CHAPTER 21

The Linux System

Practice Exercises

21.1 Dynamically loadable kernel modules give flexibility when drivers are added to a system, but do they have disadvantages too? Under what circumstances would a kernel be compiled into a single binary file, and when would it be better to keep it split into modules? Explain your answer.

Answer: There are two principal drawbacks with the use of modules. The first is size: module management consumes unpageable kernel memory, and a basic kernel with a number of modules loaded will consume more memory than an equivalent kernel with the drivers compiled into the kernel image itself. This can be a very significant issue on machines with limited physical memory.

The second drawback is that modules can increase the complexity of the kernel bootstrap process. It is hard to load up a set of modules from disk if the driver needed to access that disk itself a module that needs to be loaded. As a result, managing the kernel bootstrap with modules can require extra work on the part of the administrator: the modules required to bootstrap need to be placed into a ramdisk image that is loaded alongside the initial kernel image when the system is initialized.

In certain cases it is better to use a modular kernel, and in other cases it is better to use a kernel with its device drivers prelinked. Where minimizing the size of the kernel is important, the choice will depend on how often the various device drivers are used. If they are in constant use, then modules are unsuitable. This is especially true where drivers are needed for the boot process itself. On the other hand, if some drivers are not always needed, then the module mechanism allows those drivers to be loaded and unloaded on demand, potentially offering a net saving in physical memory.

Where a kernel is to be built that must be usable on a large variety of very different machines, then building it with modules is clearly preferable to using a single kernel with dozens of unnecessary drivers consuming memory. This is particularly the case for commercially distributed kernels, where supporting the widest variety of hardware in the simplest manner possible is a priority.

However, if a kernel is being built for a single machine whose configuration is known in advance, then compiling and using modules may simply be an unnecessary complexity. In cases like this, the use of modules may well be a matter of taste.

21.2 Multithreading is a commonly used programming technique. Describe three different ways that threads could be implemented. Explain how these ways compare to the Linux **clone** mechanism. When might each alternative mechanism be better or worse than using clones?

Answer: Thread implementations can be broadly classified into two groups: kernel-based threads and user-mode threads. User-mode thread packages rely on some kernel support—they may require timer interrupt facilities, for example—but the scheduling between threads is not performed by the kernel but by some library of user-mode code. Multiple threads in such an implementation appear to the operating system as a single execution context. When the multithreaded process is running, it decides for itself which of its threads to execute, using non-local jumps to switch between threads according to its own preemptive or non-preemptive scheduling rules.

Alternatively, the operating system kernel may provide support for threads itself. In this case, the threads may be implemented as separate processes that happen to share a complete or partial common address space, or they may be implemented as separate execution contexts within a single process. Whichever way the threads are organized, they appear as fully independent execution contexts to the application.

Hybrid implementations are also possible, where a large number of threads are made available to the application using a smaller number of kernel threads. Runnable user threads are run by the first available kernel thread.

In Linux, threads are implemented within the kernel by a clone mechanism that creates a new process within the same virtual address space as the parent process. Unlike some kernel-based thread packages, the Linux kernel does not make any distinction between threads and processes: a thread is simply a process that did not create a new virtual address space when it was initialized.

The main advantage of implementing threads in the kernel rather than in a user-mode library are that:

- kernel-threaded systems can take advantage of multiple processors if they are available; and
- if one thread blocks in a kernel service routine (for example, a system call or page fault), other threads are still able to run.

A lesser advantage is the ability to assign different security attributes to each thread.

User-mode implementations do not have these advantages. Because such implementations run entirely within a single kernel execution context, only one thread can ever be running at once, even if multiple CPUs are available. For the same reason, if one thread enters a system call, no other threads can run until that system call completes. As a result, one thread doing a blocking disk read will hold up every thread in the application. However, user-mode implementations do have their own advantages. The most obvious is performance: invoking the kernel's own scheduler to switch between threads involves entering a new protection domain as the CPU switches to kernel mode, whereas switching between threads in user mode can be achieved simply by saving and restoring the main CPU registers. User-mode threads may also consume less system memory: most UNIX systems will reserve at least a full page for a kernel stack for each kernel thread, and this stack may not be pageable.

The hybrid approach, implementing multiple user threads over a smaller number of kernel threads, allows a balance between these tradeoffs to be achieved. The kernel threads will allow multiple threads to be in blocking kernel calls at once and will permit running on multiple CPUs, and user-mode thread switching can occur within each kernel thread to perform lightweight threading without the overheads of having too many kernel threads. The downside of this approach is complexity: giving control over the tradeoff complicates the thread library's user interface.

21.3 The Linux kernel does not allow paging out of kernel memory. What effect does this restriction have on the kernel's design? What are two advantages and two disadvantages of this design decision?

Answer: The primary impact of disallowing paging of kernel memory in Linux is that the non-preemptability of the kernel is preserved. Any process taking a page fault, whether in kernel or in user mode, risks being rescheduled while the required data is paged in from disk. Because the kernel can rely on not being rescheduled during access to its primary data structures, locking requirements to protect the integrity of those data structures are very greatly simplified. Although design simplicity is a benefit in itself, it also provides an important performance advantage on uniprocessor machines due to the fact that it is not necessary to do additional locking on most internal data structures.

There are a number of disadvantages to the lack of pageable kernel memory, however. First of all, it imposes constraints on the amount of memory that the kernel can use. It is unreasonable to keep very large data structures in non-pageable memory, since that represents physical memory that absolutely cannot be used for anything else. This has two impacts: first of all, the kernel must prune back many of its internal data structures manually, instead of being able to rely on a single virtual-memory mechanism to keep physical memory usage under control. Second, it makes it infeasible to implement certain features that require large amounts of virtual memory in the kernel, such as the /tmpfilesystem (a fast virtual-memory-based file system found on some UNIX systems).

Note that the complexity of managing page faults while running kernel code is not an issue here. The Linux kernel code is already able to deal with page faults: it needs to be able to deal with system calls whose arguments reference user memory that may be paged out to disk.

21.4 What are three advantages of dynamic (shared) linkage of libraries compared to static linkage? What are two cases where static linkage is preferable?

Answer: The primary advantages of shared libraries are that they reduce the memory and disk space used by a system, and they enhance maintainability.

When shared libraries are being used by all running programs, there is only one instance of each system library routine on disk, and at most one instance in physical memory. When the library in question is one used by many applications and programs, then the disk and memory savings can be quite substantial. In addition, the startup time for running new programs can be reduced, since many of the common functions needed by that program are likely to be already loaded into physical memory.

Maintainability is also a major advantage of dynamic linkage over static. If all running programs use a shared library to access their system library routines, then upgrading those routines, either to add new functionality or to fix bugs, can be done simply by replacing that shared library. There is no need to recompile or relink any applications; any programs loaded after the upgrade is complete will automatically pick up the new versions of the libraries.

There are other advantages too. A program that uses shared libraries can often be adapted for specific purposes simply by replacing one or more of its libraries, or even (if the system allows it, and most UNIXs including Linux do) adding a new one at run time. For example, a debugging library can be substituted for a normal one to trace a problem in an application. Shared libraries also allow program binaries to be linked against commercial, proprietary library code without actually including any of that code in the program's final executable file. This is important because on most UNIX systems, many of the standard shared libraries are proprietary, and licensing issues may prevent including that code in executable files to be distributed to third parties.

In some places, however, static linkage is appropriate. One example is in rescue environments for system administrators. If a system administrator makes a mistake while installing any new libraries, or if hardware develops problems, it is quite possible for the existing shared libraries to become corrupt. As a result, often a basic set of rescue utilities are linked statically, so that there is an opportunity to correct the fault without having to rely on the shared libraries functioning correctly.

There are also performance advantages that sometimes make static linkage preferable in special cases. For a start, dynamic linkage does increase the startup time for a program, as the linking must now be done at run time rather than at compile time. Dynamic linkage can also sometimes increase the maximum working set size of a program (the total number of physical pages of memory required to run the program). In a shared library, the user has no control over where in the library binary file the various functions reside. Since most functions do not precisely fill a full page or pages of the library, loading a function will usually result in loading in parts of the surrounding functions, too. With static linkage, absolutely no functions that are not referenced (directly or indirectly) by the application need to be loaded into memory.

Other issues surrounding static linkage include ease of distribution: it is easier to distribute an executable file with static linkage than with dynamic linkage if the distributor is not certain whether the recipient will have the correct libraries installed in advance. There may also be commercial restrictions against redistributing some binaries as shared libraries. For example, the license for the UNIX “Motif” graphical environment allows binaries using Motif to be distributed freely as long as they are statically linked, but the shared libraries may not be used without a license.

21.5 Compare the use of networking sockets with the use of shared memory as a mechanism for communicating data between processes on a single computer. What are the advantages of each method? When might each be preferred?

Answer: Using network sockets rather than shared memory for local communication has a number of advantages. The main advantage is that the socket programming interface features a rich set of synchronization features. A process can easily determine when new data has arrived on a socket connection, how much data is present, and who sent it. Processes can block until new data arrives on a socket, or they can request that a signal be delivered when data arrives. A socket also manages separate connections. A process with a socket open for receive can accept multiple connections to that socket and will be told when new processes try to connect or when old processes drop their connections.

Shared memory offers none of these features. There is no way for a process to determine whether another process has delivered or changed data in shared memory other than by going to look at the contents of that memory. It is impossible for a process to block and request a wakeup when shared memory is delivered, and there is no standard mechanism for other processes to establish a shared memory link to an existing process.

However, shared memory has the advantage that it is very much faster than socket communications in many cases. When data is sent over a socket, it is typically copied from memory to memory multiple times. Shared memory updates require no data copies: if one process updates a data structure in shared memory, that update is immediately visible to all other processes sharing that memory. Sending or receiving data over a socket requires that a kernel system service call be made to initiate the transfer, but shared memory communication can be performed entirely in user mode with no transfer of control required.

Socket communication is typically preferred when connection management is important or when there is a requirement to synchronize the sender and receiver. For example, server processes will usually establish a listening socket to which clients can connect when they want to use that service. Once the socket is established, individual requests are also sent using the socket, so that the server can easily determine when a new request arrives and who it arrived from.

In some cases, however, shared memory is preferred. Shared memory is often a better solution when either large amounts of data are to be transferred or when two processes need random access to a large common data set. In this case, however, the communicating processes may still need an extra mechanism in addition to shared memory to achieve synchronization between themselves. The X Window System, a graphical display environment for UNIX, is a good example of this: most graphic requests are sent over sockets, but shared memory is offered as an additional transport in special cases where large bitmaps are to be displayed on the screen. In this case, a request to display the bitmap will still be sent over the socket, but the bulk data of the bitmap itself will be sent via shared memory.

21.6 UNIX systems used to use disk-layout optimizations based on the rotation position of disk data, but modern implementations, including Linux, simply optimize for sequential data access. Why do they do so? Of what hardware characteristics does sequential access take advantage? Why is rotational optimization no longer so useful?
Answer: The performance characteristics of disk hardware have changed substantially in recent years. In particular, many enhancements have been introduced to increase the maximum bandwidth that can be achieved on a disk. In a modern system, there can be a long pipeline between the operating system and the disk's read-write head. A disk I/O request has to pass through the computer's local disk controller, over bus logic to the disk drive itself, and then internally to the disk, where there is likely to be a complex controller that can cache data accesses and potentially optimize the order of I/O requests.

Because of this complexity, the time taken for one I/O request to be acknowledged and for the next request to be generated and received by the disk can far exceed the amount of time between one disk sector passing under the read-write head and the next sector header arriving. In order to be able efficiently to read multiple sectors at once, disks will employ a readahead cache. While one sector is being passed back to the host computer, the disk will be busy reading the next sectors in anticipation of a request to read them. If read requests start arriving in an order that breaks this readahead pipeline, performance will drop. As a result, performance benefits substantially if the operating system tries to keep I/O requests in strict sequential order.

A second feature of modern disks is that their geometry can be very complex. The number of sectors per cylinder can vary according to the position of the cylinder: more data can be squeezed into the longer tracks nearer the edge of the disk than at the center of the disk. For an operating system to optimize the rotational position of data on such disks, it would have to have complete understanding of this geometry, as well as the timing characteristics of the disk and its controller. In general, only the disk's internal logic can determine the optimal scheduling of I/Os, and the disk's geometry is likely to defeat any attempt by the operating system to perform rotational optimizations.