THE LINUX OPERATING SYSTEM

William Stallings

This document is an extract from
Operating Systems: Internals and Design Principles, Fifth Edition
Prentice Hall, 2005, ISBN 0-13-147954-7

TABLE OF CONTENTS

2.8 LINUX

History

Linux started out as a UNIX variant for the IBM PC (Intel 80386) architecture. Linus Torvalds, a Finnish student of computer science, wrote the initial version. Torvalds posted an early version of Linux on the Internet in 1991. Since then, a number of people, collaborating over the Internet, have contributed to the development of Linux, all under the control of Torvalds. Because Linux is free and the source code is available, it became an early alternative to other UNIX workstations, such as those offered by Sun Microsystems and IBM. Today, Linux is a full-featured UNIX system that runs on all of these platforms and more, including Intel Pentium and Itanium, and the Motorola/IBM PowerPC.

Key to the success of Linux has been the availability of free software packages under the auspices of the Free Software Foundation (FSF). FSF's goal is stable, platform-independent software that is free, high quality, and embraced by the user community. FSF's GNU project provides tools for software developers, and the GNU Public License (GPL) is the FSF seal of approval. Torvalds used GNU tools in developing his kernel, which he then released under the GPL. Thus, the Linux distributions that you see today are the product of FSF's GNU project, Torvald's individual effort, and many collaborators all over the world.

In addition to its use by many individual programmers, Linux has now made significant penetration into the corporate world. This is not only because of the free software, but also because of the quality of the Linux kernel. Many talented programmers have contributed to the current version, resulting in a technically impressive product. Moreover, Linux is highly modular and easily configured. This makes it easy to squeeze optimal performance from a variety of hardware platforms. Plus, with the source code available, vendors can tweak applications and utilities to meet specific requirements. Throughout this book, we will provide details of Linux kernel internals.

Modular Structure

Most UNIX kernels are monolithic. Recall from earlier in this chapter that a monolithic kernel is one that includes virtually all of the operating system functionality in one large block of code that runs as a single process with a single address space. All the functional components of the kernel have access to all of its internal data structures and routines. If changes are made to any portion of a typical monolithic operating system, all the modules and routines must be relinked and reinstalled and the system rebooted before the changes can take effect. As a result, any modification, such as adding a new device driver or file system function, is difficult. This problem is especially acute for Linux, for which development is global and done by a loosely associated group of independent programmers.

Although Linux does not use a microkernel approach, it achieves many of the potential advantages of this approach by means of its particular modular architecture. Linux is structured as a collection of modules, a number of which can be automatically loaded and unloaded on demand. These relatively independent blocks are referred to as **loadable modules** [GOYE99]. In essence, a module is an object file whose code can be linked to and unlinked from the kernel at runtime. Typically, a module implements some specific function, such as a filesystem, a device driver, or some other feature of the kernel's upper layer. A module does not execute as its own process or thread, although it can create kernel threads for various purposes as necessary. Rather, a module is executed in kernel mode on behalf of the current process.

Thus, although Linux may be considered monolithic, its modular structure overcomes some of the difficulties in developing and evolving the kernel.

The Linux loadable modules have two important characteristics:

- **Dynamic linking:** A kernel module can be loaded and linked into the kernel while the kernel is already in memory and executing. A module can also be unlinked and removed from memory at any time.

- **Stackable modules:** The modules are arranged in a hierarchy. Individual modules serve as libraries when they are referenced by client modules higher up in the hierarchy, and as clients when they reference modules further down.

Dynamic linking [FRAN97] facilitates configuration and saves kernel memory. In Linux, a user program or user can explicitly load and unload kernel modules using the `insmod` and `rmmod` commands. The kernel itself monitors the need for particular functions and can load and unload modules as needed. With stackable modules, dependencies between modules can be defined. This has two benefits:

1. Code common to a set of similar modules (e.g., drivers for similar hardware) can be moved into a single module, reducing replication.
2. The kernel can make sure that needed modules are present, refraining from unloading a module on which other running modules depend, and loading any additional required modules when a new module is loaded.

Figure 2.17 is an example that illustrates the structures used by Linux to manage modules. The figure shows the list of kernel modules after only two modules have been loaded: FAT and VFAT. Each module is defined by two tables, the module table and the symbol table. The module table includes the following elements:

- *next: Pointer to the following module. All modules are organized into a linked list. The list begins with a pseudomodule (not shown in Figure 2.17).
- *name: Pointer to module name.
- size: Module size in memory pages.
- usecount: Module usage counter. The counter is incremented when an operation involving the module's functions is started and decremented when the operation terminates.

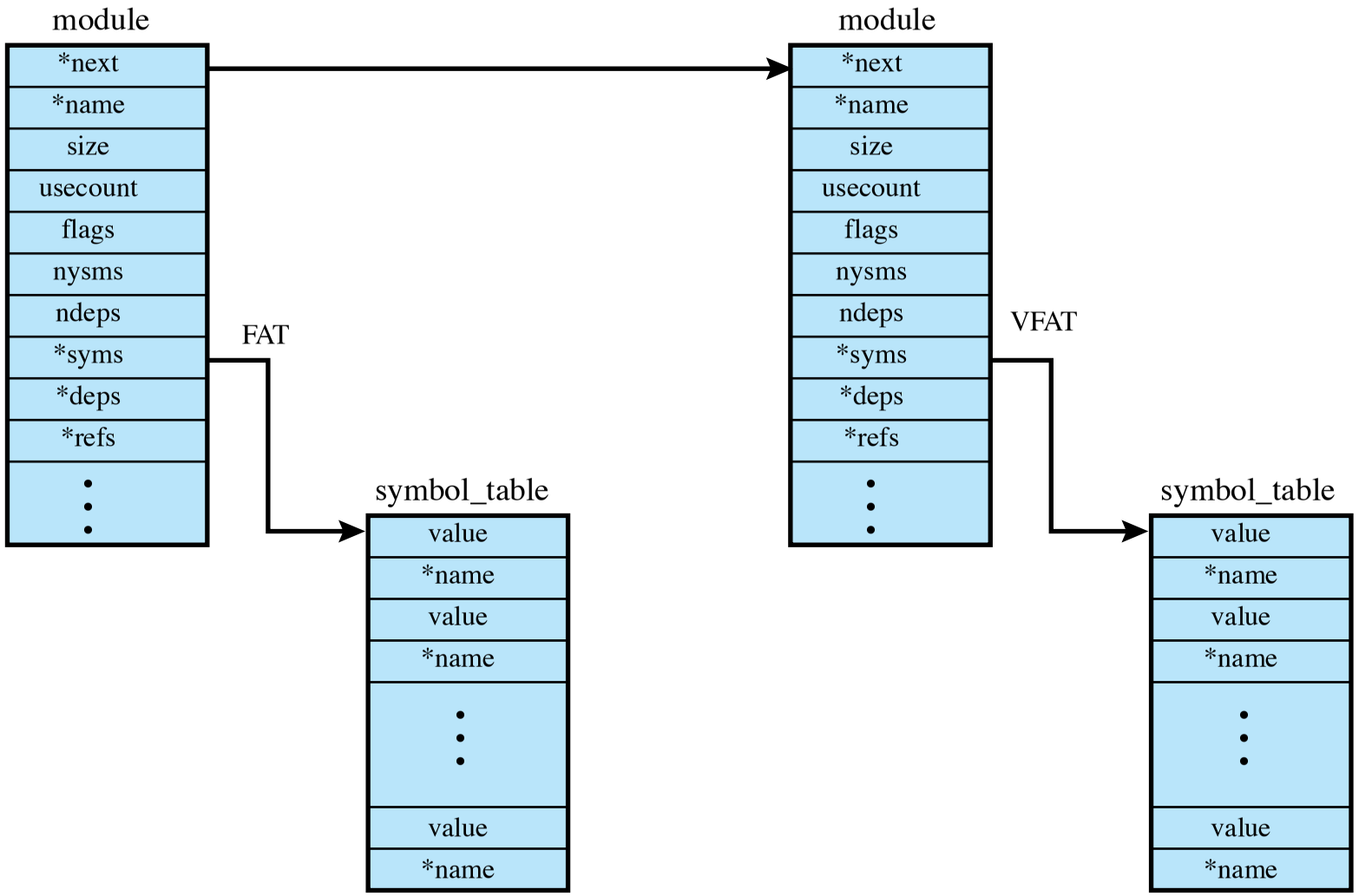

Figure 2.17 Example List of Linux Kernel Modules

- flags: Module flags.
- nsyms: Number of exported symbols.
- ndeps: Number of referenced modules
- *syms: Pointer to this module's symbol table.
- *deps: Pointer to list of modules the are referenced by this module.
- *refs: Pointer to list of modules that use this module.

The symbol table defines those symbols controlled by this module that are used elsewhere.

Figure 2.17 shows that the VFAT module was loaded after the FAT module and that the VFAT module is dependent on the FAT module.

Kernel Components

Figure 2.18, taken from [MOSB02] shows the main components of the Linux kernel as implemented on an IA-64 architecture (e.g., Intel Itanium). The figure shows several processes running on top of the kernel. Each box indicates a separate process, while each squiggly line with an arrowhead represents a thread of execution.[1] The kernel itself consists of an interacting collection of components, with arrows indicating the main interactions. The underlying hardware is also depicted as a set of components with arrows indicating which kernel components use or control which hardware components. All of the kernel components, of course, execute on the CPU but, for simplicity, these relationships are not shown.

Briefly, the principal kernel components are the following:

---

[1] In Linux, there is no distinction between the concepts of processes and threads. However, multiple threads in Linux can be grouped together in such a way that, effectively, you can have a single process comprising multiple threads. These matters are discussed in Chapter 4.

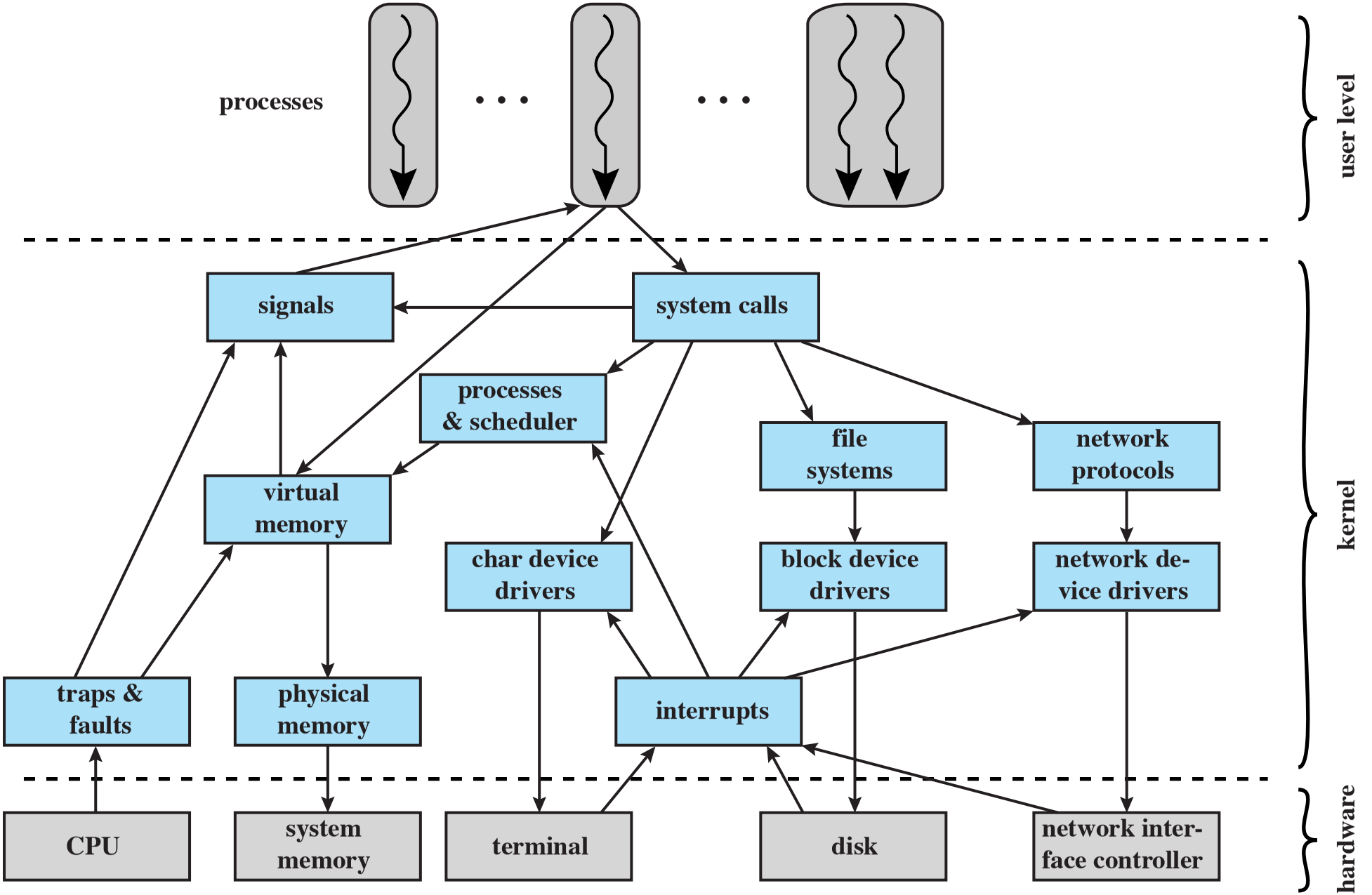

Figure 2.18 Linux Kernel Components

- **Signals:** The kernel uses signals to call into a process. For example, signals are used to notify a process of certain faults, such as division by zero. Table 2.6 gives a few examples of signals.
- **System calls:** The system call is the means by which a process requests a specific kernel service. There are several hundred system calls, which can be roughly grouped into six categories: filesystem, process, scheduling, interprocess communication, socket (networking), and miscellaneous. Table 2.7 defines a few examples in each category.
- **Processes and scheduler:** Creates, manages, and schedules processes.
- **Virtual memory:** Allocates and manages virtual memory for processes.
- **File systems:** Provides a global, hierarchical namespace for files, directories, and other file related objects and provides file system functions.
- **Network protocols:** Supports the Sockets interface to users for the TCP/IP protocol suite.
- **Character device drivers:** Manages devices that require the kernel to send or receive data one byte at a time, such as terminals, modems, and printers.
- **Block device drivers:** Manages devices that read and write data in blocks, such as various forms of secondary memory (magnetic disks, CDROMs, etc.).
- **Network device drivers:** Manages network interface cards and communications ports that connect to network devices, such as bridges and routers.
- **Traps and faults:** Handles traps and faults generated by the CPU, such as a memory fault.
- **Physical memory:** Manages the pool of page frames in real memory and allocates pages for virtual memory.
- **Interrupts:** Handles interrupts from peripheral devices.

Table 2.6 Some Linux Signals

SIGHUP	Terminal hangup	SIGCONT	Continue
SIGQUIT	Keyboard quit	SIGTSTP	Keyboard stop
SIGTRAP	Trace trap	SIGTTOU	Terminal write
SIGBUS	Bus error	SIGXCPU	CPU limit exceeded
SIGKILL	Kill signal	SIGVTALRM	Virtual alarm clock
SIGSEGV	Segmentation violation	SIGWINCH	Window size unchanged
SIGPIPT	Broken pipe	SIGPWR	Power failure
SIGTERM	Termination	SIGRTMIN	First real-time signal
SIGCHLD	Child status unchanged	SIGRTMAX	Last real-time signal

Table 2.7 Some Linux System Calls (page 1 of 2)

Filesystem related
close Close a file descriptor.
link Make a new name for a file.
open Open and possibly create a file or device.
read Read from a file descriptor.
write Read from a file descriptor
Process related
execve Execute program.
exit Terminate the calling process.
getpid Get process identification.
setuid Set user identity of the current process.
prtrace Provides a means by which a parent process my observe and control the execution of another process, and examine and change its core image and registers.
Scheduling related
sched_getparam Sets the scheduling parameters associated with the scheduling policy for the process identified by `pid`.
sched_get_priority_max Returns the maximum priority value that can be used with the scheduling algorithm identified by `policy`.
sched_setscheduler Sets both the scheduling policy (e.g., FIFO) and the associated parameters for the process `pid`.
sched_rr_get_interval Writes into the timespec structure pointed to by the parameter `tp` the round robin time quantum for the process `pid`.
sched_yield A process can relinquish the processor voluntarily without blocking via this system call. The process will then be moved to the end of the queue for its static priority and a new process gets to run.

Table 2.7 Some Linux System Calls (page 2 of 2)

Interprocess Communication (IPC) related
msgrcv A message buffer structure is allocated to receive a message. The system call then reads a message from the message queue specified by msqid into the newly created message buffer.
semctl Performs the control operation specified by `cmd` on the semaphore set `semid`.
semop Performs operations on selected members of the semaphore set `semid`.
shmat Attaches the shared memory segment identified by `shmid` to the data segment of the calling process.
shmctl Allows the user to receive information on a shared memory segment, set the owner, group, and permissions of a shared memory segment, or destroy a segment.
Socket (networking) related
bind Assigns the local IP address and port for a socket. Returns 0 for success and –1 for error.
connect Establishes a connection between the given socket and the remote socket associated with sockaddr.
gethostname Returns local host name.
send Send the bytes contained in buffer pointed to by *msg over the given socket.
setsockopt Sets the options on a socket
Miscellaneous
create_module Attempts to create a loadable module entry and reserve the kernel memory that will be needed to hold the module.
fsync Copies all in-core parts of a file to disk, and waits until the device reports that all parts are on stable storage.
query_module Requests information related to loadable modules from the kernel.
time Returns the time in seconds since January 1, 1970.
vhangup Simulates a hangup on the current terminal. This call arranges for other users to have a "clean" tty at login time.

4.6 LINUX PROCESS AND THREAD MANAGEMENT

Linux Tasks

A process, or task, in Linux is represented by a `task_struct` data structure. The `task_struct` data structure contains information in a number of categories:

- **State:** The execution state of the process (executing, ready, suspended, stopped, zombie). This is described subsequently.
- **Scheduling information:** Information needed by Linux to schedule processes. A process can be normal or real time and has a priority. Real-time processes are scheduled before normal processes, and within each category, relative priorities can be used. A counter keeps track of the amount of time a process is allowed to execute.
- **Identifiers:** Each process has a unique process identifier and also has user and group identifiers. A group identifier is used to assign resource access privileges to a group of processes.
- **Interprocess communication:** Linux supports the IPC mechanisms found in UNIX SVR4, described in Chapter 6.
- **Links:** Each process includes a link to its parent process, links to its siblings (processes with the same parent), and links to all of its children.
- **Times and timers:** Includes process creation time and the amount of processor time so far consumed by the process. A process may also have associated one or more interval timers. A process defines an interval timer by means of a system call; as a result a signal is sent to the process when the timer expires. A timer may be single use or periodic.
- **File system:** Includes pointers to any files opened by this process, as well as pointers to the current and the root directories for this process.
- **Address space:** Defines the virtual address space assigned to this process.

- **Processor-specific context:** The registers and stack information that constitute the context of this process.

Figure 4.18 shows the execution states of a process. These are:

- **Running:** This state value corresponds to two states. A Running process is either executing or it is ready to execute.
- **Interruptible:** This is a blocked state, in which the process is waiting for an event, such as the end of an I/O operation, the availability of a resource, or a signal from another process.
- **Uninterruptible:** This is another blocked state. The difference between this and the Interruptible state is that in an uninterruptible state, a process is waiting directly on hardware conditions and therefore will not handle any signals.
- **Stopped:** The process has been halted and can only resume by positive action from another process. For example, a process that is being debugged can be put into the Stopped state.
- **Zombie:** The process has been terminated but, for some reason, still must have its task structure in the process table.

Linux Threads

Traditional UNIX systems support a single thread of execution per process, while modern UNIX systems typically provide support for multiple kernel-level threads per process. As with traditional UNIX systems, older versions of the Linux kernel offered no support for multithreading. Instead, applications would need to be written with a set of user-level library functions, the most popular of which is known as *pthread (POSIX thread) libraries*, with all of the threads mapping into a single kernel-level process. We have seen that modern versions of UNIX offer kernel-level threads. Linux provides a unique solution in that it does not recognize a distinction between threads and processes. Using a mechanism similar to the lightweight processes of Solaris, user-level threads are mapped into kernel-level processes. Multiple user-

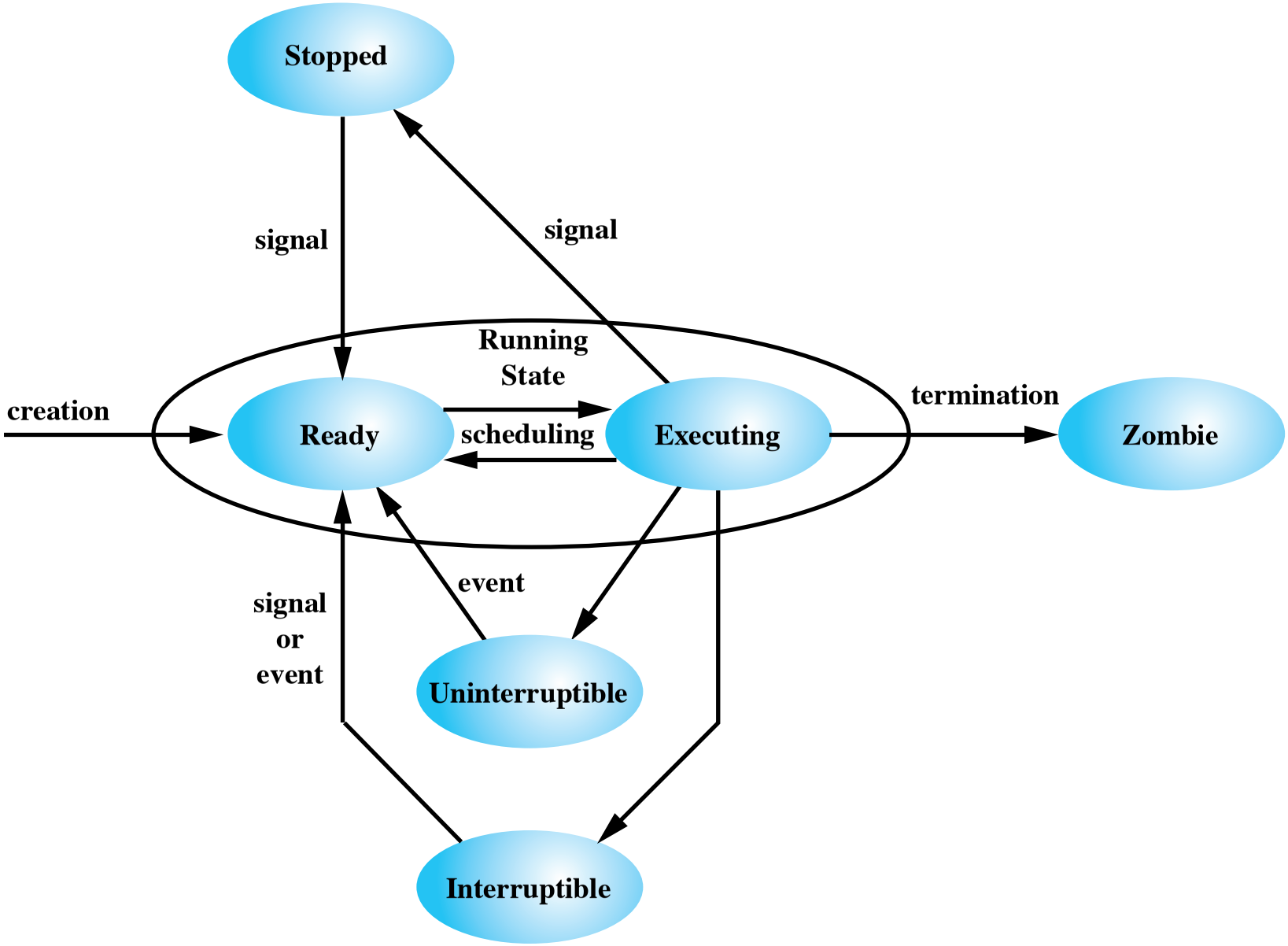

Figure 4.18 Linux Process/Thread Model

level threads that constitute a single user-level process are mapped into Linux kernel-level processes that share the same group ID. This enables these processes to share resources such as files and memory and to avoid the need for a context switch when the scheduler switches among processes in the same group.

A new process is created in Linux by copying the attributes of the current process. A new process can be *cloned* so that it shares resources, such as files, signal handlers, and virtual memory. When the two processes share the same virtual memory, they function as threads within a single process. However, no separate type of data structure is defined for a thread. In place of the usual fork() command, processes are created in Linux using the clone() command. This command includes a set of flags as arguments, defined in Table 4.5. The traditional fork() system call is implemented by Linux as a clone() system call with all of the clone flags cleared.

When the Linux kernel performs a switch from one process to another, it checks whether the address of the page directory of the current process is the same as that of the to-be-scheduled process. If they are, then they are sharing the same address space, so that a context switch is basically just a jump from one location of code to another location of code.

Although cloned processes that are part of the same process group can share the same memory space, they cannot share the same user stacks. Thus the clone() call creates separate stack spaces for each process.

Table 4.5 Linux clone () flags

CLONE_CLEARID	Clear the task ID.
CLONE_DETACHED	The parent does not want a SIGCHLD signal sent on exit.
CLONE_FILES	Shares the table that identifies the open files.
CLONE_FS	Shares the table that identifies the root directory and the current working directory, as well as the value of the bit mask used to mask the initial file permissions of a new file.
CLONE_IDLETASK	Set PID to zero, which refers to an idle task. The idle task is employed when all available tasks are blocked waiting for resources.
CLONE_NEWNS	Create a new namespace for the child.
CLONE_PARENT	Caller and new task share the same parent process.
CLONE_PTRACE	If the parent process is being traced, the child process will also be traced.
CLONE_SETTID	Write the TID back to user space.
CLONE_SETTLS	Create a new TLS for the child.
CLONE_SIGHAND	Shares the table that identifies the signal handlers.
CLONE_SYSVSEM	Shares System V SEM_UNDO semantics.
CLONE_THREAD	Inserts this process into the same thread group of the parent. If this flag is true, it implicitly enforces CLONE_PARENT.
CLONE_VFORK	If set, the parent does not get scheduled for execution until the child invokes the *execve()* system call.
CLONE_VM	Shares the address space (memory descriptor and all page tables).

6.8 LINUX KERNEL CONCURRENCY MECHANISMS

Linux includes all of the concurrency mechanisms found in other UNIX systems, such as SVR4, including pipes, messages, shared memory, and signals. In addition, Linux 2.6 includes a rich set of concurrency mechanisms specifically intended for use when a thread is executing in kernel mode. That is, these are mechanisms used within the kernel to provide concurrency in the execution of kernel code. This section examines the Linux kernel concurrency mechanisms.

Atomic Operations

Linux provides a set of operations that guarantee atomic operations on a variable. These operations can be used to avoid simple race conditions. An atomic operation executes without interruption and without interference. On a uniprocessor system, a thread performing an atomic operation cannot be interrupted once the operation has started until the operation is finished. In addition, on a multiprocessor system, the variable being operated on is locked from access by other threads until this operation is completed.

Two types of atomic operations are defined in Linux: integer operations, which operate on an integer variable, and bitmap operations, which operate on one bit in a bitmap (Table 6.3). These operations must be implemented on any architecture that implements Linux. For some architectures, there are corresponding assembly language instructions for the atomic operations. On other architectures, an operation that locks the memory bus is used to guarantee that the operation is atomic.

Table 6.3 Linux Atomic Operations

Atomic Integer Operations	
`ATOMIC_INIT (int i)`	At declaration: initialize an atomic_t to i
`int atomic_read(atomic_t *v)`	Read integer value of v
`void atomic_set(atomic_t *v, int i)`	Set the value of v to integer i
`void atomic_add(int i, atomic_t *v)`	Add i to v
`void atomic_sub(int i, atomic_t *v)`	Subtract i from v
`void atomic_inc(atomic_t *v)`	Add 1 to v
`void atomic_dec(atomic_t *v)`	Subtract 1 from v
`int atomic_sub_and_test(int i, atomic_t *v)`	Subtract i from v; return 1 if the result is zero; return 0 otherwise
`int atomic_add_negative(int i, atomic_t *v)`	Add i to v; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)
`int atomic_dec_and_test(atomic_t *v)`	Subtract 1 from v; return 1 if the result is zero; return 0 otherwise
`int atomic_inc_and_test(atomic_t *v)`	Add 1 to v; return 1 if the result is zero; return 0 otherwise
Atomic Bitmap Operations	
`void set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr
`void clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr
`void change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr
`int test_and_set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr; return the old bit value
`int test_bit(int nr, void *addr)`	Return the value of bit nr in the bitmap pointed to by addr

For **atomic integer operations**, a special data type is used, `atomic_t`. The atomic integer operations can be used only on this data type, and no other operations are allowed on this data type. [LOVE04] lists the following advantages for these restrictions:

1. The atomic operations are never used on variables that might in some circumstances be unprotected from race conditions.
2. Variables of this data type are protected from improper use by nonatomic operations.
3. The compiler cannot erroneously optimize access to the value (e.g., by using an alias rather than the correct memory address).
4. This data type serves to hide architecture-specific differences in its implementation.

A typical use of the atomic integer data type is to implement counters.

The **atomic bitmap operations** operate on one of a sequence of bits at an arbitrary memory location indicated by a pointer variable. Thus, there is no equivalent to the `atomic_t` data type needed for atomic integer operations.

Atomic operations are the simplest of the approaches to kernel synchronization. More complex locking mechanisms can be built on top of them.

Spinlocks

The most common technique used for protecting a critical section in Linux is the spinlock. Only one thread at a time can acquire a spinlock. Any other thread attempting to acquire the same lock will keep trying (spinning) until it can acquire the lock. In essence a spinlock is built on an integer location in memory that is checked by each thread before it enters its critical section. If the value is 0, the thread sets the value to 1 and enters its critical section. If the value is nonzero, the thread continually checks the value until it is zero. The spinlock is easy to implement but has the disadvantage that locked-out threads continue to execute in a busy-waiting mode. Thus spinlocks are most effective in situations where the wait time for acquiring a lock is expected to be very short, say on the order of less than two context changes.

The basic form of use of a spinlock is the following:

```
spin_lock(&lock)
```

```
/* critical section */
spin_unlock(&lock)
```

Basic Spinlocks

The basic spinlock (as opposed to the reader-writer spinlock explained subsequently) comes in four flavors (Table 6.4):

- **Plain:** If the critical section of code is not executed by interrupt handlers or if the interrupts are disabled during the execution of the critical section, then the plain spinlock can be used. It does not affect the interrupt state on the processor on which it is run.
- **_irq:** If interrupts are always enabled, then this spinlock should be used.
- **_irqsave:** If it is not known if interrupts will be enabled or disabled at the time of execution, then this version should be used. When a lock is acquired, the current state of interrupts on the local processor is saved, to be restored when the lock is released.
- **_bh:** When an interrupt occurs, the minimum amount of work necessary is performed by the corresponding interrupt handler. A piece of code, called the *bottom half*, performs the remainder of the interrupt-related work, allowing the current interrupt to be enabled as soon as possible. The `_bh` spinlock is used to disable and then enable bottom halves to avoid conflict with the protected critical section.

The plain spinlock is used if the programmer knows that the protected data is not accessed by an interrupt handler or bottom half. Otherwise, the appropriate nonplain spinlock is used.

Spinlocks are implemented differently on a uniprocessor machine versus a multiprocessor machine. For a uniprocessor system, the following considerations apply. If kernel preemption is turned off, so that a thread executing in kernel mode cannot be interrupted, then the locks are deleted at compile time; they are not needed. If kernel preemption is enabled, which does permit interrupts, then the spinlocks again compile away (that is, no test of a spinlock memory location occurs) but are simply implemented as code that enables/disables interrupts. On a multiple processor machine, the spinlock is compiled into code that does in fact test the spinlock location.

The use of the spinlock mechanism in the program allows it to be independent of whether it is executed on a uniprocessor or multiprocessor machine.

Table 6.4 Linux Spinlocks

`void spin_lock(spinlock_t *lock)`	Acquires the specified lock, spinning if needed until it is available
`void spin_lock_irq(spinlock_t *lock)`	Like spin_lock, but also disables interrupts on the local processor
`void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)`	Like spin_lock_irq, but also saves the current interrupt state in flags
`void spin_lock_bh(spinlock_t *lock)`	Like spin_lock, but also disables the execution of all bottom halves
`void spin_unlock(spinlock_t *lock)`	Releases given lock
`void spin_unlock_irq(spinlock_t *lock)`	Releases given lock and enables local interrupts
`void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)`	Releases given lock and restores local interrupts to given previous state
`void spin_unlock_bh(spinlock_t *lock)`	Releases given lock and enables bottom halves
`void spin_lock_init(spinlock_t *lock)`	Initializes given spinlock
`int spin_trylock(spinlock_t *lock)`	Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise
`int spin_is_locked(spinlock_t *lock)`	Returns nonzero if lock is currently held and zero otherwise

Reader-Writer Spinlock

The reader-writer spinlock is a mechanism that allows a greater degree of concurrency within the kernel than the basic spinlock. The reader-writer spinlock allows multiple threads to have simultaneous access to the same data structure for reading only but gives exclusive access to the spinlock for a thread that intends to update the data structure. Each reader-writer spinlock consists of a 24-bit reader counter and an unlock flag, with the following interpretation:

Counter	**Flag**	**Interpretation**

0	1	The spinlock is released and available for use
0	0	Spinlock has been acquired for writing by one thread
n ($n > 0$)	0	Spinlock has been acquired for reading by n threads
n ($n > 0$)	1	Not valid

As with the basic spinlock, there are plain, `_irq`, and `_irqsave` versions of the reader-writer spinlock.

Note that the reader-writer spinlock favors readers over writers. If the spinlock is held for readers, then so long as there is at least one reader, the spinlock cannot be preempted by a writer. Furthermore, new readers may be added to the spinlock even while a writer is waiting.

Semaphores

At the user level, Linux provides a semaphore interface corresponding to that in UNIX SVR4. Internally, Linux provides an implementation of semaphores for its own use. That is, code that is part of the kernel can invoke kernel semaphores. These kernel semaphores cannot be accessed directly by the user program via system calls. They are implemented as functions within the kernel and are thus more efficient than user-visible semaphores.

Linux provides three types of semaphore facilities in the kernel: binary semaphores, counting semaphores, and reader-writer semaphores.

Binary and Counting Semaphores

The binary and counting semaphores defined in Linux 2.6 (Table 6.5) have the same functionality as described for such semaphores in Chapter 5. The function names `down` and `up` are used for the functions referred to in Chapter 5 as `semWait` and `semSignal`, respectively.

Table 6.5 Linux Semaphores

Traditional Semaphores	
`void sema_init(struct semaphore *sem, int count)`	Initializes the dynamically created semaphore to the given count
`void init_MUTEX(struct semaphore *sem)`	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
`void init_MUTEX_LOCKED(struct semaphore *sem)`	Initializes the dynamically created semaphore with a count of 0 (initially locked)
`void down(struct semaphore *sem)`	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
`int down_interruptible(struct semaphore *sem)`	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an `up` operation is received.
`int down_trylock(struct semaphore *sem)`	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
`void up(struct semaphore *sem)`	Releases the given semaphore
Reader-Writer Semaphores	
`void init_rwsem(struct rw_semaphore, *rwsem)`	Initalizes the dynamically created semaphore with a count of 1
`void down_read(struct rw_semaphore, *rwsem)`	Down operation for readers
`void up_read(struct rw_semaphore, *rwsem)`	Up operation for readers
`void down_write(struct rw_semaphore, *rwsem)`	Down operation for writers
`void up_write(struct rw_semaphore, *rwsem)`	Up operation for writers

A counting semaphore is initialized using the `sema_init` function, which gives the semaphore a name and assigns an initial value to the semaphore. Binary semaphores, called MUTEXes in Linux, are initialized using the `init_MUTEX` and `init_MUTEX_LOCKED` functions, which initialize the semaphore to 1 or 0, respectively.

Linux provides three versions of the `down` (`semWait`) operation.

1. The `down` function corresponds to the traditional `semWait` operation. That is, the thread tests the semaphore and blocks if the semaphore is not available. The thread will awaken

when a corresponding `up` operation on this semaphore occurs. Note that this function name is used for an operation on either a counting semaphore or a binary semaphore.

2. The `down_interruptible` function allows the thread to receive and respond to a kernel signal while being blocked on the down operation. If the thread is woken up by a signal, the `down_interruptible` function increments the count value of the semaphore and returns an error code known in Linux as `-EINTR`. This alerts the thread that the invoked semaphore function has aborted. In effect, the thread has been forced to "give up" the semaphore. This feature is useful for device drivers and other services in which it is convenient to override a semaphore operation.
3. The `down_trylock` function makes it possible to try to acquire a semaphore without being blocked. If the semaphore is available, it is acquired. Otherwise, this function returns a nonzero value without blocking the thread.

Reader-Writer Semaphores

The reader-writer semaphore divides users into readers and writers; it allows multiple concurrent readers (with no writers) but only a single writer (with no concurrent readers). In effect, the semaphore functions as a counting semaphore for readers but a binary semaphore (MUTEX) for writers. Table 6.5 shows the basic reader-writer semaphore operations. The reader-writer semaphore uses uninterruptible sleep, so there is only one version of each of the `down` operations.

Barriers

In some architectures, compilers and/or the processor hardware may reorder memory accesses in source code to optimize performance. These reorderings are done to optimize the use of the instruction pipeline in the processor. The reordering algorithms contain checks to ensure that data dependencies are not violated. For example, the code:

```
a = 1;
b = 1;
```

may be reordered so that memory location `b` is updated before memory location `a` is updated. However, the code:

```
a = 1;
b = a;
```

will not be reordered. Even so, there are occasions when it is important that reads or writes are executed in the order specified because of use of the information that is made by another thread or a hardware device.

To enforce the order in which instructions are executed, Linux provides the memory barrier facility. Table 6.6 lists the most important functions that are defined for this facility. The `rmb()` operation insures that no reads occur across the barrier defined by the place of the `rmb()` in the code. Similarly, the `wmb()` operation insures that no wriets occur across the barrier defined by the place of the `wmb()` in the code. The mb() operation provides both a load and store barrier.

Two important points to note about the barrier operations:

1. The barriers relate to machine instructions, namely loads and stores. Thus the higher-level language instruction `a = b` involves both a load (read) from location `b` and a store (write) to location `a`.
2. The `rmb`, `wmb`, and `mb` operations dictate the behavior of both the compiler and the processor. In the case of the compiler, the barrier operation dictates that the compiler not reorder instructions during the compile process. In the case of the processor, the barrier operation dictates that any instructions pending in the pipeline before the barrier must be committed for execution before any instructions encountered after the barrier.

Table 6.6 Linux Memory Barrier Operations

`rmb()`	Prevents loads from being reordered across the barrier
`wmb()`	Prevents stores from being reordered across the barrier
`mb()`	Prevents loads and stores from being reordered across the barrier
`barrier()`	Prevents the compiler from reordering loads or stores across the barrier
`smp_rmb()`	On SMP, provides a `rmb()` and on UP provides a `barrier()`
`smp_wmb()`	On SMP, provides a `wmb()` and on UP provides a `barrier()`
`smp_mb()`	On SMP, provides a `mb()` and on UP provides a `barrier()`

SMP = symmetric multiprocessor
UP = uniprocessor

The `barrier()` operation is a lighter-weight version of the `mb()` operation, in that it only controls the compiler's behavior. This would be useful if it is known that the processor will not perform undesirable reorderings. For example, the Intel x86 processors do not reorder writes.

The `smp_rmb`, `smp_wmb`, and `smp_mb` operations provide an optimization for code that may be compiled on either a uniprocessor (UP) or a symmetric multiprocessor (SMP). These instructions are defined as the usual memory barriers for an SMP, but for a UP, they are all treated only as compiler barriers. The `smp_` operations are useful in situations in which the data dependencies of concern will only arise in an SMP context.

8.4 LINUX MEMORY MANAGEMENT

Linux shares many of the characteristics of the memory management schemes of other UNIX implementations but has its own unique features. Overall, the Linux memory-management scheme is quite complex [DUBE98]. In this section, we give a brief overview of the two main aspects of Linux memory management: process virtual memory, and kernel memory allocation.

Linux Virtual Memory

Virtual Memory Addressing

Linux makes use of a three-level page table structure, consisting of the following types of tables (each individual table is the size of one page):

- **Page directory:** An active process has a single page directory that is the size of one page. Each entry in the page directory points to one page of the page middle directory. The page directory must be in main memory for an active process.
- **Page middle directory:** The page middle directory may span multiple pages. Each entry in the page middle directory points to one page in the page table.
- **Page table:** The page table may also span multiple pages. Each page table entry refers to one virtual page of the process.

To use this three-level page table structure, a virtual address in Linux is viewed as consisting of four fields (Figure 8.25). The leftmost (most significant) field is used as an index into the page directory. The next field serves as an index into the page middle directory. The third field serves as an index into the page table. The fourth field gives the offset within the selected page of memory.

The Linux page table structure is platform independent and was designed to accommodate the 64-bit Alpha processor, which provides hardware support for three levels of paging. With 64-

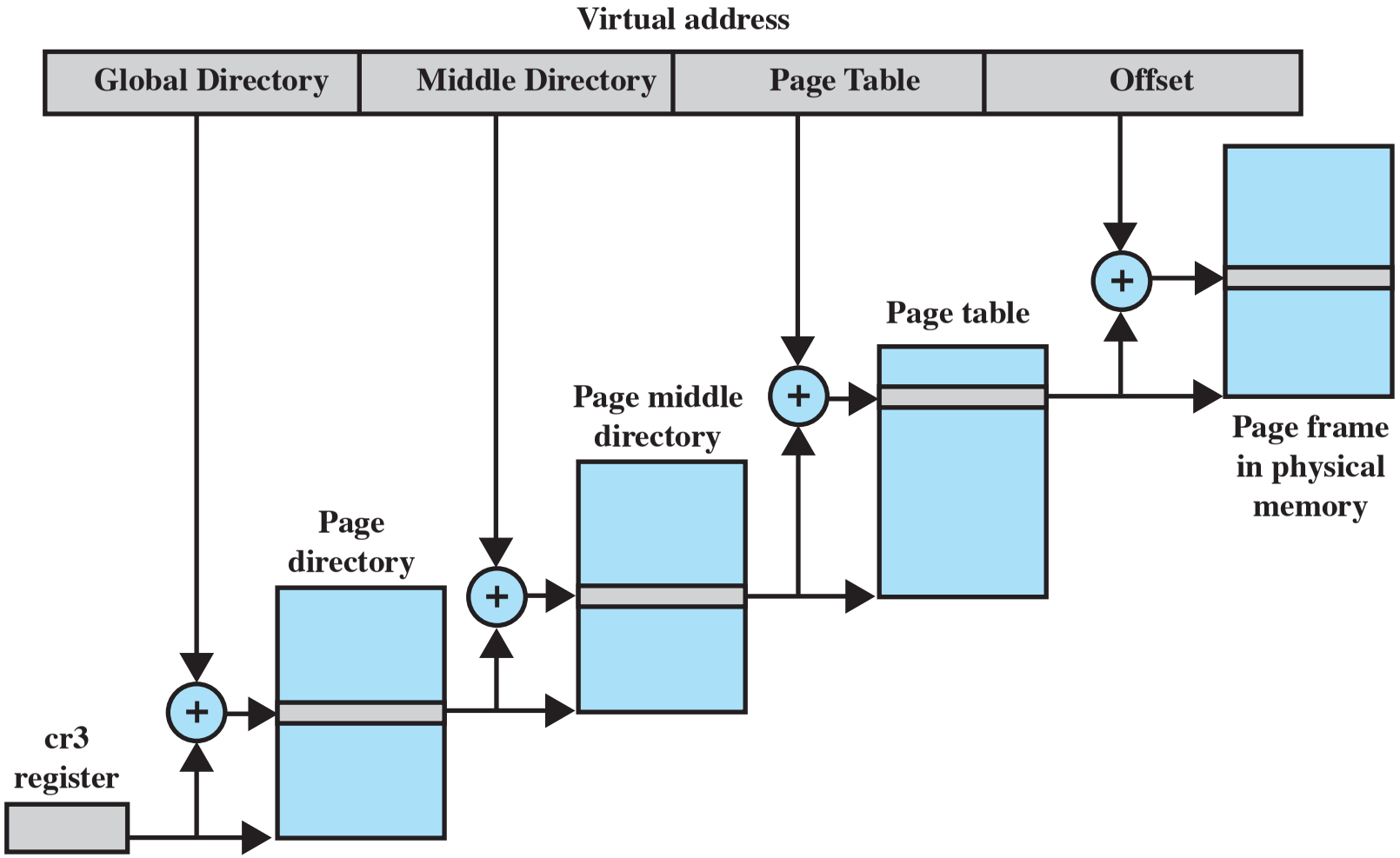

Figure 8.25 Address Translation in Linux Virtual Memory Scheme

bit addresses, the use of only two levels of pages on the Alpha would result in very large page tables and directories. The 32-bit Pentium/x86 architecture has a two-level hardware paging mechanism. The Linux software accommodates the two-level scheme by defining the size of the page middle directory as one. Note that all references to an extra level of indirection are optimized away at compile time, not at run time. Therefore, there is no performance overhead for using generic three-level design on platforms which support only two levels in hardware.

Page Allocation

To enhance the efficiency of reading in and writing out pages to and from main memory, Linux defines a mechanism for dealing with contiguous blocks of pages mapped into contiguous blocks of page frames. For this purpose, the buddy system is used. The kernel maintains a list of contiguous page frame groups of fixed size; a group may consist of 1, 2, 4, 8, 16, or 32 page frames. As pages are allocated and deallocated in main memory, the available groups are split and merged using the buddy algorithm.

Page Replacement Algorithm

The Linux page replacement algorithm is based on the clock algorithm described in Section 8.2 (see Figure 8.16). In the simple clock algorithm, a use bit and a modify bit are associated with each page in main memory. In the Linux scheme, the use bit is replaced with an 8-bit age variable. Each time that a page is accessed, the age variable is incremented. In the background, Linux periodically sweeps through the global page pool and decrements the age variable for each page as it rotates through all the pages in main memory. A page with an age of 0 is an "old" page that has not been referenced in some time and is the best candidate for replacement. The larger the value of age, the more frequently a page has been used in recent times and the less eligible it is for replacement. Thus, the Linux algorithm is a form of least frequently used policy.

Kernel Memory Allocation

The Linux kernel memory capability manages physical main memory page frames. Its primary function is to allocate and deallocate frames for particular uses. Possible owners of a frame include user-space processes (i.e., the frame is part of the virtual memory of a process that is currently resident in real memory), dynamically allocated kernel data, static kernel code, and the page cache.[2]

The foundation of kernel memory allocation for Linux is the page allocation mechanism used for user virtual memory management. As in the virtual memory scheme, a buddy algorithm is used so that memory for the kernel can be allocated and deallocated in units of one or more pages. Because the minimum amount of memory that can be allocated in this fashion is one page, the page allocator alone would be inefficient because the kernel requires small short-term memory chunks in odd sizes. To accommodate these small chunks, Linux uses a scheme known as *slab allocation* [BONW94] within an allocated page. On a Pentium/x86 machine, the page size is 4 Kbytes, and chunks within a page may be allocated of sizes 32, 64, 128, 252, 508, 2040, and 4080 bytes.

The slab allocator is relatively complex and is not examined in detail here; a good description can be found in [VAHA96]. In essence, Linux maintains a set of linked lists, one for each size of chunk. Chunks may be split and aggregated in a manner similar to the buddy algorithm, and moved between lists accordingly.

---

[2] The page cache has properties similar to a disk buffer, described in this chapter, as well as a disk cache, described in Chapter 11. We defer a discussion of the Linux page cache to Chapter 11.

10.3 LINUX SCHEDULING

For Linux 2.4 and earlier, Linux provided a real-time scheduling capability coupled with a scheduler for non-real-time processes that made use of the traditional UNIX scheduling algorithm described in Section 9.3. Linux 2.6 includes essentially the same real-time scheduling capability as previous releases and a substantially revised scheduler for non-real-time processes. We examine these two areas in turn.

Real-Time Scheduling

The three Linux scheduling classes are:

- **SCHED_FIFO:** First-in-first-out real-time threads
- **SCHED_RR:** Round-robin real-time threads
- **SCHED_OTHER:** Other, non-real-time threads

Within each class, multiple priorities may be used, with priorities in the real-time classes higher than the priorities for the SCHED_OTHER class. The default values are as follows: Real-time priority classes range from 0 to 99 inclusively, and SCHED_OTHER classes range from 100 to 139. A lower number equals a higher priority.

For FIFO threads, the following rules apply:

1. The system will not interrupt an executing FIFO thread except in the following cases:
   a. Another FIFO thread of higher priority becomes ready.
   b. The executing FIFO thread becomes blocked waiting for an event, such as I/O.
   c. The executing FIFO thread voluntarily gives up the processor following a call to the primitive `sched_yield`.

2. When an executing FIFO thread is interrupted, it is placed in the queue associated with its priority.
3. When a FIFO thread becomes ready and if that thread has a higher priority than the currently executing thread, then the currently executing thread is preempted and the highest-priority ready FIFO thread is executed. If more than one thread has that highest priority, the thread that has been waiting the longest is chosen.

The SCHED_RR policy is similar to the SCHED_FIFO policy, except for the addition of a timeslice associated with each thread. When a SCHED_RR thread has executed for its timeslice, it is suspended and a real-time thread of equal or higher priority is selected for running.

Figure 10.10 is an example that illustrates the distinction between FIFO and RR scheduling. Assume a process has four threads with three relative priorities assigned as shown in Figure 10.10a. Assume that all waiting threads are ready to execute when the current thread waits or terminates and that no higher-priority thread is awakened while a thread is executing. Figure 10.10b shows a flow in which all of the threads are in the SCHED_FIFO class. Thread D executes until it waits or terminates. Next, although threads B and C have the same priority, thread B starts because it has been waiting longer than thread C. Thread B executes until it waits or terminates, then thread C executes until it waits or terminates. Finally, thread A executes.

Figure 10.10c shows a sample flow if all of the threads are in the SCHED_RR class. Thread D executes until it waits or terminates. Next, threads B and C are time sliced, because they both have the same priority. Finally, thread A executes.

The final scheduling class is SCHED_OTHER. A thread in this class can only execute if there are no real-time threads ready to execute.

Non-Real-Time Scheduling

The Linux 2.4 scheduler for the SCHED_OTHER class did not scale well with increasing number of processors and increasing number of processes. To address this problem, Linux 2.6

A	minimum
B	middle
C	middle
D	maximum

(a) Relative thread priorities

(b) Flow with FIFO scheduling

(c) Flow with RR scheduling

Figure 10.10 Example of Linux Real-Time Scheduling

uses a completely new scheduler known as the O(1) scheduler.[3] The scheduler is designed so that the time to select the appropriate process and assign it to a processor is constant, regardless of the load on the system or the number of processors.

The kernel maintains two scheduling data structure for each processor in the system, of the following form (Figure 10.11):

```
struct prio_array {
    int               nr_active;           /* number of tasks in this array*/
    unsigned long     bitmap[BITMAP_SIZE]; /* priority bitmap */
    struct list_head  queue[MAX_PRIO];     /* priority queues */
```

A separate queue is maintained for each priority level. The total number of queues in the structure is `MAX_PRIO`, which has a default value of 140. The structure also includes a bitmap array of sufficient size to provide one bit per priority level. Thus, with 140 priority levels and 32-bit words, `BITMAP_SIZE` has a value of 5. This creates a bitmap of 160 bits, of which 20 bits are ignored. The bitmap indicates which queues are not empty. Finally, `nr_active` indicates the total number of tasks present on all queues. Two structures are maintained: an active queues structure and an expired queues structure.

Initially, both bitmaps are set to all zeroes and all queues are empty. As a process becomes ready, it is assigned to the appropriate priority queue in the active queues structure and is assigned the appropriate timeslice. If a task is preempted before it completes its timeslice, it is returned to an active queue. When a task completes its timeslice, it goes into the appropriate queue in the expired queues structure and is assigned a new timeslice. All scheduling is done from among tasks in the active queues structure. When the active queues structure is empty, a simple pointer assignment results in a switch of the active and expired queues, and scheduling continues.

---

[3] The term O(1) is an example of the "big-O" notation, used for characterizing the time complexity of algorithms. An explanation of this notation is contained in a supporting document at this book's Web site.

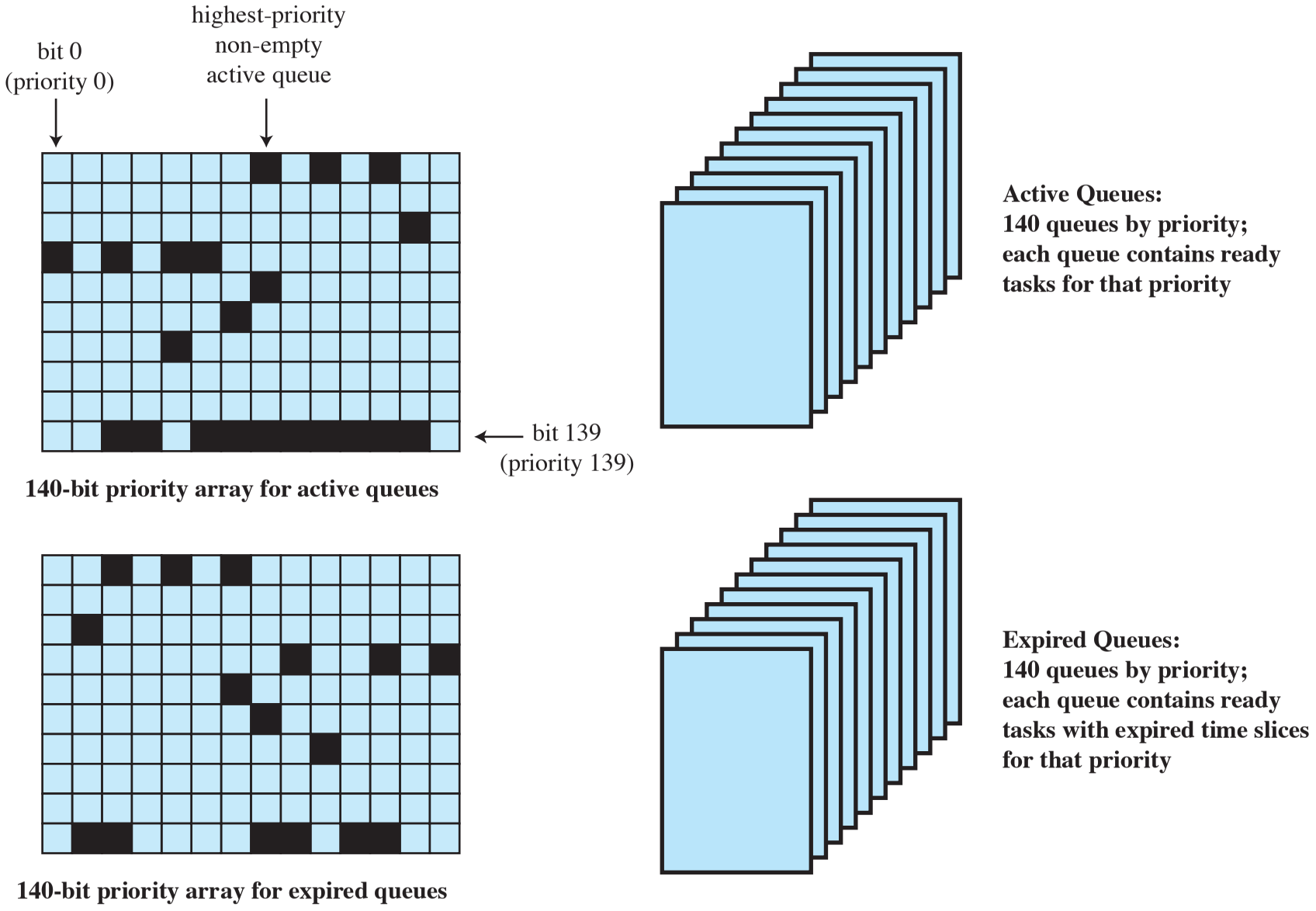

Figure 10.11 Linux Scheduling Data Structures for Each Processor

Scheduling is simple and efficient. On a given processor, the scheduler picks the highest-priority nonempty queue. If multiple tasks are in that queue, the tasks are scheduled in round-robin fashion.

Linux also includes a mechanism for moving a tasks from the queue lists of one processor to that of another. Periodically, the scheduler checks to see if there is a substantial imbalance among the number of tasks assigned to each processor. To balance the load, the schedule can transfer some tasks. The highest priority active tasks are selected for transfer, because it is more important to distribute high-priority tasks fairly.

Calculating Priorities and Timeslices

Each non-real-time task is assigned an initial priority in the range of 100 to 139, with a default of 120. This is the task's static priority and is specified by the user. As the task executes, a dynamic priority is calculated as a function of the task's static priority and its execution behavior. The Linux scheduler is designed to favor I/O-bound tasks over processor-bound tasks. This preference tends to provide good interactive response. The technique used by Linux to determine the dynamic priority is to keep a running tab on how much time a process sleeps (waiting for an event) versus how much time the process runs. In essence, a task that spends most of its time sleeping is given a higher priority.

Timeslices are assigned in the range of 10 ms to 200 ms. In general, higher-priority tasks are assigned larger timeslices.

Relationship to Real-Time Tasks

Real-time tasks are handled in a different manner from non-real-time tasks in the priority queues. The following considerations apply:

1. All real-time tasks have only a static priority; no dynamic priority changes are made.

2. `SCHED_FIFO` tasks do not have assigned timeslices. Such tasks are scheduled in FIFO discipline. If a `SHED_FIFO` task is blocked, it returns to the same priority queue in the active queue list when it becomes unblocked.
3. Although `SCHED_RR` tasks do have assigned timeslices, they also are never moved to the expired queue list. When a `SCHED_RR` task exhaust its timeslice, it is returned to its priority queue with the same timeslice value. Timeslice values are never changed.

The effect of these rules is that the switch between the active queue list and the expired queue list only happens when there are no ready real-time tasks waiting to execute.

11.9 LINUX I/O

In general terms, the Linux I/O kernel facility is very similar to that of other UNIX implementation, such as SVR4. The Linux kernel associates a special file with each I/O device driver. Block, character, and network devices are recognized. In this section, we look at several features of the Linux I/O facility.

Disk Scheduling

The default disk scheduler in Linux 2.4 is known as the Linus Elevator, which is a variation on the LOOK algorithm discussed in Section 11.3. For Linux 2.6, the Elevator algorithm has been augmented by two additional algorithms: the deadline I/O scheduler and the anticipatory I/O scheduler [LOVE04b]. We examine each of these in turn.

The Elevator Scheduler

The elevator scheduler maintains a single queue for disk read and write requests and performs both sorting and merging functions on the queue. In general terms, the elevator scheduler keeps the list of requests sorted by block number. Thus, as the disk requests are handled, the drive moves in a single direction, satisfying each request as it is encountered. This general strategy is refined in the following manner. When a new request is added to the queue, four operations are considered in order:

1. If the request is to the same on-disk sector or an immediately adjacent sector to a pending request in the queue, then the existing request and the new request are merged into one request.
2. If a request in the queue is sufficiently old, the new request is inserted at the tail of the queue.
3. If there is a suitable location, the new request is inserted in sorted order.

4. If there is no suitable location, the new request is placed at the tail of the queue.

Deadline Scheduler

Operation 2 in the preceding list is intended to prevent starvation of a request, but is not very effective [LOVE04a]. It does not attempt to service requests in a given time frame but merely stops insertion-sorting requests after a suitable delay. Two problems manifest themselves with the elevator scheme. The first problem is that a distant block request can be delayed for a substantial time because the queue is dynamically updated. For example, consider the following stream of requests for disk blocks: 20, 30, 700, 25. The elevator scheduler reorders these so that the requests are placed in the queue as 20, 25, 30, 700, with 20 being the head of the queue. If a continuous sequence of low-numbered block requests arrive, then the request for 700 continues to be delayed.

An even more serious problem concerns the distinction between read and write requests. Typically, a write request is issued asynchronously. That is, once a process issues the write request, it need not wait for the request to actually be satisfied. When an application issues a write, the kernel copies the data into an appropriate buffer, to be written out as time permits. Once the data are captured in the kernel's buffer, the application can proceed. However, for many read operations, the process must wait until the requested data are delivered to the application before proceeding. Thus, a stream of write requests (for example, to place a large file on the disk) can block a read request for a considerable time and thus block a process.

To overcome these problems, the deadline I/O scheduler makes use of three queues (Figure 11.14). Each incoming request is placed in the sorted elevator queue, as before. In addition, the same request is placed at the tail of a read FIFO queue for a read request or a write FIFO queue for a write request. Thus, the read and write queues maintain a list of requests in the sequence in which the requests were made. Associated with each request is an expiration time, with a default value of 0.5 seconds for a read request and 5 seconds for a write request. Ordinarily, the scheduler dispatches from the sorted queue. When a request is satisfied, it is removed from the

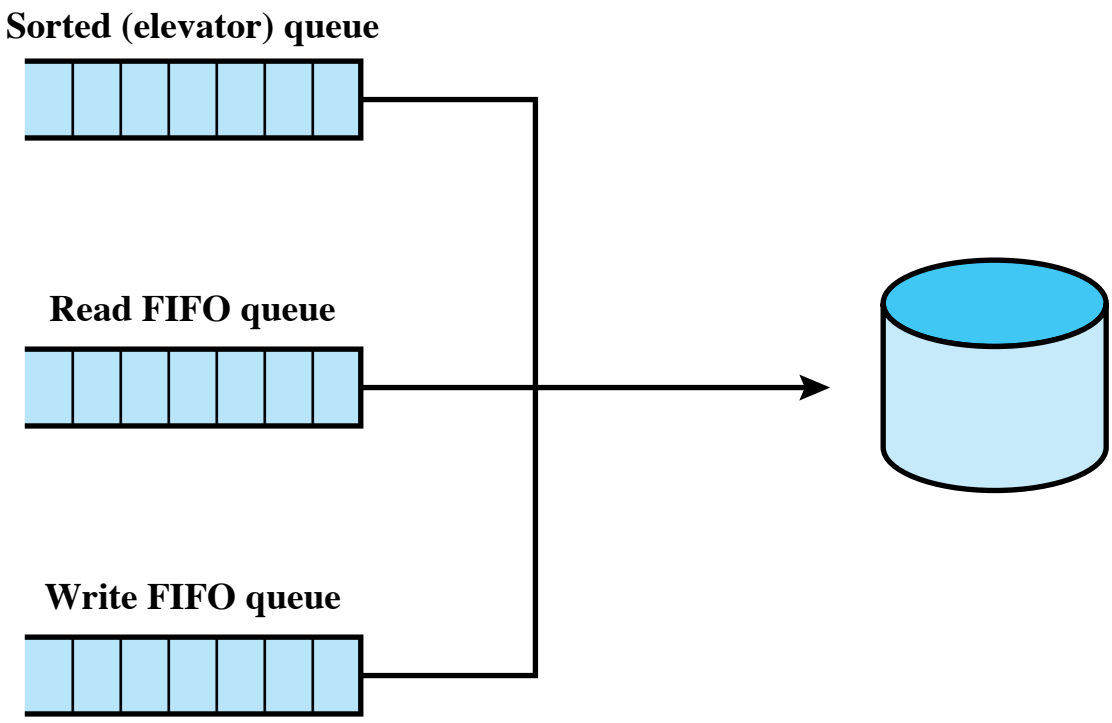

Figure 11.14 The Linux Deadline I/O Scheduler

head of the sorted queue and also from the appropriate FIFO queue. However, when the item at the head of one of the FIFO queues becomes older than its expiration time, then the scheduler next dispatches from that FIFO queue, taking the expired request, plus the next few requests from the queue. As each request is dispatched, it is also removed from the sorted queue.

The deadline I/O scheduler scheme overcomes the starvation problem and also the read versus write problem.

Anticipatory I/O Scheduler

The original elevator scheduler and the deadline scheduler both are designed to dispatch a new request as soon as the existing request is satisfied, thus keeping the disk as busy as possible. This same policy applies to all of the scheduling algorithms discussed in Section 11.5. However, such a policy can be counterproductive if there are numerous synchronous read requests. Typically, an application will wait until a read request is satisfied and the data available before issuing the next request. The small delay between receiving the data for the last read and issuing the next read enables the scheduler to turn elsewhere for a pending request and dispatch that request.

Because of the principle of locality, it is likely that successive reads from the same process will be to disk blocks that are near one another. If the scheduler were to delay a short period of time after satisfying a read request, to see if a new nearby read request is made, the overall performance of the system could be enhanced. This is the philosophy behind the anticipatory scheduler, proposed in [IYER01], and implemented in Linux 2.6.

In Linux, the anticipatory scheduler is superimposed on the deadline scheduler. When a read request is dispatched, the anticipatory scheduler causes the scheduling system to delay for up to 6 milliseconds, depending on the configuration. During this small delay, there is a good chance that the application that issued the last read request will issue another read request to the same region of the disk. If so, that request will be serviced immediately. If no such read request occurs, the scheduler resumes using the deadline scheduling algorithm.

[LOVE04b] reports on two tests of the Linux scheduling algorithms. The first test involved the reading of a 200 MB file while doing a long streaming write in the background. The second test involved doing a read of a large file in the background while reading every file in the kernel source tree. The results are listed in the following table:

I/O Scheduler and Kernel	Test 1	Test 2
Linus elevator on 2.4	45 seconds	30 minutes, 28 seconds
Deadline I/O scheduler on 2.6	40 seconds	3 minutes, 30 seconds
Anticipatory I/O scheduler on 2.6	4.6 seconds	15 seconds

As can be seen, the performance improvement depends on the nature of the workload. But in both cases, the anticipatory scheduler provides a dramatic improvement.

Linux Page Cache

In Linux 2.2 and earlier releases, the kernel maintained a page cache for reads and writes from regular filesystem files and for virtual memory pages, and a separate buffer cache for block I/O. For Linux 2.4 and later, there is a single unified page cache that is involved in all traffic between disk and main memory.

The page cache confers two benefits. First, when it is time to write back dirty pages to disk, a collection of them can be ordered properly and written out efficiently. Second, because of the principle of temporal locality, pages in the page cache are likely to be referenced again before they are flushed from the cache, thus saving a disk I/O operation.

Dirty pages are written back to disk in two situations:

- When free memory falls below a specified threshold, the kernel reduces the size of the page cache to release memory to be added to the free memory pool.

- When dirty pages grow older than a specified threshold, a number of dirty pages are written back to disk.

12.8 LINUX VIRTUAL FILE SYSTEM

Linux includes a versatile and powerful file handling facility, designed to support a wide variety of file management systems and file structures. The approach taken in Linux is to make use of a **virtual file system (VFS)**, which presents a single, uniform file system interface to user processes. The VFS defines a common file model that is capable of representing any conceivable file system's general feature and behavior. The VFS assumes that files are objects in a computer's mass storage memory that share basic properties regardless of the target file system or the underlying processor hardware. Files have symbolic names that allow them to be uniquely identified within a specific directory within the file system. A file has an owner, protection against unauthorized access or modification, and a variety of other properties. A file may be created, read from, written to, or deleted. For any specific file system, a mapping module is needed to transform the characteristics of the real file system to the characteristics expected by the virtual file system.

Figure 12.15 indicates the key ingredients of the Linux file system strategy. A user process issues a file system call (e.g., read) using the VFS file scheme. The VFS converts this into an internal (to the kernel) file system call that is passed to a mapping function for a specific file system [e.g., IBM's Journaling File System (JFS)]. In most cases, the mapping function is simply a mapping of file system functional calls from one scheme to another. In some cases, the mapping function is more complex. For example, some file systems use a file allocation table (FAT), which stores the position of each file in the directory tree. In these file systems, directories are not files. For such file systems, the mapping function for such a file system must be able to construct dynamically, and when needed, the files corresponding to the directories. In any case, the original user file system call is translated into a call that is native to the target file system. The target file system software is then invoked to perform the requested function on a file or directory under its control and secondary storage. The results of the operation are then communicated back to the user in a similar fashion.

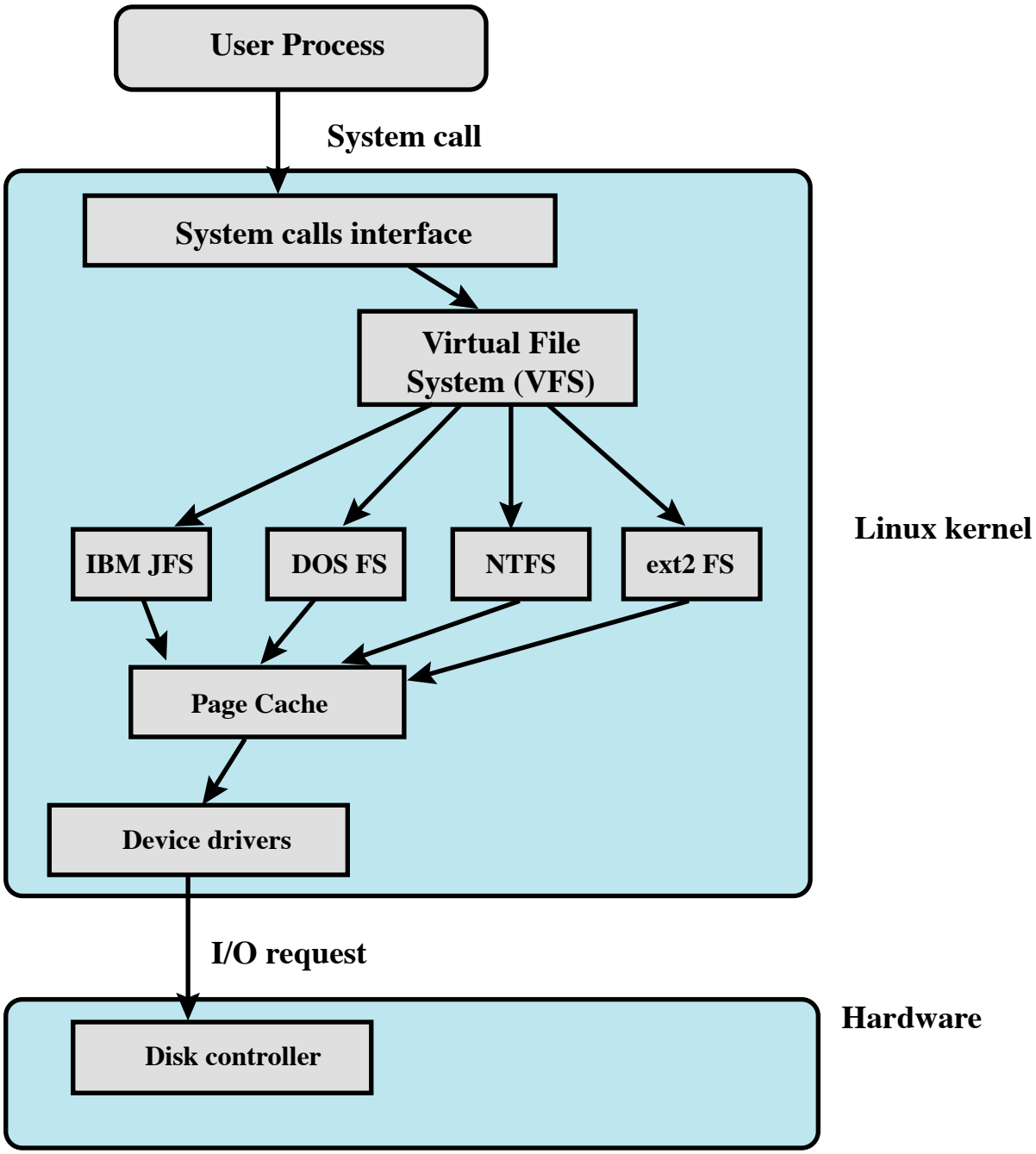

Figure 12.15 Linux Virtual File System Context

Figure 12.16 indicates the role that VFS plays within the Linux kernel. When a process initiates a file-oriented system call (e.g., read), the kernel calls a function in the VFS. This function handles the file-system-independent manipulations and initiates a call to a function in the target file system code. This call passes through a mapping function that converts the call from the VFS into a call to the target file system. The VFS is independent of any file system, so the implementation of a mapping function must be part of the implementation of a file system on Linux. The target file system converts the file system request into device-oriented instructions that are passed to a device driver by means of page cache functions.

VFS is an object-oriented scheme. Because it is written in C, rather than a language that supports object programming (such as C++ or Java), VFS objects are implemented simply as C data structures. Each object contains both data and pointers to file-system-implemented functions that operate on data. The four primary object types in VFS are as follows:

- **Superblock object:** Represents a specific mounted file system
- **Inode object:** Represents a specific file
- **Dentry object:** Represents a specific directory entry
- **File object:** Represents an open file associated with a process

This scheme is based on the concepts used in UNIX file systems, as described in Section 12.7. The key concepts of UNIX file system to remember are the following. A file system consists of a hierarchal organization of directories. A directory is the same as what is knows as a folder on many non-UNIX platforms and may contain files and/or other directories. Because a directory may contain other directories, a tree structure is formed. A path through the tree structure from the root consists of a sequence of directory entries, ending in either a directory entry (dentry) or a file name. In UNIX, a directory is implemented as a file that lists the files and directories contained within it. Thus, file operations can be performed on either files or directories.

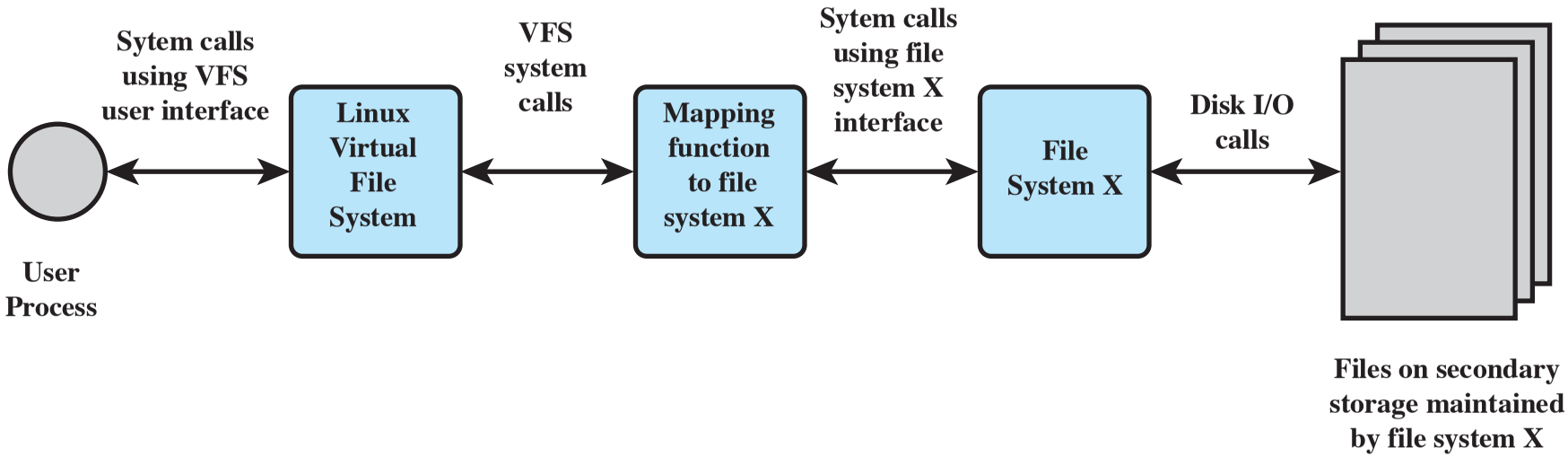

Figure 12.16 Linux Virtual File System Concept

The Superblock Object

The superblock object stores information describing a specific file system. Typically, the superblock corresponds to the file system superblock or file system control block, which is stored in a special sector on disk.

The superblock object consists a number of data items. Examples include:

- The device that this file system is mounted on
- The basic block size of the file system
- Dirty flag, to indicate that the superblock has been changed but not written back to disk
- Filesystem type
- Flags, such as a read-only flag
- Pointer to the root of the file system directory
- List of open files
- Semaphore for controlling access to the file system
- List of superblock operations

The last item on the preceding list refers to an operations object contained within the superblock object. The operation object defines the object methods (functions) that the kernel can invoke against the superblock object. The methods defined for the superblock object include:

- `*read_inode`: Read a specified inode from a mounted file system.
- `*write_inode`: Write given inode to disk.
- `*put_inode`: Release inode.
- `*delete_inode`: Delete inode from disk.
- `*notify_change`: Called when inode attributes are changed.
- `*put_super`: Called by the VFS on unmount to release the given superblock.

- `*write_super`: Called when the VFS decides that the superblock needs to be written to disk.
- `*statfs`: Obtain file system statistics.
- `*remount_fs`: Called by the VFS when the file system is remounted with new mount options.
- `*clear_inode`: Release inode and clear any pages containing related data.

The Inode Object

An inode is associated with each file. The inode object holds all the information about a named file except its name and the actual data contents of the file. Items contained in an inode object include owner, group, permissions, access times for a file, size of data it holds, and number of links.

The inode object also includes an inode operations object that describes the file system's implemented functions that the VFS can invoke on an inode. The methods defined for the inode object include:

- `create`: Create a new inode for a regular file associated with a dentry object in some directory.
- `lookup`: Searches a directory for an inode corresponding to a file name.
- `mkdir`: Creates a new inode for a directory associated with a dentry object in some directory.

The Dentry Object

A dentry (directory entry) is a specific component in a path. The component may be either a directory name or a file name. Dentry objects facilitate access to files and directories and are used in a dentry cache for that purpose.

The File Object

The file object is used to represent a file opened by a process. The object is created in response to the open() system call and destroyed in response to the close() system call. The file object consists of a number of items, including:

- dentry object associated with the file
- file system containing the file
- file objects usage counter
- user's user ID
- user's group ID
- file pointer, which is the current position in the file from which the next operation will take place

The file object also includes an inode operations object that describes the file system's implemented functions that the VFS can invoke on a file object. The methods defined for the file object include read, write, open, release, and lock.

13.4 LINUX NETWORKING

Linux supports a variety of networking architectures, in particular TCP/IP by means of Berkeley Sockets. Figure 13.7 shows the overall structure of Linux support for TCP/IP. User-level processes interact with networking devices by means of system calls to the Sockets interface. The Sockets module in turn interacts with a software package in the kernel that handles transport-layer (TCP and UDP) and IP protocol operations. This software package exchanges data with the device driver for the network interface card.

Linux implements sockets as special files. Recall from Section 12.7 that, in UNIX systems, a special file is one that contains no data but provides a mechanism to map physical devices to file names. For every new socket, the Linux kernel creates a new inode in the *sockfs* special file system.

Figure 13.7 depicts the relationships among various kernel modules involved in sending and receiving TCP/IP-based data blocks. The remainder of this section looks at the sending and receiving facilities.

Sending Data

A user process uses the sockets calls described in Section 13.3 create new sockets, set up connections to remote sockets, and send and receive data. To send data, the user process writes data to the socket with the following file system call:

```
write(sockfd, mesg, mesglen)
```

where `mesglen` is the length of the `mesg` buffer in bytes.

This call triggers the `write` method of the file object associated with the `sockfd` file descriptor. The file descriptor indicates whether this is a socket set up for TCP or UDP. The kernel allocates the appropriate data structures and invokes the appropriate sockets-level function

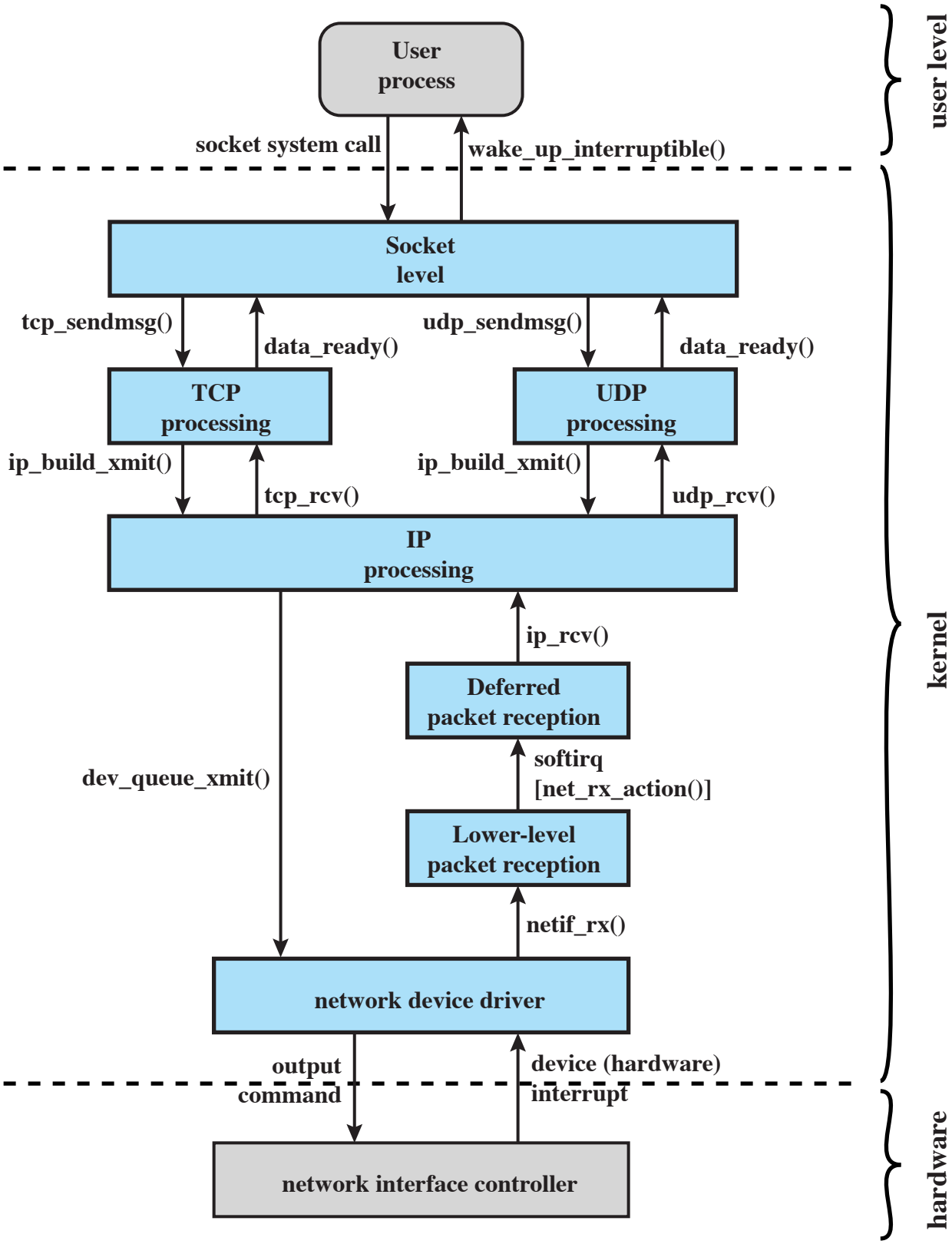

Figure 13.7 Linux Kernel Components for TCP/IP Processing

to pass data to either a TCP module or a UDP module. The corresponding functions are `tcp_sendmsg()` and `udp_sendmsg()`, respectively. The transport-layer module allocates a data structure of the TCP or UPD header and performs `ip_build_xmit()` to invoke the IP-layer processing module. This module builds an IP datagram for transmission and places it in a transmission buffer for this socket. The IP-layer module then performs `dev_queue_xmit()` to queue the socket buffer for later transmission via the network device driver. When it is available, the network device driver will transmit buffered packets.

Receiving Data

Data reception is an unpredictable event and so involves the use of interrupts and deferrable functions. When an IP datagram arrives, the network interface controller issues a hardware interrupt to the corresponding network device driver. The interrupt triggers an interrupt service routine that handles the interrupt as part of the network device driver module. The driver allocates a kernel buffer for the incoming data block and transfers the data from the device controller to the buffer. The driver then performs `netif_rx()` to invoke a lower-level packet reception routine. In essence, the `netif_rx()` function places the incoming data block in a queue and then issues a soft interrupt request (`softirq`) so that the queued data will eventually be processed. The action to be performed when the `softirq` is processed is the `net_rx_action()` function.

Once a softirq has been queued, processing of this packet is halted until the kernel executes the softirq function, which is equivalent to saying until the kernel responds to this soft interrupt request and executes the function (in this case, `net_rx_action()`) associated with this soft interrupt. There are three places in the kernel, where the kernel checks to see if any softirqs are pending: when a hardware interrupt has been processed, when an application-level process invokes a system call, and when a new process is scheduled for execution.

When the `net_rx_action()` function is performed, it retrieves the queued packet and passes it on to the IP packet handler by means of an `ip_rcv` call. The IP packet handler

processes the IP header and then uses `tcp_rcv` or `udp_rcv` to invoke the transport-layer processing module. The transport-layer module processes the transport-layer header and passes the data to the user through the sockets interface by means of a `wake_up_interruptible()` call, which awakens the receiving process.

14.7 BEOWULF AND LINUX CLUSTERS

In 1994, the Beowulf project was initiated under the sponsorship of the NASA High Performance Computing and Communications (HPCC) project. Its goal was to investigate the potential of clustered PCs for performing important computation tasks beyond the capabilities of contemporary workstations at minimum cost. Today, the Beowulf approach is widely implemented and is perhaps the most important cluster technology available.

Beowulf Features

Key features of Beowulf include [RIDG97]:

- Mass market commodity components
- Dedicated processors (rather than scavenging cycles from idle workstations)
- A dedicated, private network (LAN or WAN or internetted combination)
- No custom components
- Easy replication from multiple vendors
- Scalable I/O
- A freely available software base
- Use of freely available distribution computing tools with minimal changes
- Return of the design and improvements to the community

Although elements of Beowulf software have been implemented on a number of different platforms, the most obvious choice for a base is Linux, and most Beowulf implementations use a cluster of Linux workstations and/or PCs. Figure 14.18 depicts a representative configuration. The cluster consists of a number of workstations, perhaps of differing hardware platforms, all running the Linux operating system. Secondary storage at each workstation may be made available for distributed access (for distributed file sharing, distributed virtual memory, or other

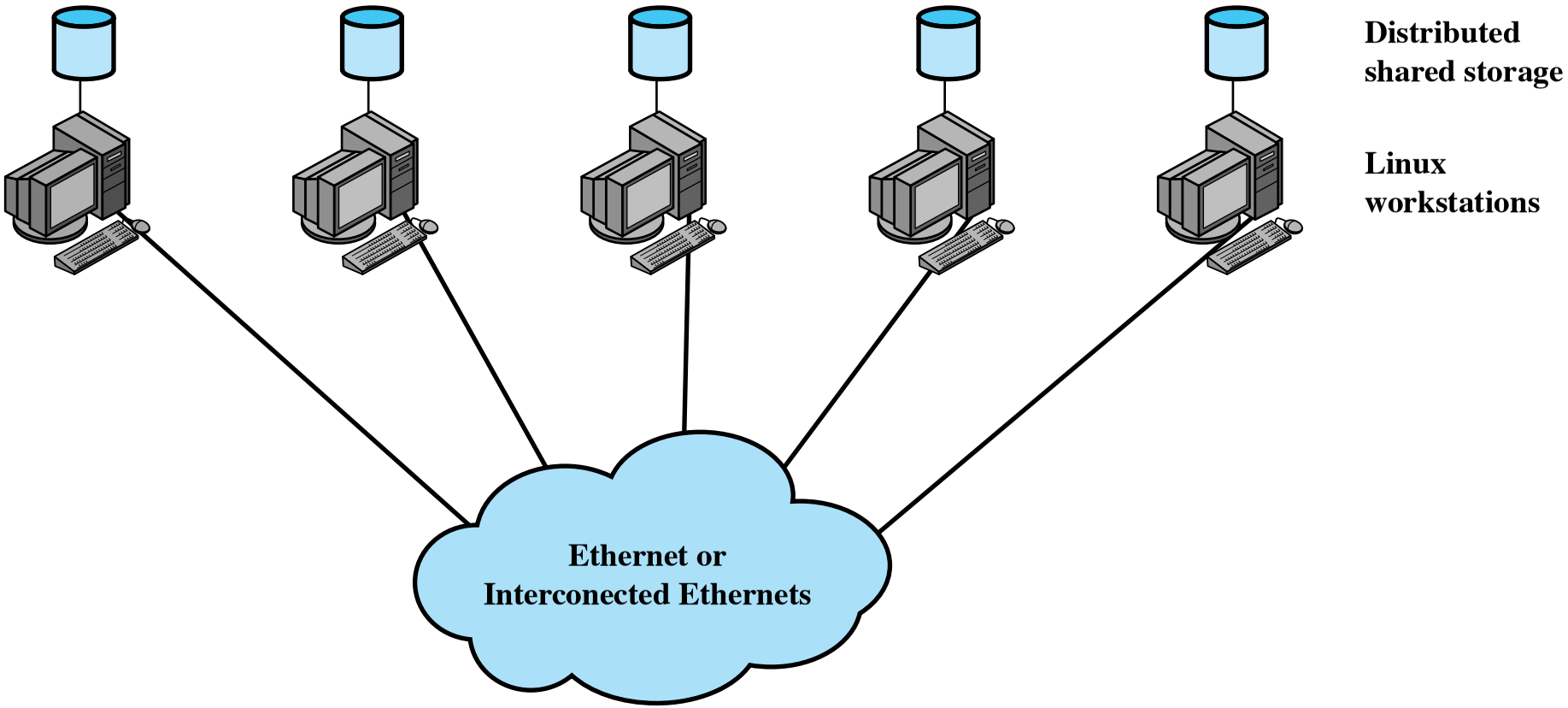

Figure 14.18 Generic Beowulf Configuration

uses). The cluster nodes (the Linux systems) are interconnected with a commodity networking approach, typically Ethernet. The Ethernet support may be in the form of a single Ethernet switch or an interconnected set of switches. Commodity Ethernet products at the standard data rates (10 Mbps, 100 Mbps, 1 Gbps) are used.

Beowulf Software

The Beowulf software environment is implemented as an add-on to commercially available, royalty-free base Linux distributions. The principal source of open-source Beowulf software is the Beowulf site at www.beowulf.org, but numerous other organizations also offer free Beowulf tools and utilities.

Each node in the Beowulf cluster runs its own copy of the Linux kernel and can function as an autonomous Linux system. To support the Beowulf cluster concept, extensions are made to the Linux kernel to allow the individual nodes to participate in a number of global namespaces. Some examples of Beowulf system software:

- **Beowulf distributed process space (BPROC):** This package allows a process ID space to span multiple nodes in a cluster environment and also provides mechanisms for starting processes on other nodes. The goal of this package is to provide key elements needed for a single system image on Beowulf cluster. BPROC provides a mechanism to start processes on remote nodes without ever logging into another node and by making all the remote processes visible in the process table of the cluster's front-end node.
- **Beowulf Ethernet Channel Bonding:** This is a mechanism that joins multiple low-cost networks into a single logical network with higher bandwidth. The only additional work over using single network interface is the computationally simple task of distributing the packets over the available device transmit queues. This approach allows load balancing over multiple Ethernets connected to Linux workstations.

- **Pvmsync:** This is a programming environment that provides synchronization mechanisms and shared data objects for processes in a Beowulf cluster.
- **EnFuzion:** EnFuzion consists of a set of tools for doing parametric computing, as described in Section 14.4. Parametric computing involves the execution of a program as a large number of jobs, each with different parameters or starting conditions. EnFusion emulates a set of robot users on a single root node machine, each of which will log into one of the many clients that form a cluster. Each job is set up to run with a unique, programmed scenario, with an appropriate set of starting conditions [KAPP00].