The Instrumentation Challenge for Linux Servers in the 21st Century

Neil J. Gunther

Abstract

There is a significant opportunity for Linux to set the pace in the scalable server marketplace. A key ingredient for scalability is manageability. To achieve a high level of performance management, better operating system instrumentation is needed. The question is, can Linux developers rise to that challenge?

1 A Brief History of Operating System Instrumentation

Since this article presents a vision of what the Linux operating system *could* look like in the future, it combines several aspects of performance instrumentation which might otherwise appear disjoint to the casual observer of Linux development. To help us see the future, we begin by briefly reviewing what has been done in the past.

The term *instrumentation*, as it will be used throughout this article, refers to the implementation of certain counters in kernel memory that are used to store sampled performance metrics, e.g., I/O counts, memory page swaps, processor busy periods and so on. These *counts* can also be converted to *rates* according to a specified timebase. More about this in Section 2.

The concept of instrumenting the operating system (O/S) in this way is not only *not* new to Linux, but even older than UNIX viz., the *Multics* project at M.I.T. circa 1965. In fact, one of the earliest forms of O/S instrumentation was logically equivalent to the oft-seen but little understood metric called the *load average*. The load average is a moving-average measure of the number of entries in the scheduler's run queue [1]. The purpose of having O/S instrumentation was to provide the Multics kernel developers with information about any significant changes in performance caused by altering the kernel code; especially that of the scheduler.

Since people like Dennis Ritchie worked on Multics, it was only natural that this same approach would be adopted in the development of the UNIX time-share kernel. Since then, as different modules have been developed to extend the capabilities of UNIX, more instrumentation counters have been included along with more *ad hoc* tools to report those metrics. The development of Linux instrumentation closely parallels that of UNIX.

The historical details notwithstanding, the important point for our discussion is that none of this instrumentation was implemented for the purposes of providing performance analysis and capacity planning information to manage transaction throughput and application response times. Even twenty years ago, no one could possibly have foreseen how ubiquitous both UNIX and Linux would become for supporting the Internet and Web-based applications. Therefore, no one could have had any inkling about a broader set of instrumentation requirements, but that is precisely the level of instrumentation that is needed today.

And let's not be chauvinistic about Linux and UNIX. Before either of those operating systems came into existence, there was the *mainframe*. Yes, the mainframe. Contrary to popular opinion, the mainframe is not a "dinosaur" that is almost extinct due to competition from cheap microprocessors. The mainframe not only survived that onslaught (by becoming cheaper itself) but it also possesses some of the most progressive O/S instrumentation and management capabilities available on the market today. In particular, the IBM *System Z* operating system [7], formerly known as *z/OS* (and *MVS* before that) provides a uniform representation of performance and capacity planning data via its Resource Measurement Facility (RMF) and System Management Facility (SMF). We certainly

do not have that in UNIX or Linux. No, the wiser view of the mainframe is to see it as a potential role model, which we shall do subsequently.

Until relatively recently, one could only monitor the sampled performance metrics collected by the O/S, no matter whether is was Linux, UNIX or System Z. Put more simply, the O/S would tell *you*, the system administrator, how it was *actually* doing as a part of its *output*, but there was no sense by which you could tell the O/S, as an *input*, what you *expected* of it. That changed under System Z (or MVS), starting about 20 years ago. IBM introduced the *System Resource Manager*, which allowed the system administrator to tell the O/S what amount of resources each user was entitled to use. This new management ability required changes to the O/S scheduler so that it became a *Fair-Share Scheduler* (FSS) rather than just a Time-Share Scheduler (TSS). UNIX vendors began providing FSS management about 10 years ago.

However, while the UNIX vendors were catching up to the mainframe, IBM introduced an even more sophisticated management concept called a *Goal Mode* scheduler (GMS). Under GMS it became possible to set performance *targets* as inputs. The targets can be either response time goals for selected applications or widows of time within which certain batch workloads should complete. Then, while the O/S is executing the entire mix of workloads, it also reports how well it is achieving those goals. To explain why this author believes a GMS-like capability is what Linux should strive to emulate, this article is organized as follows.

In Section 2 we review existing Linux performance instrumentation and its attendant weaknesses. In Section 3 we discuss the issues surrounding alternative scheduling classes for scalable server management. In Section 4 we discuss previous failed attempts to unify UNIX and Linux performance instrumentation within the *Universal Measurement Architecture*. In Section 5 we briefly consider the performance instrumentation in virtual hypervisors and recognise they also use an FSS scheduler, which we discuss more thoroughly in Sections 6 and 7. Finally, we bring these various aspects together in Section 8 to define the challenge for Linux instrumentation in the 21st century.

2 Existing Linux Performance Instrumentation

The standard snapshot of system performance in Linux is provided by the `procinfo` command. More recently, the `sysstat` package has implemented other traditional UNIX commands such as `sar, mpstat and iostat`, as well as several other performance tools in Linux. All these performance metrics are global metrics (not per-process) and, with the exception of `sar`, are not automatically time-stamped. The iostat command combines information about CPU utilization and I/O statistics for disks. The mpstat command reports both global and per-processor statistics. The sar command collects and reports system activity information including I/O transfer rates, paging activity, process-related activities, interrupts, network activity, memory and swap space utilization, CPU utilization, kernel activities and TTY statistics, and is distinguished by incorporating a time stamp when the data was sampled. sysstat works on both uniprocessor and multiprocessor machines.

Consider the number of disk IOs reported by the `iostat` command. The commands syntax is:

```
iostat [ interval [ count ] ]
```

The *interval* argument specifies the reporting period or sample period in seconds. A *count* parameter can be specified in conjunction with the interval parameter to control how many reports are generated before `iostat` exits. The default output for `count = 1` is shown in Table 1. The first report always displays information since the system was booted, while each subsequent report covers the time period since the last report. The single disk (`dev3-0`) has averaged 1.68 transfers (IOs) per second. Based on the discussion in Section 1, here is how it works.

Suppose the sample period or interval is $T = 30$ seconds, and the number of IOs during that interval is $C = 50$. The I/O rate is calculated as:

$$\frac{C}{T} = \frac{50}{30} = 1.67 \text{ TPS} \qquad (1)$$

which corresponds to the number in the third column of the last row. This is the I/O throughput, because throughput is defined as the number of completed requests C per unit time T [2].

Table 1: `iostat` output from `sysstat`

```
avg-cpu:   %user   %nice    %sys         %idle
           6.11    2.56    2.15         89.18

Device:            tps     Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
dev3-0            1.68     15.69        22.42        3117583    4454329
```

Table 2: `mpstat` output from `sysstat`

```
07:13:03 PM   CPU    %user   %nice   %system   %idle    intr/s
07:13:03 PM   all     6.40    5.84      3.29   84.47    542.47
07:13:03 PM     0     6.36    5.80      3.29   84.54    542.47
07:13:03 PM     1     6.43    5.87      3.29   84.40    542.47
```

Now consider the output of the `mpstat` command in `sysstat` shown in Table 2. In this case, the explicit command:

```
mpstat -P ALL
```

reveals that the platform has two processors labeled 0 and 1. The average CPU utilization can be determined from the second row as: $100\% - 84.47\% = 15.53\%$. Note this is the same value as: `%user + %nice + %system` $= 6.40\% + 5.84\% + 3.29\% = 15.53\%$. The calculation is performed in the Linux kernel in a way similar to that for `iostat`. Suppose the sample interval is $T = 30$ seconds and the CPU busy time is $B = 4.66$ seconds. The CPU utilization is the fraction of the sample interval for which the CPU was busy (i.e., not idle):

$$\frac{B}{T} = \frac{4.66}{30} = 0.1553 \qquad (2)$$

By definition [2], this is the same as the processor utilization. The same calculation has been done in the first two rows of Table 1. The average CPU utilization is: $100\% - 89.18\% = 10.82\%$. For comparison, the more generall output of `mpstat` for a 32-way multiprocessor is shown in Fig. 1. This kind of format has been available on many UNIX platforms for about 20 years.

The problem with these, and similar tools, is that their output format is essentially the same as it has been for more than 35 years. From the standpoint of performance management, the presentation of the 100–300 collected metrics, should be user-selectable and rendered using a more cognitively efficient visualization than ASCII text; see e.g., Fig. 2(b). Moreover, all metrics should automatically be time-stamped and stored in a database such that an analyst can drill down into more detail at any time. Although `sar` and more recent GUI-based tools

```
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
  0    2   0 4191  7150  6955 1392   93  374  573   14  1433   78  22   0   0
  1    2   0  179 11081 10956 1180  132  302 1092   13  1043   79  21   0   0
  2    1   0  159  9524  9388 1085  141  261 1249   14   897   79  21   0   0
  3    0   0 3710 10540 10466  621  231  116 1753    2   215   70  29   0   0
  4    5   0   28   355     1 2485  284  456  447   30  2263   77  23   0   0
  5    5   0   25   350     1 2541  280  534  445   26  2315   78  22   0   0
  6    3   0   26   331     0 2501  267  545  450   28  2319   78  22   0   0
  7    2   0   30   292     1 2390  232  534  475   23  2244   77  22   0   0
  8    4   0   22   265     1 2188  220  499  429   26  2118   75  25   0   0
  9    2   0   28   319     1 2348  258  513  440   26  2161   76  24   0   0
 10    4   0   23   308     0 2384  259  514  430   22  2220   76  24   0   0
 11    4   0   27   292     0 2366  237  518  438   30  2209   77  23   0   0
 12   11   0   31   314     0 2446  253  530  458   27  2290   78  22   0   0
 13    4   0   31   273     1 2334  223  523  428   25  2261   79  21   0   0
 14   12   0   29   298     1 2405  247  521  435   25  2286   78  22   0   0
 15    4   0   32   330     1 2445  272  526  450   24  2248   77  22   0   0
 16    5   0   28   271     0 2311  219  528  406   29  2188   76  23   0   0
 17    4   0   23   309     1 2387  253  537  442   25  2234   78  22   0   0
 18    3   0   25   312     1 2412  257  534  449   26  2216   78  22   0   0
 19    3   0   29   321     1 2479  262  545  462   31  2287   78  22   0   0
 20   14   0   29   347     0 2474  289  541  457   24  2253   78  22   0   0
 21    4   0   29   315     1 2406  259  534  469   24  2240   77  22   0   0
 22    4   0   27   290     1 2406  243  531  480   25  2258   77  22   0   0
 23    4   0   27   286     1 2344  235  531  445   26  2240   77  22   0   0
 24    3   0   30   279     0 2292  228  518  442   22  2160   77  23   0   0
 25    3   0   26   275     1 2340  227  538  448   25  2224   76  23   0   0
 26    4   0   22   294     1 2349  247  529  479   26  2197   77  23   0   0
 27    4   0   27   324     1 2459  270  544  476   25  2256   77  23   0   0
 28    4   0   25   300     1 2426  249  549  461   27  2253   77  23   0   0
 29    5   0   27   323     1 2463  269  541  447   23  2277   77  22   0   0
 30    2   0   27   289     1 2386  239  535  463   26  2222   77  23   0   0
 31    3   0   29   363     1 2528  304  525  446   26  2251   76  23   0   0
```

Figure 1: Sample output of Solaris *mpstat* for a 32-way multiprocessor.

do provide some of those requirements, our vision demands a more unified approach across all Linux platforms. The first step in that direction is to define a common data structure (a tree) for all the memory-resident performance counters. This has already been done in AIX and Solaris, for example, via the `rstat` and `kstat` data structures, respectively. An even more unified specification has already been formalized over a decade ago, and we shall examine that design in Section 4.

3 A Fork in the Road?

Linus Torvalds has been quoted as saying he believes the current kernel instrumentation sufficiently addresses real-world performance problems. Elsewhere, I have read that he would like to keep Linux "lean and mean." Taken in the light of the following counter points, this design goal reflects a rather narrow view of what Linux might achieve:

- How can current Linux instrumentation be sufficient when older UNIX performance instrumentation is still not sufficient for

modern multi-tier performance management?

- UNIX instrumentation was not introduced to solve real-world performance problems (see Section1). It was surely necessary but that does not make it *sufficient*. We are still living with that legacy in Linux.

- By now, I would have anticipated (naively, it seems) to have at my fingertips, a common set of useful performance metrics together with a common means for accessing them across all UNIX and Linux platforms.

- Several attempts have already been made to standardize and improve both UNIX and Linux performance instrumentation.

We take up these points in subsequent sections.

In addition, desktops, laptops and mobile devices are also becoming multiprocessor-based via the new *multicore* technology viz., multiple processor cores on a chip. Nonetheless, it can be argued that unlike servers, those platforms will continue to run essentially *single-threaded* applications. In other words, each core will run a separate application. There, I would agree that the Linux must remain lean and mean.

However, when it comes to mid-range and high-end servers, a key requirement for scalability is to be able execute *multiple threads* concurrently while accessing shared writable-data. My experience building commercial symmetric multiprocessors (SMP) at Pyramid-Siemens, showed that up to 70% of the development time was spent getting applications (e.g., relational database management systems) to scale well on a symmetric UNIX kernel. An efficient symmetric kernel requires more code (not less) in order to implement vital control structures, such as:

- processor affinity
- mutex locking
- spin-waiting
- preemption control
- processor yield

These controls provide more efficient serial access to shared resources, thereby dramatically improving the scalability of SMP applications. The same is true for Linux-based SMPs and the Linux 2.6 scheduler already supports processor affinity and preemption control. Moreover, such controls take on renewed significance when it comes to developing applications for multicores, because those chips will always be the cheapest implementation, not the most scalable implementation. Software and O/S scalability may be the only way to offset such hardware limitations.

The Linux 2.6.23 kernel comes with a modularized scheduler and a new addition called a *Completely Fair Scheduler* (CFS). It is a complete rewrite of the Linux task scheduler aimed at both desktop tasks and server workloads. As already mentioned, single-user desktops and multi-user servers lie at the extremes of a scalability spectrum. Another dimension is interactive and batch tasks; the former requiring optimal *response time* performance, while the latter needs optimal *throughput* performance.

Historically, the mainframe O/S was aimed at batch processing large amounts of data, with interactive performance being added later. The historical development of UNIX took exactly the opposite path and thus, both schedulers carry some historical bias. According to the documentation [3], CFS tries to run the task with the "gravest need" for processor time and this helps to assure that every process gets its fair share of processor. It may also benefit batch work, although the original design motivation was to improve desktop responsiveness by virtue of a *Rotating Staircase DeadLine* (RSDL) scheduler.

One sees that it is easy to become conflicted over which type of workloads, and which type of markets, Linux should aim for. Perhaps it is time for the Linux development community to take seriously Yogi Berra's famous malapropism: "When you come to a fork in the road, take it." Whether such a fork should be interpreted as a split between desktop and a sever release trees or compilable modules, is beyond the scope of this article.

4 The Universal Measurement Architecture

It is a well-kept secret that between 1990 and 1995, a group of UNIX vendors which included: Amdahl Corp. (now owned by Fujitsu), BGS (now part of BMC), Hitachi, HP, IBM, NCR, OSF, Sequent (now owned by IBM) and several other companies, designed a unified framework for the capture and transport of distributed

performance data called the *Universal Measurement Architecture* or UMA (pronounced "youmah"). This level of architectural organization is required to coherently address the difficulties of collecting performance data from the distributed instrumentation contained in multiple server tiers that support multiple software applications. It is also more efficient than SNMP (Simple Network Management Protocol).

Ironically, the motivation for UMA came from the mainframe world. Amdahl Corporation was in the business of making mainframe hardware clones that could run MVS from IBM. But Amdahl also wanted to sell into the Telecom industry and their requirement was that all machines must run UNIX. So, Amdahl developed a version of UNIX called *Unix Time Share* or UTS. Because Amdahl was aware of the RMF and SMF instrumentation under MVS (see Section 1), they originally designed UMA specifically for UTS. The generalized UMA specification is available online from the Open Group [4].

> O/S vendors avoid investing in alternative instrumentation if they do not perceive any demand, whilst system administrators cannot demand instrumentation they have not conceived.

The UMA reference model defines four layers and two interfaces as shown in Fig. 2(a). These layers and interfaces are briefly described from the bottom up, starting with the Data Capture Layer.

Data Capture Layer: The Data Capture Layer is responsible for collecting raw data. Its architecture together with the Data Capture Interface (DCI) allow data from multiple sources to be collected by a single consumer above the DCI, and this in turn improves the synchronization of the data collection.

Data Capture Interface: The Data Capture Interface is the interface between the Measurement Control Layer and the Data Capture Layer. It provides the means for dynamically extending data collection to new providers such as databases without affecting existing programs.

Measurement Control Layer: The Measurement Control Layer schedules and synchronizes data collection through the Data Capture Interface.

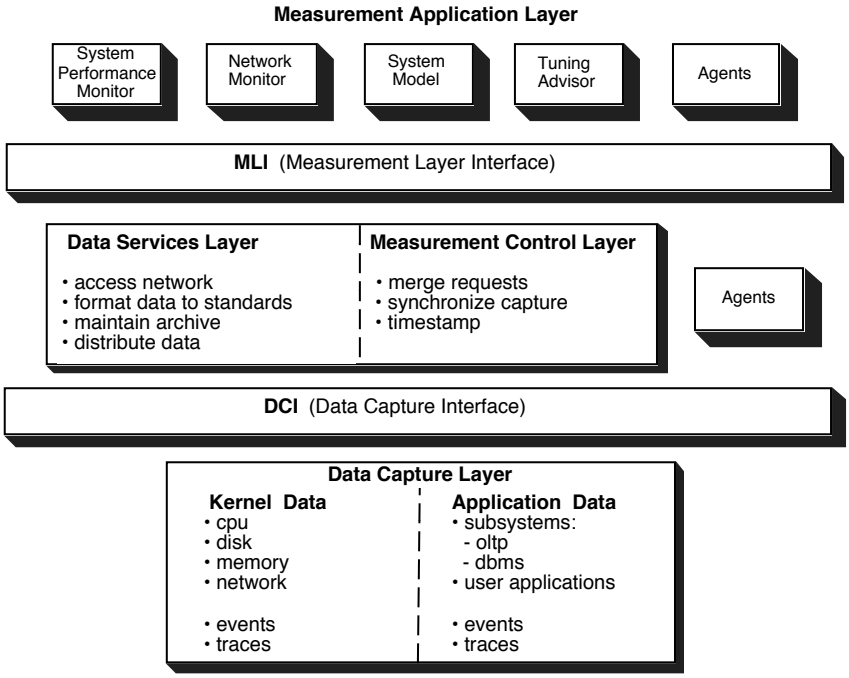
Data Services Layer: The Data Services Layer accepts measurement requests from a Measurement Application Program or MAP (Fig. 2(b)) through the Measurement Layer Interface (MLI), and distributes data to the destination requested by the MAP. A destination may include, the MAP itself, a private file or the UMA Data Storage (UMADS), which will be described later.

The `/dev/kmem` interface has historically been the primary interface used by UNIX system performance measurement utilities for extracting data from the kernel. If a program is aware of the name of a particular data structure, it can find the virtual address of that data structure by looking at the symbol table of the bootable object file. It can then open `/dev/kmem` to seek to and read the value of that data structure. The advantage of this approach is its generality: if the address of a data structure can be found, its value can be read, but its generality is also a disadvantage. Since almost any data structure can be used to provide performance data, the tendency is to do so without regard to whether it is supported. This makes it very difficult to maintain a performance application across releases when data structures change. More recently, data structures for performance metrics, such as `kstat` on Solaris and `rstat` on AIX have made data collection more portable. A similar data structure organization has been proposed [5] for Linux but as far as this author is aware, it has not been implemented. To address some of these issues and limitations, the UMA specifications includes definitions for:

- Data Pool Specification to define common metrics
- Data Capture Interface (DCI) for data providers
- Measurement Layer Interface (MLI) for performance tools

The obvious question that follows from all this is, why is UMA still not available from the same UNIX vendors who helped to define it?

The real answer is complicated but part of it is that O/S vendors, being commercial enterprises, are generally loath to promote alternative



(a) Open Group UMA reference model.

(b) UMA tool (MAP) showing processor utilization as a time series and as a histogram. A sliding control enables seemless transition between both monitored and historical data.

Figure 2: UMA architecture and tools

instrumentation and tools if they do not perceive any demand that will ensure a significant return on their development investment. The UMA working group should also have begun marketing their conception before and during its development. Instead they just assumed it would sell itself once the UMA specification was complete. And, like ships in the night, system administrators cannot demand better instrumentation which they have not yet conceived. UMA was a vendor conception, not a user conception, but it ended up being a solution in search of problem. The problem is still there, but UMA remains unrecognized as a possible solution.

Whether or not one agrees with the details of the UMA specification, the fact remains that conceptually it represents one way by which to achieve a unified interface to both UNIX and Linux server instrumentation such that a system administrator no longer need be concerned with whether their performance management scripts will run on both platform A and platform B without modification in a multitiered environment. In retrospect, it is rather depressing to think that many people thought we would already be using natural speech to communicate with computers by the turn of this century, when in fact we cannot even guarantee that a textual script will run on any two Linux distributions.

5 Virtual Machine Managers and Hypervisors

Virtualization (of services) is a hot topic because it offers server consolidation, co-located hosting, distributed web services, isolation, secure computing platforms and application mobility, without the need to be concerned about how all that gets accomplished. At least, that is the marketing pitch and to a large extent the current perception. But as anyone who has tried to tune virtual machine managers (e.g., XenServer or VMWare) for performance knows, the realities can be a little different.

All virtualization is about illusions and although it is perfectly reasonable to perpetrate such illusions upon a blissfully unaware user, it is should be considered forbidden to propagate those same illusions to a system administrator.

For those not already familiar, virtualization, in the sense it is being used here, means that an arbitrary number of different O/S instances or guests (e.g., Windows XP, MacOS X, and Linux) can run concurrently on the same platform under the overarching supervision of a virtual machine manager or hypervisor, which provides the interface between each O/S and the actual hardware resources. Beyond the O/S instance seen by each user, the details of the hypervisor are generally hidden. Like all things *virtual*, virtual machine managers are really about *illusions* and those illusions can be a source of real problems because too many important details remain hidden from the system administrator due to a lack of proper instrumentation.

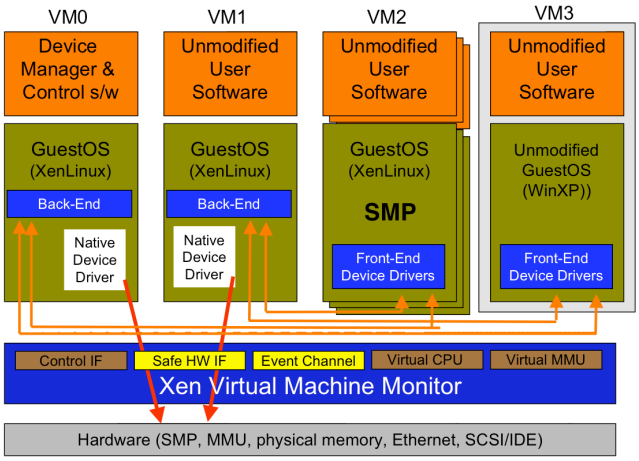


Figure 3: Organization of XenServer 3.0 hypervisor supporting Linux, Linux SMP, and Windows XP O/S guests.

What would proper instrumentation look like? To answer that question, one should understand something about the basic principles of operation of a virtual machine manager. We discuss that in the next section.

6 Getting Beyond Time Share

The partitioning of resources in a physical machine to support the concurrent execution of multiple O/S guests poses several challenges (Fig. 3). First, each O/S must be truly isolated from one another. It is unacceptable for the execution of one O/S to adversely affect the performance of another. This is particularly true when virtual machines are owned by mutually untrust-

ing users. Second, it is necessary to support a variety of different guests to accommodate the heterogeneity of popular applications. Third, and most importantly from a management standpoint, the performance overhead introduced by each guest should be small.

Historically, Linux and all UNIX schedulers have been time-share schedulers by design. In Section 3, we noted that Linux 2.6 now incorporates other scheduling classes, such as: `kernel/sched_fair.c` for CFS and `kernel/sched_rt.c` for real-time scheduling. Another type of scheduler, which has yet to be implemented in Linux is FSS (see Section 1); not to be confused with CFS. FSS requires the explicit awarding of resource shares to users by the system administrator. Moreover, it is not commonly recognized that the FSS also forms the underpinnings of the virtual machine managers in Section 5 (see [6] for a complete discussion). To better understand FSS, we first review TSS operations.

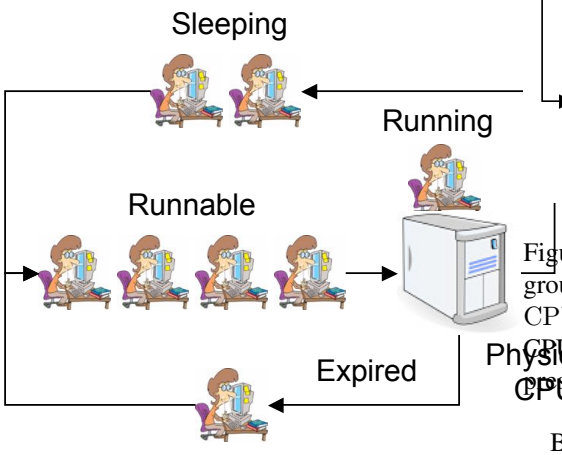


Figure 4: Time-share scheduler model including time-quantum expiration.

The purpose of a TSS is simply to provide each user with the *illusion* that they are the only person using the physical platform. In Linux, each user process is in one of three possible states: *running*, *runnable* or *sleeping*. If a process is running, it will be in the lower part of Fig. 4 executing on the physical CPU. If the process is runnable but not executing, then it will reside in the waiting line or run-queue [2]; shown immediately to the left of the physical CPU. If a process has not completed execution when the *time-quantum* expires (e.g., 10 ms or 50 ms in VMWare) it is returned to the tail of the run-queue. Otherwise, the process is sleeping because it is not ready to execute, perhaps waiting on an I/O to complete, as shown in the upper part of diagram.

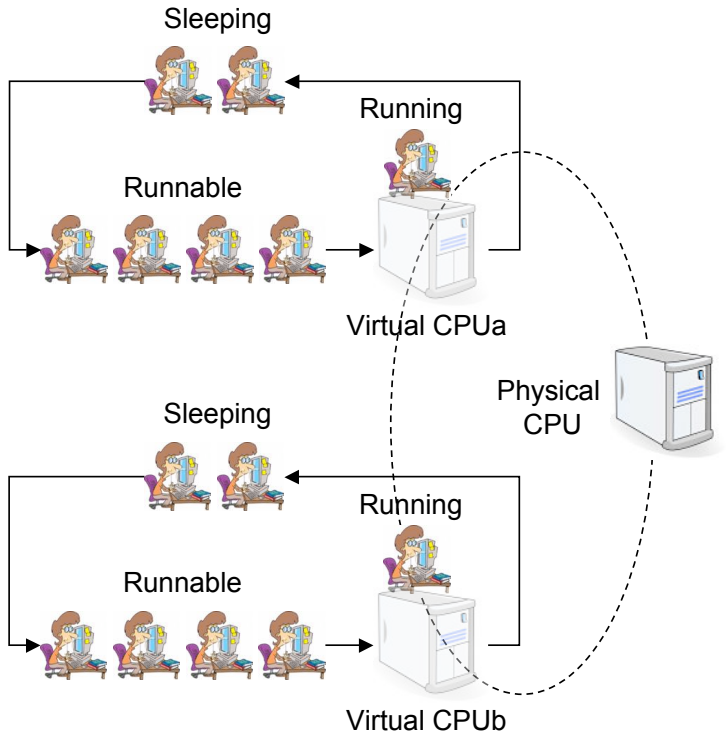


Figure 5: Fair-share scheduler model of two user groups $\mathrm{Group_a}$ and $\mathrm{Group_b}$ each with virtual $\mathrm{CPU_a}$ or $\mathrm{CPU_b}$ which share the same physical CPU. Time-quantum expiration has been suppressed for clarity.

By contrast, the FSS shown schematically in Fig. 5, provides each of the users in either $\mathrm{Group_a}$ or $\mathrm{Group_b}$ with the illusion that they possess an entire platform of their own—*a virtual machine*—whose performance is scaled according to their resource entitlement ($\mathcal{E}_{guest}$) i.e., the effective speed of *virtual* $\mathrm{CPU_a}$ or $\mathrm{CPU_b}$. Entitlement is awarded by the system administrator through the allocation of *shares*; just like owning equity shares in a corporation. The physical service time S_{guest} for each guest process (if it were to run in isolation) becomes a *virtual* service time:

$$S^V_{guest} = \frac{S_{guest}}{\mathcal{E}_{guest}}, \qquad (3)$$

which is either faster or slower than S_{guest}, ac-

cording to how it is scaled by the awarded share entitlement.

XenServer in Fig. 3 uses a form of FSS called *Borrowed Virtual Time* (BVT) as the default scheduler. Other options are also available e.g. real-time scheduling. BVT provides proportional FSS for processor scheduling based on *weights*. Each runnable guest receives a share of the processor in proportion to its weight. For example, a single processor VMWare guest OS is allocated 1000 shares by default [6]. Share allocation can have a significant impact on overall performance.

FSS introduces a scheduling superstructure on top of conventional TSS to connect processes with users and their resource entitlements as represented in the following, highly simplified, pseudocode:

VM Share Scheduling: Polls every 4000 ms ($f = 250$ mHz) to compare physical processor usage per user entitlement (Fig. 5).

```
for(i = 0; i < USERS; i++) {
    usage[i] *= decayUsage;
    usage[i] += cost[i];
    cost[i] = 0;
}
```

VM Priority Adjustment: Polls every 1000 ms and decays internal FSS process priority values (Fig. 5).

```
priDecay = Value in [0..1];
for(k = 0; k < PROCS; k++) {
    sharepri[k] *= priDecay;
}
priDecay = a * p_nice[k] + b;
```

Time Share Scheduling: Polls every physical processor tick to adjust process priorities (Fig. 4).

```
for(i=0; i<USERS; i++) {
    sharepri[i] +=
       usage[i] * p_active[i];
}
```

Process-level polling is essentially the same as standard TSS, while VM-share polling controls process-level capacity consumption.

7 Capping Resources

For any pair of users with entitlements $\mathcal{E}_1$ and $\mathcal{E}_2$, the goal of FSS can be stated as:

$$\lim_{t\to\infty} \frac{\rho_1}{\rho_2} = \frac{\mathcal{E}_1}{\mathcal{E}_2}\,, \tag{4}$$

where ρ_1 and ρ_2 are the respective resource utilizations of each user as defined in Section 2. In other words, the long run goal of FSS is to try and match the sampled ratio of utilizations (expressed as a percentage) to the ratio of their entitlements (expressed as a percentage). Among the reasons for wanting to do this are, resource management and cost accounting. The latter concept is not only unfamiliar to Linux users, but is probably anathema to them. The reality is, however, that cycles are not *free* and some datacenters already charge department managers for computer resource consumption. There is every reason to believe this trend will become more widespread as all of IT endeavors to become greener.

Suppose you are entitled to receive 10% of processing resources by virtue of being allocated 10 out of a possible 100 system wide processor shares. In other words, your processing entitlement would be 10%. Further suppose you are the only user on the system. Should you be entitled to access 100% of the processing resources? Most system administrators believe it makes sense (and is fair) to use all of the resources rather than have a 90% idle server. But can you access 100% of the processing resources if you only have 10 shares? You can if the FSS only considers active shares to calculate your entitlement. As the only user active on the system, owning 10 shares out of 10 active shares is tantamount to a 100% processing entitlement.

This natural inclination to make use of otherwise idle processing resources really rests on two assumptions:

1. You are not being charged for the consumption of processing resources. If your manager only has a budget to pay for a maximum of 10% processing on a shared server, then it would be fiscally undesirable to exceed that limit.

2. You are unconcerned about service targets. Its a law of nature that users complain about perceived changes in response time.

If there are operational periods where response time is significantly better than at other times, those periods will define the future service target.

There can, however, be potential problems due to user-perceived changes in performance. Consider the following example based on the analogy of capacity planning for a business reception.

You decide to throw a big business celebration in a hotel and invite 200 guests. You pay $2000 for 2 reception rooms along with catering from the hotel. Each reception room is designed to comfortably hold 100 people.

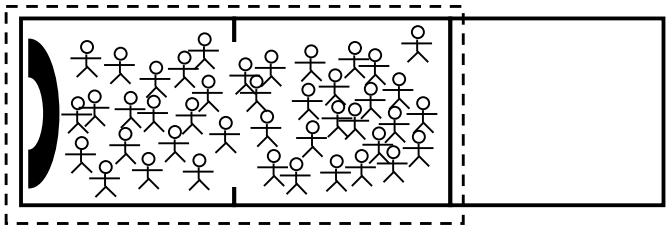
Capping Enabled: It turns out, unexpectedly, that 220 guests actually show up space becomes a little cramped (Fig. 6(a)). You are, however, getting what you paid for. It would not have been wise to purchase an extra room for such a small overflow. This situation is analogous to FSS with capping enabled.

Capping Disabled: The hotel grants you permission to occupy the third reception room (Fig. 6(b)) until the other party, who have purchased that room, actually arrives. This eases your congestion temporarily until other party does arrive. Then, your guests are going to feel uncomfortable dealing with the congestion of regrouping back into your two rooms. This situation is analogous to FS scheduling with capping disabled or not implemented.

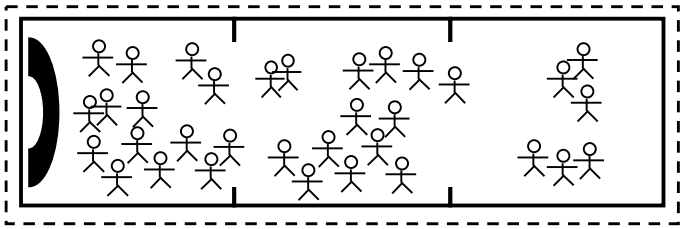
Capping Inactive: Another possibility is that only 100 guests show up to your party (Fig. 6(c)). You have now purchased more capacity than you actually needed but your guests are comfortable. This is analogous to TS scheduling, since the capping boundary is not exercised.

The state of entitlement capping can have a critical impact on the observed performance of applications running under FS control.

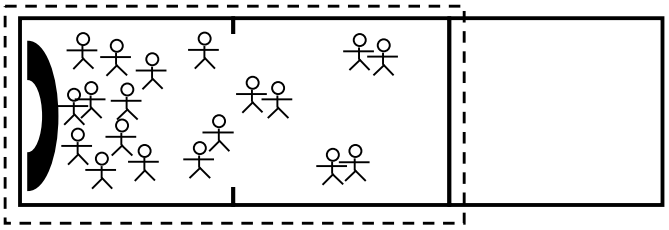
In this example, if capping depends on the number of *active* shares in the total pool of shares (as opposed to the total allocated pool), then Fig. 6(a) corresponds to the *least upper bound* on capacity, while Fig. 6(b) corresponds to the *greatest upper bound*. Such dynamically changing capacity can have detrimental consequences for both performance perceived by the users, and the overall capacity allocation strategy.



(a) An unexpected 220 guests occupy the two purchased reception rooms (*dashed line*) each of which is intended to hold 100 people comfortably.



(b) An unexpected 220 guests can temporarily occupy three rooms (*dashed line*) until another party arrives to claim the third room. The uncomfortable congestion upon regrouping into two rooms will be noticed by the guests.



(c) Only 100 guests show up and although they occupy both purchased rooms (*dashed line*), they tend to congregate around the buffet table to the left. The purchased double-room capacity is underutilized.

Figure 6: Reception room analog of various capping scenarios under FSS

In summary, understanding FSS and capping is important for the following reasons. It is important to understand FSS in its own right because if offers real-time capacity management; something that current Linux does not provide. FSS is a preliminary requirement for higher levels of capacity management such as GMS,

which we discuss in Section 8. It not widely known that FSS forms the underpinnings of all virtual machine managers like XenServer and VMWare. It is also important to understand that with the implementation of alternative schedulers comes the requirement for additional instrumentation.

8 The Challenge

Based on the preceding discussion, it does not take much imagination to guess what the future of Linux server instrumentation could and should look like. We should turn to the mainframe as a role model. Moreover, Linux is no stranger to the IBM mainframe [8], where it can run in a System Z logical partition or LPAR.

> The mainframe is not a dinosaur that became extinct due to competition from cheap microprocessors. Instead, it remains today as another powerful data-processing server on the network. Linux has the opportunity to match it and perhaps eventually surpass it.

The relevant mainframe features for Linux servers are:

1. Performance management via a fair-share scheduler and its appropriate instrumentation.

2. Performance management via a goal-mode scheduler and its appropriate instrumentation.

3. Single image cluster management.

The first of these features has already been discussed in Sections 6 and 7 and at least one variant already exists as XenServer. Although the second feature is entirely unknown in UNIX and Linux, GMS is the logical extension of the implicit goal (Eqn. 4) for FSS. GMS requires that the system administrator input performance targets to the O/S. These targets can be either response time goals or batch execution goals (e.g., scientific workloads). Using the same kind of statistical sampling mechanism described in Section 2, the actual resource consumption (*output*) is reported along with the goal (*input*).

The details of how GMS works would take us too far afield, suffice to say that it must not only monitor resource utilization, as does FSS, but it must also monitor *wait-time* in the run-queue, because wait time is directly proportional to queue length [1, 2]. If, for example, a workload is *over*-achieving with respect to its processing performance goal in the previous sample period (i.e., it got too many CPU cycles) , GMS will penalize that process by placing it closer to the tail of the run-queue than the head. As a consequence, it will take longer to return to the processor in the next sample period, and vice versa for *under*-achievers.

Of course, none of this comes for free. The cycles have to come from somewhere and the system administrator must prioritize the workloads in such a way that the favored work gets enough cycles to achieve the desired performance goals. Within the system performance reporting, there is a comparison of the specified goal and the actual value achieved. The ratio of these two metrics is called the *performance index*. You can expect that this will all become more automatic (*self-tuning*) in future releases of System Z.

The third feature, *cluster management*, is used to aggregate the resources of multiple mainframes in such a way that they can be managed as single platform. The IBM terminology for such a collection of mainframes is *Sysplex* and it requires an additional piece of hardware, called "the coupling facility," to act as both a memory interconnect and cache. At the hardware level, it shares a lot of features in common with the *Beowulf* platform [9]. Beyond the platform, however, the Sysplex is capable of running multiple applications, e.g., billing and database workloads, as though it were a single system from a management perspective.

The purpose of this section is not to suggest that Linux should aim to replace the mainframe, although there would be nothing wrong with that. Good system management facilities are currently missing in Linux and the mainframe gives us a good idea of what those future facilities could be.

9 Conclusion

A significant opportunity exists for Linux in the server marketplace. We have attempted to

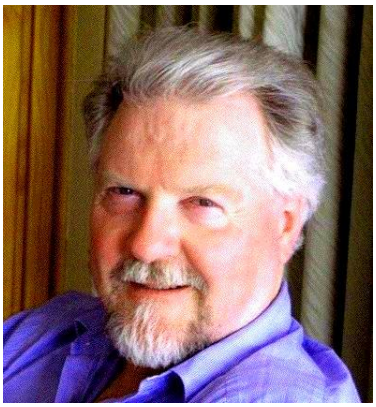
present a vision of how that opportunity might be met. Grasping that opportunity does not involve the typical adrenalin rush associated with faster speeds and feeds or more GigaHertz and MegaBytes. The real challenge is achieving more coherent system management through better O/S instrumentation than is available today on Linux servers. In this article, we have turned to the mainframe as a plausible role model for what true server management should look like. Whereas development on commercial UNIX servers is severely constrained by the need to show return-on-investment, Linux is relatively free of such financial demands.

But with freedom comes responsibility and, in the case of Linux, highly organized development. Prima facie, the required organization would seem to run counter to the motivations and culture of the Linux development community at large, so perhaps there is no will or consensus to do things differently and the aforementioned opportunity will be lost. On the other hand, there may be a role for university and other research groups to pool both organizational and research talent to drive the development process in the required way. A successful example from that arena is the *Beowulf* cluster project sponsored by NASA. Obviously, IBM has also sponsored some non-commercial Linux-related development projects.

Returning to the issues raised in Section 3. A certain faction of developers would like to keep Linux lean and mean. In this article, I have tried to survey what I see as an opportunity for server-side Linux. If the view of desktop performance is analogous to a *sports car*, then what I have been discussing here is closer to a vision of a Linux *train* or *jetliner*. Logically, it seems incompatible to have both modes of transport contained in the same code base.

In any case, there is a crying need for more coherent server management in the IT industry, but we are generally bogged down by fragmented proprietary interests on the part of commercial vendors. As a consequence, we are now 10–20 years behind mainframe system management. Linux, on the other hand, is relatively free of such concerns and therefore offers a unique way out of this impasse and a road forward to 21st century server management through better O/S instrumentation.

The Author



Neil Gunther, M.Sc., Ph.D., is an internationally recognized IT consultant who founded Performance Dynamics Company (www.perfdynamics.com) in 1994. Prior to that, Dr. Gunther held research and management positions at JPL/NASA (Voyager and Galileo missions), Xerox PARC and Pyramid/Siemens Technology. Currently, Dr. Gunther is working on Quantum Information Technologies. He is a member of the AMS, APS, ACM, CMG and IEEE.

References

[1] N. J. Gunther, “Load Average enträtselt und erweitert Leistungsdiagnostik,” Linux Magazin, p. 84, 08/2007

[2] N. J. Gunther, “Berechenbare Performance,” Linux Technical Review, Ausgabe 02, p. 112, 2007

[3] A. Kumar, “Multiprocessing with the Completely Fair Scheduler,” `http://www.ibm.com/developerworks/linux/library/l-cfs/`

[4] “Systems Management: Universal Measurement Architecture,” `http://www.opengroup.org/pubs/catalog/c427.htm`

[5] R. Pettit, “Formalizing Performance Metrics in Linux,” CMG Conference, p. 262, 1999

[6] N. J. Gunther, *Guerrilla Capacity Planning*, Springer-Verlag, 2007

[7] `http://www-03.ibm.com/servers/eserver/zseries/zos/wlm/`

[8] `http://www-03.ibm.com/systems/z/os/linux/`

[9] `http://en.wikipedia.org/wiki/Beowulf_%28computing%29`