Installing and Configuring System Software in Linux

Copyright © 1998 P. Tobin Maginnis
This document is free; you can redistribute it and/or modify it under the terms
of the GNU General Public License as published by the Free Software Foundation.

1. **Explain the dilemma of simplicity versus functionality and how Linux addresses the issue.**

   For everyday users, Linux offers prepackaged distributions like Red Hat that make installing very easy. But if additional functionality is required, the user can progress through deeper and deeper layers of control until the desired element is found and integrated. Six examples of layers of control are:

   i. Mastery of the default GUI and associated applications.
   ii. Mastering system commands and common arguments to the commands.
   iii. Mastery of on-line documentation to learn new commands and file formats that, in turn, increase the functionality of Linux.
   iv. Mastering the downloading and installation of new versions of software packages.
   v. Modification and re-compilation of a software package.
   vi. Joining the news (discussion) groups and finding new or experimental applications that can be installed.

2. **Describe the default action of gzip and contrast zcat *versus* gunzip.**

   Gzip automatically removes the uncompressed file and leaves in its place the compressed version. The compressed version includes the original uncompressed file name & its original size.

   Gunzip uncompresses the file, replacing the compressed version with the uncompressed version. Zcat reads the uncompressed version without replacing the compressed version.

3. **Explain the role of the gzip arguments -1 through -9.**
4. **Contrast the compress and gzip "problem" and its relationship to zcat.**
5. **Describe the role of an archiver and explain why tar seems to always require the "f" argument.**

   An archiver is an essential operating system utility that gathers or scatters many smaller files to/from one larger file. The archiver must also preserve critical system information such as who owns the files, the permission settings, the directory path, as well as hard and symbolic links.

   By default, tar reads from the Unix "raw" tape drive "/dev/rmt0" (raw devices are obsolete in Linux) and the "f" argument redirects the archive to the file name following the argument.

6. **Contrast the tar arguments t,v,vv,c,x, and f.**
7. **Describe the two opinions about including directories in the distribution of a software package and describe a habit one must develop to accommodate these differences.**
8. **Explain the nature of the special Unix file called "-" and explain how it relates to tar.**

The "-" is used as a file name in command line arguments but it is not a file. Instead, it's a Unix convention that is required when a system utility read/writes to more than one file and the user wants to direct the "other" file to/from stdin or stdout.

In the case of tar, it will accept a series of file names to archive from stdin with redirection or dump files to stdout. But if the archive itself is to come from or go to stdin or stdout, then using the dash file "f -" will accomplish the goal.

9. For the GNU version of tar, contrast the command `tar cvf - files | gzip -9 > archive.tar.gz` and the command `tar cvzf archive.tz files`

In the first command, tar reads and archives "files" to stdout listing each file name as it is archived. The output is piped to gzip which compresses the archive (optimizing for space at the cost of compression time), and the output is then redirected to the file "archive.tar.gz."

The second command uses the "z" option of GNU tar and compresses (with a default "-6" time/space tradeoff) the archive directly. The ".tz" extension is just as meaningful as ".tar.gz" and tends to be used more often.

10. Explain why one can move a directory with the "mv" command, but not across a mounted volume. Explain how to move directories across mounted volumes.

The mv command does not copy and delete files. It simply "adjusts" inode numbers in the parent directories to simulate the move operation. But when the move crosses a volume boundary, mv generates an error message since inode numbers in one volume cannot reference inodes in another volume. To overcome these complexities, one uses a combination of tar and shell programming to physically copy the files.

```
tar cf - . | (cd ../destination; tar xvf -).
```

In the above display, the tar command archives the current directory contents and pipes the (unnamed) archive to another shell. The second shell changes directory to the "destination" directory with one command and the second command employs tar again to unarchive the unnamed archive.

11. Support or refute the idea that the best way to upgrade is simply to load a new distribution.

Support: distributions like Red Hat's have become very easy to run. Red Hat 5.1 installs "everything" off a CD-ROM in about 12 minutes! Just tar up those key configuration files in /etc, your directory, your mail files, and your Web pages, then reload them and you are ready to go again!

Refute: Over time you forget how much customization is done. You forgot about that bug fix in sendmail, the extra features you configured in pine, the extra options configured for vim, etc. Installing a new distribution erases all of these features and you must begin again learning the quirks of the new versions.

12. Contrast "swapped" *versus* "shared" memory and relate this to "static," "shared," and "dynamic" linking libraries.

Programs only see virtual memory that is mapped onto physical memory by the memory management unit (MMU) hardware. The MMU divides virtual memory into pages. When a program tries to access

virtual memory that is not mapped to physical memory by the MMU, an interrupt occurs and causes an unused physical page to be moved or "swapped" to the hard disk. The new page is read from the file system and loaded into memory. Since pages are logically sequential, two or more programs may see a pure or read-only page logically in the program; but in fact, the same physical page is being "shared" with the other programs. Generally, shared memory is frequently accessed and not swapped.

A "static" linked library module is attached to the program at compile-link time and the program does not share the library module with other programs. Since the library module is physically attached to the program it is always there when the program invokes a function in the module.

In the case of a "shared" library module, only the name of the module to be attached and a loader (*i.e.*, ld.so) are static linked into the existing program. When the first program that has a given module name executes, it loads the shared library module. The module loads on a page boundary and, in this way, the same library module page can be shared (through the MMU) by many programs or shared by multiple copies of the same program.

A "dynamic" library module is similar to a shared module except that the attached loader (*i.e.*, ld-linux.so) waits for the program to explicitly call for loading the library module. Put another way, the dynamic loader allows the program to decide moment by moment how many library modules it wishes to load or unload.

In summary, compile-time static linked programs waste memory but always run. Shared library modules save overall system space and run-time dynamic linked programs use memory extremely efficiently, but if the library module is missing (or the wrong version), the program will not run.

As an aside, this is a far cry from the PC development environments (MS-C, MS-VB, and Borland C++) that staticly attach the whole library the program even though only module in the library was called.

13. Explain the role of dynamically linked libraries. Describe the two constraints in the use of dynamically linked library modules, and explain the system management issues associated with these constraints.

Efficient memory usage (*i.e.*, a small process footprint) and the use of a large data cache are fundamental reasons why Linux performs as well as it does; however, this speed is traded for additional system management requirements.

The first constraint is that the attached loader (ld-linux.so) has only the name of the dynamically linked module and it must be able to find the correct module for the program to continue executing. The second constraint is that the shared object name (soname) and version number of the dynamically linked module must also match the same name and version number in the library.

To assist in finding dynamic library modules quickly, the system administrator uses the program `ldconfig` to read the /etc/ld.so.config file and set up a list of links to library locations and names for ld-linux.so.

14. Define "object module" and describe the three types of libraries.

An object module (also called a relocatable object module) is an unfinished program. Although its high-level source code has been compiled into assembly language and its assembly language has been

assembled into ones and zeros, it has not been joined with other modules and it has not been assigned memory addresses. These object modules could be in one (or all) of the three library formats: static, shared, or dynamic, and each library format can have many libraries.

Static libraries employ an archiver. The archiver, ar, works in the same way as tar but it combines object module names, additional "entry points," and a table of contents into the archive. The linking loader (also called a linking editor) searches the table of contents for a module name or entry point and, when found, joins (edits) the library module with the program. The static library is usually housed in /usr/lib and takes the form /usr/lib/libxxx.a where xxx is the library name and ".a" indicates the archive format.

Linux libraries are moving from static to dynamic and the "less efficient" shared library archive format ".sa" is no longer used.

As of now, Linux dynamic libraries do not have an archiver; instead, modules are included in the library by the GNU compiler and loader. The dynamic library is usually housed in /lib and takes the form /lib/libxxx.so.version where xxx is the library name, ".so" indicates the shared object Executable and Linking Format (ELF) format, and ".version" is the major version number.

15. Explain why there are symbolic links to dynamic libraries in the same directory. Can the dynamic library not be directly accessed?

The loader that attaches to the program (ld-linux.so) only looks for the major version number, but libraries come with a major, minor, and patch level number to keep track of versions. The symbolic link redirects references to the most recent library version without the need to recompile the programs that call the new library version.

```
-rwxr-xr-x 1 root root 651112 Jul 16 20:38 /lib/libc-2.0.7.so
lrwxrwxrwx 1 root root     14 Sep  8 12:07 /lib/libc.so.5 -> libc.so.5.4.38
-rwxr-xr-x 1 root root 584776 Jun  7 07:09 /lib/libc.so.5.4.38
lrwxrwxrwx 1 root root     13 Sep  4 10:46 /lib/libc.so.6 -> libc-2.0.7.so
```

For example, in the above display we see that the shared object library has two versions of libc, versions 5 & 6. Version 5 is really "libc.so.5.4.38" and compatible with older program binaries while newer programs use version 6 which points to the real library "libc-2.0.7.so."

Thus, the symbolic links refer the shared object name (soname) to the current version of the library and these links are built by ldconfig at system boot time. The ldconfig reads the header and file names of libraries located in /lib, /usr/lib, as well as other directories listed in the /etc/ld.so.conf configuration file. When ldconfig encounters libraries with new versions, it updates the soname symbolic link to the new version. The program ldd examines programs and reports which dynamic linked soname libraries are required for the program to execute.

16. Explain the dilemma of deleting an old symbolic link to the dynamic library and replacing it with a symbolic link to a newer version of the library.

Normally, one would rm the old symbolic links and use the link program, ln, to create a new link. But the ln program may use the /lib/libc.so.6 dynamic link library and, therefore, will not run once the old symbolic link is deleted. There are three solutions to this dilemma:

i. Use a special version of the ln command that deletes the old link and installs a new link without having to re-execute or recall the dynamic library module. An example command for one-step relinking is:
ln -sf /lib/libc-2.0.8.so /lib/libc.so.6

ii. The first solution only works until someone forgets and deletes the link, then few if any system programs work. Alternatively, one can recompile the ln program with a static /usr/lib/libc.a library. Thus, the ln program will always execute.

iii. Another possibility is to use ldconfig directly since it employs a static linked library.

17. Describe the eight basic steps of installing any new software into a Linux system.

1. Make a sub-directory in /tmp for the compressed package image.
2. Check the file system in the package with the "tar tvf package" command.
3. Make a sub-directory in /usr/src for the new software and un-archive the software.
4. Read all the README installation directions.
5. Examine the Makefile to be sure there is nothing being done to your files that you do not agree with.
6. Add or delete parameters to the "config.h" files.
7. Run the configuration script, if any.
8. Execute make with the argument specified in the README file.

18. Explain why one would want to recompile the Linux kernel.

Even for a non-programmer, re-compilation is not the forbidding and complex task that it appears to be. Most of the details are handled through batch control files called "Makefiles." Second, re-compilation provides complete mastery over how the Linux system is to be configured. Unneeded components can be removed to conserve memory usage while unique file managers or various options can be enabled to improve functionality and performance for your particular system. In other words, re-compilation provides complete control over configuration of the operating system.

19. Contrast the Linux kernel "major," "minor," and "patch" numbers and relate them to "oddness."

Software versions come in dotted decimal numbers like 5.0.12. The "5" is the "major" number and relates to compatibility among a large group of modules. The "0" is the "minor" number and indicates that some of the modules have been modified, but not in a way that makes them incompatible with the group as a whole. The "12" refers to the "patch" level where minor corrections to one or more modules was made.

By convention, versions with even-numbered "minor" numbers have been tested and are said to be stable, while odd-numbered minor numbers are said to be experimental.

20. Where does one get a new kernel, and what are patches?

New Linux kernels are downloaded from The Linux Kernel Archives at http://www.kernel.org/. However the most recent versions are only the source code "differences" among an earlier patch version and subsequent patch versions. To construct the current source code version of the kernel, one starts with the latest kernel version (usually in /usr/src/linux) and runs the patch program with next patch version as input The patch program reads file names and "diffs" and searches for the same file names. When a file is found, old source code is removed and new lines are added. Since one patch version is unaware of another, the various patch input files must be applied in strict sequence to the results of the earlier pass. When the process is complete, the kernel source will be up-to-date.

Errors in the update process can be detected by searching for files created with the suffix of ".rej" or "#".

21. Describe the basic steps in building the kernel.

0. Change directory to /usr/src/linux and read README.
1. Run "make config," "make menuconfig," or "make xconfig." These are three versions of the same program that prompts for the enabling or disabling of 100 plus kernel options. Make config prompts from the command line, make menuconfig uses a colorful ASCII-based GUI (the curses library) for configuration, and make xconfig uses the X-Window GUI interface.
2. Run make dep to add "#include header.h" files for the various selected options.
3. Optionally run make clean to remove objects from a previous compilation. Use this option on the first compilation. Do not use this option if the previous configuration was unsuitable and a new configuration is being compiled.
4. Run make zImage to begin the compilation process. This can take more than an hour on a 33MHz 486 with 16 MB of RAM or about 7.5 minutes on a 200MHz Pentium Pro with 64 MB.

22. Explain how to boot the kernel image after it has been compiled.

0. The make file leaves the kernel image in /usr/src/linux/arch/i386/boot/zImage.
1. The zImage file is normally moved to /boot/vmlinuz-major.minor.patch and a shorter symbolic link file name like "myvmlinuz" is made to it.
2. The file /etc/lilo.conf is edited and a new stanza for "myvmlinuz" is added to the configuration file.
3. The command lilo reads /etc/lilo.conf and writes the new boot information to the MBR on the hard disk.
4. When the operating system is rebooted and the LILO: prompt appears, the Tab key is pressed and "myvmlinuz" is listed as one of the boot images.
5. Typing myvmlinuz boots the new kernel.

As an aside, the original PDP-11 Unix kernel images were simply called "unix." When Unix was ported to the DEC Virtual Address Extended (VAX) architecture with its 4 GB virtual memory space, paging was added to the kernel (separate from the original swapping code) and the name "vmunix" was born.

23. Contrast "insmod," "lsmod," and "rmmod" and explain why a new "class" of utilities had to be created.

System administrators probably dislike nothing more than reconfiguration to accommodate additional peripherals. For example, Microsoft's plug and play in combination with Windows 9x "sniffing" to detect "lost" or new hardware creates complex interactions which may take days to resolve. Traditional Unix implementations are not any better since they require kernel re-compilation to accommodate new hardware.

Linux takes a different approach to the problem with the concept of loadable device drivers. The insmod program loads a device driver from the /lib/modules directory. Upon loading, the driver attempts to initialize the hardware. If successful, the driver remains in the kernel, otherwise it terminates with an error message. The lsmod program lists currently loaded drivers and the rmmod program removes a driver from the kernel.

These functions could not be carried out by existing utilities since they execute as user-level programs. Device drivers must be in the same kernel "address space" so that they can have direct access to kernel data structures.

Loadable device drivers suffer a small run-time penalty since they are reached indirectly though a lookup table. However, any driver can be included directly in the kernel at compile time to improve run-time performance.