), to name a few.

More importantly, we will be looking at averages over time or time-dependent averages. A particular example of such a time-dependent average is the **load average** metric that appears in certain UNIX commands. In Part 1 I shall look at what the load average is and how it gets calculated. In Part 2 I'll compare it with other averaging techniques as they apply in capacity planning and performance analysis. This article does not assume you are a familiar with UNIX commands, so I will begin by reviewing those commands which display the load average metric. By Section 4, however, I'll be submerging into the UNIX kernel code that does all the work.

1 UNIX Commands

Actually, *load average* is not a UNIX command in the conventional sense. Rather it's an embedded metric that appears in the output of other UNIX commands like `uptime` and `procinfo`. These commands are commonly used by UNIX sysadmin's to observe system resource consumption. Let's look at some of them in more detail.

1.1 Classic Output

The generic ASCII textual format appears in a variety of UNIX shell commands. Here are some common examples.

uptime

The `uptime` shell command produces the following output:

```
[pax:~]% uptime
9:40am  up 9 days, 10:36,  4 users,  load average: 0.02, 0.01, 0.00
```

It shows the time since the system was last booted, the number of active user processes and something called the *load average*.

procinfo

On Linux systems, the `procinfo` command produces the following output:

```
[pax:~]% procinfo
Linux 2.0.36 (root@pax) (gcc 2.7.2.3) #1 Wed Jul 25 21:40:16 EST 2001 [pax]

Memory:      Total        Used        Free      Shared     Buffers      Cached
Mem:         95564       90252        5312       31412       33104       26412
Swap:        68508           0       68508

Bootup: Sun Jul 21 15:21:15 2002    Load average: 0.15 0.03 0.01 2/58 8557
...
```

The *load average* appears in the lower left corner of this output.

w

The `w(ho)` command produces the following output:

```
[pax:~]% w
  9:40am  up 9 days, 10:35,  4 users,  load average: 0.02, 0.01, 0.00
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
mir      ttyp0    :0.0             Fri10pm  3days  0.09s  0.09s  bash
neil     ttyp2    12-35-86-1.ea.co  9:40am  0.00s  0.29s  0.15s  w
...
```

Notice that the first line of the output is identical to the output of the `uptime` command.

top

The `top` command is a more recent addition to the UNIX command set that ranks processes according to the amount of CPU time they consume. It produces the following output:

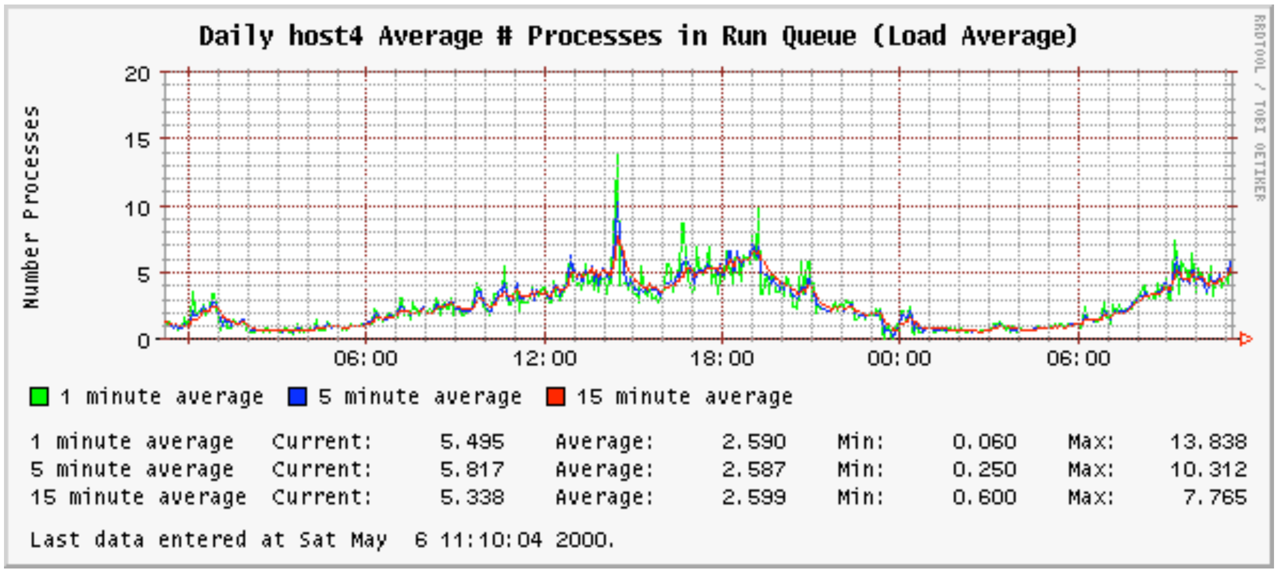
```
  4:09am  up 12:48,  1 user,  load average: 0.02, 0.27, 0.17
58 processes: 57 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  0.5% user,  0.9% system,  0.0% nice, 98.5% idle
Mem:   95564K av,  78704K used,  16860K free,  32836K shrd,  40132K buff
Swap:  68508K av,      0K used,  68508K free                 14508K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT  LIB %CPU %MEM   TIME COMMAND
 5909 neil      13   0   720  720   552 R       0  1.5  0.7   0:01 top
    1 root       0   0   396  396   328 S       0  0.0  0.4   0:02 init
    2 root       0   0     0    0     0 SW      0  0.0  0.0   0:00 kflushd
    3 root     -12 -12     0    0     0 SW<     0  0.0  0.0   0:00 kswapd
...
```

In each of these commands, note that there are **three** numbers reported as part of the `load average` output. Quite commonly, these numbers show a descending order from left to right. Occasionally, however, an ascending order appears e.g., like that shown in the `top` output above.

1.2 GUI Output

The load average can also be displayed as a time series like that shown here in some output from a tool called ORACA.



```
1 minute average   Current:    5.495   Average:    2.590   Min:     0.060   Max:    13.838
5 minute average   Current:    5.817   Average:    2.587   Min:     0.250   Max:    10.312
15 minute average  Current:    5.338   Average:    2.599   Min:     0.600   Max:     7.765

Last data entered at Sat May  6 11:10:04 2000.
```

Figure 1: ORCA plot of the 3 daily load averages.

Although such visual aids help us to see that the green curve is more spikey and has more variability than the red curve, and it allows us to see a complete day's worth of data, it's not clear how useful this is for capacity planning or performance analysis. We need to understand more about how the load average metric is defined and calculated.

2 So What Is It?

So, exactly what is this thing called *load average* that is reported by all these various commands? Let's look at the official UNIX documentation.

2.1 The man Page

```
[pax:~]% man "load average"
No manual entry for load average
```

Oops! There *is* no man page! The `load average` metric is an output embedded in other commands so it doesn't get its own man entry. Alright, let's look at the man page for `uptime`, for example, and see if we can learn more that way.

```
...
DESCRIPTION
       uptime  gives a one line display of the following informa-
       tion.  The current time, how long the system has been run-
       ning, how many users are currently logged on, and the sys-
       tem load averages for the past 1, 5, and 15 minutes.
...
```

So, that explains the three metrics. They are the ``... load averages for the past 1, 5, and 15 minutes.''
Which are the **GREEN**, **BLUE** and **RED** curves, respectively, in Figure 1 above.
Unfortunately, that still begs the question ``What is the load?''

2.2 What the Gurus Have to Say

Let's turn to some UNIX hot-shots for more enlightenment.

Tim O'Reilly and Crew

The book *UNIX Power Tools* [], tell us on p.726 **The CPU**:

> *The load average tries to measure the number of active processes at any time. As a measure of CPU utilization, the load average is simplistic, poorly defined, but far from useless.*

That's encouraging! Anyway, it does help to explain what is being measured: the number of active processes. On p.720 **39.07 Checking System Load: uptime** it continues ...

> *... High load averages usually mean that the system is being used heavily and the response time is correspondingly slow.*
>
> *What's high? ... Ideally, you'd like a load average under, say, 3, ... Ultimately, 'high' means high enough so that you don't need uptime to tell you that the system is overloaded.*

Hmmm ... where did that number ``3'' come from? And which of the three averages (1, 5, 15 minutes) are they referring to?

Adrian Cockcroft on Solaris

In *Sun Performance and Tuning* [] in the section on p.97 entitled: **Understanding and Using the Load Average**, Adrian Cockcroft states:

> *The load average is the sum of the run queue length and the number of jobs currently running on the CPUs. In Solaris 2.0 and 2.2 the load average did not include the running jobs but this bug was fixed in Solaris 2.3.*

So, even the ``big boys'' at Sun can get it wrong. Nonetheless, the idea that the load average is associated with the CPU run queue is an important point.

O'Reilly et al. also note some potential gotchas with using load average ...

> *...different systems will behave differently under the same load average. ... running a single cpu-bound background job .... can bring response to a crawl even though the load avg remains quite low.*

As I will demonstrate, this depends on when you look. If the CPU-bound process runs long enough, it will drive the load average up because its always either running or runnable. The obscurities stem from the fact that the load average is not your *average kind of average*. As we alluded to in the above introduction, it's a **time-dependent**average. Not only that, but it's a **damped** time-dependent average. To find out more, let's do some controlled experiments.

3 Performance Experiments

The experiments described in this section involved running some workloads in background on single-CPU Linux box. There were two phases in the test which has a duration of 1 hour:

- CPU was pegged for 2100 seconds and then the processes were killed.
- CPU was quiescent for the remaining 1500 seconds.

A Perl script sampled the load average every 5 minutes using the `uptime` command. Here are the details.

3.1 Test Load

Two hot-loops were fired up as background tasks on a single CPU Linux box. There were two phases in the test:

1. The CPU is pegged by these tasks for 2,100 seconds.
2. The CPU is (relatively) quiescent for the remaining 1,500 seconds.

The 1-minute average reaches a value of 2 around 300 seconds into the test. The 5-minute average reaches 2 around 1,200 seconds into the test and the 15-minute average would reach 2 at around 3,600 seconds but the processes are killed after 35 minutes (i.e., 2,100 seconds).

3.2 Process Sampling

As the authors [] explain about the Linux kernel, because both of our test processes are CPU-bound they will be in a `TASK_RUNNING` state. This means they are either:

- *running* i.e., currently executing on the CPU
- *runnable* i.e., waiting in the run_queue for the CPU

The Linux kernel also checks to see if there are any tasks in a short-term sleep state called `TASK_UNINTERRUPTIBLE`. If there are, they are also included in the load average sample. There were none in our test load.

The following source fragment reveals more details about how this is done.

```
 * Nr of active tasks - counted in fixed-point numbers
 */
static unsigned long count_active_tasks(void)
{
        struct task_struct *p;
        unsigned long nr = 0;

        read_lock(&tasklist_lock);
        for_each_task(p) {
                if ((p->state == TASK_RUNNING ||
                     (p->state & TASK_UNINTERRUPTIBLE)))
                        nr += FIXED_1;
        }
        read_unlock(&tasklist_lock);
        return nr;
}
```

So, `uptime` is sampled every 5 seconds which is the linux kernel's intrinsic timebase for updating the load average calculations.

3.3 Test Results

The results of these experiments are plotted in Fig. 2. NOTE: These colors do not correspond to those used in the ORCA plots like Figure1.

Although the workload starts up instantaneously and is abruptly stopped later at 2100 seconds, the load average values have to catch up with the instantaneous state. The 1-minute samples track the most quickly while the 15-minute samples lag the furthest.

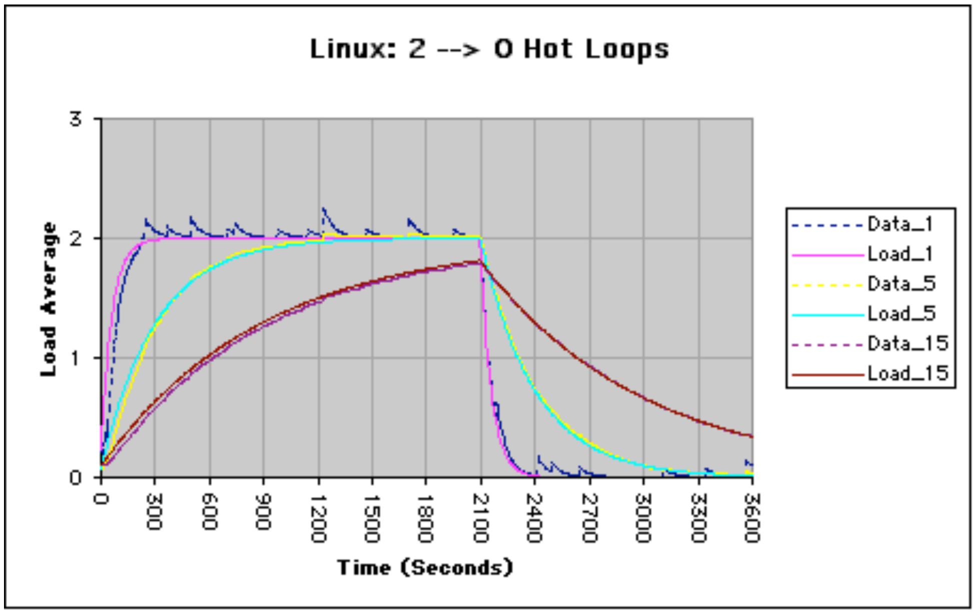


Figure 2: Linux load average test results.

For comparison, here's how it looks for a single hot-loop running on a single-CPU Solaris system.

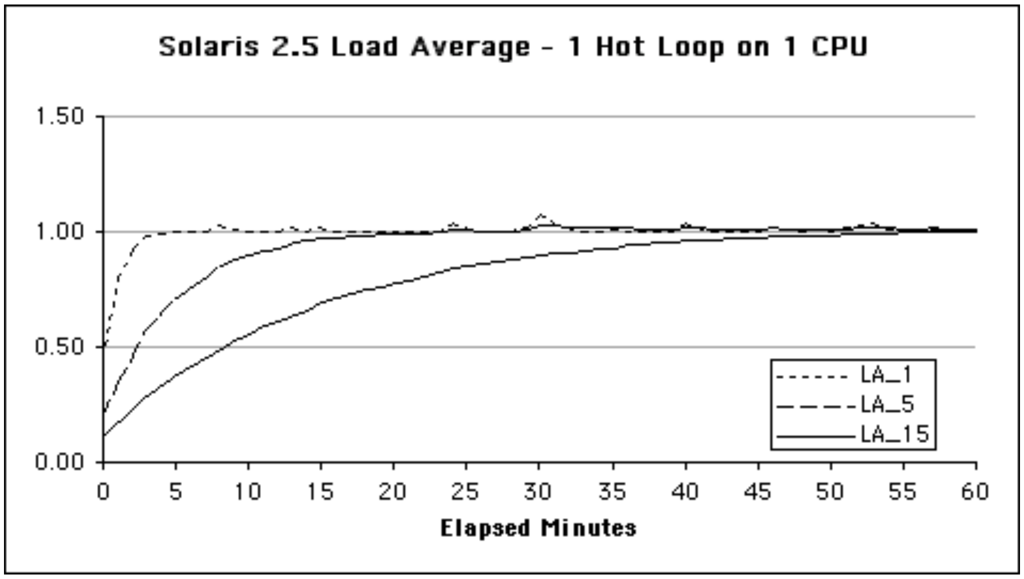


Figure 3: Solaris load average test results.

You would be forgiven for jumping to the conclusion that the ``load'' is the same thing as the CPU utilization. As the Linux results show, when two hot processes are running, the maximum load is two (not one) on a single CPU. So, load is **not** equivalent to CPU utilization.

From another perspective, Fig. 2 resembles the charging

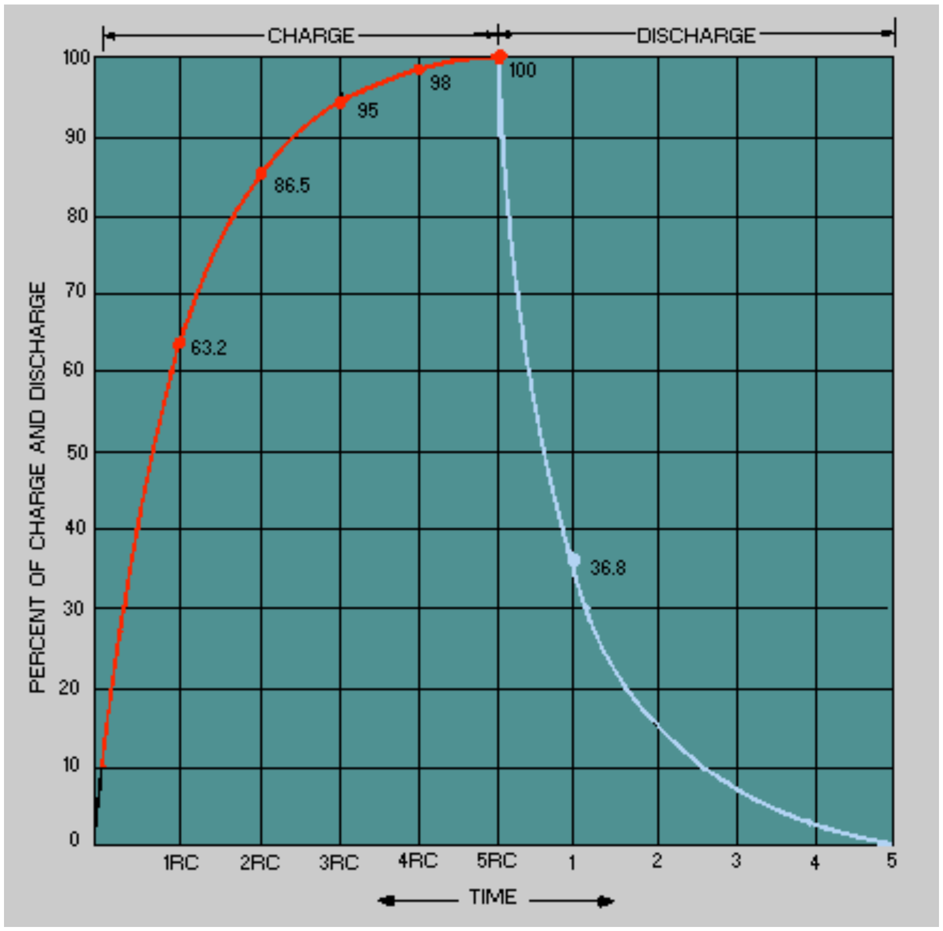


Figure 4: Charging and discharging of a capacitor.

and discharging of a capacitive RC circuit.

4 Kernel Magic

Now let's go inside the Linux kernel and see what it is doing to generate these load average numbers.

```
unsigned long avenrun[3];

static inline void calc_load(unsigned long ticks)
{
        unsigned long active_tasks; /* fixed-point */
        static int count = LOAD_FREQ;

        count -= ticks;
        if (count < 0) {
                count += LOAD_FREQ;
                active_tasks = count_active_tasks();
                CALC_LOAD(avenrun[0], EXP_1, active_tasks);
                CALC_LOAD(avenrun[1], EXP_5, active_tasks);
                CALC_LOAD(avenrun[2], EXP_15, active_tasks);
        }
}
```

The countdown is over a `LOAD_FREQ` of 5 HZ. How often is that?

```
  1 HZ    =   100 ticks
  5 HZ    =   500 ticks
  1 tick  =    10 milliseconds
500 ticks =  5000 milliseconds (or 5 seconds)
```

So, 5 HZ means that `CALC_LOAD` is called every **5 seconds**.

4.1 Magic Numbers

The function `CALC_LOAD` is a macro defined in sched.h

```
extern unsigned long avenrun[];         /* Load averages */

#define FSHIFT          11              /* nr of bits of precision */
#define FIXED_1         (1<<FSHIFT)     /* 1.0 as fixed-point */
#define LOAD_FREQ       (5*HZ)          /* 5 sec intervals */
#define EXP_1           1884            /* 1/exp(5sec/1min) as fixed-point */
#define EXP_5           2014            /* 1/exp(5sec/5min) */
#define EXP_15          2037            /* 1/exp(5sec/15min) */

#define CALC_LOAD(load,exp,n) \
        load *= exp; \
        load += n*(FIXED_1-exp); \
        load >>= FSHIFT;
```

A noteable curiosity is the appearance of those magic numbers: 1884, 2014, 2037. What do they mean? If we look at the preamble to the code we learn,

```
/*
```

```
 * These are the constant used to fake the fixed-point load-average
 * counting. Some notes:
 *  - 11 bit fractions expand to 22 bits by the multiplies: this gives
 *    a load-average precision of 10 bits integer + 11 bits fractional
 *  - if you want to count load-averages more often, you need more
 *    precision, or rounding will get you. With 2-second counting freq,
 *    the EXP_n values would be 1981, 2034 and 2043 if still using only
 *    11 bit fractions.
 */
```

These magic numbers are a result of using a fixed-point (rather than a floating-point) representation.

Using the 1 minute sampling as an example, the conversion of exp(5/60) into base-2 with 11 bits of precision occurs like this:

$$e^{5/60} \circledR \frac{e^{5/60}}{2^{11}} \qquad (1)$$

But EXP_M represents the inverse function exp(-5/60). Therefore, we can calculate these magic numbers directly from the formula,

$$\text{EXP\_M} = \frac{2^{11}}{2^{5 \log_2(e) / 60M}} \qquad (2)$$

where M = 1 for 1 minute sampling. Table 1 summarizes some relevant results.

T	EXP_T	Rounded
5/60	1884.25	1884
5/300	2014.15	2014
5/900	2036.65	2037
2/60	1980.86	1981
2/300	2034.39	2034
2/900	2043.45	2043

Table 1: Load Average magic numbers.

These numbers are in complete agreement with those mentioned in the kernel comments above. The fixed-point representation is used presumably for efficiency reasons since these calculations are performed in kernel space rather than user space.

One question still remains, however. Where do the ratios like exp(5/60) come from?

4.2 Magic Revealed

Taking the 1-minute average as the example, `CALC_LOAD` is identical to the mathematical expression:

$$\text{load}(t) = \text{load}(t-1)\, e^{-5/60} + n\,(1 - e^{-5/60}) \qquad (3)$$

If we consider the case n = 0, eqn.(3) becomes simply:

$$\text{load}(t) = \text{load}(t-1)\, e^{-5/60} \qquad (4)$$

If we iterate eqn.(4), between t = t_0 and t = T we get:

$$\text{load}(t_T) = \text{load}(t_0)\, e^{-5t/60} \qquad (5)$$

which is pure exponential decay, just as we see in Fig. 2 for times between t_0 = 2100 and t_T = 3600.

Conversely, when n = 2 as it was in our experiments, the load average is dominated by the second term such that:

$$\text{load}(t_T) = 2\, \text{load}(t_0)\, (1 - e^{-5t/60}) \qquad (6)$$

which is a monotonically increasing function just like that in Fig. 2 between t_0 = 0 and t_T = 2100.

5 Summary

So, what have we learned? Those three innocious looking numbers in the LA triplet have a surprising amount of depth behind them.

The triplet is intended to provide you with some kind of information about how much work has been done on the system in the recent past (1 minute), the past (5 minutes) and the distant past (15 minutes).

As you will now appreciate, there are some issues:

1. The ``load'' is not the utilization but the total queue length.
2. They are point samples of three different time series.
3. They are exponentially-damped moving averages.
4. They are in the wrong order to represent trend information.

These inherited limitations are significant if you try to use them for capacity planning purposes. I'll have more to say about all this in the next online column *Load Average Part II: Not Your Average Average*.

File translated from T$_E$X by T$_T$H, version 2.25.
On 29 May 2003, 11:24.