ROSS BRUNSON
SEAN WALBERG

CompTIA Linux+ POWERED BY LPI

Cert Guide

Learn, prepare, and practice for exam success

CompTIA® Linux+™/ LPIC-1

Exams LX0-103 & LX0-104/ 101-400 & 102-400

Save 10% on Exam Voucher

See Inside

DVD INCLUDED

PEARSON IT CERTIFICATION

CompTIA® Linux+ / LPIC-1 Cert Guide

Ross Brunson
Sean Walberg

PEARSON

800 East 96th Street
Indianapolis, Indiana 46240 USA

The important number is the one reported by the **wc** command: It's the number of lines in the output. Each one represents a running process. This machine has 66 processes found by the **ps** command.

Key Topic

More **ps** command switches to know are as follows:

- **a—**Shows all processes for all users
- **u—**Shows user information for processes
- **x—**Shows processes without a controlling tty

What's the Diff?

There's a lot of confusion among junior sysadmins about why you use **ps aux** sometimes and **ps -aux** at other times.

The man page isn't very helpful either, but it does tell you why the dash is used and why it's not. Linux's version of the **ps** command is a latecomer. There are two main parents to this version: BSD and POSIX, offering differing but similar ways of doing commands and options.

The BSD style of using options with the **ps** command is that you can group them, aka "aux," but they must not use a preceding dash. The POSIX or UNIX method is that you can group them, again "aux," but they must be preceded with a dash, as in **ps -aux**.

In short, the two methods are similar, and Linux's version of **ps** allows for either method.

Use the **pstree** command to show the hierarchical nature of the processes on the system, such as the following example:

```
init-+-apmd
     |-atd
     |-bdflush
     |-bonobo-activati
     |-crond
     |-cupsd
     |-dhcpcd
     |-evolution-alarm
     |-gconfd-2
     |-gnome-cups-mana
     |-gnome-name-serv
     |-gnome-panel
     |-gnome-settings-
```

```
|-gnome-smproxy
|-gnome-terminal-+-bash
|                `-mgt-pty-helper
```

NOTE The **ps** and **pstree** commands have many options. It's good to be aware of as many as possible, especially for when you take the exam. Both commands have the capability of showing the system's running processes in a treelike or hierarchical fashion.

Key Topic

Attention, older or seasoned Unix users: There will likely be questions that test how well you know the **ps** command, including what Linux uses as the equivalent of the command:

```
ps -ef
```

Verify how similar **ps –ef** is to **ps aux** with the following:

```
ps -ef > psef
ps -aux > psaux
vimdiff psef psaux
```

You find a number of similarities between the two output streams. The **ps aux** command is from Linux, whereas **ps –ef** originates from Unix.

Key Topic

The free Command

The **free** command is used to determine the amount of free memory on the system, not just by displaying the amount of unused system RAM, but also by giving you more detailed information about how much free and in-use physical memory you have, how much swap space or disk-based fake memory is in use, and finally how much of the used system RAM is being taken up by buffers and caches.

There is a moment in every class we teach where the student/attendee looks at the output of the **free** command and gets a quizzical look on her face, because she can't reconcile the actual amount of RAM in use on the system with the current load. We always take the time to explain that system RAM on a lightly utilized system is like a bus that is only half full of passengers. Everyone has room in the seat next to them for magazines, snacks, and drinks and ample room to stretch out a bit.

We then make the correlation between the passenger and a running process, and the room the passenger has allocated to her is similar to the working set (everything needed to run) for a process. The key here is that when lightly loaded, the system can allocate otherwise unused system RAM to handy and useful items such as cache

and buffers, and the processes can stretch out a bit and have fully loaded working sets for efficient running of that process.

Blocks and Buffers

Key to the running of processes and the speed of processing on the system are blocks and buffers. Block devices such as disks have an addressable unit called a *block*. This is typically (not always, but often) 512 bytes in size. System software also uses a construct called a block, but it's typically much larger than the physical size of a block on a hard disk or other block device.

When a disk block is read into system memory, it's stored in what is called a "buffer." Buffers are associated with one and only one block, and that buffer is how the data contained in that block is addressed while in memory.

Pages, Slabs, and Caches

Pages are how the kernel manages memory on your system. The processor can address very small units of memory, aka a "word" or "byte," but the memory management unit uses pages and only addresses memory in page-sized chunks. The kernel addresses every single page of the memory in the struct_page table and includes information critical to managing that page in each page's entry there.

Pages are frequently populated with the same data, read from buffers and written back to *caches* for later reading. The system manages its own structure and marks pages as being used, free, or a number of other flags or parameters entirely secondary to our purpose here.

Caches are made up of *slabs*, which are one or more contiguous pages, although most of the time a slab is only one page. To illustrate the relationship between a cache and a slab, we can say the cache is a city, the slab is a neighborhood, and the page is a block in that neighborhood.

To sum this all up, blocks are a data location on disk, buffers are what blocks are read into when a file or set of files is requested from disk, and then when those buffers are read into a page in memory, that page is a part of a slab or set of slabs that make up a cache.

To continue the bus and passenger analogy, remember that a lightly loaded system shows a large utilization of RAM but also a correspondingly hefty use of cache and buffers, while a heavily loaded system, one with many processes running, shows a lot of RAM in use but relatively low cache and buffers. Effectively, as you load up the system with more processes, all the passengers/processes have to tighten up their working sets, put their magazines, snacks, extra luggage, and drinks in the overhead rack or under the seat and be respectful of their fellow passengers/processes.

Interpreting Displayed Information from free

Key Topic

free has a number of options, but before showing you execution examples, it's important to discuss and define the columns of output you see.

```
       total  Total installed memory (MemTotal and SwapTotal in /proc/
meminfo)
       used   memory (calculated as total - free)
       free   Unused memory (MemFree and SwapFree in /proc/meminfo)
       shared    Memory  used (mostly) by tmpfs (Shmem in /proc/
meminfo, available on kernels 2.6.32, displayed as zero if not
available)
       buffers   Memory used by kernel buffers (Buffers in /proc/
meminfo)
       cached   Memory used by the page cache (Cached in /proc/
meminfo)
```

These definitions are from the **free** man page, and as you look at the following, remember that all this information is available in raw formats in the **/proc/meminfo** directory in the various files mentioned. All the **free** command does is read that data and display it in a more organized and configurable manner for your viewing pleasure.

NOTE The **free** command is like a number of other system utilities. It allows you to display its counts or data in a couple of modes, the first being "here's a huge number in bytes, good luck converting that to something mere humans can understand." Or you can just use the -h option; it converts the numeric values into bytes, kilobytes, megabytes, gigabytes, and even terabytes, as shown in Figure 7-1. You may see this referenced on an exam item, so pay close attention to the examples.

```
rbrunson@linux-ab2x:~> free -h
             total       used       free     shared    buffers     cached
Mem:          2.0G       1.2G       802M        20M       876K       744M
-/+ buffers/cache:       458M       1.5G
Swap:         2.0G         0B       2.0G
rbrunson@linux-ab2x:~>
```

Figure 7-1 Running the free command with the **-h** option

Look at the total column in Figure 7-1. That's the total RAM on the virtual machine we are running the **free** command on. The next two columns, used and free, tell you exactly how much RAM is being used and what remains not in use.

The buffers and cached columns are the amount of RAM currently being used to store the cache and buffers, and these will likely change over time and with differing amounts of loads. If you want to see some of these items change, in particular the cached column, execute the **find / -iname ".txt"** command to find every text file on your system you have access to, and you'll see the cached column expand as it predictively loads a lot of files that the system now assumes you might read or use another utility upon.

Then run the **free** command again to see the changes made to the numbers, particularly the cached column, as many files were found and predictively cached in case you might need to read them, and that read would be from memory, not from disk.

Key Topic

Additionally, you can use the **--lohi** long option to get a more detailed summary of the information. It's helpful to also include the **-h** option to get the summary information in **kb/mb/gb etc** (see Figure 7-2).

```
rbrunson@linux-ab2x:~> free --lohi -h
             total       used       free     shared    buffers     cached
Mem:          2.0G       1.2G       801M        20M       876K       745M
Low:          2.0G       1.2G       801M
High:           0B         0B         0B
-/+ buffers/cache:       458M       1.5G
Swap:         2.0G         0B       2.0G
rbrunson@linux-ab2x:~>
```

Figure 7-2 Running the **free** command with the **--lohi** option

System Uptime

The obvious usage for the **uptime** command is to determine how long the system has been "up" or running since reboot.

What's not obvious but is handy information is the number of users currently on the system and the average number of *jobs* in the run *queue* for the processor(s) over the last 1/5/15 minutes.

Key Topic

The jobs in the run queue information can be helpful in determining why a system might be slow, or seem sluggish. Too many jobs in the queue means that either the system is heavily overloaded or undergoing a lot of wait time for data to be read from disk, rather than from caches in memory, as per the previous section.

A possible fix for this is to start monitoring the number of processes and how much swap space is being used. A higher number of processes and an active amount of swap in use, not allocated but actually in use, indicates the system is overloaded.

A lightly loaded system returns uptime output similar to the following:

```
14:22pm  up 3 days  3:34,  3 users,  load average:  0.00,  0.02,  0.05
```

NOTE I know some sysadmins who have a cron job that runs **top** and **uptime** once every 30 minutes, and sends the output to them as an email, just so they can periodically see what is happening without constantly monitoring the system. This is a quick and easy way to get important information.

Sending Signals to Processes

Traditionally, when learning about *signals* and processes, you're introduced to a list of signals, with the explanations to the right of the signal, and then taught about the **kill** and **killall** commands. If you're lucky and your teacher likes to use **pgrep** or **pkill**, you learn a bit about those as well.

Table 7-2 shows the commonly used signal names, their numeric equivalents, and what the signal does to a process.

Key Topic

Table 7-2 List of Common Linux Signals

SIGINT	2	Interrupt a process (used by **Ctrl-C**)
SIGHUP	1	Hang up or shut down and restart process
SIGKILL	9	Kill the process (cannot be ignored or caught elsewhere)
SIGTERM	15	Terminate signal, (can be ignored or caught)
SIGTSTP	20	Stop the terminal (used by **Ctrl-z**)
SIGSTOP	19	Stop execution (cannot be caught or ignored)

The most common way to send a signal to a given process is by pressing the **Ctrl-C** keys while the command or script is processing and not interrupting the process unnecessarily. If you wait a reasonable amount of time and nothing happens, or the process seems locked up, by all means interrupt the process; just be aware you might lose a small bit of data or suffer other unintended consequences.

NOTE Obviously, if you are running a data-generating command, interruption might be of consequence, but merely interrupting a long file search or output command is relatively risk free.