LPI exam 301 prep, Topic 306: Capacity planning
Senior Level Linux Professional (LPIC-3) exam study guide

Skill Level: Intermediate

Sean Walberg (sean@ertw.com)
Senior Network Engineer
Freelance

15 Apr 2008

In this tutorial, Sean Walberg helps you prepare to take the Linux Professional Institute Senior Level Linux® Professional (LPIC-3) exam. In this last in a series of six tutorials, Sean walks you through monitoring your system resources, troubleshooting resource problems, and analyzing system capacity.

Section 1. Before you start

Learn what these tutorials can teach you and how you can get the most from them.

About this series

The Linux Professional Institute (LPI) certifies Linux system administrators at three levels: junior level (also called "certification level 1"), advanced level (also called "certification level 2"), and senior level (also called "certification level 3"). To attain certification level 1, you must pass exams 101 and 102. To attain certification level 2, you must pass exams 201 and 202. To attain certification level 3, you must have an active advanced-level certification and pass exam 301 ("core"). You may also need to pass additional specialty exams at the senior level.

developerWorks offers tutorials to help you prepare for the five junior, advanced, and senior certification exams. Each exam covers several topics, and each topic has a corresponding self-study tutorial on developerWorks. Table 1 lists the six topics and
corresponding developerWorks tutorials for LPI exam 301.

### Table 1. LPI exam 301: Tutorials and topics

<table>
<thead>
<tr>
<th>LPI exam 301 topic</th>
<th>developerWorks tutorial</th>
<th>Tutorial summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic 301</td>
<td>LPI exam 301 prep: Concepts, architecture, and design</td>
<td>Learn about LDAP concepts and architecture, how to design and implement an LDAP directory, and about schemas.</td>
</tr>
<tr>
<td>Topic 302</td>
<td>LPI exam 301 prep: Installation and development</td>
<td>Learn how to install, configure, and use the OpenLDAP software.</td>
</tr>
<tr>
<td>Topic 303</td>
<td>LPI exam 301 prep: Configuration</td>
<td>Learn how to configure the OpenLDAP software in detail.</td>
</tr>
<tr>
<td>Topic 304</td>
<td>LPI exam 301 prep: Usage</td>
<td>Learn how to search the directory and use the OpenLDAP tools.</td>
</tr>
<tr>
<td>Topic 305</td>
<td>LPI exam 301 prep: Integration and migration</td>
<td>Learn how to use LDAP as the source of data for your systems and applications.</td>
</tr>
<tr>
<td>Topic 306</td>
<td>LPI exam 301 prep: Capacity planning</td>
<td>(This tutorial.) Measure resources, troubleshoot resource problems, and plan for future growth. See the detailed objectives.</td>
</tr>
</tbody>
</table>

To pass exam 301 (and attain certification level 3), the following should be true:

- You should have several years experience with installing and maintaining Linux on a number of computers for various purposes.
- You should have integration experience with diverse technologies and operating systems.
- You should have professional experience as, or training to be, an enterprise-level Linux professional (including having experience as a part of another role).
- You should know advanced and enterprise levels of Linux administration including installation, management, security, troubleshooting, and maintenance.
- You should be able to use open source tools to measure capacity planning and troubleshoot resource problems.
- You should have professional experience using LDAP to integrate with UNIX® services and Microsoft® Windows® services, including Samba,
Pluggable Authentication Modules (PAM), e-mail, and Active Directory.

• You should be able to plan, architect, design, build, and implement a full environment using Samba and LDAP as well as measure the capacity planning and security of the services.

• You should be able create scripts in Bash or Perl or have knowledge of at least one system programming language (such as C).

To continue preparing for certification level 3, see the series developerWorks tutorials for LPI exam 301, as well as the entire set of developerWorks LPI tutorials.

The Linux Professional Institute doesn’t endorse any third-party exam preparation material or techniques in particular.

About this tutorial

Welcome to "Capacity planning," the last of six tutorials designed to prepare you for LPI exam 301. In this tutorial, you'll learn all about measuring UNIX resources, analyzing requirements, and predicting future resource requirements.

This tutorial is organized according to the LPI objectives for this topic. Very roughly, expect more questions on the exam for objectives with higher weights.

Objectives

Table 2 shows the detailed objectives for this tutorial.

Table 2. Capacity planning: Exam objectives covered in this tutorial

<table>
<thead>
<tr>
<th>LPI exam objective</th>
<th>Objective weight</th>
<th>Objective summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>306.1 Measure resource usage</td>
<td>4</td>
<td>Measure hardware and network usage.</td>
</tr>
<tr>
<td>306.2 Troubleshoot resource problems</td>
<td>4</td>
<td>Identify and troubleshoot resource problems.</td>
</tr>
<tr>
<td>306.3 Analyze demand</td>
<td>2</td>
<td>Identify the capacity demands of your software.</td>
</tr>
<tr>
<td>306.4 Predict future resource needs</td>
<td>1</td>
<td>Plan for the future by trending usage and predicting when your applications will need more resources.</td>
</tr>
</tbody>
</table>
Prerequisites

To get the most from this tutorial, you should have advanced knowledge of Linux and a working Linux system on which to practice the commands covered.

If your fundamental Linux skills are a bit rusty, you may want to first review the tutorials for the LPIC-1 and LPIC-2 exams.

Different versions of a program may format output differently, so your results may not look exactly like the listings and figures in this tutorial.

System requirements

To follow along with the examples in these tutorials, you'll need a Linux workstation with the OpenLDAP package and support for PAM. Most modern distributions meet these requirements.

Section 2. Measure resource usage

This section covers material for topic 306.1 for the Senior Level Linux Professional (LPIC-3) exam 301. This topic has a weight of 4.

In this section, learn how to:

- Measure CPU usage
- Measure memory usage
- Measure disk I/O
- Measure network I/O
- Measure firewalling and routing throughput
- Map client bandwidth usage

A computer relies on hardware resources: central processing unit (CPU), memory, disk, and network. You measure these resources to get an idea of how the computer is doing at the present moment and where any trouble spots may lurk. Looking at these measurements over a period of time, such as a few months, gives you some interesting history. It's often possible to extrapolate these readings to the future,
which helps you predict when one of the resources will run out. Alternatively, you can develop a mathematical model of your system, using historical information to validate the model, which you can then use to more accurately predict future usage.

Servers always require more than one hardware resource to complete a task. A task may require disk access to retrieve data, and memory to store it while the CPU processes it. If one of the resources is constrained, performance suffers. The CPU can't process information until it's read from disk; nor can the information be stored if memory is full. These concepts are related. As memory fills up, the operating system starts swapping other memory to disk. Memory is also taken away from buffers, which are used to speed up disk activity.

Understanding resources

Before measurements are useful, you must understand what you're measuring. Then you can begin to draw useful information about your system: current information, history, or future predictions.

CPU

The computer's CPU performs all the calculations an application needs, causes commands to be issued to disks and other peripherals, and takes care of running the operating system kernel. Only one task runs on the CPU at a time, whether the task is running the kernel or a single application. The current task can be interrupted by a hardware signal called an interrupt. Interrupts are triggered by external events, such as a network packet being received; or internal events, such as the system clock (called a tick in Linux). When an interrupt happens, the current running process is suspended, and a routine is run to determine what the system should do next.

When the currently running process has exceed its allotted time, the kernel can swap it out for another process using a procedure called a context switch. A process can be switched out before its allotted time if the process issues any I/O commands such as a read to disk. The computer is so much faster than the disk that the CPU can run other tasks while waiting for the suspended process's disk request to return.

When talking about the CPU of a Linux system, you should be concerned with several factors. The first is the percentage of time the CPU is idle compared to the time it's doing work (in reality, the CPU is always doing something—it's considered idle if no tasks are waiting to be executed). The CPU is running at maximum when the idle percentage is zero. The non-idle part of the CPU is split into system and user time, where system time refers to the time spent in the kernel, and user time is the time spent doing work requested by the user. The idle time is split into the time the kernel is idle because it has nothing to do, and the time it's idle because it's waiting on some bit of I/O.
Measuring these counters is tricky because getting an accurate number would require the CPU to spend all of its time determining what it's doing! The kernel checks the current status (system, user, iowait, idle) about 100 times per second and uses these measurements to calculate the percentages.

Another metric that Linux uses to convey CPU usage is the load average. This metric doesn't tie directly to the CPU utilization; it represents an exponential weighting of the number of tasks in the kernel's run queue for the past minute, 5 minutes, and 15 minutes. This metric is investigated more closely later.

Other things to consider about the kernel are the interrupt load and the context switches. There are no upper bounds on these figures, but the more interrupts and context switches are performed, the less time the CPU has to do the user's work.

**Memory**

The system has two types of memory: real memory and swap space. Real memory refers to the sticks of RAM on the motherboard. Swap space is a temporary holding spot that is used when the system tries to allocate more RAM than physically exists. In this situation, pages of RAM are swapped to disk to free up space for the current allocation. The data is swapped back to RAM when the data is needed again.

RAM can be used by applications, by the system, or not at all. The system uses RAM in two ways: as a buffer for raw disk blocks (incoming or outgoing) and as a file cache. The sizes of buffers and cache are dynamic so that memory can be given back to applications if needed. This is why most people see their Linux systems as having no free memory: the system has allocated the unused memory for buffers and cache.

Swap memory resides on disk. A lot of swapping slows things down and is a sign that the system is out of RAM.

**Disk**

Disk is where long-term data is stored, such as on a hard drive, a flash disk, or tape (collectively referred to as block devices). One exception is a RAM disk, which behaves like a block device but resides in RAM; this data is gone when the system shuts down. The most prevalent form of disk is the hard drive, so the discussion of disk in this tutorial focuses on this medium.

Two categories of measurements are used to describe disk: space and speed. The free space on a disk refers to the number of bytes on the disk that are available for use. The overhead on a disk includes any space used by the file system or that is otherwise unavailable for use. Keep in mind that most manufacturers report disks in terms of 1,000,000,000-byte gigabytes, whereas your operating system uses the base 2 value of 1,073,741,824; this results in a 93% "loss" for the consumer. This isn't overhead, but if you don't account for it, your calculations will be incorrect.
The second metric of a disk is speed, which measures how fast data is returned from the disk. When the CPU issues a request, several things must come together to get the data back to the CPU:

1. The kernel puts the request in a queue, where it waits for its turn to be sent to disk (wait time).
2. The command is sent to the disk controller.
3. The disk seeks the disk heads to the required block (seek time).
4. The disk heads read the data from disk.
5. The data is returned to the CPU.

Each of these steps is measured differently, or sometimes not at all. The service time encompasses the last three steps and represents how long a request takes to service once the request has been issued. The wait time represents the entire procedure, which includes the time in queue and the service time.

One bit of optimization the kernel performs is to reorder and merge requests in the queue from step 1 to minimize the number of disk seeks. This is called an elevator, and several different algorithms have been used over the years.

**Network**

Linux plays two broad roles with respect to the network: a client, where packets are sent and received by applications on the server; and a router (or firewall, or bridge). Packets are received on one interface and sent out on another (perhaps after some filtering or inspection has happened).

Networks are most often measured in terms of bits per second (or kilobits, megabits, or gigabits) and in packets per second. Measuring packets per second is often less useful because computers have a fixed overhead per packet, resulting in poorer throughput at smaller packet sizes. Don't confuse the speed of the network card (100Mbit/sec, or gigabit) with the expected speed of data transfers from or through the machine. Several outside factors come into play, including latency and the remote side of the connection, not to mention tuning on the server.

**Queues**

Queues don't fit well with the other resources, but they appear so often in performance monitoring that must be mentioned. A queue is a line in which requests wait until they're processed. The kernel uses queues in a variety of ways, from the run queue that holds the list of processes to be run, to disk queues, network queues, and hardware queues. Generally, a queue refers to a spot in memory that the kernel
uses to keep track of a particular set of tasks, but it can also refer to a piece of memory on a hardware component that is managed by the hardware.

Queues appear two ways in performance tuning. First, when too much work comes into a queue, any new work is lost. For example, if too many packets come into a network interface, some get dropped (in networking circles, this is caused a tail drop). Second, if a queue is being used excessively (or, sometimes, not enough), then another component isn't performing as well as necessary. A large number of processes in the run queue on a frequent basis may mean the CPU is overloaded.

Measure performance

Several tools are available to measure performance on a Linux box. Some of these tools measure CPU, disk, memory, and network directly, and others show indicators such as queue usage, process creation, and errors. Some tools show instantaneous values, and some show values that have been averaged over a period of time. It's equally important to understand both how the measurement was taken and what is being measured.

vmstat

vmstat is a helpful tool for showing the most-often-used performance metrics in real time. The most important thing to understand about vmstat is that the first display it produces represents the average values since the system was booted, which you should generally ignore. Specify a repeat time (in seconds) at the command line to have vmstat repeatedly report the information using current data. Listing 1 shows the output of vmstat 5.

Listing 1. The output of vmstat 5

```
# vmstat 5
procs -----------memory---------- ---swap-- -----io---- --system--
-----cpu-----
  r  b  swpd  free  buff  cache  si  so  bi  bo  in  cs  us  sy  id  wa  st
 0  3  17780  10304  18108  586076  0  0  2779  332  1  1  3  4
 76 17  0
 1  2  17780  10088  19796  556172  0  0  7803  3940  2257  4093  25  28
14 34  0
 0  2  17780  9568  19848  577496  0  0  18060  1217  1610  910  0  3
48 49  0
 0  0  17780  51696  20804  582396  0  0  9237  3490  1736  839  0  3
55 41  0
```

Listing 1 shows the output of the vmstat command, with measurements taken every 5 seconds. The first line of values represents the average since the system booted, so it should be ignored. The first two columns refer to processes. The number under the r heading is the number of processes in the run queue at the time...
of the measurement. Processes in the run queue are waiting on the CPU. The next column is the number of processes blocked on I/O, meaning they're sleeping until some piece of I/O is returned, and they can't be interrupted.

The columns under the memory heading are instantaneous measurements about the system memory and are in kilobytes (1024 bytes). swpd is the amount of memory that has been swapped to disk. free is the amount of free memory that isn't used by applications, buffers, or cache. Don't be surprised if this number is low (see the discussion on free for more information about what free memory really is). buff and cache indicate the amount of memory devoted to buffers and cache. Buffers store raw disk blocks, and cache stores files.

The first two categories are instantaneous measurements. It's possible that for a brief period, all free memory was consumed but returned before the next interval. The rest of the values are averaged over the sampling period.

swap is the average amount of memory swapped in from disk (si) and out to disk (so) per second; it's reported in kilobytes. io is the number of disk blocks per second read in from all block devices and sent out to block devices.

The system category describes the number of interrupts per second (in) and context switches (cs) per second. Interrupts come from devices (such as a network card signaling the kernel that a packet is waiting) and the system timer. In some kernels, the system timer fires 1,000 times per second, so this number can be quite high.

The final category of measurements shows what's going on with the CPU, reported as a percent of total CPU time. These five values should add to 100. us is the average time the CPU spent on user tasks over the sampling period, and sy is the average time the CPU spent on system tasks. id is the time the CPU was idle, and wa is the time the CPU was waiting for I/O (Listing 1 was taken from a heavily I/O bound system, you can see that 34-49% of the CPU's time is spent waiting for data to return from disk). The final value, st (the steal time) is for servers running a hypervisor and virtual machines. It refers to the percentage of time the hypervisor could have run a virtual machine but had something else to do.

As you can see from Listing 1, vmstat provides a wealth of information across a broad spectrum of metrics. If something is going on while you're logged in, vmstat is an excellent way to narrow down the source.

vmstat can also show some interesting information about disk usage on a per-device basis, which can shed more light on the swap and io categories from Listing 1. The -d parameter reports some disk statistics, including the total number of reads and writes, on a per-disk basis. Listing 2 shows part of the output from vmstat -d 5 (with unused devices filtered out).
Listing 2. Using vmstat to show disk usage

<table>
<thead>
<tr>
<th>disk</th>
<th>reads</th>
<th>total merged sectors</th>
<th>ms</th>
<th>writes</th>
<th>total merged sectors</th>
<th>ms</th>
<th>cur</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-----IO------sec</td>
<td>hda</td>
<td>186212</td>
<td>28646</td>
<td>3721794</td>
<td>737428</td>
<td>246503</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hdd</td>
<td>181471</td>
<td>27062</td>
<td>3582080</td>
<td>789856</td>
<td>246450</td>
</tr>
</tbody>
</table>

Each disk is displayed on a separate line, and the output is broken down into reads and writes. Reads and writes are further split into the total numbers of requests issued, how many requests were merged in the disk elevator, the number of sectors read from or written to, and the total service time. All these numbers are counters, so they will increase until the next reboot, as opposed to the average values seen without the `-d` option.

The final group of measurements in Listing 2, under the **IO** heading, show the current number of I/O operations in progress for the disk and the total number of seconds spent in I/O since boot.

Listing 2 shows that read volume is similar across the two disks and that write volume is almost identical. These two disks happen to form a software mirror, so this behavior is expected. The information from Listing 2 can be used to indicate slower disks or disks with higher usage than others.

**iostat**

Closely tied to the `vmstat -d` example from Listing 2 is `iostat`. This command provides details about disk usage on a per-device basis. `iostat` improves on `vmstat -d` by giving more details. Just as with `vmstat`, you can pass a number to `iostat` that indicates the refresh interval. Also, as with `vmstat`, the first output represents the values since the system was started and therefore is usually ignored. Listing 3 shows the output of `iostat` with 5-second intervals.

Listing 3. Output of the iostat command

```
$ iostat 5
Linux 26.20-1.3002.fc6xen (bob.ertw.com) 02/08/2008
avg-cpu:  %user  %nice  %system  %iowait  %steal  %idle
      0.85  0.13  0.35  0.75  0.01  97.90
Device:     tps  Blk_read/s  Blk_wrtn/s  Blk_read  Blk_wrtn
hda  1.86  15.24  13351  4549745  38981340
  8456728  0  2583
hdd  1.85  14.69  133.51  4570088  38981624  8855516
  0  2652
```

The first part of every measurement interval shows the CPU usage, which is also shown by `vmstat`. However, two decimal places are presented here. The second
part of the output shows all the block devices on the system (to limit the number of
devices shown, pass the names of the devices on the command line, such as
iostat 5 hda sda). The first column, tps, represents the transfers per second to
the device after the requests have been merged by the elevator. The sizes of the
transfers aren’t specified. The last four columns deal in 512-byte blocks and show
the blocks read per second, written per second, total blocks read, and total blocks
written, respectively. If you’d rather see values reported in kilobytes or megabytes,
use -k or -m, respectively. The -p option displays details down to the partition level,
if you need that data.

You can get a great deal more information by using the -x parameter, which is
shown in Listing 4. Listing 4 also limits the output to one drive. The formatting was
adjusted to fit page-width constraints.

Listing 4. Extended information from iostat

```bash
# iostat -x 5 hda
.... CPU information removed ...
Device: rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz
hda  16669.31  1.49  756.93  1.49  139287.13  27.72  18369
  avgqu-sz  await  svctm  %util
       1.58     208   1.28  96.83
```

The first six values are concerned with reads and writes per second. rrqm/s and
wrqm/s refer to the number of read and write requests that were merged. By
contrast, r/s and w/s represent the number of reads and writes sent to disk.
Therefore, the percentage of disk requests merged is 16669 / (16669 + 757) = 95%.
rsec/s and wsec/s show the read and write rate in terms of sectors per second.

The next four columns display information about disk queues and times. avgrq-sz is
the average size of requests issued to the device (in sectors). avgqu-sz is the
average length of the disk queue over the measurement interval. The await is the
average wait time (in milliseconds), which represents the average time a request
takes from being sent to the kernel to the time it’s returned. svctm is the average
service time (in milliseconds), which is the time a disk request takes from when it’s
out of the queues and sent to disk to the time it’s returned.

The final value, %util, is the percentage of time the system was performing I/O on
that device, also referred to as the saturation. The 96.83% reported in Listing 4
shows that the disk was almost at capacity during that time.

mpstat

mpstat reports detailed information about the CPU (or all CPUs in a multiprocessor
machine). Much of this information is reported by iostat and vmstat in some
form, but mpstat provides data for all processors separately. Listing 5 shows
mpstat with 5-second measurement intervals. Unlike with iostat and vmstat,
you shouldn't ignore the first line.

**Listing 5. Showing CPU information with mpstat**

```
# mpstat -P 0 5
Linux 2.6.20-1.3002.fc6xen (bob.ertw.com) 02/09/2008
09:45:23 PM CPU %user %nice %sys %iowait %irq %soft %steal %idle intr/s
09:45:25 PM 0 77.61 21.89 0.00 0.00 0.50 0.00 0.00 155.22
09:45:27 PM 0 68.16 30.85 1.00 0.00 0.00 0.00 0.00 154.73
```

The addition of `-P 0` specifies that the first CPU (starting at 0) should be shown. You can also specify `-P ALL` for all CPUs separately. The fields returned by `mpstat` are as follows:

- **%user**: The percentage of time spent in user tasks, excluding the nice tasks
- **%nice**: The percentage of time spent in nice (lower priority) user tasks
- **%sys**: The percentage of time spent in kernel tasks
- **%iowait**: The percentage of time spent waiting for I/O while idle
- **%irq**: The percentage of time servicing hardware interrupts
- **%soft**: The percentage of time spent in software interrupts
- **%steal**: The percentage of time the hypervisor stole from a virtual machine
- **intr/s**: The average number of interrupts per second

**pstree**

Understanding which processes spawned another process is helpful when you're tracking down resource usage. One way to find this is to use the output of `ps -ef` and use the parent processid to work your way back to PID 1 (`init`). You can also use `ps -efjH`, which sorts the output into a parent-child tree, and include CPU time usage.

A utility called `pstree` prints the process tree in a more graphical format and also rolls multiple instances of the same process into one line. Listing 6 shows the output of `pstree` after it's passed the PID of the Postfix daemon.

**Listing 6. pstree output**
The master process, cleverly called **master**, has spawned several other processes such as **anvil**, **cleanup**, and **local**. The last two lines of the output are of the format $N^*[something]$, where *something* is the name of a process, and $N$ is the number of children by that name. If *something* was enclosed in curly brackets ({{}}) in addition to the square brackets ([[]]), that would indicate $N$ threads running ($ps$ doesn't normally show threads unless you use -l).

**w, uptime, and top**

These utilities are grouped together because they're the first utilities people tend to reach for when investigating a problem. Listing 7 shows the output of the **w** command.

**Listing 7. Output of the w command**

```
# w
12:14:15 up 33 days, 15:09, 2 users, load average: 0.06, 0.12, 0.09
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
root tty2 - 17Jan08 18days 0.29s 0.04s login --
root
root pts/0 bob Sat22 0.00s 0.57s 0.56s -bash
```

The first line of **w** provides the bulk of the information. The first part, "12:14:15 up 33 days, 15:09" gives the current time, followed by the uptime of 33 days, 15 hours, and 9 minutes. The second part, "2 users", gives the number of logged-in users. The final part is the load average, given as a 1-minute, 5-minute, and 15-minute averaging.

The load average is a weighted average of the number of processes in the run queue over a given period of time. The higher the load average, the more processes are trying to contend for the CPUs. Load averages aren't normalized to the number of CPUs, meaning that the load average and the number of CPUs aren't related.

To use the load average, you must also understand the weighting. The load average is updated every 5 seconds, with the older information playing a less prominent role in the calculation. If your system were to go immediately from 0 processes in the run queue to 1 process, a graph of the 1-minute load average over the next minute would be not a straight line but a curve that rises quickly at first and then tapers off until the 60-second mark. For a more detailed look at how the load average calculation is done, see the **Resources** section.
The practical application of weighting the load average is that variations in the actual load at the time of measurement are smoothed out; but the current state is reflected more in the numbers, especially the 1-minute average.

After the first line comes a list of logged-in users, including their login time, their location, and information about CPU usage. The first user, root, is logged in from tty2 (a local console) and has been idle for 18 days. The second user is again root, but logged in over the network and currently at the shell. The JCPU and PCPU columns give you an idea of how much CPU time the user has used; the first column includes jobs in the past, whereas PCPU is for the process the user is currently using.

The output of `uptime` shows exactly the same first line of `w`, but no information about the users. In practical terms, `w` is the more helpful of the two because of the additional information about users and because it's shorter to type!

Another popular command is `top`, which shows a continuously updating list of the top processes (sorted by either memory or CPU usage), in addition to some other metrics. Figure 1 shows a screenshot of `top` in action.

**Figure 1. top in action**

[![Top in action screenshot](image.png)](image.png)

The first line shows the same thing as `uptime`, such as the uptime and load average. The second line is the number of processes. *Running* processes are in the run queue; *sleeping* processes are waiting for something to wake them up. A *stopped* process has been paused, likely because it's being traced or debugged. A
zombie process has exited, but the parent process hasn’t acknowledged the death.

**Something doesn't add up**

VIRT = RES + SWAP, which means that SWAP = VIRT - RES. Looking at PID 13435, you can see that VIRT is 164m, and RES is 76m, meaning SWAP must be 88m. However, the swap stats from the top of the screen indicate that only 144K of swap are used! This can be verified by using the f key when inside top and enabling more fields, such as swap.

As it turns out, swap means more than just pages swapped to disk. An application's binary and libraries need not stay in memory the whole time. The kernel can mark some of the memory pages as unnecessary at the time; but because the binary is at a known location on disk, there is no need to use the swap file. This is still counted as swap because the part of the code isn't resident. In addition, memory can be mapped to a disk file by the application. Because the whole size of the application (VIRT) includes the mapped memory, but it isn't resident (RES), it's counted as swap.

The third line gives the CPU usage: in order, you see user, system, nice, idle, I/O wait, hardware interrupt, software interrupt, and steal time. These numbers are percentages of time in the last measurement interval (3 seconds by default).

The last two lines of the top section show memory statistics. The first gives information about real memory; in Figure 1, you can see the system has 961,780K of RAM (after the kernel overhead). All but 6,728K have been used, with around 30MB of buffers and 456M of cache (cache is displayed at the end of the second line). The second line displays the swap usage: the system has almost 3G of swap, with only 144K used.

The rest of the screen shows information about the currently running processes. top displays as many processes as possible to fill the size of the window. Each process gets its own line, and the list is updated every measurement interval with the tasks that used the most CPU at the top. The columns are as follows:

- **PID**: The processid of the process
- **USER**: The effective username of the process (if the program uses `setuid(2)` to change the user, the new user is displayed)
- **PR**: The priority of the task, used by the kernel to determine which process gets the CPU first
- **NI**: The nice level of the task, set by the system administrator to influence which processes get the CPU first
- **VIRT**: The size of the process's virtual image, which is the sum of the space used by RAM (the resident size) and the size of the data in swap
(swapped size)

- **RES**: The resident size of the process, which is the amount of real RAM used by your process
- **SHR**: The amount of memory shared by the application, such as SysV shared memory or dynamic libraries (*.so)
- **S**: The state, such as sleeping, running, or zombie
- **%CPU**: The percentage of CPU used in the last measurement interval
- **%MEM**: The percentage of RAM (excluding swap) used when last measured
- **TIME+**: The time in minutes:seconds:hundredths used by the process
- **COMMAND**: The name of the command running

*top* provides a fast way to see which processes are using the most CPU and also gives you a good dashboard view of the system's CPU and memory. You can have *top* sort by memory usage by typing **M** in the *top* display.

### free

After you've seen *top*, you should understand *free* right away. Listing 8 shows the output of *free*, using the **-m** option to report all values in megabytes.

**Listing 8. Using the free command**

<table>
<thead>
<tr>
<th></th>
<th>total</th>
<th>used</th>
<th>free</th>
<th>shared</th>
<th>buffers</th>
</tr>
</thead>
<tbody>
<tr>
<td>cached</td>
<td>939</td>
<td>904</td>
<td>34</td>
<td>0</td>
<td>107</td>
</tr>
<tr>
<td>Mem:</td>
<td>310</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>310</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+/- buffers/cache:</td>
<td>486</td>
<td>452</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swap:</td>
<td>2847</td>
<td>0</td>
<td>2847</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*free* lays out the memory usage in a few directions. The first line shows the same information you saw for *top*. The second line represents used and free memory without considering buffers and cache. In Listing 8, 452M of memory is free for an application to use; this memory will come from the free memory (34M), buffers (107M), or cache (310M).

The final line shows the same swap statistics as *top*.

### Show network statistics

Getting network statistics is less direct than for CPU, memory, and disk. The primary method is to read counters from `/proc/net/dev`, which reports transfers per interface on both a packet and byte basis. If you want a per-second value, you must calculate...
it yourself by dividing the difference between two successive measurements by the
interval. Alternatively, you can use a tool like `bwm` to automate the collection and
display of total bandwidth. Figure 2 shows `bwm` in action.

**Figure 2. Using the bwm command**

![bwm output](image)

`bwm` shows interface usage in a variety of ways. Figure 2 shows the instantaneous
rate every half second, although 30-second averages, maximum bandwidth, and
counting bytes are available. You can see from Figure 2 that eth0 is receiving about
20K/sec of traffic, which seems to be coming from vif0.0. If bytes per second aren't
what you're looking for, you can cycle between bits, packets, and errors with the `u`
key.

To get more details about which hosts are responsible for the traffic, you need
`iftop`, which provides a `top`-like interface for your network traffic. The kernel
doesn't provide this information directly, so `iftop` uses the pcap library to inspect
packets on the wire, which requires root privileges. Figure 3 shows `iftop` in action,
when attached to the eth2 device.

**Figure 3. The output of iftop -i eth2**
iftop shows the top talkers on your network. By default, each conversation takes two lines: one for the sending half and the other for the receiving half. Looking at the first conversation from mybox to pub1.kernel.org, the top row shows the traffic sent from mybox, and the second line shows the traffic received by mybox. The numbers to the right indicate the average traffic over the last 2 seconds, 10 seconds, and 40 seconds, respectively. You can also see a black bar overlaying the hostnames, which is a visual indication of the 10-second average (the scale is displayed at the top of the screen).

Looking more closely at Figure 3, the first transfer is probably a download because of the large amount of traffic received (averaging around half a megabit per second over the last 10 seconds) compared to the small amount of upload traffic. The second talker has an equal amount of traffic, which has been relatively steady at around 75-78k/sec. This is a G.711 voice call through les.net, my VoIP provider. The third transfer shows a 128K download with a small upload: it was an Internet radio stream.

The choice of interface you attach to is important. Figure 3 uses the outside interface on a firewall, which sees all packets after they have passed through IP masquerading. This causes the internal address to be lost. Using a different interface, such as an internal interface, would preserve this information.

sar

sar is the topic of an entire article (see the Resources section). sar measures dozens of key metrics every 10 minutes and provides a method to retrieve the measurements. You can use the preceding tools to determine "what is going on now?"; sar answers, "what happened this week?" Note that sar purges its data to keep only the last 7 days of data.
You must configure data collection by adding two lines to root's crontab. Listing 9 shows a typical crontab for `sar`.

### Listing 9. root's crontab for sar data collection

```bash
# Collect measurements at 10-minute intervals
0,10,20,30,40,50 * * * * /usr/lib/sa/sa1 -d 1 1
# Create daily reports and purge old files
0 0 * * * /usr/lib/sa/sa2 -A
```

The first line executes the `sa1` command to collect data every 10 minutes; this command runs `sadc` to do the actual collection. This job is self-contained: it knows which file to write to and needs no other configuration. The second line calls `sa2` at midnight to purge older data files and collect the day's data into a readable text file.

It's worth checking to see how your system runs `sar` before you rely on the data. Some systems disable collection of disk statistics; to fix this, you must add `-d` to the call to `sa1` (Listing 9 has this added).

With some data collected, you can now run `sar` without any options to see the day's CPU usage. Listing 10 shows part of the output.

### Listing 10. Sample output from sar

```
[root@bob cron.d]# sar | head
Linux 2.6.20-1.3002.fc6xen (bob.ertw.com) 02/11/2008
12:00:01 AM CPU %user %nice %system %iowait %steal %idle
12:10:01 AM all 0.18 0.00 0.18 3.67 0.01 95.97
12:20:01 AM all 0.08 0.00 0.04 0.02 0.01 99.85
12:30:01 AM all 0.11 0.00 0.03 0.02 0.01 99.82
12:40:01 AM all 0.12 0.00 0.02 0.01 0.01 99.83
12:50:01 AM all 0.11 0.00 0.03 0.05 0.01 99.81
01:00:01 AM all 0.12 0.00 0.02 0.02 0.01 99.83
01:10:01 AM all 0.11 0.00 0.02 0.03 0.01 99.83
```

The numbers shown in Listing 10 should be familiar by now: they're the various CPU counters shown by `top`, `vmstat`, and `mpstat`. You can view much more information by using one or more of the command-line parameters shown in Table 3.

### Table 3. A synopsis of sar options

<table>
<thead>
<tr>
<th>Option</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-A</code></td>
<td><code>sar -A</code></td>
<td>Displays everything. Unless you're dumping</td>
</tr>
</tbody>
</table>
this result to a text file, you probably don't need it. If you do need it, this process is run nightly as part of sa2 anyway.

<table>
<thead>
<tr>
<th>Option</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-b</td>
<td>sar -b</td>
<td>Shows transactions and blocks sent to, and read from, block devices, much like iostat.</td>
</tr>
<tr>
<td>-B</td>
<td>sar -B</td>
<td>Shows paging (swap) statistics such as those reported by vmstat.</td>
</tr>
<tr>
<td>-d</td>
<td>sar -d</td>
<td>Shows disk activity much like iostat -x, which includes wait and service times, and queue length.</td>
</tr>
<tr>
<td>-n</td>
<td>sar -n DEV or sar -n NFS</td>
<td>Shows interface activity (like bwmon) when using DEV, or NFS client statistics when using the NFS keyword (use the NFSD keyword for the NFS server daemon stats). The EDEV keyword shows error information from the network cards.</td>
</tr>
<tr>
<td>-q</td>
<td>sar -q</td>
<td>Shows information about the run queue and total process list sizes, and load averages, such as those reported by vmstat and uptime.</td>
</tr>
<tr>
<td>-r</td>
<td>sar -r</td>
<td>Shows information about memory, swap, cache, and buffer usage (like free).</td>
</tr>
<tr>
<td>-f</td>
<td>sar -f /var/log/sa/sall</td>
<td>Reads information from a different file. Files are named after the day of the month.</td>
</tr>
<tr>
<td>-s</td>
<td>sar -s 08:59:00</td>
<td>Starts displaying information at the first measurement after the given time. If you specify 09:00:00, the first measurement will be at 09:10, so subtract a minute from the time you want.</td>
</tr>
</tbody>
</table>
You may also combine several parameters to get more than one report, or a different file with a starting and an ending time.

**df**

Your hard disks are a finite resource. If a partition runs out of space, be prepared for problems. The `df` command shows the disk-space situation. Listing 11 shows the output of `df -h`, which forces output to be in a more friendly format.

**Listing 11. Checking disk space usage with df -h**

```bash
$ df -h
Filesystem Size Used Avail Use% Mounted on
/dev/mapper/VolGroup00-LogVol00 225G 169G 44G 80% /
/dev/hda1 99M 30M 64M 32% /boot
tmpfs 474M 0 474M 0% /dev/shm
```

Listing 11 shows a single file system on the root that is 225G in size, with 44G free. The /boot partition is 99M with 64M free. tmpfs is a special file system and doesn't refer to any particular device. If a partition is full, you see no available space and 100% usage.

### Section 3. Troubleshoot resource problems

This section covers material for topic 306.2 for the Senior Level Linux Professional (LPIC-3) exam 301. This topic has a weight of 4.

In this section, learn how to:

- Match/correlate system symptoms with likely problems
- Identify bottlenecks in a system

The previous section showed how to display different performance counters within the Linux system. It's now time to apply these commands to solving resource-related
problems in your systems.

Troubleshooting methodology

Lay out your strategy for problem solving before getting into the details of resource constraints and your system. The many strategies for problem solving boil down to four steps:

1. Identify the symptoms.
2. Determine the root cause.
3. Implement a fix.
4. Evaluate the results.

Identify the symptoms

The first step to solving a problem is to identify the symptoms. An example of a symptom is "e-mail is slow" or "I'm out of disk space." The symptoms are causing your users to complain, and those users won't be happy until the symptoms go away. Don't confuse the symptoms with the problem, though: most often, the problem is different than what your users are reporting, although the problem is causing the symptoms.

Once you've collected the symptoms, try to quantify the complaint and clarify the conditions under which it occurs. Rather than the e-mail system being slow, you may learn that e-mails used to be received within seconds of being sent but now take hours. And a user who is out of disk space has to be doing something, such as saving a file or processing a batch job.

This final step of quantifying the complaint has two purposes. The first is to let you reproduce the problem, which will help later in determining when the problem has been solved. The second purpose is to get more details from the user that will help you determine the root cause. After learning that a job that once took 5 minutes to execute now takes an hour, you might inquire about the nature of the job. This might lead you to learn that it pulls information from a database on another server, which you must include in the scope of your search for the root cause.

Determine the root cause

Determining the root cause involves using the commands learned in the Measure resource usage section to find the cause of the problem. To do so, you must investigate the resources, such as CPU, memory, disk, and network. Ideally, you'll be able to collect data while the problem is occurring with real-time tools like
vmstat, iostat, and top. If not, something that produces historical information such as sar may have to do.

If the problem is resource related, then one of two results will appear: one (or more) of your resources will be at 100% utilization, and the cause should be obvious; or nothing is obviously being overused.

In the second case, you should refer to a baseline. The baseline is a set of reference data you can use to compare what's "normal" to what you're seeing. Your baseline might be a series of graphs, or archived sar reports showing normal activity. The baseline will also be helpful later when you learn about predicting growth.

As you use the administration tools, you'll start to develop a picture of the problem's root cause. It may be that your mail server is stuck on a message, causing it to stop processing other messages. Or perhaps a batch job is consuming all the CPU on the system.

At this point, you must be careful that you have properly identified the root cause of the problem. An application that generates a huge logfile may have caused you to run out of space. If you identify the logfile as the root cause and decided to delete it, the application may still fill up the disk at some point in the future.

**Implement a fix**

You'll often have several ways to fix a problem. Take, for instance, a batch job that is consuming all CPU resources. If you kill the job, then the user running the job will probably lose his or her work, although the other users will have their server back. You may decide to renice the process to give other processes more time on the CPU. This is usually a judgment call, depending on the needs of the business and the urgency of the situation.

**Evaluate the results**

Once you've implemented your solution, you must go back and check to see if the problem was solved. Are e-mails delivered near-instantly? Are users able to log in? If not, then you must step back and look at the root cause again, which will lead to another fix that must be evaluated. If your fix failed, you must also check to see that you didn't make things worse!

After the problem is fixed, determine if you need to take any longer-term actions. Do you have to consider a bigger disk, or moving a user's batch job to another host? Unexplained processes on a machine mat prompt you to do a more in-depth security check of the server to make sure it hasn't been exploited.

**Compound problems**
Some performance problems are obvious. A user complains that something is running slowly: you log in and fire up `top`. You see a process unnecessarily hogging the CPU: you kill it, and the system returns to normal. After showering you with praise, your boss gives you a raise and the rest of the day off. (OK, maybe the last part is made up.)

What happens when your problem isn't obvious? Sometimes problems are caused by more than one thing, or a symptom is caused by something that may seem unrelated at first.

**The swap spiral**

Memory is fast, and you probably have lots of it in your system. But sometimes an application needs more memory than the system has, or a handful of processes combined end up using more memory than the system has. In this case, virtual memory is used. The kernel allocates a spot on disk and swaps the resident memory pages to disk so that the active application can use it. When the memory on disk is needed, it's brought back to RAM, optionally swapping out some other memory to disk to make room.

The problem with that process is that disk is slow. If you briefly dip into swap, you may not notice. But when the system starts aggressively swapping memory to disk in order to satisfy a growing demand for memory, you've got problems. You'll find your disk I/O skyrocketing, and it will seem that the system isn't responding. In fact, the system probably isn't responding, because your applications are waiting for their memory to be transferred from disk to RAM.

UNIX admins call this the *swap spiral* (or sometimes, more grimly, the *swap death spiral*). Eventually, the system grinds to a halt as the disks are running at capacity trying to swap memory in and out. If your swap device is on the same physical disk as data, things get even worse. Once your application makes it onto the CPU and issues an I/O request, it has to wait longer while the swap activity is serviced.

The obvious symptom of the swap spiral is absurdly long waits to do anything, even getting the uptime. You also see a high load average, because many processes are in the run queue due to the backed-up system. To differentiate the problem from a high-CPU problem, you can check `top` to see if processes are using the CPU heavily, or you can check `vmstat` to see if there is a lot of swap activity. The solution is usually to start killing off processes until the system returns to order, although depending on the nature of the problem, you may be able to wait it out.

**Out of disk space**

Applications aren't required to check for errors. Many applications go through life assuming that every disk access executes perfectly and quickly. A disk volume that fills up often causes applications to behave in weird ways. For example, an
application may consume all available CPU as it tries to do an operation over and over without realizing it's not going to work. You can use the `strace` command to see what the application is doing (if it's using system calls).

Other times, applications simply stop working. A Web application may return blank pages if it can't access its database.

Logging in and checking your available disk (with `du`) is the quickest way to see if disk space is the culprit.

**Blocked on I/O**

When a process requests some form of I/O, the kernel puts the process to sleep until the I/O request returns. If something happens with the disk (sometimes as part of a swap spiral, disk failure, or network failure on networked file systems), many applications are put to sleep at the same time.

A process that's put to sleep can be put into an interruptible sleep or an uninterpretable sleep. The former can be killed by a signal, but the second can't. Running `ps aux` shows the state. Listing 12 shows one process in uninterpretable sleep and another in interruptible sleep.

### Listing 12. Two processes in a sleep state

<table>
<thead>
<tr>
<th>User</th>
<th>PID</th>
<th>CPU</th>
<th>Mem</th>
<th>Swap</th>
<th>FS</th>
<th>State</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache</td>
<td>26575</td>
<td>0.2</td>
<td>19.6</td>
<td>132572</td>
<td>50104</td>
<td>?</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>/usr/sbin/httpd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Feb13</td>
</tr>
<tr>
<td>root</td>
<td>8381</td>
<td>57.8</td>
<td>0.2</td>
<td>3844</td>
<td>532</td>
<td>pts/1</td>
<td>D</td>
</tr>
</tbody>
</table>

The first process in Listing 12, `httpd`, is in an interruptible sleep state, indicated by the letter `S` just after the question mark. The second process, `dd`, is in an uninterpretable sleep state. Uninterpretable sleeps are most often associated with hard disk accesses, whereas interruptible sleeps are for operations that take comparably longer to execute, such as NFS and socket operations.

If you find a high load average (meaning a lot of processes in the run queue) and a lot of processes in an uninterpretable sleep state, then you may have a problem with hard drive I/O, either because the device is failing or because you're trying to get too many reads/writes out of the drive at a time.

---

**Section 4. Analyze demand**

This section covers material for topic 306.3 for the Senior Level Linux Professional
(LPIC-3) exam 301. This topic has a weight of 2.

In this section, learn how to:

- Identify capacity demands
- Detail capacity needs of programs
- Determine CPU/memory needs of programs
- Assemble program needs into a complete analysis

Fixing immediate problems is a key task for the system admin. Another task involves analyzing how systems are currently performing in the hope that you can foresee resource constraints and address them before they become a problem. This section looks at analyzing the current demand, and the next section builds on that to predict future usage.

You can use two approaches to analyze current demand: measure the current demand over a period of time (like a baseline), or model the system and come up with a set of parameters that makes the model reflect current behavior. The first approach is easier and reasonably good. The second is more accurate but requires a lot of work. The real benefit of modeling comes when you need to predict future behavior. When you have a model of your system, you can change certain parameters to match growth projections and see how performance will change.

In practice, both of these approaches are used together. In some cases, it’s too difficult to model a particular system, so measurements are the only basis on which to base demand and growth projections. Measurements are still required to generate models.

Model system behavior

The activity in a computer can be modeled as a series of queues. A queue is a construct where requests come in one end and are held until a resource is available. Once the resource is available, the task is executed and exits the queue.

Multiple queues can be attached together to form a bigger system. A disk can be modeled as a queue where requests come in to a buffer. When the request is ready to be serviced, it’s passed to the disk. This request generally comes from the CPU, which is a single resource with multiple tasks contending for the use of the CPU. The study of queues and their applications is called queuing theory.

The book *Analyzing Computer System Performance with Perl::PDQ* (see Resources for a link) introduces queuing theory and shows how to model a computer system as a series of queues. It further describes a C library called PDQ and an associated
Perl interface that lets you define and solve the queues to give performance estimates. You can then estimate the result of changes to the system by changing parameters.

**Introducing queues**

Figure 4 shows a single queue. A request comes in from the left and enters the queue. As requests are processed by the circle, they leave the queue. The blocks to the left of the circle represent the queued objects.

**Figure 4. A simple queue**

The queue’s behavior is measured in terms of times, rates, and sizes. The *arrival rate* is denoted as $\text{Lambda} (\Lambda)$ and is usually expressed in terms of items per second. You can determine $\Lambda$ by observing your system over a reasonable period of time and counting the arrivals. A reasonable amount of time is defined to be at least 100 times the *service time*, which is the length of time that the request is processed. The *residence time* is the total time a request spends in the queue, including the time it takes to be processed.

The arrival rate describes the rate at which items enter the queue, and the *throughput* defines the rate at which the items leave. In a more complex system, the *nodal throughput* defines the throughput of a single queuing node, and the *system throughput* refers to the system as a whole.

The size of the buffer doesn’t matter in most cases because it will have a finite and predictable size as long as the following conditions hold true:

- The buffer is big enough to handle the queued objects.
- The queue doesn’t grow unbounded.

The second constraint is the most important. If a queue can dispatch requests at the
rate of one request per second, but requests come in more often than one per second, then the queue will grow unbounded. In reality, the arrival rate will fluctuate, but performance analysis is concerned with the steady state, so averages are used. Perhaps at one point, 10 requests per second come in, and at other times no requests come in. As long as the average is less than one per second, then the queue will have a finite length. If the average arrival rate exceeds the rate at which requests are dispatched, then the queue length will continue to grow and never reach a steady state.

The queue in Figure 4 is called an open queue because an unlimited population of requests is arriving, and they don't necessarily come back after they're processed. A closed queue feeds back to the input; there is a finite number of requests in the system. Once the requests have been processed, they go back to the arrival queue.

The classic example of a queue is a grocery store. The number of people entering the line divided by the measurement period is the arrival rate. The number of people leaving the line divided by the measurement period is the throughput. The average time it takes a cashier to process a customer is the service time. The average time a customer waits in line, plus the service time, is the residence time.

To move into the PDQ realm, consider the following scenario. A Web service sees 30,000 requests over the course of 1 hour. Through some tracing of an unloaded system, the service time is found to be 0.08 seconds. Figure 5 shows this drawn as a queue.

**Figure 5. The Web service modeled as a queue**

![Queue Diagram](image)

What information can PDQ provide? Listing 13 shows the required PDQ program and its output.

**Listing 13. A PDQ program and its output**

```perl
#!/usr/bin/perl
use strict;
```

© Copyright IBM Corporation 1994, 2008. All rights reserved.
use pdq;
# Observations
my $arrivals = 30000; # requests
my $period = 3600; # seconds
my $serviceTime = 0.08; # seconds

# Derived
my $arrivalRate = $arrivals / $period;
my $throughput = 1 / $serviceTime;
# Sanity check -- make sure arrival rate < throughput
if ($arrivalRate > $throughput) {
die "Arrival rate $arrivalRate > throughput $throughput";
}

# Create the PDQ model and define some units
pdq::Init("Web Service");
pdq::SetWUnit("Requests");
pdq::SetTUnit("Seconds");
# The queuing node
pdq::CreateNode("webservice", $pdq::CEN, $pdq::FCFS);

# The circuit
pdq::CreateOpen("system", $arrivalRate);

# Set the service demand
pdq::SetDemand("webservice", "system", $serviceTime);

# Run the report
pdq::Solve($pdq::CANON);
pdq::Report();

..... output ..

***************************************
****** Pretty Damn Quick REPORT *******
*******************************************************************************
*** of : Sat Feb 16 11:24:54 2008 ***
*** for: Web Service ***
*** Ver: PDQ Analyzer v4.2 20070228***
*******************************************************************************
*******************************************************************************

********************************************************************************
***** PDQ Model INPUTS ********
********************************************************************************

Node Sched Resource Workload Class Demand
---- ----- -------- -------- ----- -----
CEN FCFS webservice system TRANS 0.0800

Queueing Circuit Totals:

Streams: 1
Nodes: 1

WORKLOAD Parameters

Source per Sec Demand
----- ------ ------
system 8.3333 0.0800
### Solution Method: CANON

#### SYSTEM Performance

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload: &quot;system&quot;</td>
<td>8.3333</td>
<td>Requests/Seconds</td>
</tr>
<tr>
<td>Mean Throughput</td>
<td>8.3333</td>
<td>Requests/Seconds</td>
</tr>
<tr>
<td>Response Time</td>
<td>0.2400</td>
<td>Seconds</td>
</tr>
</tbody>
</table>

#### Bounds Analysis:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Demand</td>
<td>12.5000</td>
<td>Requests/Seconds</td>
</tr>
<tr>
<td>Max Throughput</td>
<td>12.5000</td>
<td>Requests/Seconds</td>
</tr>
</tbody>
</table>

#### RESOURCE Performance

<table>
<thead>
<tr>
<th>Metric</th>
<th>Resource</th>
<th>Work</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>webservice</td>
<td>system</td>
<td>8.3333</td>
<td>Requests/Seconds</td>
</tr>
<tr>
<td>Utilization</td>
<td>webservice</td>
<td>system</td>
<td>66.6667</td>
<td>Percent</td>
</tr>
<tr>
<td>Queue Length</td>
<td>webservice</td>
<td>system</td>
<td>2.0000</td>
<td>Requests</td>
</tr>
<tr>
<td>Residence Time</td>
<td>webservice</td>
<td>system</td>
<td>0.2400</td>
<td>Seconds</td>
</tr>
</tbody>
</table>

Listing 13 begins with the UNIX "shebang" line that defines the interpreter for the rest of the program. The first two lines of Perl code call for the use of the PDQ module and the strict module. PDQ offers the PDQ functions, whereas strict is a module that enforces good Perl programming behavior.

The next section of Listing 13 defines variables associated with the observations of the system. Given this information, the section that follows the observations calculates the arrival rate and the throughput. The latter is the inverse of the service time—if you can serve one request in $N$ seconds, then you can serve $1/N$ requests per second.

#### Installing PDQ

You can download the PDQ tarball from the author’s Web site (see the Resources). Unpack it in a temporary directory with `tar -xzf pdq.tar.gz`, and change into the newly created directory with `cd pdq42`. Then, run `make` to compile the C code and the Perl module. Finally, `cd perl5` and then run `./setup.sh` to finish building the Perl module and install it in your system directory.

The sanity test checks to make sure the queue length is bounded. Most of the PDQ functions flag an error anyway, but the author of the module recommends an explicit
check. If more requests per second come in than are leaving, then the program dies with an error.

The rest of the program calls the PDQ functions directly. First, the module is initialized with the title of the model. Then, the time unit and work units are set so the reports show information the way you expect.

Each queue is created with the CreateNode function. In Listing 13, a queue called webservice is created (the names are tags to help you understand the final report) that is of type CEN (a queuing center, as opposed to a delay node that doesn’t do any work). The queue is a standard first in, first out (FIFO) queue that PDQ calls a first-come first-served queue.

Next, CreateOpen is called to define the circuit (a collection of queues). The arrival rate to the circuit has already been calculated. Finally, the demand for the queue is set with SetDemand. SetDemand defines the time taken to complete a particular workload (a queue within a circuit).

The circuit is finally solved with the Solve function and reported with the Report function. Note that PDQ takes your model, turns it into a series of equations, and then solves them. PDQ doesn’t simulate the model in any way.

Interpreting the output is straightforward. The report first starts with a header and a summary of the model. The WORKLOAD Parameters section provides more interesting information. The circuit’s service time is 0.08 seconds, which was defined. The per second rate is the input rate.

The SYSTEM performance section calculates performance of the system as a whole. The circuit managed to keep up with the input rate of 8.3333 requests per second. The response time, which includes the 0.08 seconds of service time, and the time spent in queue was 0.24 seconds (more on this later). The maximum performance of the circuit was deemed to be 12.5 requests per second.

Looking closely at the queue, you can see that it’s 66.6667% used. The average queue length is two requests. This means that a request coming in can expect to have two requests queued ahead of it, plus the request being executed. At 0.08 seconds per request, the average wait is then the 0.24 seconds reported earlier.

This model could be extended to show the components of the Web service. Rather than a single queue representing the Web service, you might have a queue to process the request, a queue to access a database, and a queue to package the response. The system performance should stay the same if your model is valid, but then you’ll have insight into the inner workings of the Web service. From there, you can play “what if” and model a faster database or more Web servers to see what sort of improvement in response you get. The individual resource numbers will tell you if a particular queue is the bottleneck, and how much room you have to grow.
Listing 13 is a basic example of using the PDQ libraries. Read Analyzing Computer System Performance with Perl::PDQ (see Resources for a link) to learn how to build more complex models.

---

Section 5. Predict future resource needs

This section covers material for topic 306.4 for the Senior Level Linux Professional (LPIC-3) exam 301. This topic has a weight of 1.

In this section, learn how to:

- Predict capacity break point of a configuration
- Observe growth rate of capacity usage
- Graph the trend of capacity usage

The previous section introduced the PDQ library and a sample report. The report showed the calculated values for the utilization and maximum load of a queue and the system as a whole. You can use the same method to predict the break point of a configuration. You can also use graphs to show the growth of a system over time and predict when it will reach capacity.

More on PDQ

Listing 14 shows the same Web service as Listing 13, but it's broken into two queues: one representing the CPU time on the Web server for processing the request and response, and one showing the time waiting for the database request to return.

**Listing 14. A new PDQ program for the sample Web service**

```perl
#!/usr/bin/perl
use strict;
use pdq;

# Observations
my $arrivals = 30000;  # requests
my $period = 3600;     # seconds

# Derived
my $arrivalRate = $arrivals / $period;

# Create the PDQ model and define some units
pdq::Init("Web Service");
```

The code in Listing 14 adds another queue to the system. The total service time is still 0.08 seconds, comprising 0.05 seconds for a database lookup and 0.03 seconds for CPU processing. Listing 15 shows the generated report.

**Listing 15. The PDQ report from Listing 14**

```
****************************
***** Pretty Damn Quick REPORT *****
****************************
*** of : Sun Feb 17 11:35:35 2008 ***
*** for: Web Service ***
*** Ver: PDQ Analyzer v4.2 20070228***
****************************

****** PDQ Model INPUTS *******

<table>
<thead>
<tr>
<th>Node</th>
<th>Sched</th>
<th>Resource</th>
<th>Workload</th>
<th>Class</th>
<th>Demand</th>
</tr>
</thead>
<tbody>
<tr>
<td>CEN</td>
<td>FCFS</td>
<td>dblookup</td>
<td>system</td>
<td>TRANS</td>
<td>0.0500</td>
</tr>
<tr>
<td>CEN</td>
<td>FCFS</td>
<td>process</td>
<td>system</td>
<td>TRANS</td>
<td>0.0300</td>
</tr>
</tbody>
</table>

Queueing Circuit Totals:

Streams: 1
Nodes: 2

WORKLOAD Parameters

<table>
<thead>
<tr>
<th>Source</th>
<th>per Sec</th>
<th>Demand</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>8.3333</td>
<td>0.0800</td>
</tr>
</tbody>
</table>
```
Solution Method: CANON

***** SYSTEM Performance *****

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workload: &quot;system&quot;</td>
<td>8.3333</td>
<td>Requests/Seconds</td>
</tr>
<tr>
<td>Mean Throughput</td>
<td>0.1257</td>
<td>Seconds</td>
</tr>
</tbody>
</table>

Bounds Analysis:
- Max Demand: 20.0000 Requests/Seconds
- Max Throughput: 20.0000 Requests/Seconds

***** RESOURCE Performance *****

<table>
<thead>
<tr>
<th>Metric</th>
<th>Resource</th>
<th>Work</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>dblookup</td>
<td>system</td>
<td>8.3333</td>
<td>Requests/Seconds</td>
</tr>
<tr>
<td>Utilization</td>
<td>dblookup</td>
<td>system</td>
<td>41.6667</td>
<td>Percent</td>
</tr>
<tr>
<td>Queue Length</td>
<td>dblookup</td>
<td>system</td>
<td>0.7143</td>
<td>Requests</td>
</tr>
<tr>
<td>Residence Time</td>
<td>dblookup</td>
<td>system</td>
<td>0.0857</td>
<td>Seconds</td>
</tr>
<tr>
<td>Throughput</td>
<td>process</td>
<td>system</td>
<td>8.3333</td>
<td>Requests/Seconds</td>
</tr>
<tr>
<td>Utilization</td>
<td>process</td>
<td>system</td>
<td>25.0000</td>
<td>Percent</td>
</tr>
<tr>
<td>Queue Length</td>
<td>process</td>
<td>system</td>
<td>0.3333</td>
<td>Requests</td>
</tr>
<tr>
<td>Residence Time</td>
<td>process</td>
<td>system</td>
<td>0.0400</td>
<td>Seconds</td>
</tr>
</tbody>
</table>

Look at the output side of the report, and note that the average response time has decreased from Listing 13, and the maximum requests per second has gone from 12.5 to 20. This is because the new model allows for pipelining. While a request is being dispatched to the database, another request can be processed by the CPU. In the older model, this wasn't possible to calculate because only one queue was used.

More important, you can see that the database is 42% utilized and the CPU is only 25% utilized. Thus the database will be the first to hit capacity as the system falls under higher load.

Change the arrivals to be 60,000 over the course of an hour, and you'll find that the average response time increases to 0.36 seconds, with the database hitting 83% utilization. You'll also see that out of the 0.36 second, .30 is spent waiting on the database. Thus, your time would be better spent speeding up database access.

You may define maximum capacity in different ways. At 20 requests per second (from the top of the report), the system is at 100% capacity. You may also choose to define your capacity in terms of average response time. At roughly 15 requests per second, the response time will exceed a quarter of a second. If your goal is to keep response to under 0.25 second, your system will hit capacity at that point even though you still have room to grow on the hardware.
Use graphs for analysis

Graphs are an excellent way to show historical information. You can look at the graph over a long period of time, such as 6 months to a year, and get an idea of your growth rate. Figure 6 represents the CPU usage of an application server over the course of a year. The average daily usage measurements were brought into a spreadsheet and graphed. A trendline was also added to show the growth.

**Figure 6. Graphing the CPU usage of a server**

![CPU Usage Graph](image)

From this graph, you can project the future usage (assuming growth stays constant). Growth on the server in Figure 6 is approximately 10% every 3 months. The effects of queuing are more pronounced at higher utilization, so you may find you need to upgrade prior to reaching 100% usage.

**How to graph**

The spreadsheet method doesn't scale well to many servers with many different measurements. One method takes the output from `sar` and passes it through a graphing tool like GNUplot. You may also look at the graphing tools available, many of which are open source. The open source category includes a series of tools based on the RRDTool package.

RRDTool is a series of programs and libraries that put data into a round-robin
database (RRD). An RRD continually archives data as it comes in, so you can have hourly data for the past year and 5-minute averages for the week. This gives you a database that never grows and that constantly prunes old data. RRDTOOL also comes with tools to make graphs.

See the Resources section for several good graphing tools.

What to graph

You should graph any information that is important to your service, and anything you could potentially use to make decisions. Graphs also play a secondary role by helping you find out what happened in the past, so you may end up graphing items like fan speeds. Normally, though, you'll focus your attention on graphing CPU, memory, disk, and network stats. If possible, graph response times from services. Not only will this help you make better decisions based on what your users will expect, but the information will also help you if you develop any models of your system.

Section 6. Summary

In this tutorial, you learned about measuring and analyzing performance. You also learned to use your measurements to troubleshoot problems.

Linux provides a wealth of information regarding the health of the system. Tools like vmstat, iostat, and ps provide real-time information. Tools like sar provide longer-term information. Remember that when you're using vmstat and iostat, the first value reported isn't real-time!

When troubleshooting a system, you should first try to identify the symptoms of the problem, both to help you understand the problem and to know when it has been solved. Then, measure system resources while the problem is ongoing (if possible) to determine the source of the problem. Once you've identified a fix, implement it and then evaluate the results.

The PDQ Perl module allows you to solve queuing problems. After replacing your system with a series of queues, you can write a Perl script using the PDQ functions. You can then use this model to calculate demand based on current usage and on future predicted usage.

Both models and graphs can be used to predict growth. Ideally, you should use both methods and compare the results.
This concludes the series on preparing for the LPIC 3 exam. If you'll be writing the exam, I wish you success and hope this series is helpful to you.
Resources

Learn

- Review the previous tutorial in this 301 series, "LPI exam 301 prep, Topic 305: Integration and migration" (developerWorks, April 2008), or all tutorials in the 301 series.

- Review the entire LPI exam prep tutorial series on developerWorks to learn Linux fundamentals and prepare for system administrator certification.

- At the LPIC Program, find task lists, sample questions, and detailed objectives for the three levels of the Linux Professional Institute’s Linux system administration certification.

- Check out UNIX Load Average Part 1: How It Works and UNIX Load Average Part 2: Not Your Average Average, by Neil Gunther, who runs the Performance Dynamics Company and has written several other articles and books entitled Analyzing Computer System Performance with Perl::PDQ and Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services.

- Sean’s article "Easy system monitoring with SAR" (developerWorks, February 2006) introduces SAR. It was written with Solaris in mind, so the command-line parameters are somewhat different, but the theory is the same.

- Sean’s tutorial "Expose Web performance problems with the RRDTool" (developerWorks, March 2006) shows you how to monitor the performance of a Web site and graph the results with RRDTool.

- The sysstat FAQ is a good place to go for answers about SAR and friends.

- In the developerWorks Linux zone, find more resources for Linux developers, including Linux tutorials, as well as our readers’ favorite Linux articles and tutorials over the last month.

- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Get the source for PDQ, along with instructions for installing it on various operating systems.

- RRDTool, the basis of most open source network monitoring systems, is a spinoff of the famous Multi Router Traffic Grapher package. RRDTool can be used as part of another package or in your own scripts.

- Cacti is one of the best open source monitoring packages around. It’s made primarily for network devices but has been extended to perform a variety of systems tasks. Cacti has an active user community that is always willing to help
in the forums.

- **ZABBIX** is another open source systems-monitoring package worth investigating.

- Check out the **Tivoli** area of developerWorks for more information on IBM’s enterprise systems, network, security, and application management tools. In particular, the **Tivoli Composite Application Management** solution helps you increase the performance and availability of today’s business-critical composite applications, including portal and SOA-based technologies.

- **Order the SEK for Linux**, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

- With **IBM trial software**, available for download directly from developerWorks, build your next development project on Linux.

**Discuss**

- Participate in the discussion forum for this content.

- Get involved in the **developerWorks community** through our developer blogs, forums, podcasts, and community topics in our new **developerWorks spaces**.

**About the author**

Sean Walberg
Sean Walberg has been working with Linux and UNIX since 1994 in academic, corporate, and Internet service provider environments. He has written extensively about systems administration over the past several years.

**Trademark**

DB2, Lotus, Rational, Tivoli, and WebSphere are trademarks of IBM Corporation in the United States, other countries, or both.
Linux is a trademark of Linus Torvalds in the United States, other countries, or both.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.