The Linux System

Linux is a UNIX-like system that has gained popularity in recent years. In this chapter, we look at the history and development of Linux, and cover the user and programmer interfaces that Linux presents, interfaces that owe a great deal to the UNIX tradition. We also discuss the internal methods by which Linux implements these interfaces. However, since Linux has been designed to run as many standard UNIX applications as possible, it has much in common with existing UNIX implementations. We do not duplicate the basic description of UNIX given in the previous chapter.

Linux is a rapidly evolving operating system. This chapter describes specifically the Linux 2.6 kernel, released in late 2003.

21.1 The advantage of making only some of the symbols defined inside a kernel accessible to a loadable kernel module is that there is a fixed set of entry points made available to the kernel module. This ensures that loadable modules cannot invoke arbitrary code within the kernel and thereby interfere with the kernel's execution. By restricting the set of entry points, the kernel is guaranteed that the interactions with the module take place at controlled points where certain invariants hold. The disadvantage of making only a small set of the symbols defined accessible to the kernel module is the loss in flexibility and might sometimes lead to a performance issue as some of the details of the kernel are hidden from the module.

21.2 Linux's "soft" real-time scheduling provides ordering guarantees concerning the priorities of runnable processes: real-time processes will always be given a higher priority by the scheduler than normal time-sharing processes, and a real-time process will never be interrupted by another process with a lower real-time priority.

However, the Linux kernel does not support "hard" real-time functionality. That is, when a process is executing a kernel service routine, that routine will always execute to completion unless it yields control back to the scheduler either explicitly or implicitly (by waiting for some asynchronous event). There is no support for preemptive scheduling of kernel-mode processes. As a result, any kernel system call that runs for

a significant amount of time without rescheduling will block execution of any real-time processes.

Many real-time applications require such hard real-time scheduling. In particular, they often require guaranteed worst-case response times to external events. To achieve these guarantees, and to give user-mode real-time processes a true higher priority than kernel-mode lower-priority processes, it is necessary to find a way to avoid having to wait for low-priority kernel calls to complete before scheduling a real-time process. For example, if a device driver generates an interrupt that wakes up a high-priority real-time process, then the kernel needs to be able to schedule that process as soon as possible, even if some other process is already executing in kernel mode.

Such preemptive rescheduling of kernel-mode routines comes at a cost. If the kernel cannot rely on non-preemption to ensure atomic updates of shared data structures, then reads of or updates to those structures must be protected by some other, finer-granularity locking mechanism. This fine-grained locking of kernel resources is the main requirement for provision of tight scheduling guarantees.

Many other kernel features could be added to support real-time programming. Deadline-based scheduling could be achieved by making modifications to the scheduler. Prioritization of IO operations could be implemented in the block-device IO request layer.

21.3 Linux augments the standard setuid feature in two ways. First, it allows a program to drop and reacquire its effective uid repeatedly. In order to minimize the amount of time that a program executes with all of its privileges, a program might drop to a lower privilege level and thereby prevent the exploitation of security loopholes at the lower-level. However, when it needs to perform privileged operations, it can switch to its effective uid. Second, Linux allows a process to take on only a subset of the rights of the effective uid. For instance, an user can use a process that serves files without having control over the process in terms of being able to kill or suspend the process.

21.4 Sockets of type SOCK_STREAM use the TCP protocol for communicating data. The TCP protocol is appropriate for implementing an intercomputer file-transfer program since it provides a reliable, flow-controlled, and congestion-friendly communication channel. If data packets corresponding to a file transfer are lost, then they are retransmitted. Furthermore, the file transfer does not overrun buffer resources at the receiver and adapts to the available bandwidth along the channel. Sockets of type SOCK_DGRAM use the UDP protocol for communicating data. The UDP protocol is more appropriate for checking whether another computer is up on the network. Since a connection-oriented communication channel is not required and since there might not be any active entities on the other side to establish a communication channel with, the UDP protocol is more appropriate.

21.5 When a process performs a fork operation, a new process is created based on the original binary but with a new address space that is a clone

of the original address space. One possibility is to not to create a new address space but instead to share the address space between the old process and the newly created process. The pages of the address space are mapped with the copy-on-write attribute enabled. Then, when one of the processes performs an update on the shared address space, a new copy is made and the processes no longer share the same page of the address space.

21.6 In Linux, creation of a thread involves only the creation of some very simple data structures to describe the new thread. Space must be reserved for the new thread's execution context, its saved registers, its kernel stack page and dynamic information such as its security profile and signal state, but no new virtual address space is created.

Creating this new virtual address space is the most expensive part of the creation of a new process. The entire page table of the parent process must be copied, with each page being examined so that copy-on-write semantics can be achieved and so that reference counts to physical pages can be updated. The parent process's virtual memory is also affected by the process creation: any private read/write pages owned by the parent must be marked read-only so that copy-on-write can happen (copy-on-write relies on a page fault being generated when a write to the page occurs).

Scheduling of threads and processes also differs in this respect. The decision algorithm performed when deciding what process to run next is the same regardless of whether the process is a fully independent process or just a thread, but the action of context switching to a separate process is much more costly than switching to a thread. A process requires that the CPU's virtual memory control registers be updated to point to the new virtual address space's page tables.

In both cases—creation of a process or context switching between processes—the extra virtual memory operations have a significant cost. On many CPUs, changing page tables or swapping between page tables is not cheap: all or part of the virtual address translation lookaside buffers in the CPU must be purged when the page tables are changed. These costs are not incurred when creating or scheduling between threads.

21.7 The organization of architecture-dependent and architecture-independent code in the Linux kernel is designed to satisfy two design goals: to keep as much code as possible common between architectures and to provide a clean way of defining architecture-specific properties and code. The solution must of course be consistent with the overriding aims of code maintainability and performance.

There are different levels of architecture dependence in the kernel, and different techniques are appropriate in each case to comply with the design requirements. These levels include:

a. **CPU word size and endianness**. These are issues that affect the portability of all software written in C, but especially so for an operating system, where the size and alignment of data must be carefully arranged.

b. **CPU process architecture**. Linux relies on many forms of hardware support for its process and memory management. Different processors have their own mechanisms for changing between protection domains (e.g., entering kernel mode from user mode), rescheduling processes, managing virtual memory, and handling incoming interrupts.

The Linux kernel source code is organized so as to allow as much of the kernel as possible to be independent of the details of these architecture-specific features. To this end, the kernel keeps not one but two separate subdirectory hierarchies for each hardware architecture. One contains the code that is appropriate only for that architecture, including such functionality as the system call interface and low-level interrupt-management code.

The second architecture-specific directory tree contains C header files that are descriptive of the architecture. These header files contain type definitions and macros designed to hide the differences between architectures. They provide standard types for obtaining words of a given length, macro constants defining such things as the architecture word size or page size, and function macros to perform common tasks such as converting a word to a given byte order or doing standard manipulations to a page-table entry.

Given these two architecture-specific subdirectory trees, a large portion of the Linux kernel can be made portable between architectures. An attention to detail is required: when a 32-bit integer is required, the programmer must use the explicit __int32 type rather than assume than an int is a given size, for example. However, as long as the architecture-specific header files are used, then most process and page-table manipulation can be performed using common code between the architectures. Code that definitely cannot be shared is kept safely detached from the main common kernel code.

21.8 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:

a. kernel-threaded systems can take advantage of multiple processors if they are available; and

b. 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.

21.9 The open availability of an operating system's source code has both positive and negative impacts on security, and it is probably a mistake to say that it is definitely a good thing or a bad thing.

Linux's source code is open to scrutiny by both the good guys and the bad guys. In its favor, this has resulted in the code being inspected by a large number of people who are concerned about security and who have eliminated any vulnerabilities they have found.

On the other hand is the "security through obscurity" argument, which states that attackers' jobs are made easier if they have access to the source code of the system they are trying to penetrate. By denying attackers information about a system, the hope is that it will be harder for those attackers to find and exploit any security weaknesses that may be present.

In other words, open source code implies both that security weaknesses can be found and fixed faster by the Linux community, increasing the security of the system; and that attackers can more easily find any weaknesses that do remain in Linux.

There are other implications for source code availability, however. One is that if a weakness in Linux is found and exploited, then a fix for that problem can be created and distributed very quickly. (Typically, security problems in Linux tend to have fixes available to the public within 24 hours of their discovery.) Another is that if security is a major concern to particular users, then it is possible for those users to review the source code to satisfy themselves of its level of security or to make any changes that they wish to add new security measures.

21.10 Uninitialized data can be backed by demand-zero memory regions in a process's virtual address space. In addition, newly malloced space can also be backed by a demand-zero memory region.

21.11 Conflict resolution prevents different modules from having conflicting access to hardware resources. In particular, when multiple drivers are trying to access the same hardware, it resolves the resulting conflict.

21.12 There are many advantages to writing an operating system in a high-level language such as C. First, by programming at a higher abstraction, the number of programming errors is reduced as the code becomes more compact. Second, many high-level languages provide advanced features such as bounds checking that further minimize programming errors and security loopholes. Also, high-level programming languages have powerful programming environments that include tools such as debuggers and performance profilers that could be handy for developing code. The disadvantage with using a high-level language is that the programmer is distanced from the underlying machine, which could cause a few problems. First, there could be a performance overhead introduced by the compiler and run-time system used for the high-level language. Second, certain operations and instructions that are available at the machine level might not be accessible from the language level, thereby limiting some of the functionality available to the programmer.

21.13 There are a number of reasons for keeping functionality in shared libraries rather than in the kernel itself. These include:

a. **Reliability**. Kernel-mode programming is inherently higher risk than user-mode programming. If the kernel is coded correctly so that protection between processes is enforced, then an occurrence of a bug in a user-mode library is likely to affect only the currently executing process, whereas a similar bug in the kernel could conceivably bring down the entire operating system.

b. **Performance**. Keeping as much functionality as possible in user-mode shared libraries helps performance in two ways. First of all, it reduces physical memory consumption: kernel memory is non-pageable, so every kernel function is permanently resident in physical memory, but a library function can be paged in from disk on demand and does not need to be physically present all of the time. Although the library function may be resident in many processes at once, page sharing by the virtual memory system means that it is loaded at most once into physical memory.

   Second, calling a function in a loaded library is a very fast operation, but calling a kernel function through a kernel system service call is much more expensive. Entering the kernel involves changing the CPU protection domain, and once in the kernel, all of the arguments supplied by the process must be very carefully checked for correctness: the kernel cannot afford to make any assumptions about the validity of the arguments passed in, whereas a library function might reasonably do so. Both of these factors make calling a kernel function much slower than calling the same function in a library.

c. **Manageability**. Many different shared libraries can be loaded by an application. If new functionality is required in a running system, shared libraries to provide that functionality can be installed without interrupting any already running processes. Similarly, existing shared libraries can generally be upgraded without requiring any system down time. Unprivileged users can create shared libraries to be run by their own programs. All of these attributes make shared libraries generally easier to manage than kernel code.

There are, however, a few disadvantages to having code in a shared library. There are obvious examples of code that is completely unsuitable for implementation in a library, including low-level functionality such as device drivers or file systems. In general, services shared around the entire system are better implemented in the kernel if they are performance-critical, since the alternative—running the shared service in a separate process and communicating with it through interprocess communication—requires two context switches for every service requested by a process. In some cases, it may be appropriate to prototype a service in user mode but implement the final version as a kernel routine.

Security is also an issue. A shared library runs with the privileges of the process calling the library. It cannot directly access any resources inaccessible to the calling process, and the calling process has full access to all of the data structures maintained by the shared library. If the service being provided requires any privileges outside of a normal process's, or if the data managed by the library needs to be protected from normal user processes, then libraries are inappropriate and a separate server process (if performance permits) or a kernel implementation is required.

21.14 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 /tmp-filesystem (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.

21.15 There are many implications to having to support different file system types within the Linux kernel. For one thing, the file system interface should be independent of the data layouts and data structures used within the file system to store file data. For another thing, it might have to provide interfaces to file systems where the file data is not static data and is not even stored on the disk; instead, the filedata could be computed every time an operation is invoked to access it, as is the case with the /proc file system. These call for a fairly general virtual interface to sit on top of the different file systems.

21.16 Linux threads are kernel-level threads. The threads are visible to the kernel and are independently schedule-able. User-level threads, on the other hand, are not visible to the kernel and are instead manipulated by user-level schedulers. In addition, the threads used in the Linux kernel are used to support both the thread abstraction and the process abstraction. A new process is created by simply associating a newly created kernel thread with a distinct address space, whereas a new thread is created by simply creating a new kernel thread with the same address space. This further indicates that the thread abstaction is intimately tied into the kernel.

21.17 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.

21.18 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:

a. kernel-threaded systems can take advantage of multiple processors if they are available; and

b. if one thread blocks in a kernel service routine (for example, a system call or page fault), other threads are still able to run.

21.19 vfork() is a special case of clone and is used to create new processes without copying the page tables of the parent process. vfork() differs from fork in that the parent is suspended until the child makes a call to exec() or exit(). The child shares all memory with its parent, including the stack, until the child makes the call. This implies constraints on the program that it should be able to make progress without requiring the parent process to execute and is not suitable for certain programs where the parent and child processes interact before the child performs an exec. For such programs, the system-call sequence `fork() exec()` more appropriate.